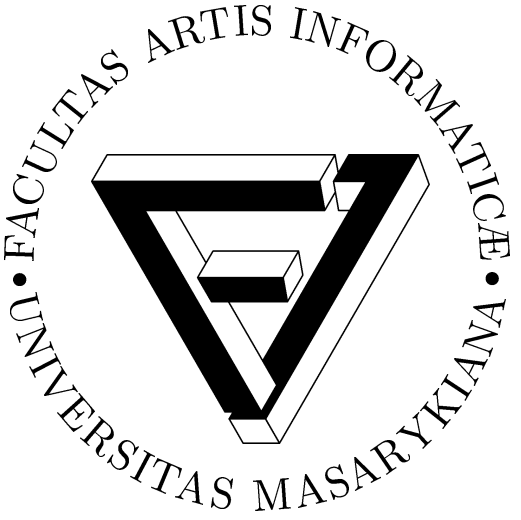
Masaryk University
Faculty of Informatics



Inheritance of SOFA Components

Master Thesis

by
Tomáš Opluštil

supervised by
Prof. Ing. František Plášil, DrSc.

Brno, 2002

Acknowledgements

I would like to express my deepest thanks to my supervisor Prof. Ing. František Plášil, DrSc., from Charles University in Prague, Czech Republic, who assigned me the subject of this thesis and thus invited me to the world of the cutting--edge software engineering research and whose wise guidance supported by his rich experience helped me not to lose my way in that world.

My warmest thanks go to Assoc. Prof. RNDr. Renata Ochranová, CSc., from Masaryk University in Brno, Czech Republic, for her immense multilateral support and encouragement during the whole process of writing this thesis.

I would also like to thank Prof. Markku Sakkinen from University of Jyväskylä, Finland, for his initial hints concerning inheritance in programming languages.

Declaration

I hereby declare that this master thesis has been written solely by me and that all the sources I have used in this thesis have been explicitly cited and included in the bibliographical references.

Brno, April 8^{th}, 2002

Abstract

Component software construction has been a very intensively researched branch of the nowadays software engineering discipline because it seems to be the best answer to the increasing demands on complexity, reliability, maintainability and configurability of software systems which is due to its ability to integrate all these otherwise contradictory requirements. Therefore, a lot of component frameworks have emerged recently, often as results of theoretical studies in this field. SOFA/DCUP component model that has been founded and developed by the Distributed Systems Research Group at Charles University in Prague belongs to one of the most promising ones. This thesis contributes to this project by analyzing the assets of incorporating inheritance into each of three main abstractions of the SOFA's component specification language CDL, analyzing various inheritance mechanisms how they suit needs of each of those three abstractions and proposing and elaborating the particular inheritance mechanisms that suit best. Also, some other issues concerning SOFA/DCUP, inheritance, programming languages and component software have been discussed in this work.

Keywords

inheritance, SOFA/DCUP, component software, software specifications, architecture description languages, interface definition languages, object oriented programming, programming languages, software engineering

Contents

Chapter 1

Introduction

There is nothing more difficult to take in hand, more perilous to conduct or more uncertain in its success, than to take the lead in the introduction of a new order of things.
— Niccolò Machiavelli (The Prince, 1513)

1.1 The main goal of this thesis and its context

In the last decade, we experience massive development in the area of the IT industry. The words like e-business, digital economy or e-government have become buzz-words. Although it is mostly due to the vast progress in technologies and related computer hardware and network capabilities, this progress necessarily induces significant changes in the software engineering. Information technologies have been used in completely new areas by many thousand times more people than only a decade ago. However, the demands on software systems are commensurate. Now, sophisticated reliable software systems that fully support security, dynamic reconfiguration and automatic updating, ensure quality of services and take advantage of latest technologies and, at the same time, systems that are developed quickly and maintainable easily, are demanded. People from the software industry have been in a quest for the best answer to this challenge and so far component technologies seem to be the optimal solution.

Several component models have been introduced but neither is fully satisfiable yet and their development is still in progress. One of such component models called SOFA/DCUP has been in development at the Department of Software Engineering at the Charles University in Prague. This thesis can be viewed as a small part of this effort. The part of this effort consists of analyzing issues concerning usefulness and practical applicability of inheritance for the Software Appliances (SOFA) Component Definition Language (CDL) abstractions.

This fact determines the nature of the whole work. We cannot start from the scratch but we must at least roughly introduce the current situation in the software engineering, the basic notions from the SOFA/DCUP component model and basic facts about the inheritance in programming languages, mostly object oriented.

1.2 Structure of the rest of this thesis

1.2.1 General structure of the thesis

Chapters two, three and four are kind of overview chapters that represent a very important groundwork for our further research. They summarize basic knowledge from the fields which essentially touch subjects of our further work and they provide readers with the basic information without having to consult the papers referred. The three main fields of study are as follows: contemporary software engineering, inheritance in object oriented programming languages and the SOFA/DCUP component model. These chapters are not necessarily comprehensive and mostly represent the author's view of these topics gained from bunch of papers, however, they contain enough information to understand the rest of this thesis.

Chapters five, six, seven and eight represent the core of this thesis. They deal with various problems component systems bring and the chapters six to eight are devoted to individual SOFA CDL abstractions and the elaboration of the possibilities of incorporating inheritance into individual SOFA CDL abstractions.

Chapters nine and ten are devoted to evaluations and conclusions.

From the readers' viewpoint, chapters two, three and four represent compact units that can be read independently and make sense per se. The remaining chapters are not recommended to be read without having previously read the first five chapters (without being really well acquaint with the problems presented in those chapters). Moreover, these chapters are recommended to be read in order they are written because readers can often find references to previous parts of the work.

1.2.2 Structure of individual chapters

Chapter two will be devoted to looking back at development in the area of software engineering, including the component technology. We will try to learn lessons from history of software engineering and, in the following sections, we focus on two main approaches in software engineering which seem to withstand the demands of current fast development in the IT industry: the object oriented approach and the component-based approach. Even though this chapter may seem a bit off-topic, we do not recommend readers to skip it because it provides us with basic trends and directions we should follow in this work and presents concepts upon which nowadays software technologies rely.

Chapter three is devoted to the notion of inheritance in object oriented programming languages. Readers can find there initial explanations of fundamental theoretical and practical benefits and origins of inheritance followed by two main sections: One is devoted to various classifications of inheritance from its usage viewpoint and the other presents various inheritance mechanisms and their variations that can be used in object oriented programming languages and can be useful for our thesis. An initial brief evaluation of the presented concepts and mechanisms in the relation to the SOFA CDL is presented at the end of the chapter.

Chapter four is devoted to the SOFA/DCUP component model. First, it introduces fundamental concepts SOFA/DCUP component model is based on

and parts it consist of. The major part of this thesis is devoted to the SOFA CDL, thus readers can get acquaint with the abstractions of SOFA CDL for which the inheritance should be discussed and other related important things, such as protocols. This chapter can provide readers with a very basic imagination of other parts of the project, including Dynamic Component Updating and the SOFA's implementation and deployment facilities. At the end of this chapter, an example how to create a component specification is shown.

Chapter five recapitulates the main objectives and a basic analysis of the problems we are going to solve from the general viewpoint and, subsequently some common aspects which are important to be clarified prior dealing with the concrete problems or some other interesting topics somehow related to the field of study are discussed. These common aspects include problems of types and subtypes, problems of semantics, and also a problem o protocol canonical form is sketched (however, not solved).

Chapter six presents the first and lowest-level abstraction of the three ones for which we should discuss the inheritance: interfaces. First, the notion of interfaces is introduced, subsequently various advantages and disadvantages of implementing inheritance into this abstraction are discussed, followed by a presentation of various aspects of possible implementations of various inheritance mechanisms. The chapter is concluded by some proposals what inheritance type seems to be optimal, including some syntax proposals.

Chapter seven presents the second abstraction: component template frames. First, problems of subtypes and substitutability are presented upon which we introduce an example of how the formal representation of this abstraction could look like. Then we discuss possible inheritance from many viewpoints, similarly as in the previous chapter, and, also similarly as in the previous chapter, some proposals what inheritance mechanism seems optimal and its syntax are presented.

Chapter eight is devoted to the last abstraction we need to discuss the inheritance for: component template architectures. After a summarization of main problem domains that are related to architectures, two main ones are elaborated: substitutability of subcomponents in architectures and then issues concerning inheritance in architectures are thoroughly discussed and the inheritance type that seems optimal is proposed at the end of the chapter. Also some additional problems are discussed and some additional concepts (problems of ties, initial formalization of the architecture notion, etc.) are introduced during the elaboration of these main task.

Chapter nine presents an example of specifying software components for passive banking services. In this example, we take advantage of the proposed inheritance mechanisms and show that the proposed concepts are viable.

The last chapter brings an evaluation of the current state of the work and summarizes some further tasks that have to be finished in order to be the work also practically useful, and research issues that have been initiated in this work and should be further examined. We also briefly introduce related component systems and highlight abilities or inabilities of incremental modification of their abstractions. The conclusion is made at the end of the chapter.

Chapter 2

Analysis of the contemporary software engineering

Taken by and large, programmers are a rude, irritable, intolerant, arrogant, insufferable, prejudicial and bigoted lot.
— Dr. Stenley Gill, pioneer of computing

2.1 Principles of software engineering and lessons from its evolution

Computer science has been undoubtedly the most evolving (and exciting) domain of human endeavor for the last fifty years. There is hardly any scientific area that led in such a short time to so many achievements in both theoretical and practical fields and whose results changed the face of the world so dramatically.

To understand the objective of this master thesis, we need to look back at the development in a subdomain of computer science called *software engineering*, which is not an exception in the sense of development rapidity mentioned in the previous paragraph.

2.1.1 Software engineering as a computer science discipline

Software engineering is quite an empirical discipline. As stated in [Kral–98] or [KraDem–91], the primary aim of software engineering is to find modus operandi in creating software of demanded properties and quality and managing the pieces of software as workable software systems. Proposals for effective methods concerning software should come out of knowledge of three aspects: hardware, technological know-how and workflow management (rules of team cooperations, documentation creation standards, etc.). It is important especially in case of large-scale enterprise software systems, where demands for security, reliability,

efficiency, etc. are very high. Therefore, the whole lifecycle of those software systems should be controlled and preferably handled in a standardized way to minimize the risks of failure (in any form). This all brings the development of large-scale software systems closer to other engineering areas which also deal with problems of lifecycle phases, quality management, etc. By the way, this fact is reflected even in the terminology used: for example *software architecture* obviously originates in terminology of civil engineering.

2.1.2 Problems of the software lifecycle

The software lifecycle comprises a lot of individual phases, starting with the phase of analysis of business needs of the organization for which the software is to be developed, through the software creation itself to the maintenance of working systems and monitoring the needs of their replacement with new ones. It is important to realize that the proposed methods, in addition to trying to automate and standardize as many activities as possible, should also facilitate the transitions between the individual software lifecycle phases. For dozens of obvious reasons it is desirable to make the transitions "invisible" (the downside of the opposite will be shown in next paragraphs).

As we look through the history of software engineering (for example in [RiSoch–94]), we can find out that individual phases of software development were often dealt with separately. It all began in the late sixties with the discussions concerning basic methodologies of software development e.g. top-down programming, continued with discussions concerning structural programming and the appropriateness of "goto" and "goto-less" programs in the early seventies, consecutively focusing on the possibilities of verification of software soundness in middle seventies, to return back to the early phases of software lifecycle discussing how to design and specify pieces of software before they are going to be programmed (which resulted in modular decomposition, structure charts, etc).

Naturally, the goal of reaching fine transitions between the individual lifecycle phases and thus advancing on the goal of reducing possible risks associated with software, could not be achieved this way. To show the consequences, let's discuss the case of two important phases of the software lifecycle: *software analysis & design*[1] and *software programming*[2]:

For many years, they had been representing two very different areas with their own abstractions that had nothing in common except for the fact that the piece of software had to be programmed according to the specifications obtained from the analysis & design phase. This fact had a lot of inherent drawbacks: First, there were different people with their own — in the terminology used in [Stanic–99] — ideal worlds in their heads working on each of the phases, which often led to misunderstanding and inconsistencies during the transformation of the design phase abstractions to the abstractions of the selected programming language. Second, since the transformation had to be done by hands, it was slow, and, since a technique called *waterfall*[3](see e.g. [Kral–98]) was the major

[1] in a closer look, analysis and design are two disticnt phases but for now, we will consider them as one

[2] often called *coding*

[3] which also has this consequence: once your design specifications are not correct, all other phases will be useless

technique of software development, and, moreover, the software was developed ad hoc and at once, it was totally inflexible. Such a way of development was, of course, also expensive and the final software system was very hard to maintain.

2.1.3 Recent trends in software engineering

Meanwhile, along with the rapid development of hardware capabilities, the demands for software capabilities had been increasing and thus software became larger and larger, which naturally led to bigger demands on efficiency of software development and maintenance process. Fortunately, as we have already mentioned, software engineering can take advantage of hardware advances (and also workflow management improvements) as-well and thus some first tools, so called *CASE*s[4], were devised in the early eighties to support software designers' task to design the software specifications and also (a bit later) *IDE*s[5] to support developers' task to implement these specifications. However, since originally each of these two types of tools focused solely on its own task, which undoubtedly helped to increase the speed and the quality of these two phases as such, the transition between them remained still quite coarse and error-prone. Obviously, a more unified view was needed and object oriented programming was the first step following this concept.

2.2 Object oriented programming — fundamentals

Object oriented programming — a topic about which thousands of pages have been written and other thousands are yet to be written. Therefore, we will focus merely on main OOP aspects that are significant for better understanding of concepts we will deal with further in this thesis.

2.2.1 Object oriented programming: a revolution or an evolution?

Object oriented style of programming is quite an old programming paradigm, in fact, much older than the first CASEs a IDEs mentioned in the previous section. Its origins date back to late sixties of the twentienth century when *Simula 67* — the first programming language with object oriented features — was invented to solve simulation (model building) problems. The need for such a language originally arose from the need to model objects of the real world (i.e. physical systems) using some software abstractions. Simula's main concepts were further refined in *Smalltalk* where this approach was originally meant as a pedagogical tool. But it took two more decades before commercially successful languages with object oriented features — such as C++, Object Pascal or Java — appeared and, together with even more recently adopted object oriented methods of analysis & design of which Unified Modeling Language (UML) is

[4] which is an acronym for Computer Aided Software Engineering
[5] which is, in this case, an acronym for Integrated Development Environment

the best known, caused the boom of object oriented systems, we experience today.

As for the question from this subsection's title: If we have a look at those commercially successful object oriented languages like C++ or Object Pascal, the answer is seemingly simple: those languages were in an evolutionary process derived from C or Pascal, respectively, by adding some new object oriented features to the language syntax. Programmers are free to decide whether or not to use these features and even about the extent of their usage[6]. Such a reality is in our opinion the real danger to the proper understanding of object oriented programming because, in our opinion, the object oriented approach to programming definitely is a revolution, despite the fact that it is often considered as an evolution only. Now, we will explain why.

2.2.2 Nature of the object oriented revolution

The object oriented programming is often characterized as *programming with taxonomically organized data* (e.g. [Meyer–97]), which expresses the fact that data-centric decompositions of programs are involved, unlike formerly used action-centric procedural decompositions (as presented in e.g. [Ochran–79]). According to the way data are handled, two main types of OOPLs can be recognized: *class-based languages* and *object-based languages*. We will explain those two notions further in this chapter.

We should notice that this represents the first real attempt to create a link between the software analysis & design on the one hand and the software programming on the other. We should also realize that for the full use of the object oriented approach, following two conditions are necessary to be satisfied:

1. A programmer must analyze and design the given problem in the object oriented way.

2. A programming language which has a syntactic support for object oriented features (abstractions) must be used.

This fact is very clearly documented in the comparison in [OchKoz–93], where a single problem is solved by three approaches: the procedural one, the object oriented design only approach and the approach where full object oriented design plus object language features are involved. Let's examine the two items: If the former condition is satisfied only, it is still better for larger systems than when neither of these is satisfied. For instance, the whole first version of Windows NT was programmed this way. If the latter condition is satisfied only, then it is a gross misuse of such an object oriented language because it breaks the rules of object oriented programming, confuses readers of such a program and brings more troubles than benefits. Unfortunately, the author of this master thesis has gained a lot of experience with such programs during his teaching in introductory object oriented programming seminars.

We can conclude that the fact that you have to think in a completely different way while creating a real object oriented program (often based upon constructions from procedural languages) is the heart of the matter. The revolution

[6] of course, there are dozens of experimental OOPLs that force programmers to programm in the object oriented way but they have kind of little impact outside the academic community

involves a completely new approach to programming. Researchers in object oriented field indeed realized this fact as-well and, as emphasized by Prof. Niklaus Wirth on the occasion of his being awarded honoris causa at Masaryk University in 1999, the researchers tried to avoid possible misunderstanding by completely changing the terminology for well-known abstractions from procedural programming (e.g. "procedure call" was renamed as "message despatch", etc.). But it did not help much for languages derived from procedural ones mentioned above. Java, as the third commercially successful OOPL, changed this fact a bit because it was developed from scratch, however, not all programmers use Java (especially for its slowness given by its implementation design for the needs of Internet applications[7]).

2.2.3 Class-based object oriented programming languages

Class-based languages form the mainstream of object oriented programming. They are based on the old Aristotle idea of classifing things into categories. Every class consists of individual things (objects) of the same category. And that is how class-based languages work: there is an abstraction referred to as *a class* which describes a structure (i.e. particularly characteristic features) of individual concrete *objects* which can be created (instantiated) from that class. Therefore, such objects are often refered to as *instances of a class*. In a bit different view, each instance (of a class) is characterized by its state which must be compatible with its mother class specifications but different instances of the same class can be found at the same time in different states (but, again, these states must be allowed by the class description). In the terminology of programming languages, a class is *a type* and individual objects (instances) of such a class are *variables* of that type. The question of types in object oriented programming languages will also be mentioned in subsection 5.2.1 on page 41.

2.2.4 Object-based object oriented programming languages

Object-based languages represent the more recent and more gradually evolving part of the object oriented dichotomy. Their main advantages over class-based languages are simplicity and flexibility. Thus, they represent the response to the calls of occasional programmers to simplify object oriented programming. Their characteristic feature is that object-based languages do not have classes but just simple full-fledged objects. Programmers do not design descriptions first (classes) and then instantiate them but they just create true objects. In most object-based languages, every object can serve as a *prototype* for the creation of its clones. Such clones can be further modified (each separately). As pointed out in [CardAb–96], since object-based languages originated in Lisp and artificial intelligence community, little attention was devoted to creating typed object-based languages. Therefore, most object based languages are very simple supporting only the notions of objects and dynamic dispatch; when typed, they support only object type, subtyping and subsumption[8].

From these facts we can conclude that object-based languages are not much suitable for programming in large, because they lose most advantages appreciat-

[7] most compilers compile so called bytecode for the Java virtual machine which is subsequently interpreted by a Java run-time specific for a particular platform

[8] we will explain the notions of subtyping and subsumption in 5.2.4 on page 44

ed by enterprise, such as reliability, maintainability, and also the interconnection between the design phase and the implementation phase is limited. Therefore, most object-based programming languages are valuable only for their theoretical aspects (e.g. Emerald, Cecil, Omega, Kevo and Self)[9]. And, since the above mentioned properties are considered as essential for the object oriented approach, some researchers reject object-based languages. For example, Prof. Sakkinen in [Sakkin–89] claims that to reject classes is to throw the child out with the wash.

In this study, we will take into consideration some interesting inheritance principles of object-based languages, however, as the nature and purpose of component systems are much closer to the class-based object oriented languages, we will not deal with object-based languages anymore.

2.2.5 Is object oriented programming worth using?

We have talked about a new approach to programming, etc. so far, but is this style really worth using in the real software construction? Well, we think that in a serious software creation, if it is done well, then definitely yes[10]. A lot of books have been written about this matter, e.g. [Meyer–95] is a good one which deals with the assets of the object oriented approach for enterprise. Now, we will briefly bring some of them out: The central motto of object oriented programming is: *If you think writing software is difficult, try rewriting software.* The object oriented approach allows us to

- make software analysis, design and implementation seamless
- make better software architectures[11]
- make programs which more closely correspond to reality (thanks to object oriented modeling (from the design point of view) and polymorphism (from the implementation point of view)
- make programs transparent and readable (thanks to coherent design and the selfishness principle: Tell me not what you are; tell me what you can do for me!)
- make software extendable and open (by involving inheritance and dynamic binding)
- make and modify software with a lesser effort

These things altogether make, for serious software usage, software more reliable (less error-prone, ...) more reusable, more portable, cheaper to maintain and bring a lot of other benefits.

Of course, while programming "in small", the need for the initial good design is something occasional programmers cannot overcome, therefore they often choose to create their programs procedurally. But this fact cannot degrade the value of object oriented programming in the eyes of professional programmers. However, there is still a place for improvements, especially as for the even better

[9] although e.g. Self is supported by such a big corporation as Sun Microsystems

[10] especially using class-based languages, see next subsections

[11] from [Meyer–95]: architecture is an organization to coherent pieces and description of how these pieces interact with each other

cooperation between the design and the implementation phases and the distribution of very large software systems deploying pieces of a system on various places over the net. These are the main aspects the component based approach promises to solve.

2.3 Component-based software development

This is a kind of paradoxical situation with component-based development: it is a much more evolutionary (than revolutionary) step from the object oriented approach, compared with the case of the relation between the procedural and the object oriented approaches described above, but it is not essential to use any object oriented programming language as the underlying technology.

This section, in which we will try to outline the main facts about and principles of the component software construction, should provide us with the answer of how this is possible, and it should also mention other things relevant to further progress of this thesis.

2.3.1 Definition of software components

Component based development is a matter of the last decade; therefore, this area has been intensively studied and definitions refined. However, let's cite a widely accepted and exact enough definition of the component (from Workshop on Component-Oriented Programming at ECOOP 1996):

A *software component* is a unit of composition with contractually specified interfaces and only explicit context dependencies. Such a software component can be deployed independently and it is a subject of composition by third parties

A software system consists of a set of such independently deployed components which communicate in order to be able to provide intended functionality.

As you could notice, only interfaces (with operations) are mentioned in the component definition. This is one point, where the concept of components radically differs from the concept of objects. As we stated in subsection 2.2.3, a key characteristic of objects is that they have states (given by values of their member variables). However, since components are determined by their services, they do not have states. This complication will be discussed later.

2.3.2 Motivation for component-based development

As mentioned at the end of the previous section, even the object oriented approach gasps while facing the demands of contemporary intricate software systems. Let's enumerate several major aspects that ought to be improved over the object oriented approach:

1. Reuse of not only source code but also complete binary software pieces.
2. Better utilization of hardware power; including utilization of computer networks.
3. Better verification of software soundness and thus improving reliability.
4. Better flexibility in changing and updating software configurations.

These targets require some technological and structural requirements to be met. We will describe them in the next subsections.

Before we proceed, let's briefly mention three, a bit more specific, arguments why to use components introduced by Clemens Szypeski in [Szyper–00].

1. Baseline argument: combining of self-created strategic components with general-purpose third party components, perhaps even bought as off-the-shelf ones.

2. Enterprise argument: by skillful component factoring, several product lines can be covered by configuring a core set of components plus some specific ones. Product creation is thus based on a specific component configuration and can be controlled by versioning.

3. Dynamic computing argument: Modern software systems challenge growing set of content types to be processed. Well designed systems can be dynamically extended (and upgraded) to meet the new requirements.

However, there can be hundreds of arguments found why to use components for particular solutions and we presented the three of them as exemplary ones only. Therefore, as promised, those more general goals above are going to be examined further.

2.3.3 Specification of component design and component definition languages

Component-based software creation makes a step backwards and strictly separates the design (specification) phase and the programming (implementation) phase, albeit on a new evolutionary level. This allows us to specify software design more precisely, consider software pieces independently on a programming language and shift most work (from functional apportionment to verification) onto the specification side, thus setting a base for better flexibility and reliability of software.

Therefore, a special new type of programming languages, so called *specification languages* or *definition languages* must be established. Their characteristic feature is that they are not compiled to a binary executable form because no implementation is involved. You can (a bit inaccurately) imagine this like taking a program structure skeleton plus formalized replacement of informal comments and plain text descriptions, enriched with dozens of new features. Such languages are used for lots of purposes, e.g. functionality layout, architecture description, soundness verification, etc.

The idea of such languages is not completely new; they are mostly based on, mostly single-purposed languages used in particular cases earlier. As for some examples, software specification languages include *Z*, *OBJ* or *Vienna Development Method*. Languages used to specify communication protocols include *Language of Temporal Ordering Specifications*, *Estelle*, *Process Meta Language*, or *Specification and description language*. However, the best known specification languages are *Interface Definition Languages* (IDLs) used in CORBA or COM[12]. We will show usage and possibilities of one such particular language

[12] all these notions will be explained later

(called Component Description Language, or CDL for short) in Chapter 4 while introducing SOFA/DCUP Component Model.

2.3.4 Component architectures

While describing a particular component framework, it is important to describe its architecture. As mentioned earlier, architecture is a specification of the components of the system and communication between them. As emphasized in [LuVeMe–00], such an architecture guarantees certain behavioral properties of conforming systems and can be a powerful tool which is to aid the process of predicting behavior of the system with that architecture, managing the construction of such a system and maintaining it.

In other words (with a bit of simplification): We have a set of component descriptions and we want to describe how they communicate. In most cases, this goal is achieved by determining what services a component provides and what services it requires from the external environment to be able to provide them. And while projecting a component system, we have to describe links between a required service of one component and a *conforming* provided service of another. It may seem simple but it is not. We can call this a *service connection* or, in the terminology of SOFA/DCUP component model, a *tie*. Such a tie has to be correct (or *sound*). What it means: you can notice that we did not use the word "corresponding" but "conforming" several lines above. To express the nature of this notion, let's borrow the general definition from [LuVeMe–00] again: If the contstraints of two components connected by a service connection together are sufficient to ensure that the constraints given in the service definition are satisfied, then the connection is correct. In other words (a bit simplified again): in such a connection, an involved service of the component that provides the given service can have at least the same and possibly better capabilities than what is actually required from the service by the component on the requirement side of the connection. This is, of course, much complicated and it might cause troubles while deliberating the architecture inheritance further in this thesis.

Such architecture descriptions are created in special *Architecture Definition Languages* (or ADLs for short). There are a lot of architecture types, even hierarchical (so called multi-tier) ones, and similarly a lot of ADLs. We will show the usage of a particular ADL while describing the SOFA/DCUP's CDL in Chapter 4, which can be viewed as an ADL as-well.

2.3.5 Software distribution and component deployment

Another important point in the goals of contemporary software engineering was the better utilization of hardware, including computer networks. The solution of this goal is an inherent feature of the component-based software because one of software components' characteristic features is that they are units of independent deployment.

Thus, we have binary components that are deployed somewhere on a given computer or across a network. However, we need low-level connection standards and facilities "in the middle" to locate such components, find if they are appropriate or assist the software system functionality in another way. Therefore, some middleware systems had to be developed to accomplish this task. Component models are equipped with such middleware systems. OMG CORBA, Microsoft

COM/DCOM/COM+ or Sun EJB can be mentioned as the best known. Such models include not only type repositories and interoperability facilities, but often even transaction monitors, load balancing mechanisms and other things necessary for reasonable software distribution and hardware utilization.

With the component design and deployment another issue that has a great importance for this thesis is connected: the weight of components. One of the main reasons why software is divided into components is that we do not want heavy-weighted software that is cumbersome, hard to maintain and update, unnecessarily repeated (due to its self-containment and — as termed by C. Szyperski — "introversion"). The inherently distributed component software offers a good solution. However, inapt component design could negate this advantage. Therefore, it is essential to pay attention to commonsensical separation of functionalities into individual software components. We will discuss this problem in the chapters devoted to finding optimal solution of the component inheritance problems.

2.3.6 Component assembly and component markets

Component-based software construction allows changes even in such nontechnical aspects as forms of software marketing and trading. There are supposed to be lots of prefabricated off-the-shelf components from various vendors that could be, provided good specifications are available and after checking their compatibility, assembled into larger systems. This task can be done by *assemblers* that are independent on component vendors and component system users. Such a system could be subsequently distributed across the net according to the demands of the contractor.

The working models of such component trading have not been established yet, however, it is obvious that the component trading would help the software reuse (even at the binary level), which is a very important issue because, if we look at the composition of typical applications, they often use a great deal of the same procedures and techniques. If such a code is reused, it is done mostly in a form of libraries, the procedures from which are used in the source code of the given application. But assembling the binary forms of the code would be much easier. Moreover, great possibilities of reuse are apparent even on a higher level of software construction: As emphasized in [Stanic–99], functional decomposition of a typical organization is always the same — finance and accounting, human resources management, material possession, marketing and trading and a primary process; with only the primary process significantly differing depending on individual organizations. This can also be achieved by component composition (see the baseline argument in subsection 2.3.2).

By now, we can meet component assembly in quite primitive forms of visual assembly of classes written in a standardized manner. Such visual assembly capabilities are incorporated in various development environments (e.g. Visual Component Library by Borland into Delphi or C++Builder or JavaBeans by Sun into several development environments for Java). Despite the primitiveness of such a visual assembly, we can find dozens of electronic marketplaces over the Internet, where VCLs, JavaBeans, or ActiveX components are traded, that's why we presume that such a concept is not an utopian vision but a viable form of software trading in the near future.

2.3.7 Software versioning & updating

Another commendable property of software components is their flexibility. When we have a good component model (like SOFA/DCUP[13]), updating and changing configuration of even running applications is quite easy and it is supposed to be quite a frequent task. This fact has to be taken into consideration when deliberating the inheritance problems further in this study.

However, while replacing one component with another, it is always necessary to be sure that the involved components are compatible to preserve configuration consistency. Besides their behavioral compliance automatically detectable from behavioral protocols of such components (see next subsection), it is felicitous to have a standardized way to keep track of changes, which would support automatic analysis of their compatibility — e.g. by properly used revision numbers. Attempts to introduce such a version identification system for SOFA/DCUP component model are described in [Brada–99] and [Brada–00].

2.3.8 Software soundness and its verification

The way of creating software introduced by a component based approach, i.e. starting with designing a precise specification of components themselves and also their particular architecture, brings another advantage: better verification of correctness of large software systems because this process occurs mostly on the specification level as well.

An absolutely precise description of software and its semantics is utterly essential for all serious software systems, especially mission critical ones. Unfortunately, just this aspect of software construction has been the sticking point of software engineering for a long time. This fact is given in [CicRot–99] (originally discussed in [MeyJez–97]) remarking that inadequate specification of reusable software can result in a disaster which is backed up by the case of the launch of the $500 million rocket Ariane 5 in 1996, where a code that was originally intended to convert a number less than 2^{16} from a 64-bit floating point to a 16-bit unsigned integer, was applied to a greater number, causing the software (and the rocket) to crash.

There are quite a lot of approaches to the problem of bringing better semantics description. As stressed in [PlaVis–02], even the very practical UML offers a semi-formal way how to describe software semantics and communication nature of objects: UML collaboration, interaction and state diagrams. Another approach termed *design by contract* originally introduced in [Meyer–87] and used in Eiffel, consists mostly of *assertions*, constraints of individual features, etc. Now, it is quite quickly evolving under the name of *Behavioral specification.*

Another approach that is more interesting for us because it is used in the SOFA/DCUP component model, is using *behavior protocols.* The idea of behavior protocols is not new in the field of communicating systems (they are mostly based on Hoare's CSP or Milner's CCS), however, applying them in the field of component based development changes their role and they prove to be very important in the task of keeping component-based systems consistency. For the examples of such languages, check page 11. Every component has its protocol and there must be a way how to decide, comparing those protocols, whether such components are compatible. Unlike most such protocol languages,

[13]DCUP means Dynamic Component Updating

SOFA/DCUP uses behavioral protocols based on regular-like expressions and defines a special type of protocols for every abstraction of that component model and rules for the conformance of respective protocols. We will present this protocol in section 4.3 and discuss it throughout the study because we have to keep track of behavior protocols behavior in inheritance mechanisms that are to be proposed.

2.3.9 Levels of component specifications

When considering the practical aspects of component usage, some problems with the component specification appear. Except for a dispute if binary components should be created from the precise component specification only or if it is appropriate to allow adding of these specification to the existing software pieces ex-post to maximize the code reuse and accelerate bringing the component way of programming into the real life, there is a problem that is pointed out and further discussed in [Mencl–01].

In brief, it claims that specifications can be used either as means of the application design (termed *design specifications*) or as descriptions of ready-to-use software components for the purposes of using the components (termed *use-specificition*). That's because a specification deep enough to decide reliability would be costly in terms of human resources and, besides, vendors might not often be interested in releasing detailed specification, and thus opening their products completely. The thesis concludes that optimal component model should support multiple levels of depth of component specifications.

This problem is presented here as another example of problems to be solved before component-based development can be fully accepted. However, for now, we will not consider this complication in this thesis anymore.

2.3.10 Résumé of the component notion

Although not revolutionary, components represent a promising concept that appears to heal most sores that are placed before contemporary software engineering in forms of increasing demands on complexity and reliability of software. A lot of positive properties that are inherent to components (provide higher level of abstraction, are flexible when updating and configuring, allow reuse on the binary level, allow precise specification and verification of such a specification), speak clearly for their bright future.

However, readers should be aware that software component technologies are still emerging. It implies that this field is not standardized enough yet and that there are quite a lot of competing concepts, component models and component-related products; however, except CORBA3, EJB and COM/DCOM/COM+, most are only academic. Likewise some basic conceptual questions have been still discussed; this includes ways of trading components, even better semantic description of components and architectures, specification levels. This work is affected by this fact as-well.

Anyway, only the best ones (or, unfortunately, those who will be supported by most powerful corporations) will be allowed to survive. This thesis should help the SOFA/DCUP component model (introduced in Chapter 4) to be among the survivors.

Chapter 3

Inheritance in OOPLs — an overview

If I have seen a little farther
than others, it is because I have
stood on the shoulders of giants.
— Isaac Newton

3.1 Motivation for writing this overview

We argue for the fact that the idea of writing an consistent overview together with some reasoning concerning the issues of inheritance in object oriented programming languages is a very important and beneficial deed for further progress in this thesis because the proposals for the inheritance in SOFA should be based on a good knowledge of the inheritance mechanisms used in current languages. Moreover, compared to the number of papers from other areas of object oriented programming, the inheritance has been quite marginalized and there are known only inheritance concepts used in commercially successful languages like Object Pascal, C++ or Java among the wide programmers community.

Therefore, instead of cutting this notion to pieces and introducing individual pieces when deliberating individual inheritance mechanisms for SOFA abstractions, we have decided to devote this chapter to bringing a consistent overview of inheritance. Moreover, we try to compose this chapter in such a way that it should be useful separately from the rest of the thesis.

We will use three main resources for writing this overview: [Sakkin–89], [Taival–96] and two large chapters devoted to inheritance in [Meyer–97]. Since this overview will be far from being as exhausting as those resources are and since we will omit some basic (often language-specific) issues (such as a specific terminology, etc.), we recommend to readers seriously interested in this topic to get that articles and read them right through.

3.2 Why to use inheritance

3.2.1 What is inheritance anyway

Inheritance has been one of the pillars of object oriented programming since its very beginning, albeit there is a general acceptance that several other properties of object oriented programming are regarded as more important. Single inheritance is well defined already in the first language with object oriented features — *Simula 67* — although it is used under the name *concatenation* or *prefixing*. A lot of particular OOPLs use their own synonyms for inheritance (except *prefixing* also *derivation*, *subclassing* or *subtyping*, etc.), however, in fact, such names often represent only special forms of inheritance which are implemented in the particular languages (for example, we cannot say that subclassing is equal to inheritance).

The term *inheritance* is an apposite word because its meaning in object oriented languages is very close to its intuitive meaning, albeit, as M. Sakkinen remarks in [Sakkin–89], there seem to be no common definition of inheritance: even in the ordinary meaning there are many kinds of inheritance, at least biological, juridical and cultural with each of them having completely different rules.

However, we try to come from a general definition of the word *inherit*:
To inherit is to receive properties or characteristics of another, normally as a result of some special relationship between the giver and the receiver.

From this definition we can clearly deduce several simple things that are important to realize when both, using inheritance in programming languages and trying to find a proper inheritance mechanism:

- the inheritance is a tool for an incremental creation of new entities based upon old ones
- we have to find characteristic properties that will be the subjects of the inheritance
- the giver and the receiver should not come from much different areas because they share some characteristics
- the inheritance is about finding a proper relation between givers and receivers

Another important matter to realize for our further deliberations is that, in general, inheritance is not an independent language feature, but it usually operates with tight interaction with other language mechanisms.

To summarize this topic, let's present a delimitation of inheritance in programming languages that can be found in [Taival–96]:

Inheritance is a facility for differential or incremental programming because it allows newly created objects to be based upon existing ones — only those properties that differ from the properties of the original object need to be declared explicitly, while the others are automatically extracted from the existing ones.

3.2.2 Troubles of inheritance usage

It is not trouble-free to use inheritance even in the OOP, despite it has been used in OOP for quite a long time and a lot of experience has been gained.

The reasons follow not only from the great amount of particular inheritance mechanisms but also from the fact that it is used for creating new entities in many different situations for many particular purposes; unfortunately, often in a wrong way.

For example, as stressed in [Sakkin–89], there is too much use of inheritance in object-oriented programming because programmers sometimes apply inheritance when plain aggregation would be more suitable.

We will devote next sections to classifications of cases of inheritance usage, inheritance forms and inheritance mechanisms to better understand the inheritance notion in both its usage and its application in the area of component systems.

3.3 Taxonomies of inheritance

3.3.1 Inheritance usage dichotomy

There can be recognized a basic dichotomy in usage of inheritance. Various researchers label this dichotomy in various ways. For example, M. Sakkinen terms it, following the Aristotlean tradition of dividing things into essential and accidental, the *essential (use of) inheritance* and the *accidental (use of) inheritance*, other researchers name it according to purposes it is used for as *inheritance for conceptual modeling* and *inheritance for convenience*. Generally, the former can be viewed as a *specification inheritance* and the latter as a *implementation inheritance*

However, the terminologies are not fully equivalent, so let us discuss their relation: Inheritance of implementation only is always incidental. Inheritance of specification is essential, whether implementation is inherited also or not. Let's present an interpretation proposed by Ian Holland: Incidental inheritance seems to appear as a result of software engineering and program design. Essential inheritance occurs as a result of domain analysis and system design.

Thus, essential inheritance is more important for the software maintainability and its design at all, but the accidental inheritance is used for the convenience of software programmers.

Not considering these two conceptions can result in many misunderstandings. As an example, let's present the case of ellipse-circle inheritance relation. As claimed in [Meyer–97], when introducing the concept of *restriction inheritance*, which is a form of conceptual specialization[1], to follow the rules of restriction inheritance, the circle should inherit from the ellipse because circles have properties of ellipses (i.e. a circle *is a* special type of an ellipse) but circles have the extra property that both focuses of ellipses are at the same point in circles. However, often we can find a usage of inheritance done "for convenience", in case of which the inheritance is quite opposite: ellipses inherit from circles by adding the second focus as a new data field.

[1] see next sections for the discussion about conceptual specialization

Generally, essential and accidental inheritance can be done using the inheritance mechanism which a particular language possesses. As an curiosity, the difference between the views of inheriting specification (behavior) and implementation was mentioned several times at ECOOP'88, often saying the former to be typical European and the latter typically American, which, in our opinion, corresponds to the widely believed American preference of pragmatism.

However, as for specification and implementation inheritances as such, some limitations follow from the nature of particular languages. For example, specification languages (among which IDLs including CDL belong), from their nature, cannot take advantages of some widely used implementation inheritance concepts, e.g. subclassing and related polymorphism (this issue is also presented in subsection 5.2.4 when discussing subsumption). Since the subclassing is quite widely used, inability to consider it can reduce the importance of proposed inheritance.

3.3.2 Conceptual modeling and inheritance

As we have already said, conceptual modeling belongs to the essential purposes of the inheritance concept and it is often done during the phase of designing software specifications. We can use it for modeling the correspondence relationships between the program and the problem domain (for which purpose the original Simula 67 was developed). Therefore, the inheritance was initially introduced to represent certain kinds of modeling relationships.

Let's present a more precise definition used in [Taival–96]: conceptual modeling is defined as a process of organizing our knowledge of an application domain into hierarchical rankings or orderings of abstractions, in order to obtain a better understanding of the phenomena in concern.

Various object oriented languages provide various abstraction principles for conceptual modeling. Let's itemize some of important abstraction principles provided by most languages as described in [Taival–96]:

Classification/instantiation — grouping like things together into *classes* or *categories* over which uniform conditions hold. Classes should share at least one such characteristic that others do not have.
Example: individual concrete cars (instances) can be grouped to a class car that has properties characteristic for all cars.

Aggregation/decomposition — treating collection of concepts as single higher-level concepts, so called *aggregates*. Aggregation means creating *part-whole hierarchies*. Practically it is accomplished by using objects as variables in other objects.
Example: a car (a whole) consists of a chassis, a body, an engine, etc. (parts); the body (a whole in this case) of doors and a hood, etc.

Generalization/specialization — a concept C_s can be regarded as a specialization of concept C, if all phenomena belonging to the extension of the specialized concept C_s also belong to the extension of C; i.e. C a C_s are otherwise similar, but C_s may also possess some additional, more specific properties.
Generalization, on the other hand, captures the commonalities but suppresses some of the detailed differences.

The specialization — allowing new concepts to be derived from less specific classes — has traditionally been considered as a different view of inheritance; however, it has been recently observed that relationship between inheritance and specialization may be confusing.
Example: a sedan is a specific car, the car is a specific vehicle, the vehicle is a specific product, etc.

Grouping/individualization — also known as *association*, *partitioning* or *cover aggregation*. It is used for creating possibly non-homogenous collections of things related by their *extensional* rather than intensional properties. Practically it is enabled by allowing definition of collection classes such as lists, sets, bags and dictionaries.
Example: my car and Bob's car are white therefore are grouped to White-Cars, etc.

3.3.3 Conceptual specialization and subtyping

The generalization/specification is the abstraction principle which is relevant for the inheritance. It expresses the well-known *is a* relationship (compare to the *has a* relationship which expresses the nature of the aggregation principle).

There are various inheritance forms suitable for usage in the case we want to express the conceptual specialization. The most important representative of mechanisms which can be used to express the conceptual specialization on the specification level is *subtyping* (some people even consider subtyping as a synonym for the specification inheritance). In fact, in most commercial OOPLs, the inheritance is basically restricted to satisfy the requirements of subtyping.

However, subtyping is kind of a broader term. Apart from the object oriented understanding of subtyping, there can be recognized e.g. subset subtyping or isomorphic copy subtyping, etc. The object oriented subtyping possesses the *is a* property which can be also viewed as a substitutability relationship.

The substitutability principle says that — let's cite from [Sakkin–89] — an instance of subtypes can always be used in contexts in which an instance of a supertype is expected and, for all operations of the supertype, corresponding arguments yield corresponding results. A typical example can be a class Student which is (an object oriented) subtype of a person.

When this *is a* relationship is applied together with the late binding and the self reference to the implementation (e.g. using subclassing), it is the base for another fundamental OO property — the polymorphism.

Anyway, suptyping will play a key role in our further reasoning and proposals, therefore we will deliberate some additional issues concerning subtyping in next chapters (for example in subsection 5.2.4).

3.3.4 Strict vs. nonstrict inheritance

A classification of inheritance can be also done from another viewpoint: a viewpoint which would reflect the degree of conceptual correspondence between ancestors and descendants. This taxonomy is based on four *compatibilty rules* for the relation of classes and subclasses in OOPLs introduced by Wegner in 1990.

These rules are as follows:

1. *Cancellation* — the weakest, allows operation of the class to be redefined and even cancelled in a subclass.
2. *Name compatibility* — operations may be redefined but the set of names has to be preserved.
3. *Signature compatibility* — full syntactic compatibility between classes and their subclasses.
4. *Behavioral compatibility* — subclasses are not allowed to change the behavior of operations radically.

The levels 1 – 3 are considered as *non-strict inheritance*. The fourth level is called a *strict inheritance* but it is difficult to reach it (see the attempts how to reach it further in this chapter) and besides it is too restrictive (in the terms of expressive power). Therefore, cases in which strict inheritance mechanisms are used, are limited.

Of course, for different abstractions it is necessary to reformulate these rules a bit. Note that it is quite hard to achieve the behavioral compatibility for the specification inheritance because there is no real implementation available (the rules above require subclassing). The only way is to describe somehow the intended behavior to the specification (e.g. using assertions, see subsection 5.2.3).

3.3.5 Other inheritance taxonomies

From the viewpoint of types of usage of the inheritance in OOPLs, a lot of other — more detailed — taxonomies can be created. Such taxonomies take into consideration various types of modifications (similarly as the compatibility levels in the previous subsection) and many other aspects. As an example, we present, without any details, a simplified version of the valid inheritance usage tree presented in [Meyer–97]. It generally follows the specification and implementation inheritance dichotomy — termed as model and software inheritance here — (and adds another one) but each is divided into several more specific cases:

- model inheritance
  - subtype inheritance
  - restriction inheritance
  - view inheritance
  - extension inheritance
- variation inheritance
  - functional variation inheritance
  - type variation inheritance
  - uneffecting inheritance
- software inheritance
  - reification inheritance
  - structure inheritance

– implementation inheritance
– constant inheritance
– machine inheritance

3.4 Selected issues from techniques of implementing inheritance into OOPLs

Since we have to propose a suitable inheritance mechanism for the SOFA CDL abstractions, it is useful to look at the existing possibilities of practical inheritance techniques in OOPLs, despite the fact that not everything is fully applicable to those abstractions.

3.4.1 Single vs. multiple inheritance

Whether to use or not to use multiple inheritance has been subject of quite strong disputes. Multiple inheritance (i.e. inheritance from more direct ancestors at the same time) can bring considerably more modification possibilities, however, it brings also some inherent problems. These problems derive from the fact that the inheritance DAG might be disrupted and some cycles in the graph might appear, which can result in name collisions.

The simplest form of such a disruption is so called *fork-join inheritance* and it is described e.g. by M. Sakkinen. It is the case when B and C are parents of D and A is parent of both B and C. Thus, D inherits attributes of A twice. It is problem for both, methods (how the message dispatch shall be done) and even more important problem for data fields because they carry a state (a value) and it is necessary to solve which one is correct.

In general, using inheritance, two types of overlapping properties can appear: vertically overlapping properties (along a unique inheritance path) which is a normal consequence of redefinition of properties in descendants. This is correct and it is solved by a common message lookup strategy. The other type is represented by horizontally overlapping properties (overlapping on the same level in the inheritance directed graph), which may cause problems in the case of multiple inheritance.

Some languages try to find ways how to avoid this problem, some languages leave it unsolved and suppose programmers will mind this possible problem and do not allow it to happen.

Among techniques used to avoid this problem belong selective inheritance, ordered inheritance, repeated inheritance together with renaming, mixin inheritance, etc. We will discuss some of them later in this chapter.

Another problem is that multiple inheritance is often misused. As an example presented in [Taival–96] is the Stack example which was used by B. Mayer in one of his books: He defines a class *Fixed_Stack* by inheriting two previously defined classes *Stack* and *Array*. However, this is questionable because it implies that the *Fixed_Stack*, except for being a specialization of a *Stack* is also a specialization of an *Array*. But some operations (e.g. indexed access) on arrays are not generally applicable to stacks.

3.4.2 Delegation vs. concatenation

In its basic form, inheritance can be viewed as a record combination, but there are two ways how things can be related in a computer memory: via a reference pointer or contiguity. Thus, inherited attributes can be physically located in different places using pointers or in the same place in case of concatenation.

The concatenation represents the original concept introduced in Simula 67, where inheritance was originally defined in terms of textual concatenation of program blocks. Concatenation represents the *creation-time sharing* which means that ancestors and descendants share things only in time being created, then they are self-contained and do not share anything. As a consequence, the inheritance DAG is flattened. This situation makes possible several actions which are difficult or even unable to achieve in the delegation concept such as the independent modification, renaming and selective inheritance.

The delegation concept, on the other hand, uses references to share attributes of ancestors with their descendants (even transitively) and the descendants physically contain only their own specific attributes. This concept is implemented in most object oriented languages and represents the life-time sharing, which means that dependency between ancestors and descendants is permanent.

It is done by a particular message lookup technique because, in this concept, messages that are not accepted by the (via self-reference) addressed object have to be *delegated* to another one. This task can be accomplished in various ways, however, practically there is only one technique used: incremental transitive traverse through the inheritance DAG. There can be two directions: the most used technique is so called *descendant-driven* because the lookup starts with the most specific class that is addressed and it progresses to its direct ancestor, etc. On the other hand, e.g. Beta uses so called *parent-driven* technique, where the lookup starts with the topmost subpattern (Beta's equivalent to the term *superclass*) and since it only supports a single inheritance, the inheritance path is unique and all properties of the specified name along the path are executed. Thus, the behavior compatibility is achieved because descendant properties have to take into consideration that all ancestors' properties of the same name will be executed before[2].

Speaking about this mechanism, we can realize that another decision has to be made when implementing the delegation based message lookup: the lookup exhaustion, i.e. when to stop the lookup. Most languages stop when the first matching property is found (so called *asymmetric* message lookup), some execute all properties along the path (so called *composing* message lookup) and some are implicitly asymmetric but give a possibility to explicitly decide to programmers (keywords *super*, *inherited*, *inner*, etc).

3.4.3 Ordered vs. unordered inheritance

Ordered inheritance is one of the ways how to solve problems of name collisions when using multiple inheritance. The inheritance DAG is somehow linearized and the first matching property is executed regardless of the other properties of the same name. Such a mechanism is used in most Lisp-based systems.

[2]note that this effect can be achieved in the descendant-driven technique using a mechanism which allows to call an ancestor in the beginning of methods, e.g. *super* in Java

3.4.4 Dynamic inheritance

Dynamic inheritance means the ability to change parents at runtime. From its nature, it is applicable only in the case of the delegation-based inheritance. Dynamic inheritance is quite a dangerous type of inheritance because improper assignment of a parent might cause a run-time error. However, it may be useful to implement logical states (see [Taival–96] for further discussion). It is used mostly in object-based languages (e.g. Self), however, some proposals appear even for the case of class-based languages, e.g. so called *predicate classes*.

3.4.5 Selective inheritance

Unlike the case of dynamic inheritance, selective inheritance is an inheritance modification used together with concatenation-based inheritance. Selective inheritance, in addition to renaming which allows the property names modification in the descendant (without any negative impact because of independence of objects), gives to the descendant an explicit possibility to decide which properties of its parents it wants to inherit. However, selective inheritance may lead to conceptual problems (incautious omissions, etc.), that's why it should be used only with additional verification mechanisms.

3.4.6 Mixin inheritance

The basic idea of the mixin inheritance is that the modification parts distinct from the parents are not directly embedded to the descendant, but separate so called *mixin classes* are created to hold the modifications.

A mixin class is syntactically similar to a normal class but its intent is different. New concrete classes are constructed by combining primary parent classes with secondary mixin classes using multiple inheritance.

Mixin classes do not have subclasses that's why they are not bound to a particular place in the inheritance hierarchy. Thus, we can add those modification encapsulated in the mixin classes to any place without both having to rewrite the same code again as in the case of single inheritance and being aware of name collisions as in the case of common multiple inheritance. However, it requires that the methods in the mixin classes are implemented as open to extensions and they are able to invoke corresponding methods in their surrounding environment, which is not always easy to achieve.

In addition to the practical aspects, mixin inheritance is important for the theoretical research of inheritance because it is capable of capturing functionality of other forms of inheritance. On the other hand, it brings some inherent problems given by the necessity to distinguish three types of classes (base, mixin and combinational), which brings confusion to some basic concepts of inheritance.

3.5 Some consequences of this chapter for the inheritance in CDL

Most of the inheritance concepts presented here were primarily intended for classes or objects. CDL, as a specification language for component systems, has

a bit different abstractions and a bit different needs. To consider various possibilities how to transform an inheritance type to suite the CDL abstractions (if at all) is a task of this thesis. The specification nature of the CDL suggests a kind of concatenation inheritance should be preferred because it has more advantages over the delegation in such a case. Although the concepts trichotomy is a considerable drawback of mixin inheritance, it should also be carefully considered in our research.

As for the usage viewpoint, we should prefer the conceptual purity which is essential in specification languages, however some use "for convenience" that would significantly reduce the creation effort should not be dismissed as well.

Another aspect to realize and that was also mentioned in [Taival–96] is quite a subtle difference between inheriting properties from an abstraction and using it as a variable via aggregation, especially in — but not limited to — the case of specification languages (which use concatenation inheritance type). Except for the indirect addressing of properties in the aggregation, the main difference is in the proper conceptual modeling understanding of things. When using selective inheritance, this difference is blurred even more.

Chapter 4

SOFA/DCUP framework — an overview

Software business is binary: you are either one or zero, alive or dead
— from the movie of Antitrust

4.1 SOFA/DCUP — basic facts

4.1.1 What is SOFA/DCUP component framework

The SOFA/DCUP (SOFtware Appliances/Dynamic Component UPdating) project has been running by the Distributed Systems Research Group at the Department of Software Engineering, Faculty of Mathematics and Physics, Charles University, Prague under the supervision of prof. Frantisek Plasil. The creators took advantage of many years of research in the field of object oriented programming and software systems and based especially on their Java and CORBA experience (see list of their publications [PublicList]), they started to design a complex software environment that should support full provider-user relation. They initially introduced this project in [PlaBaJ–98] introducing the SOFA architecture, SOFA component model and the SOFA component model extension called DCUP. The project has been running ever since and now first software tools have been already developed.

4.1.2 Major goals of the SOFA/DCUP project

The SOFA wants to provide a small set of well scaling orthogonal abstractions to model trading using software components over a network, and, at the same time, to support their instantiation into running applications where they can be subjects of updating. Thus, an application in the SOFA is composed as a set of components deployed on a network that can be dynamically downloaded and updated.

This task is quite complex because, as stressed in [PlaBaJ–98], SOFA encompasses a lot of software domains, e.g. the communications middleware, component management, component design, electronic commerce, security, etc. Thus,

a lot of issues have to be solved. The following are recognized as primary (most of the issues have been described in a lot of papers, that's why none will be presented here and readers are encouraged to consult the [PublicList]):

- dynamic component downloading
- dynamic component updating
- component description
- component versioning
- component transmission protocols
- support for component trading, licensing, accounting and billing
- security issues
- quality of service issues

Our effort in this thesis will be focused on enhancing the component description issue, however the other issues have to be taken into considerations as well.

4.2 Component descriptions

4.2.1 CDL and its motivation

A component in the SOFA is described using a CDL (Component Definition Language). The CDL is based on the CORBA IDL and Java syntax but it introduces a lot of novel features that meet demands on full-featured components. This condition is achieved especially by separating interfaces (as service definitions) from architectures (as communications), even though both may represent a view of a single component but at a different level. This brings better support for versioning and for better research of individual goals as defined in the previous subsection. The CDL was initially introduced in [Mencl–98] and even this part of the SOFA project has been still under quite an intensive research. So far, the integration of behavior protocols into CDL can be considered as the most significant result of that continuing research.

4.2.2 The CDL basics

In analogy of the classical concept of object as an instance of a class in object oriented programming, a *software component* (a component for short) has been introduced as an instance of a *component template* (a template for short). Such a template T is a pair consisting of a *template frame* (a frame for short) and a *template architecture* (an architecture for short). For more detailed discussion about these abstractions, see next subsections. Briefly, we can say that frames serve for the specification of services a component provides to an environment and also services the component requires from the environment to be able to provide the promised services. Architectures serve for the specification of communication between a component and its immediate subcomponents. Such subcomponents consist of instances of other frames. Thus we can see two elementary facts:

1. An unlimited number of architectures can be based on a single frame (a particular architecture is determined by what subcomponents are instantiated and the way they communicate).

2. A component can be viewed in this conception as a hierarchy of nested frames and architectures, in particular as a subtree of a component tree[1] (with unlimited number of branches) consisting of a sequence of nodes frame→architecture and for each of the architecture's subcomponents a frame→architecture, etc; leaves are represented by primitive architectures (architectures which have no subcomponents and are implemented by implementation objects); on the implementation level, an application (a highest level component in a particular subtree) can be viewed as a series of expressly communicating implementation objects[2].

Moreover, we have to realize that we can have a lot of components providing the same functionality but requiring different services (which are determined by its architecture's subcomponents requirements) that's why frames, as the least approximations of components, can be viewed as supportive abstractions (uniquely determined by an architecture) and have little value per se. This is important for deliberating the frame inheritance mechanisms.

4.2.3 CDL interfaces

We will speak about interfaces in depth in Chapter 6.1. The reason is that unlike other notions explained here, interfaces are not SOFA/DCUP specific. Briefly, interfaces are the lowest and most fundamental abstraction of the three main abstractions in CDL. Their syntax originates in the CORBA IDL. Interfaces serve for definition of individual services. Each of such services is characterized by a set of operations (called *methods*) and possibly exceptions, etc.

CDL interfaces are enhanced with interface behavior protocols which specify the acceptable order of method invocations for a given interface. Such a protocol represents a behavior of the component on a single interface only. More about behavior protocols can be found in section 4.3.

4.2.4 CDL template frames

The notion of *frame* represents a higher-level abstraction and it is introduced in the CDL to denote the contract nature of components. Let's clarify it in the following paragraph.

Components are units of independent deployment (cf. section 2.3) and, as such, they are deployed in various environments. A component is supposed to *provide* well-defined services (typically in the form of interfaces). However, such services typically need other services to accomplish their tasks. Since it would contradict the idea of the well-structured separation of functionalities into small deployable pieces if the components were self-contained (i.e. if every component contained all functionalities), and it would be unrealistic to expect the given environment to have all possible functionalities implicitly available, it is necessary to explicitly say, what services are *required* from the environment to supply possible subcomponents' needs.

In our view, frames are black-box views of components. A frame specifies, what services a component makes available (to the environment) provided that

[1] we will consider it intuitively only; it will not be specified formally nor any of its properties will be discussed

[2] objects in this case generally do not have to strictly correspond to objects in the sense of OOP

services it requires are available (from the environment). At this level, nothing more is known about how these services interoperate (therefore the term black-box view is used).

From the technical viewpoint, CDL frames consist of *interface instances* which have an interface as a type (cf. subsection 5.2.1 for the discussion about types). Interface instance is a name that is used to denote a concrete interface in behavior protocols and in architecture ties. The reason for introduction of interface instances is that a frame can generally have more instances of the same interface (however, connectors should solve this problem [PlaBal–01]).

As for the frame syntax, it is quite simple and is shown in the CDL example several sections bellow. However, we will itemize several rules that may come into one's mind while turning the topic over.

- for the reasons of universality, frames do not have to contain any required interface nor any provided interfaces; however, this does not make much sense and a usable frame should contain at least one interface in the *provides* role
- more interfaces of an identical interface type can be in the same role in a frame; however, introduction of *connector frames* makes this possibility obsolete and it should be forbidden for the reasons of clarity.
- it is not allowed to have more interface instances of the same name in a frame, even if they are placed in different roles
- there are no strict rules how to generate names of frames or interfaces except for some rules of thumb, however, we realize that such rules may be quite important if the rapidly growing number of interfaces and frames should be manageable

4.2.5 CDL template architectures

We have already defined roles of component software architectures in general in subsection 2.3.4 on page 12. Now, we describe basics of the particular component architecture employed by the SOFA/DCUP component model.

Frames provide us with a list of interfaces a component provides and a list of interfaces a component requires. However, on condition we have a hierarchical component model as presented earlier, we have to describe how the provided interfaces will actually be accomplished and who will actually be using the required interfaces. It is enough to describe it on the first level of nesting only, because each subcomponent can be viewed as another component, and thus we can choose an architecture for each of the subcomponents independently. Therefore, architectures in the SOFA/DCUP are sometimes called *gray-box views* of components. However, the nesting cannot be done ad infinitum, that's why a special type of architectures that contain no subcomponents must exist. That's why SOFA CDL distinguishes two forms of architectures:

1. A primitive architecture — has no internal structure and is supposed be implemented by an implementation object in a supported implementation language. Its description is empty.
2. A compound architecture — consists of subcomponents and description of communication among them.

A subcomponent has the form of an instantiated frame. It is sufficient for description of the highest-level communication because frames unambiguously determine the services required and provided by a subcomponent but their architectures can be arbitrary and they are not essential for the highest-level communication description. The communication is described using the notion of *ties*. There are three types of the ties in the compound architectures:

Bind — binds a required interface of one subcomponent to a provided interface of another. This is a tie of type *requires—provides* and represents an internal cooperation within a component.

Subsume — subsumes an interface required by a subcomponent to requirements of the component itself. This is a tie of type *requires—requires* and represents utilization of interfaces required from the environment by the component on behalf of its subcomponents.

Delegate — delegates the accomplishment of an interface that is granted to be provided by the component to a subcomponent. This is a tie of type *provides—provides* and represents a way how a component fulfills tasks it claims to provide.

Moreover, there we can present three additional rules for the architectures concerning ties:

1. Each interface has to be bound to at most one other interface but we can use *connectors* to avoid this limitation, therefore, on the abstract level, we can assume more interface instances can be bound to a single interface instance in an architecture.
2. Each interface has to be described in the architecture even though it is not a part of any tie.
3. On some conditions, both sides of a tie do not have to be instances of the same interface type.

The second rule coerces introduction of another binding type *exempt*, semantics of which is obvious: it exempts thus described interface from any ties (in that architecture). The first mechanism behind the first rule is introduced in [PlaBal–01] and the third rule will be discussed later.

The concrete semantics and typical way architectures are created are presented in the CDL example shown in section 4.6.

4.3 Behavior protocols in SOFA/DCUP

We have already mentioned the problem of component behavior verification and the role of protocols in this task in subsection 2.3.8 on page 14. Now, we will informally present the essential facts about protocol utilization in SOFA/DCUP framework.

Behavior protocols for SOFA CDL were originally introduced in [Visnov–99], they were further elaborated in [PlaViB–99] and the latest revision reflecting experience acquired from their practical use is described in [PlaVis–02]. In this work, we will consider the latest revision. For a thorough discussion of this topic and formal introduction of notions mentioned here, see [PlaVis–02], which is the main source of information for this section.

4.3.1 Basic concepts

The behavior protocols were introduced to bound possible behavior of the three main abstractions in CDL, which is essential to keep the complex component system in a manageable and verifiable state. Although a protocol for each of them must be a little different, it would be useful if they were based on the same basic conception. Thus, an abstraction-independent idea of behavior protocols was introduced in [Visnov–99].

The conception is based on a communication model which consists of *agents* as computational entities emitting or absorbing external communication events or performing internal ones (in general, we say that agents exhibit actions). Such an agent can communicate via bidirectional peer-to-peer connections with a finite number of other agents. A finite number of actions an agent exhibits on a set of connections is called an *activity* and all possible activities of an agent on a set of connections are called *behavior of an agent on a set of connections*. A sequence of action tokens is called *a trace*. Agents can be primitive or composed, which leads to introduction of internal and external connections and events. We also distinguish requests, responses and general events.

The set of traces representing the behavior of an agent on a set of its connections is typically an infinite language. After some reasoning how to represent it, the behavior protocols as regular-like expressions over action tokens which syntactically generate traces, were introduced.
The following operators can be used in such expressions[3]:

1. $*$ — *repetition* — the only unary operator;
2. ; — *sequencing*
3. $+$ — *alternative*
4. $|$ — *and-parallel*
5. $\|$ — *or-parallel*
6. $\sqcap_x$ — *composition*
7. $|T|$ — *adjustment*
8. $/$ — *restriction*

Such a conception has a lot of theoretical and practically useful properties. We will not discuss them because such a discussion requires to introduce the whole formal notation, which is not the task of this work. But we mention at least *behavior compliance* which specifies on what conditions one agent (or protocol) is behavior-compatible with another and may serve as a substitution of it.

4.3.2 Particular protocols in SOFA/DCUP

The general model of behavior protocols can applied to interfaces, frames and architectures, and thus we get:

Interface protocols — specify the acceptable order of method invocations on an interface. They represent the behavior of components on single interfaces only. Name of events are denoted by method names.

[3] their exact meaning is defined in the [PlaVis–02]

Frame protocols — specify the acceptable interplay of method invocations on the provided interfaces and reactions on the required interfaces of frames. Names of events are denoted by method names prefixed by respective interface instance name and specifications, whether the event is a request or a response (which is determined from the role of the interface in a particular frame).

Architecture protocols — describe the interplay of method invocations on interfaces of the frame and the outmost interfaces of subcomponents in the architecture of a component template. These protocols are not specified directly but are inferred from specification of the architecture and generated automatically by a CDL compiler by appropriately combining frame protocols of ties participants using the composition operator.

It is obvious that there must be a relationship between interface protocols, frame protocols and architecture protocols. The way interfaces are utilized in frames must correspond to intentions interfaces were created with and also the architecture protocol of a component should follow the intentions described in the frame protocol of that component. The above mentioned notion of compliance proved to be a good basis for describing this relationship, thus the notion of *protocol conformance* was established using the notion of compliance. We will only informally describe the intentions behind three types of the protocol conformance, precise definitions are presented in [PlaVis–02]:

Interface-interface protocol conformance — an interface protocol conforms to another interface protocol if the other interface is compliant with it on an appropriate alphabet.

Interface-frame protocols conformance — whatever the frame protocol allows to do on a provides-interface, it must do it in such a way that a component can exhibit at least the events as specified by the interface protocol. For requires-interfaces it is vice versa: whatever the requires-interface allows to do, it must do it in such a way that a component may exhibit at least the events as specified by the frame protocol on the requires-interfaces.

Frame-architecture protocols conformance — an architecture protocol cannot generate traces not allowed by the frame protocol, under assumption that the provides interfaces from the frame are used in the architecture in a way the frame protocols allows to. At the same time, the architecture protocol can be "less demanding" on the requieres-interfaces.

4.4 Dynamic Component Updating

DCUP should allow safe updating of SOFA components at run-time. It extends the SOFA component model in the following way:

- it introduces specific implementation objects
- it makes the way components are interconnected more specific
- it presents techniques for the updating of a component inside a running application

- it specifies the necessary interaction between a running application and the Run part of a SOFAnode (see the next section)

A DCUP component is divided into a *permanent part* and a *replaceable part* with respect to an update operation and into *control part* and a *functional part* with respect to the nature of operation. The control part is uniform across all DCUP components and it is used for managing purposes only, the functional part correspond to the classic component specifications of SOFA.

Since DCUP elaborates especially implementation issues and is implementation environment specific, it is not essential for this work and thus we will not describe DCUP anymore (with exception of some possible allusions within other parts of this work) and we encourage readers interested in this topic to read [PlaBaJ–98] (from which we took information for this section) and [Hnetyn–00].

4.5 Implementation and deployment of the SOFA components

Now, we will very briefly present basic conception of the SOFA implementation. This section is based on [MenHne–01] where you can find quite a detailed elaboration of this topic.

As mentioned earlier, the development process of a component has two parts: creating a software specification and implementing it later which can be done by independent vendors. The whole component lifecycle has several stages

1. Designing the component's architecture.
2. Recursive binding of all nested component frames to a concrete architecture — selection of concrete subcomponents is done using a component assembly infrastructure.
3. Dividing application into distribution units which results into a deployment form.
4. Filling this form with the exact location for execution, which results into a deployment descriptor.

A single deployment environment in SOFA (an individual computer) is called a *SOFAnode*. A set of interconnected SOFAnodes forms the SOFAnet. A SOFAnode can consist of several parts. Not all parts have to be present in a single SOFAnode. This is determined by the role of the SOFAnode in the SOFAnet. Let's describe the individual parts:

Template repository (TR) — the only obligatory part of a SOFA node. It contains component implementations together with their descriptions.

Run-part — provides environment for running the component instances. At least this part is supposed to be distributed across multiple hosts.

Made-part — is used to create new components and insert them into the TR.

Out-part — is used for a transfer of components from the SOFAnode.

In-part — is used for a transfer of components to the SOFAnode.

When the application (i.e. top-level component) is launched, component instances are created in so called *deployment docks* (in the Run-part of the SOFAnodes) according to the application's deployment descriptor. Components can be either already installed at the given deployment dock, or their implementation can be downloaded from the TR and connectors (stub and skeleton) for interconnecting the components via a middleware can be generated on-the-fly.

4.6 A CDL example

Now, at the end of the chapter and as a starting point for our further work, we create a sample component specification written in the SOFA CDL. It will be referred to in further text. The example should represent a fragment of a simplified banking application and is projected in a manner standard software analysis does.

4.6.1 Definition of services

First, we have to realize what business activities make a bank to be a bank and what supportive activities are needed for them. This is important in any analysis, however, in the case of SOFA component model, they should be designed with a special care because they represent cornerstones for both, the component specification and its implementation.

The activities are generally labeled as services and each service consists of several related operations. Services constitute the core functionality of our application. As we already know, they are represented by interfaces in SOFA. We also present interface behavior protocols associated with each of the interfaces.

We will design ten services (remember, it is a fragment of an application only). As for their names, we will try to use self-explanatory names and utilize some kind of uniform naming conventions.

```
typedef float currency;

interface ITAccountBasicAdministration
{ long CreateAccount(in string accountType, in string customerID);
  void DeleteAccount(in long account);
  void AssociateCreditCard(in long account, in long creditCard);
  void SetAccountReportPeriod(in int period)

 protocol: (CreateAccount + DeleteAccount + AssociateCreditCard +
   SetAccountReportPeriod)*
}
```

```
interface ITAccountAdvancedAdministration
{ long ChangeOverdraftLimit(in long account, in currency newLimit);
  void ChangeOwner(in long account, in string newCustomerID);
  void ChangeCreditCard(in long account, in long creditCard);
  void FreezeAccount(in long account);
  void UnFreezeAccount(in long account);

protocol: (ChangeOwerdraftLimit + ChangeOwner + ChangeCreditCard +
  FreezeAccount + UnFreezeAccount)*
}
```

```
interface ITAccountManipulation
{ void Deposit(in long account, in currency amount);
  void Withdraw(in long account, in currency amount);
  void Transfer(in long sourceAccount, in long destinationAccount,
                in currency amonount);

protocol: (Deposit + Withdraw + Transfer)*
}
```

```
interface ITAccountInformation
{ string   GetCustomerID(in long account);
  string   GetAccountType(in long account);
  currency GetBalance(in long account);
  currency GetOverdraftLimit(in long account);
  long     GetCreditCard(in long account);
  int      GetCurrentState(in long account);

protocol:  (GetCustomerID || GetAccountType || GetBalance ||
GetOverdraftLimit || GetCreditCard || GetCurrentState)*;
}
```

```
interface ITCreditCardAdministration
{ long CreateCreditCard(in string creditCardType);
  long SetDispositionalRightsToAccount(in long creditCard, in long account);
  long RemoveDispositionalRightsToAccount(in long creditCard, in long account);
  void InvalidateCreditCard(in long creditCard);
  void RevalidateCreditCard(in long creditCard);
  void SetDailyWithdrawalLimit(in long creditCard, in currency Limit);
  void SetWeeklyWithdrawalLimit(in long creditCard, in currency Limit);
  void SetPerSessionWithdrawalLimit(in long creditCard, in currency Limit);

protocol: (CreateCreditCard + InvalidateCreditCard +
   SetDispositionalRightsToAccount + RemoveDispositionalRightsToAccount +
   SetDailyWithdrawalLimit +
   SetWeeklyWithdrawalLimit + SetPerSessionWithdrawalLimit)*
}
```

```
interface ITCreditCardInformation
{ string  GetAccount(in long creditCard);
  string  GetCreditCardType(in long creditCard);
  void    GetValidity(in long creditCard, out date validSince out date validUntil);
  bool    IsValidNow(in long creditCard);
  void    GetLimits(in long account, out currency dailyLimit,
                    out currency monthlyLimit, out currency perSessionLimit);

protocol: (GetAccount || GetCreditCardType || GetValidity ||
   IsValidNow || GetLimits)*

}
```

```
interface ITDataManipulation
{ void  Insert(in any key, in any data);
  void  Delete(in any key);
  void  Query(in string query, out any data);

protocol: (Insert + Delete + Query)*
}
```

```
interface ITRestrictedDatabaManipulation
{  void  Query(in string query, out any data);

protocol: Query*
}
```

```
interface ITDatabaseAccess
{ void  Open();
  int   GetTransactionModel();
  void  SetTransactionModel(int model);
  void  Close();

protocol: (Open ; GetTransactionModel* ; Close + SetTransactionModel)*
}
```

```
interface ITLogging
{ void  LogEvent(in string event);
  void  ClearLog();

protocol: (LogEvent; LogEvent*; ClearLog)*
}
```

4.6.2 Distribution of services — raw component design

Now, we have to constitute the black-box view of components — frames. This comprises finding subjects that provide one or more services (i.e. interfaces) defined in previous subsection and possibly require some services to be able to do that. We should preferably consider existing functional subjects in a typical bank but it is not essential in this case because we have to realize that a bank is not the bank for it has a teller etc. but simply for it provides banking services. Therefore the internal structure of the bank can vary. Instead, we should design a component with the suitable component weight requirement on mind and with respect to properties of the environment (hardware utilization, network utilization, etc.)

To spare some space, we will write only a few frames and behavioral protocols even for fewer of them, even though theoretically lots of frames can be created as various combinations of interfaces defined in the previous subsection and even though a frame protocol should be part of all of them. Notice that database services are specified as requirements of the bank application (more precisely, outside the SimplifiedBankFragment component), thus they must be supplied from the environment (which makes sense because, in most cases, databases are not bank-specific).

```
frame SimplifiedBankFragment
{ provides:
    ITAccountBasicAdministration      piiABA;
    ITAccountAdvancedAdministration   piiAAA;
    ITAccountManipulation             piiAM;
    ITCreditCardAdministration        piiCCA;
  requires:
    ITDatabaseAccess                  riiDA;
    ITDataManipulation                riiDM;
    ITRestrictedDataManipulation      riiRDM;
}
```

```
frame Teller
{ provides:
     ITAccountBasicAdministration piiABA;
     ITAccountManipulation        piiAM;
     ITCreditCardAdministration   piiCCA;
   requires:
     ITAccountInformation         riiAI;
     ITCreditCardInformation      riiCCI;
     ITDataManipulation           riiDM;
}
```

```
frame Superindtendent
{ provides:
    ITAccountAdvancedAdministration   piiAAA;
  requires:
    ITAccountInformation              riiAI;
    ITCreditCardInformation           riiCCI;
    ITDataManipulation                riiDM;
    ITCreditCardAdministration        riiCCA

  protocol:
   (?piiAAA.ChangeOverdraftLimit
   {!riiAI.GetOverdraftLimit;!riiDM.Insert} +
    ?piiAAA.ChangeOwner{!riiDM.Insert} +
    ?piiAAA.ChangeCreditCard{!riiCCA.RemoveDispositionalRightToAccount;
    !riiCCA.SetDispositionalRightToAccount} +
    ?piiAAA.FreezeAccount{!riiAI.GetCreditCard; !riiAI.GetCurrentState;
!riiCCI.RemoveDispositionalRightsToAccount; !riiDM.Insert} +
    ?piiAAA.UnfreezeAccount{(!riiAI.GetCreditCard;
       !riiCCI!riiAI.GetCurrentState;
    !riiCCI.SetDispositionalRightsToAccount; !riiDM.Insert})*
}
```

```
frame InformationCenter
{ provides:
    ITAccountInformation        piiAI;
    ITCreditCardInformation     piiCCI;
  requires:
    ITDatabaseAccess             riiDA;
    ITRestrictedDataManipulation riiRDM;

protocol:
  (!riiDA.Open; !riiDA.setTransactionModel;
  (?piiAI.getCostumerID{!riiRDM.Query} ||
   ?piiAI.getAccountType{!riiRDM.Query} ||
   ?piiAI.getBalance{!riiRDM.Query} ||
   ?piiAI.getOverdraftLimit{!riiRDM.Query} ||
   ?piiAI.getCreditCard{!riiRDM.Query})*;
   !riiDA.Close)*
}
```

4.6.3 Making contracts — particular component design

Finally, we have to a create a concrete component architecture and constitute thus the gray-box view of components. This process may include possible instantiation of subcomponents, delegating component provisions to its subcomponents, subsuming subcomponents' requirements to the component's requirements, and constituting internal communication between subcomponents. The architecture can also be primitive but we will not do it in order to present the syntax of architecture notation. Besides, we will not write the architecture because this is done automatically from frame protocols. Notice that we use multiple ties (which is allowed thanks to connectors).

```
architecture SimplifiedBankFragment version v1
{ inst Teller    fiTeller;
  inst Superintendent    fiSuperintendent;
  inst InformationCenter   fiInformationCenter;

  bind fiTeller:riiAI  to fiInformationCenter:piiAI;
  bind fiTeller:riiCCI to fiInformationCenter:piiCCI;
  bind fiSuperintendent:riCCA to fiTeller:piCCA;
  bind fiSuperintendent:riiAI  to fiInformationCenter:piiAI;
  bind fiSuperintednet:riiCCI to fiInformationCenter:piiCCI;

  delegate piiABA to fiTeller:piiABA;
```

```
  delegate piiAM to fiTeller:piiAM;
  delegate piiCCA to fiTeller:piiCCA;
  delegate piiAAA to fiSuperintendent:piAAA;

  subsume fiInformationCenter:riiDA to riiDA;
  subsume fiInformationCenter:riiRDM to riiRDM;
  subsume fiTeller:riiDM to riiDM;
  subsume fiSuperintendent:riiDM to riiDM;
}
```

Chapter 5

Analysis of challenges

I do not believe my father ever was (or ever could have been) such a poet as I shall be an analyst.
— Ada Augusta, Countess of Lovelace, daughter of lord George G. Byron

5.1 The primary objectives

Let us recall the main objective of this thesis: to discuss application of inheritance to SOFA components. This includes analyzing of assets and drawbacks of using inheritance in each of the three SOFA CDL abstractions (interfaces, frames and architectures), presenting various inheritance mechanisms for the individual SOFA CDL abstractions and analyzing appropriateness and possibilities of incorporating these mechanisms to those abstractions. For each of the individual abstractions, the option that seems to be most appropriate should be highlighted and justified and initial proposals for syntax and semantics should be presented. Also, some possible problems related to inheritance in SOFA CDL should be identified and sketched.

This thesis should serve as a foundation for possible broader discussion among SOFA/DCUP project contributors whether to or not to incorporate inheritance to the SOFA CDL abstractions therefore no implementation is required for now.

In spite of this fact, this task is not a bit as straightforward as it may seem at the first glance. Most of the important issues were in depth discussed in previous chapters (so let the reader consult them if necessary), however, let us summarize the major facets, we should take into consideration if we want the solution to be beneficial.

5.1.1 Summarization of preconditions

The aspect of inheritance — since there is a widespread consensus among researchers in the field of object oriented programming that inheritance is the most intricate and controversial issue in the object oriented programming itself, its application to components is even more intricate because of a bit different conception of component-based development which makes

some common inheritance notions like subclassing or dynamic dispatch inapplicable (at least for most abstractions).

The aspect of CBD basic concepts — although component-based development (CBD) originates in the object oriented programming, key concepts differ. It is necessary to fully understand the entirely new abstractions whose introduction was enforced especially by the separation of specification and implementation (and component interoperability mechanisms) and the component composition, including the design by contract mentioned in the second chapter.

The aspect of trends in CBD — the solution to be competitive should take advantage of experience gained during the quite extensive research, the component-based development is subject of recently.

The aspect of SOFA component framework — since SOFA is a dynamically evolving component framework which is trying to gather the best of other component technologies and software architectures and enhance it with a bunch of new ideas, this fact implies that it is necessary to take possible changes in particular details into consideration, and thus the solution should be quite robust and should not depend on too much details. On the other hand, since the key concepts and abstractions have been already assessed and a lot of work has been done so far, the solution should minimize the need of changes in the existing framework enforced by the concepts suggested.

5.1.2 Main directions of the research to follow

Based on the existing SOFA component framework proposals and facts mentioned in this thesis so far, two main directions of further research can be recognized:

Inheritance — this direction tries to find suitable inheritance mechanisms for individual SOFA abstractions and incorporate them into the Component Definition Language. The decision on what inheritance mechanism is suitable depends on a lot of aspects including the philosophy of the SOFA framework and it will have to be carefully deliberated. Work in this direction of research has not been started yet and this thesis, for which inheritance is the main goal, is supposed to bring some foundations.

Substitutability — this direction tries to consider possibilities of replacing instances of some SOFA abstractions within other abstractions with another instances of that abstractions that do not have to fully correspond to the original ones. For example, binding partially corresponding interfaces into mutual ties and replacing frames (subcomponents) in architectures. This direction is partially covered by behavior protocols. We will marginally touch this direction as well, however, no essential results in this direction are expected to be achieved by this thesis.

5.2 General problems related to the solution

Before we begin to solve the particular issues, let us try to discuss some problems generally related to the issues, we will deal with in the next chapters. Most of those discussed problems are kind of knotty and remain open, but some conclusions have to be made at least for the purposes of this thesis. We will discuss these problems solely at an intuitive level (without introducing formalisms) because the problems are minor (albeit inherent) ones.

5.2.1 Problems of type definition

Here, we will turn over questions that relate to types because introduction of higher-level abstractions, such as interfaces and frames, may bring confusion to the concept of types.

The first and cardinal question is: How can be characterized the notion of a (data) type? The usual, simple answer is (e.g. [Skarva–99]): a set of values a data object can acquire with operations that can be used for handling these values. This answer perfectly holds for simple data types (integer, etc.) and, of course, also for user defined types, namely classes. In the case of user defined types like classes, the user has to specify and implement also the permitted operations. In all these cases, a variable or a constant is used to hold a value from the set of values, which can be viewed as representing a state.

Now, that we have defined the notion of a data type, another question arises: Can we consider abstractions that do not have states as types? This question is important if we want to use notions such as *subtyping* further in this work because, as mentioned in section 2.3, one of the most characteristic properties of components is that they do not have states. We will illustrate this issue using the the case of interfaces[1]. Other abstractions like frames could be treated similarly. To find the answer easily without jamming with details, we will reduce interfaces only to methods with their signatures and omit other features described in Chapter 6.

Thus, we ask if interfaces can be taken as types. We have to find their values and their operations. We argue that, on a theoretical level, maybe the best solution would be to introduce a single type termed *Interface* and consider a set of *method signatures*, a single interface consists of, as a value. Then, a set of all possible method signature combinations would form a domain. In the same way, we could suppose common set operations (union, intersection, ...) as operations upon those values. Another possibility in this approach would be to consider only method names instead of method signatures, but since interfaces can generally consist of overloaded operations, those interfaces (i.e. values) could not be distinguished, which would lead to the loss of information and some other undesirable consequences.

As the main advantage of the above presented approach, we would state that it forms a set inclusion partial order upon these values, thus the problem of interface equivalence would be solved in a quite natural way. An interface name is a pure alias for a value in this approach and possible more aliases of a single value still represent the same value. Besides, this approach generally makes possible to grasp the notion of interfaces much easier and possibly allows better

[1] those who are not too familiar with the notion of interface should consult section 6.1

subsequent theoretic work with this notion. However, let's remark that this is only an initial reasoning not considered in details because it is not supposed to be used.

That is because the practical use of interfaces is much closer to classes (i.e types) than to values and SOFA/DCUP treats interfaces as types for practical reasons. Therefore we will use an approach similar to classes and consider every single interface as a special user-defined type with method signatures as "values". Operations will be defined for each type (i.e. interface) by behavior protocols describing permitted method calls order. Instances of interface types can be informally imagined as having state *waiting* (before one of the initially permitted operations was called) and then each subsequent state is determined by the interface behavior protocol. Notice that enabling parallel execution of operations, interfaces can acquire a state in which two "values" are together. However, since this anomaly is not critical to solution of our problem, we will allow it. Ergo, the answer to the question above is: yes, we can but with some tricks.

5.2.2 Problems of type equivalence

Anyway, the approach from the end of the previous subsection brings some inherent problems that the original approach eliminated. As an example, we will take the problem of an assignment. SOFA's version of this problem is the substitution. We will show it on the example of interfaces again.

When can we replace an instance of an interface type with another (which can be imagined as assignment between variables of different types known from general imperative programming)? The answer should be looked for mainly from the viewpoint of particular needs of SOFA/DCUP component model, but now, we will ask a more specific question more generally: when are these two types (that have no values in the classical sense) equivalent? There are two most common equivalences in the theory of programming languages: an *equivalence by name* and a *structural equivalence*.

First, let's consider the equivalence by name: types are equivalent when they have the same name. But this seems too restrictive: we can have interfaces consisting of the same method signatures but renamed. This situation is, of course, undesirable and should be avoided. In fact, there are some facilities that might be able to eliminate this problem, e.g. type repositories (cf. section 4.5 for basic introduction of that SOFA/DCUP's one), however, taken generally, we will consider these particular solutions only as supportive. But this problem relates to broader semantics problems (which will be deliberated more generally in the next subsection) and may prove the equivalence by name meaningful. In this case, the intended semantics of operations come into consideration: if there are two interfaces with different name, albeit they are otherwise completely the same, they might (or, of course, might not) be intended for different purposes and their mapping onto the implementation object should reflect this fact. That is why they should not be generally considered as equal.

Another possibility is the structural equivalence. It claims that two types are equivalent iff their inner "components" are equivalent; i.e. this approach omits the interface (i.e. type) names and considers them as pure aliases (as in the approach of general interface as a type) and also ignores the objection above saying that interfaces considered for different purposes should contain

different methods. The idea favoring this approach in the case of interfaces is as follows: if interfaces have the same structure, they can be mapped onto the same sets of implementation objects (to support the binary reuse in component-based development, there should not be any tight relations between a particular interface specification and a particular implementation object).

The answer to the question which approach to use, is not generally decidable per se, neither of these approaches can be the most advantageous in all cases. For purposes of this work, we decided to use the structural equivalence. The main reasons are practical and mainly because SOFA's behavioral protocols are a helpful tool for further semantic decisions: if methods and ordering of their calls are the same, then interfaces are with a great probability intended for the same use. This approach also better justifies the substitutability.

5.2.3 Problems of semantics

This category of problems was mentioned in the previous subsection. The troubles are caused by purely accidental name collisions. A lot of examples are presented in the literature. The most classic one is the "push" in an interface for a stack and for a button.

We have already come across this problem while we were discussing inheritance in Chapter 3. As an example, let's present an observation by Zdonik from 1986. He says that the redefined operations in a subclass usually do not have to bear any semantic relationship to the replaced operations in the superclass; the only semantic tie is that they share the same names. However important the issue is, most object oriented languages do not solve it, albeit there are some attempts (cf. the definition of inheritance compatibility levels in subsection 3.3.4.

Separation of specification from implementation as introduced by the component approach makes these semantic issues even more important. There appeared two main approaches to this problem: behavior protocols and assertions.

Behavior protocols were mentioned in subsection 2.3.8 and the SOFA behavior protocols including examples were presented within the presentation of the SOFA/DCUP component model in Chapter 4. We will briefly remind essentials: Protocols capture semantics of communicating systems by modeling the system's communication by specifying permitted atomic communication actions and their orderings. This modeling has a discrete nature: it is based on states of the communicating system and transitions to different states which are determined by the actions. There are two main advantages of such an approach:

1. In most cases, it is possible to check if the communication conforms to the protocol, algorithmically. This means that automatic or semi-automatic tools that check this can be created. These checks can be performed either statically or dynamically (when the communication is performed) via guards. See [PlaViB–99].

2. Since protocols represent a formal calculus, some properties of communicating systems on a specification level (i.e. before they are implemented) can be proved via such protocols. Especially Milner's CCS is good for this purpose and SOFA/DCUP's behavior protocols were inspired by it to some extent too. For more, see Chapter 4, a paper about SOFA/DCUP's protocols motivation [Visnov–99] and CCS basic usage in [Brim–00].

As we can easily see, communication protocols do not solve issues that have non-communicational nature, e.g. the problem of "push" mentioned at the beginning of this subsection. The other approach — represented by assertions — cannot generally solve that problem as well but it can be of help while determining similar accidental name collisions. We briefly sketch how assertions work:

The idea of assertions originates in the Eiffel language where assertions were implemented under the name *design by contract* (see [Meyer–87] or [Meyer–97]). According to [BJPW–99], assertions were adopted by several other languages including UML (where it is known as OCL — Object Constraint Language) and Java (known as iContract). The mechanism is based on boolean assertions: in case of interfaces, preconditions and postconditions for each method as well as invariants for the whole interface can be such assertions. There is a nice example of that shown in the [BJPW–99] using BankAccount interface. In that example, the assertions are represented by keywords `require` (precondition), `ensure` (postcondition) and `invariant` and boolean expressions using method arguments or methods, e.g. a part of the `deposit` method is an assertions `require amount` $>$ `0` or `invariant balance()` $\geq$ `overdraftLimit()`.

The main advantage of this mechanism is that it can contribute to more reliable behavior of component systems and to preventing some accidental name collisions. More reasonings and information can be found in the cited literature and also in [CicRot–99].

Anyway, again neither of these two approaches solves the problems completely and, as stressed in the résumé of the component technology in the second chapter, problems of semantics belong to those that have been studied very intensively recently. In our opinion, it may be promising to combine behavior protocols with assertions, however, the way how to do it and all consequences have not been examined yet.

As for the consequences of these semantic issues to this work: since behavior protocols are now the integral part of the CDL abstractions used in SOFA/DCUP component model, they will play a key role in our decisions of inheritance and substitutability. Assertions are not a part of the CDL abstractions, therefore they will be omitted from our reasonings, albeit their role would have probably been quite important for the decisions made if they had been part of the language. For now, we have to count with those accidental name collisions as potential problems.

5.2.4 Problems of subtypes

The rule of *subsumption* is one of the most important notions in object oriented programming because it allows polymorphic behavior of programs. This rule breaks the standard of most commonly used non-object oriented high-level programming languages that are strongly typed. The subsumption says that any instance of any class that is a descendant of another class can also be viewed and handled as an instance of that parent class/classes (and transitively their parents, etc.). Thus, we can view this subclassing as a reflexive, asymmetric and transitive relation on classes.

In Chapter 3, we mentioned that subclassing is not applicable to abstractions representing specifications because, as the name implies, this notion is bound to classes. That is why we need to find another relation that would serve as a

base for inheritance within such abstractions. There was the inheritance usage dichotomy problem stressed in Chapter 3 and it is more or less obvious that in the case of specifications, the conceptual modeling purpose should be favored (however, the practical use of inheritance should not come out of mind).

We discussed problems of types in subsection 5.2.1 and type equivalence in 5.2.2. In those subsections together with subsection 5.2.3, we have also mentioned that we can expect semantic problems (homonymy relation on the one hand and synonymy relation on the other). The subtyping issues and their relation to the conceptual modeling were introduced in subsection 3.3.3. Now, we have to find a suitable subtype relation definition.

One possible answer to the question how to define the subtype relation can be found in [WinOck–00]. This subtype relation is based on Liskov and Wing's constraint-based subtype definition. This definition is only partially suitable for our problems, therefore, we will present it only briefly and informally. The subtype relation is defined in terms of the checklist of properties that must hold between the specifications of a type and a subtype of the type. This definition takes into consideration the problems mentioned in the previous paragraph and defines:

- *abstraction function* that transforms value-spaces — hardly applicable; method signatures are considered as "values" (cf. 5.2.1)
- *renaming map* is a function that maps method names of a subtype to method names of a supertype

And generally those conditions must hold:

- subtype invariants must ensure supertype invariants — not applicable; we do not have assertions in CDL (cf. 5.2.3)
- subtype constraints must ensure supertype constraints — not applicable; we do not have assertions in CDL (cf. 5.2.3)
- subtype methods must preserve the supertype methods' behavior — applicable partially — only the Signature rule: there are two rules that must hold for the corresponding methods given by the renaming map:
  - ◇ Signature rule
    - ♣ contravariance of arguments — both corresponding methods have the same number of arguments and the subtype method ith argument's type must be a supertype of the corresponding supertype ith one's type.
    - ♣ covariance of results — either both corresponding methods return a result or neither does; if they do then the subtype method result's type is a subtype of the corresponding supertype one's type.
    - ♣ exception rule — exceptions signaled by the subtype unified method must be contained in the set of exceptions of the corresponding supertype unified method.
  - ◇ Method rule — not applicable
    - precondition rule
    - postcondition rule

When we have a look at the definition, we can see that it is quite straightforward and a good compromise between robustness and realization. As an example, we can name introduction of the renaming map function: not to use renaming map at all strongly favors the implementation ease but is kind of particular, on the other hand, to use renaming map as a general relation (unifying one or more methods of a subtype with one or more methods of a supertype) would complicate and change radically both, the nature of the subtype definition and the finding of an instance of such a renaming relation.

As for the usefulness to our work, we may take the Signature rule into consideration when reasoning about substitutability. The rest of the definition is not applicable for now because assertions are not part of the abstraction on which the SOFA/DCUP component is built, but it is outlined for possible use in the future. The usefulness of the Signature rule for inheritance may be discussed in chapters devoted to the particular SOFA CDL abstractions.

5.2.5 Problems of protocol canonical form

Before we finish this chapter, we will present one more problem which came into our mind when studying SOFA/DCUP component model and proposing the inheritance mechanisms.

The motivation is following: while creating component specifications, a component designer might often come across a problem that he wants to create an abstraction which has the same structure as an existing one and he has a conception of that abstraction's behavior represented by the abstraction's protocol (this goes especially for interfaces) or, generally, that the designer wants to compare syntactically different protocols if they are semantically the same (i.e. if they represent the same communication behavior).

Although such protocols are regular languages (enriched with parallelism) that may be analyzed by a finite automaton and therefore automatic tools should help, there might be situations when this task should be done quickly and by hand. Then, it is quite demanding, especially in large quantities and complex real-life applications. One of these situations is inheritance in case it is done solely for purposes of changing communication behavior, see subsection 6.2.5.

This canonical form could be used either as a primary form of protocol expression or as a secondary additional form.

However, difficulties with finding such a canonical form that could be both created and read easily enough and also with transforming such a canonical form to a vital integral part of the existing CDL, unfortunately lead to refusing this idea — at least for now — and only deliberating releasing some rules-of-thumb how to write protocols.

That is why we define it only as a problem that might be possibly solved sometimes in the future:

Goal: Let A and B be behavior protocols which generate the same language. We want to find a reasonably simple algorithm that transforms these two protocols to protocols that are denoted identically.

The technique should be probably based on eliminating unnecessary parentheses and unnecessary operators, introducing a uniform ordering, etc.

Chapter 6

Interfaces

Let's look at the record!
— Al-Smith (1928)

6.1 Interfaces: basic facts

6.1.1 Evolution

Let us recall some basic facts concerning interfaces. Originally, there were classes in class-based object oriented languages that contained data fields and methods. Classes' goal is to specify and implement particular services class instances (i.e. objects) provide. The methods encapsulated the data fields in such a way that the user of an object called methods only (or — in object oriented terminology — sent messages) and the methods used the fields for storing information, etc. Moreover, only some methods could be called by the rest of the world (i.e. were public), others were hidden from outside an object. Those public methods formed an interface to the object, via which the rest of the world communicated with the object (requesting services, etc.). This technique improved clarity significantly.

Complete separation of interfaces from implementation was the next step in the evolution of interfaces. It was necessary if we wanted to use binary software components (deployed somewhere on the same computer or on the net using various middleware technologies, e.g. CORBA or COM/DCOM), see Chapter 2.

As interfaces gained importance per se, *Interface Definition Languages* (IDLs) began to spring up. Those languages serve for writing software specifications, which are then (preferably automatically) mapped onto an implementation code (e.g. in CORBA, client stub and server skeleton will be generated from a CORBA IDL source file). Unlike pure interfaces, quite a lot of IDLs also offer to utilize attributes (data fields) and exceptions, etc. in the software specifications (however, using attributes is not recommended — cf. [MowRuh–97] — because of maximization of reusability and unavailability of exceptions for data fields.

Since specifications use a generic form of methods and generic (data) types, precise role of individual method arguments appeared to be useful and therefore their more precise description has been established. Now, we will present an anatomy of a typical interface.

6.1.2 Anatomy

As a typical interface we chose the form of interface which is used in CDL (and also in CORBA IDL). Confer Chapter 4 and [Mencl–98] for CDL specifications and [PlaVis–02] for interface protocols definition. This decision is a consequence of the fact that CDL interfaces belong to primary objects of interest in our work. But anyway, we will try to point out the common features of interfaces and newly added features in CDL.

Every interface primarily consists of operations (or — taken from the object oriented terminology — methods). A method is quite a fuzzy notion, therefore we will speak about *method signatures*. At the basic level, we will understand a method name, a possible return value type and an ordered set of individual method argument types under this term. In situations, we want to emphasize a complete method declaration, we will use the term *method header* which includes also identifiers of arguments. Method signatures represent the way methods are uniquely identified in an interface. We have already mentioned that interfaces mean a specification (not an implementation), therefore we need method headers to be precise enough and detailed enough to describe the intended use (semantics) of given methods. That's why they are enriched with additional features in most cases and that's why they differ in various programming (and definition) languages. Let us summarize basic elements of method headers:

- **method type specification** — is a return type of the respective method. For languages that also support procedures (without return types) or functions having type *void*, IDLs make possible to use a type *void*, which can be subsequently mapped onto a procedure or a function returning void
- **identifier** — name of the method — should resemble the intended semantics of the respective method
- **declaration of arguments (parameters)** — such a declaration is usually written as a list in parenthesis after the method identifier and each element of the list consists of a parameter type and a parameter identifier. The list cannot contain two parameters with the same parameter identifier (regardless of their types)

Every interface should contain at least one method because, unlike classes, interfaces without methods make no sense (even data fields — which should be avoided in interfaces as mentioned earlier — should not be declared without operations which manipulate them). Description languages such as SOFA CDL restrict types, an argument can be of, to several exactly given which are then mapped onto the actual ones the particular implementation language supports.

Apart from the three signature elements, IDLs bring some more elements, the description to be more accurate[1]. We will itemize additional elements of OMG CORBA IDL interfaces:

- **modifiers** `in`, `out` and `inout` of method arguments; the modifier `inout` (which is used to indicate an argument which is used to input a value into a computation and, simultaneously, to output a computed value from the computation) is recommended to be used sparingly. This notion was

[1] note that method modifers *public*, *private*, *protected* are not used in interfaces (unlike in classes) because all methods are inherently public

coined in Ada but it existed in the form of calls by value, name or result in procedural programming languages from their origins.

- **a method attribute** `oneway` used to denote an asynchronous call of the respective method; not part of CDL
- **exceptions** that methods can throw
- **attributes** — behave like properties, often compile to getter and/or setter methods of the same name; they can be prefixed by a `readonly` modifier; CDL contains properties, cf. [Mencl–98]
- **type definitions**
- **context** — a client can contain one or more CORBA context objects, which provide mapping from identifiers onto string values; an IDL method can specify that it must be provided with the clients mapping for particular identifiers; not part of CDL

6.1.3 Additional features of interfaces

But this description of methods is not accurate enough yet. For instance, according to [MowRuh–97], OMG recommends at least two pieces of additional specification: better specification of the semantics of methods and classes and the sequencing of operations in the IDL (i.e. communication modeling). Both are typically specified informally by using plain textual description.

As mentioned earlier, in SOFA CDL, the latter is formalized into the notion of *interface behavior protocols*. Every interface is now associated with just one interface protocol. Although interfaces will be instantiated for reasons of higher-level abstraction definitions, protocols are associated with interface types (cf. [Visnov–99] for reasons). As a consequence, whenever we want to instantiate an interface with another specification of communication capabilities, we must create a new interface with the requested protocol, even if the interface is otherwise the same as an existing one. This fact should be considered while deliberating interface inheritance.

As for the former additional issue (the better formal specification of method semantics), there are a lot of approaches to solve this problem in various languages. We discussed the frequently used approach of assertions in subsection 5.2.3 on page 43. Such an approach can influence the interface anatomy as well. This approach adds constraints concerning method argument values and invariants (relations among return values of methods and/or arguments of various methods that must be valid in every use case) to interface definitions. For a concrete example, cf. [BJPW–99].

Let's emphasize again what mentioned in subsection 5.2.3: from the additional features of interfaces, we will consider the behavior protocols only in our further deliberations because no other additional feature is an integral part of SOFA CDL. In general, we will focus most of our effort on methods and other elements will be considered only marginally.

6.2 Inheritance of interfaces in SOFA

6.2.1 Why interface inheritance is not straightforward

Adding inheritance to interfaces may seem quite simple. But since we have already in-depth discussed the notion of inheritance in Chapter 3, we know that there are dozens of ways inheritance can be implemented and dozens of reasons why we may want to do it. However, since interfaces represent the specification — unlike classes that also represent the implementation — and can be viewed as rather immutable windows through which we look at implementation objects, many inheritance models used in full-fledged object oriented programming languages are often hard to use. For instance, as we have already stressed, there is a wide acceptance of the opinion that the polymorphism, which can be achieved due to the subsumption together with the notion of *self* and late bound message dispatch, is the main asset of inheritance (although some people emphasize the necessity to put the inheritance's role in object oriented modeling above this). Anyhow, this cannot be considered in the case of such static structures as interfaces. We have to find a suitable inheritance relation, probably based on a form of subtyping, as indicated in Chapter 3. But first, we ought to reason whether inheritance is suitable for component interfaces at all.

Let's try to estimate the price/value ratio of the interface inheritance for SOFA, i.e. how extensively the interface inheritance is supposed to be used and what benefits it brings, in proportion to the price paid for the introduction of this feature in terms of the language clarity, manageability, implementation effort, usage overhead, etc. This estimation is not intended to be absolute but only as a supportive aspect in our decisions.

6.2.2 The value of interface inheritance for SOFA CDL

There is a dispute whether inheritance should be used in component-based systems. For instance, in [SzyWec–96] it is recommended to investigate whether (immutable) interface aggregation can replace subtyping. As an example of such an approach Microsoft COM is presented for it relies entirely on non-hierarchical, non-extensible, immutable interfaces. This fact is presented and espoused in [EddEdd–99]. We will mention this topic also in section 7.2 while speaking about inheritance of frames. However, now let's notice that in COM, interfaces themselves do not support any form of inheritance and they are supposed to be immutable pieces that serve for purposes of functional aggregation and composition. Only other IDL source files may be imported to a source file (from which then header files and stubs are generated).

On the other hand, OMG CORBA IDL does support the inheritance of interfaces. It takes form of multiple inheritance, where IDL interfaces can be created as descendants of other interfaces inheriting the attributes and methods of their ancestor interfaces. Interfaces support multiple inheritance but only if ancestor interfaces do not include definitions with identical names. This "undisciplined" inheritance is subject of quite a strong criticism from many researchers and users. Let us name, for example, prof. Markku Sakkinen who has been dealing with the inheritance issues for a long time, (cf. e.g. [Sakkin–89]) and who generally does not recommend to use pure multiple inheritance mechanisms.

We can see that various component models are split in opinions about the

interface inheritance value. But if we want to make an eligible decision, we have to consider additional factors. The main factor is the real intended use of SOFA component framework. The two main possible ways, proposals of concepts of which are sketched in [Mencl–01], are also briefly mentioned in subsection 2.3.9.

Although the multitude of a legacy code already written in object oriented languages which can be given a better specification and which can then be used for composing component applications, speaks for the bottom-up approach, it is generally more reasonable to tend to the top-down approach. Therefore, unlike in the case of bottom-up approach, the interface inheritance can become handy for conceptual modeling (cf. subsection 3.3.2) because while considering creation of specifications, good capabilities of conceptual modeling become very important, even though we admit that the main burden of a good application design is assigned to components themselves.

As for other benefits, although expressive power of CDL obviously remains the same (e.g. no subclassing-based polymorphism as in the case of classes), as mentioned in the previous paragraph, it can bring new expressive forms (e.g. conceptual specialization). Another advantage that is inherent to all inheritance mechanisms and that can be considered as the practical part of inheritance in the inheritance dichotomy as described in Chapter 3, is that it can save quite a lot of typing effort. This is very handy especially in cases when only few features are going to be added or redefined (e.g. most typically if we want to change the interface protocol, add a new method or change a method signature).

6.2.3 The price for addition of interface inheritance to the SOFA CDL

Any addition of any new feature to a programming language inherently brings the aggravation of its clarity and manageability. The significance of this fact grows very rapidly in cases when such a new feature affects the most fundamental cornerstones from which all higher-level concepts of the language are built. And interfaces are such a case. Therefore, we will follow, especially in the case of interfaces, the well-known KISS[2] maxim and restrict the inheritance capabilities only to justifiable cases. Practically, this means that we have, after a long deliberation, found inappropriate to allow to redefine (i.e. change the signature of) or remove an interface method, which means that we allow only such an inheritance mechanism that keeps quite a strict subtype relation between parent and child interfaces.

When we look at the usable parts of the subtype relation definition in subsection 5.2.4, we realize that even allowing modifications given solely by the signature rule can be confusing and can cause problems. For example, while mapping such a modified descendant interface onto an implementation language: if there is an ancestor interface and a corresponding class in the implementation language, the descendant interface could not generally be mapped onto a descendant of that class because many often used programming languages do not support method signature modification in descendants. And as for the renaming function, it is absolutely unsuitable in this case. Except for problems with mapping of such renamed methods onto an implementation language like in the previous case, there is also another problem: a descendant interface should be

[2]Keep It Simple, Stupid!

created with the basic semantic behavior compatibility in mind, and thus the capability to change method names would bring too much freedom and lead to disturbing this property.

This decision to use such a strict inheritance might seem to be quite drastic but we have to realize that interfaces are grouped into a *provides-requires* dichotomy in frames and they are further subjects of ties in architectures (cf. subsections 4.2.5 and 4.2.4) which can be nested to an arbitrary finite depth level. Allowing interfaces to change arbitrarily would be very messy in cases we would like to create ties between frames containing different (but similar) interfaces and if we wanted the inheritance to be of help in compatibility decisions.

Behavior protocols can be considered to be the only exception to this rule, because they can be automatically checked and the need for their change while keeping the rest intact, can be quite frequent.

Now, let us discuss the implementation effort. The SOFA/DCUP implementation has to modified. If we consider solely interfaces, the modification affects mainly the CDL compiler and the change should be realizable. However, if we consider its influence on higher concepts and the inheritance mechanisms and compatibility rules presented further in this work, the change will be quite substantial. Therefore we have to decide carefully, so that the implementation change could be a one-off effort.

And finally, as for the usage overhead, there will be almost none. SOFA/DCUP component framework should know all available interfaces and, since the interfaces are specifications only, no message lookup is necessary and the interface methods' details (the design level specification) should be stored in type interface repositories.

Discussing implementation price of inheritance, we have to think about another aspect that was mentioned several times earlier: Should the *concatenation* model or *delegation* model be used? These terms are explained in subsection 3.4.2. Even though they also affect message lookup, etc., we will reduce this issue to a question if descendants ought to be dependent on their ancestors or be self-contained. This depends highly on how the CDL is implemented and how the mapping of those specifications created in CDL onto an implementation language is done. If all interfaces were traceable from a repository and if there were a requirement for a mapping homomorphism between specifications and implementations (i.e. if the inheritance relation on interfaces homomorphically corresponded to the inheritance relation on respective implementation objects), we would recommend the delegation-based model.

6.2.4 Interface inheritance aimed at protocol replacement

We have already mentioned that interface protocols can be viewed as an integral part of interfaces in SOFA CDL, albeit they are not obligatory parts of CDL interfaces in the current implementation of the CDL compiler. But their use is highly recommended in practical applications because they help with checking the communication correctness. Therefore, we will handle protocols as obligatory parts of interfaces. At this moment, a situation occurs when a form of inheritance of interfaces becomes very handy because there often might be cases in which we want to change a communication behavior of an existing interface without changing the rest of the interface. For instance, we can imagine a situation when an interface that originally allowed to use some methods only sequentially, wants

to extend possibilities of their usage by allowing their parallel execution.

Such an inheritance aimed at a protocol replacement may be advantageous for two main reasons:

1. We need not write the whole interface again (which is a typical practical reason for using inheritance in general).

2. We will be sure that the involved interfaces are really the same except for protocols, without lengthy checking of all interface elements.

But there is a problem: if we have a requirement that a descendant must be a subtype of its ancestor and if we subsumed protocols to the interface definition then replacing one protocol with arbitrary another generally breaks that rule. The question is if we want to subsume the protocol in the definition of subtype.

As for mapping, such two interfaces could be theoretically mapped onto both, the same implementation objects (because only the rules how their methods are allowed to be called, change) or the descendant can be mapped onto a subclass (reflecting the fact that changing communication behavior may change implementation of that methods). However, the latter case should be avoided by creating a completely new interface with changed method names which express the actual semantics more accurately. That's why this aspect should not represent a problem.

But if we wanted to automatically decide substitutability in higher-level abstractions (e.g. i.e. an ancestor can be substituted by a descendant in *provides* part of a frame and vice versa) then allowing arbitrary change of protocols should be rejected.

We should only allow to define a new protocol in such a way that it has at least the same capabilities as its ancestor. For example, if an ancestor allowed a parallel execution of some methods, then its descendants should allow it as-well. If an ancestor allowed a sequential execution only, then its descendants should allow either sequential or parallel execution (because parallel execution can be done sequentially by the requirement counterpart in an architecture). We prefer this purer solution even though it decreases number of situations in which such an inheritance will be used.

Notice that according to [PlaMik–97] — in which sound enrichment rules for protocol inheritance are introduced — this form of inheritance in which existing communication of methods is modified (even by pure extending capabilities) and not only "soundly enriched" (which requires preserving ancestor "flavor" in descendants), is prohibited. However, we believe that descendants should have the same properties and capabilities as their ancestors and possibly some more, which is satisfied in the case proposed by us as well.

We will consider only a mechanism which uses a single inheritance because the intended result is to have an interface with a different protocol.

The syntax could look like this:

```
interface InterfaceName
  inherits InheritedInterfaceName redefines protocol
{
            the definition of a new protocol goes here...
}
```

The semantics of this code is straightforward: we have a new interface InterfaceName which is otherwise the same as the InheritedInterfaceName except for a different protocol which has to allow at least the same communication as the original protocol.

Of course, some minor problems may appear which should be solved quite easily. For instance, defining a new interface with the same protocol as the original one or as another existing interface, i.e we will have completely identical interfaces with different names only. Such situations are a bit confusing but otherwise harmless and could be solved by repositories.

6.2.5 Interface inheritance aimed at protocol modification

Now, we may want to take advantage of a modification (not a complete replacement) of protocols. The two advantages enumerated in the previous subsection will generally remain preserved but they a bit change their weight: an inheritance aimed at a protocol modification may (or may not) save time necessary for writing the protocol code, but it surely more explicitly shows changes made to the protocol. This can be also useful when deliberating compatibility of interfaces.

We will aim at the same thing as in the previous subsection (with preserving the same properties of the inheritance relation), however using a bit different mechanism which has some additional properties (e.g. explicitly shown incremental modification) but which also is a bit more demanding.

We will take the ideas of *enrichable protocols* introduced in [PlaMik–97] as an inspiration, however, we will omit some of their properties and allow modification of existing communication of methods in such a way that descendants have at least the same capabilities (as presented in the previous subsection).

The syntax could look like this:

```
interface InterfaceName
  inherits InheritedInterfaceName modifies protocol
{
        OriginalProtocolSubexpression1 ⇝ NewProtocolSubexpression1;
        OriginalProtocolSubexpression2 ⇝ NewProtocolSubexpression2;
                        .....
}
```

This code has the following semantics: we inherit all elements (including a protocol) from InheritedInterfaceName but we consider a modified protocol to be a valid part of the new interface. The modification is done by marking parts of the original protocol which are to be changed (here denoted as OriginalProtocolSubexpression$X$) and replacing them with new subexpressions (here donoted as NewProtocolSubexpression$X$).

The protocol elements consist of method names (upon which the unary and binary operators as presented in section 4.3 are defined) and those elements appear just once in the protocol. We can always select a subexpression that is unique in the protocol and modify it. Notice that also the whole protocol is a subexpression and it is unique for sure. That is why we can disallow the protocol replacement from the previous subsection as redundant and, for better clarity and simplicity, allow only this modification mechanism.

We should ensure that this modified protocol satisfies the subtype relation, which assumes two basic steps:

1. We must ensure that all methods of the interface will be contained in the resultant protocol.

2. We must ensure that the new protocol preserves and possibly extends the communication capabilities of interface methods

This should be secured by the CDL compiler which should have the capabilities to parse such modifications, internally creating a new protocol, and to compare it with the list of interface methods to find if there is any missing, and subsequently, by comparing the languages generated by the original (pointer to which must be kept if delegation inheritance model is involved) and the new protocols, the compiler decides the latter property. This is usable even in a general case (some new methods which are reflected in the protocol, are added), which will be discussed in the next subsection.

The fact that the protocol modification-based inheritance is usable in a general case, is important because we admit that this limited version of inheritance (for purpose of changing protocols increasing interface communication capabilities) will not be used too often.

Another consequence of this approach is that the inheritance direction matters, which may be in cases of protocol modification confusing. We will present it using an example, for which purpose we use a fragment of the CDL example created in section 4.6.

The original protocol has the following form:

protocol: `(Deposit + Withdraw + Transfer)`*

We will denote the modification in this way:

```
interface ITAccountManipulationWithWeakTransfer
  inherits ITAccountManipulation modifies protocol
{
      Transfer ⇝ Transfer ; Withdraw
}
```

The resultant protocol is then: `(Deposit + Withdraw + Transfer;Withdraw)`*. Notice three things:

1. The new protocol limits the capabilities of the original protocol (after the method `Transfer`, the method `Withdraw` must be called — in the original, an arbitrary method was allowed), that's why this use of inheritance is forbidden!

2. This change of the protocol assumes different semantics of the involved interface methods — it assumes that the method `Transfer` only checks whether the transfer of the given amount of money is allowed and implements the transfer, nevertheless it is not able to decrease the transferred money from the account and assumes the Withdraw method is called.

3. If this inheritance had been done vice-versa, it would have been (at least formally) O.K.

The second item may seem a strong argument against possibility of such a protocol modification and for allowing only its sound enrichment in cases some new methods are added (which is discussed in the next subsection). But we believe that the component designer should be given a possibility to free decide whether the semantic changes are so significant that creating a new interface from the scratch is the preferred solution or whether it is sufficient to use a protocol modification of an existing interface.

6.2.6 Interface inheritance in general

Now, we are going to reason the interface inheritance in general. It should involve the complete structure of interfaces, however, as we have already mentioned, we will focus on methods because other interface elements are used only rarely.

We have already argued for the refusal of allowing other interface modifications than those that preserve the subtype relation because this property can be useful further while using interfaces in higher-level abstractions. Moreover, we have emphasized that inheritance should be primarily used for the conceptual modeling purposes in cases of specification languages.

One of the major questions is what inheritance mechanism should be used because some of them are intrinsically excluded from the nature of interfaces. From the seemingly usable ones, especially the mixin inheritance comes to mind (cf. subsection 3.4.6). However, we argue that mixin inheritance — and the same goes for any form of multiple inheritance(!) — is not suitable for interfaces because interfaces should represent small well-defined specifications of services and their combination is inappropriate for the interface level because the interface concept in SOFA is much closer to mixins themselves (as independent pieces of functionalities) rather than combinations of functionalities represented by often pragmatic use of inheritance in classes. This combination should be accomplished in frames by instantiating (as required or provided) all interfaces that would be otherwise combined by multiple inheritance. Mixin inheritance, moreover, introduces mixins as new types of limited non-self-contained classes (interfaces in our case). Such a concept is quite nonsensical and confusing in this case.

Therefore, we propose only a classical single inheritance that preserves the subtype relation and that should be used primarily for creating conceptual specialization hierarchies of services.

Now, we should reason how the subtype relation for these CDL interfaces should be constructed. Let's discuss three basic forms we have already come across:

1. Inheritance that preserves all method signatures of all methods from an ancestor and that may add only new ones.

2. The same as in the previous case except for possible changes of method signatures in descendants in compliance with the Signature rule (cf. subsection 5.2.4).

3. The same as in the previous case except for possible renaming of methods.

As already mentioned, the third option is an outsider. But first some advantages. There are situations when such a renaming would be useful. For example,

some methods of an descendant may change semantics a little and the renaming may be used to describe this subtle change.

But there are a lot more disadvantages. The practical implementation of the renaming function is intolerably hard and would increase complexity of such inheritance. Besides, such an inheritance would be confusing. Another disadvantage is that such a renaming may be misused (improperly renamed methods that do not semantically relate but have only an accidental signature compliance). Also the mapping onto implementation objects and substitutability would be uneasy. That's why we reject this possibility.

Compared with the first option, the second option is much more acceptable: the inheritance form will be more difficult only slightly (only a few more inheritance correctness checks in the CDL compiler), possibly no additional syntactic requirements, substitutability of interface inheritance-based hierarchy in components should also be probably possible (we will discuss substitutability in the next chapter).

Mapping onto implementation objects preserving inheritance relation would generally be quite problematic (as discussed earlier). The possibility of misuse still remains but on an acceptable level.

The first option is the most conservative one and its main advantage is its simplicity. Besides, it is the least misuse-prone solution (it is closest to the notion of strict inheritance from all three options). Mapping to implementation objects should be O.K. as well, even for OOPLs that require descendants to preserve their ancestors' method signatures.

If we estimate how these three options compare from the viewpoint of usage increment, we argue that for correct usage, the increment between options one and two is quite low and between options two and three even lower (cases of incorrect use, on the contrary, increase significantly).

We summarize this discussion using a table:

	Implem. ease	*Lucidity*	*Correct usage*	*Mapping*	*Corr. uses accrual*	*Substitutability*	*Semantic accur.*
1^{st}	Excellent	Excellent	Excellent	Excellent	N/A	Good	Low
2^{nd}	Good	Good	Good	Possibly problematic	Satisfactory	Good	Low
3^{rd}	Unsatisfactory	Unsatisfactory	Satisfactory	Difficult	Low	Difficult	Excellent

From this table, we can conclude that the most conservative inheritance form is probably the best.

6.2.7 Complete interface inheritance — the final form

As for the syntax of the interface inheritance, we will follow the syntax proposed for the inheritance aimed at protocol modification earlier in this chapter and we will unify the protocol modification and addition of methods in a single inheritance mechanism.

We will disallow the protocol replacement as considered in subsection 6.2.4 and we will allow only the protocol modification as suggested in 6.2.5 because replacement can be done by a modification of the whole protocol. Notice that the `modifies protocol` keyword is obsolete because there is no other option. That is why we will not include this keyword in the final form of the proposed inheritance.

In the previous subsection, we decided to use the form of inheritance that only allows to add some methods in descendants. The same may go for attributes (which is not supposed to be used often). We could also allow the Exception rule (defined in subsection 5.2.4).

We suggest the following syntax:

```
interface InterfaceName
  inherits InheritedInterfaceName
{
        AddedMethodHeader1;
        AddedMethodHeader2;
                        .....

        OriginalProtocolSubexpression1 ⇝ NewProtocolSubexpression1
        OriginalProtocolSubexpression2 ⇝ NewProtocolSubexpression2
                        .....
}
```

The notation has the following semantics: a newly created interface InterfaceName inherits all elements (including the protocol) from the InheritedInterfaceName, in addition to that, new methods are added and the protocol is modified similarly as shown in subsection 6.2.5.

Newly added method names must be different from the inherited method names because method overloading is not allowed in SOFA CDL (likewise in CORBA IDL), because there exist implementation languages that do not support method overloading (cf. Chapter 4).

Before we finish this section, we will present a simple example. The example uses an interface taken from the large CDL example in section 4.6:

```
interface ITAccountInformation
{ string   GetCustomerID(in long account);
  string   GetAccountType(in long account);
  currency GetBalance(in long account);
  currency GetOverdraftLimit(in long account);
  long    GetCreditCard(in long account);

protocol: GetCustomerID* || GetAccountType* || GetBalance* ||
GetOverdraftLimit* || GetCreditCard*

}
```

Now, we will create a more specialized interface which can be used in the cases, when the current account state can be surveyed (which does not have to be allowed in all cases).

```
interface ITAccountExtendedInformation
   inherits ITAccountInformation
{
    int GetAccountState(in long account);

    GetCreditCard* ⇝ GetCreditCard* || GetAccountState*

}
```

Chapter 7

Frame-level problems

In this given situation, it is difficult
to decide which sorting algorithm is better.
— Prof. Donald Ervin Knuth (1973)

We introduced the CDL component template frames in Chapter 4 while introducing the SOFA/DCUP project. Now, our goal is to analyze issues concerning inheritance at the frame level. But first, we also mention some issues concerning substitutability.

7.1 Substitutability of SOFA components

This section will present the only substitutability relation considered in SOFA framework so far and elaborate it a bit. We would like to emphasize that this is an initial attempt to introduce a bit more formalized views of some of the SOFA CDL abstractions and that substantial elaborations and revisions might be done in the future.

7.1.1 Why we need substitutability

As mentioned in Chapter 4, one of the main goals of the SOFA/DCUP project is to create easy-to-compose and easy-to-upgrade components (for the component definition, see subsection 2.3.1).

Regardless of the philosophy of the SOFA framework usage, it is clear that a lot of components will not be created to be directly bindable and that new enhanced versions of existing components with extended or corrected functionality will appear. It implies that we have to deal with the problem of components of which services and possible contracts need not fully correspond but that should cooperate. Now, let's introduce the only type of substitutability considered in SOFA framework so far (for example here [Mencl–98]) and let us try to make it more precise.

7.1.2 A simple subtype relation on frames

Let there be two components which are otherwise the same except one component *provides* the same and possibly more interfaces than the other and *requires*

the same or possibly less interfaces than the other. Then, such a component can substitute the other.

Let us formalize and generalize this notion by introducing a bit more rigorous terminology. We will try to show that frames can form a CPO which is a standard notion in the domain theory (cf. [Zlatus–93]) and even that they form a lattice.

Let $\mathcal{U}_\mathcal{I}$ denote an infinite but countable set of all possible interfaces. Since we are going to deal with real life problems, let $\mathcal{I} \subseteq \mathcal{U}_\mathcal{I}$ denote a set of all actually existing interfaces (i.e. interfaces created so far). The cardinality of this set can increase with the time passing but since, at any point of time, there cannot be infinite number of actually existing interfaces, we can be sure that $\mathcal{I}$ is a finite set.

We would like to define the notion of the component frame using the set $\mathcal{I}$. But there is no straightforward way how to do it because a frame can be constituted from repeating instances of the same interfaces. Therefore, we use a method analogical to that in the previous case: we denote $\mathcal{U}_\mathcal{F}$ the universe of all possible component frames and $\mathcal{F}$ a finite set that represents its restriction to all really existing frames. Moreover, we denote $\mathcal{T}$ a set of all individual interface instance names that any existing frame consists of (prefixed with respective frame names to ensure that the names are unique) and $\mathcal{M}$ a multiset of corresponding interfaces (i.e. interface types). This ensures that we will deal with at least all existing frames in our further considerations and, at the same time, we will stay in finite domains. Now, we can particularize[1] the notion of component frame using $\mathcal{M}$:

Observation 1: Let $\mathcal{M}$ be a multiset of all interface types of interface instances in any existing component frame. A component frame can be viewed as $\xi \in 2^\mathcal{M} \times 2^\mathcal{M}$. We label the set of all possible thus defined frames $\Xi \equiv 2^\mathcal{M} \times 2^\mathcal{M}$.

Let's realize that this definition makes all frame types with the same sets of interfaces identical. This makes sense because it corresponds to the structural equivalence of types as discussed in subsection 5.2.2 and it can be done without detriment to universality and it is important if we want to consider theoretical properties of frames. Further notice that if we apply the structural equivalence to $\mathcal{F}$ and denote the resulted subset $\mathcal{F}_{\mathcal{SU}}$, then $\mathcal{F}_{\mathcal{SU}} \subseteq \Xi$, i.e. all existing frames structurally unified can be viewed in such a way.

We can also define the first and the second projections which extract the *provides* respectively *requires* interfaces of the given frame.

Definition 2: Let $\mathcal{M}$ be a multiset of all interface types of interface instances in any existing component frame and ξ a frame. We define a function

$$\psi : 2^\mathcal{M} \times 2^\mathcal{M} \rightarrow 2^\mathcal{M}$$

in such a way that $\forall\, \xi \equiv (P, R)$ where P is a set of its *provides* interfaces and R is a set of its *requires* interfaces, it returns P. Similarly, we define

$$\rho : 2^\mathcal{M} \times 2^\mathcal{M} \rightarrow 2^\mathcal{M}$$

[1] although not define

as a function that returns (in the case above) the set R representing the *requires* interfaces of the frame ξ.

Now, we are ready to define the relation mentioned in the beginning of this subsection:

Definition 3: Let Ξ be the set of all frames. We define a binary relation $\prec \subseteq \Xi \times \Xi$ in such a way that $\xi_1, \xi_2 \in \Xi$ are in relation $(\xi_1, \xi_2) \in \prec$ iff

$$\psi(\xi_1) \subseteq \psi(\xi_2) \wedge \rho(\xi_1) \supseteq \rho(\xi_2)$$

We will use the infix notation for this binary relation in further text, i.e. we will write $\xi_1 \prec \xi_2$ instead of $(\xi_1, \xi_2) \in \prec$.

Lemma 4: $\prec$ is a reflexive, antisymetric and transitive relation on $\Xi \times \Xi$.

Proof:
Ad reflexiveness: We have to show that $\xi \prec \xi$ holds for any ξ. This is obviously true because $\mathbf{S} \subseteq \mathbf{S}$ and $\mathbf{S} \supseteq \mathbf{S}$ holds for any set $\mathbf{S}$.

Ad antisymmetry: We have to show that for $\xi_1, \xi_2 \in \Xi$ the following holds: $\xi_1 \prec \xi_2 \wedge \xi_2 \prec \xi_1 \Rightarrow \xi_1 = \xi_2$. This is true as well because $\mathbf{S}_1 \subseteq \mathbf{S}_2$ and $\mathbf{S}_2 \subseteq \mathbf{S}_1$ implies $\mathbf{S}_1 = \mathbf{S}_2$ for any sets $\mathbf{S}_1$ and $\mathbf{S}_2$. Correspondingly, we can show it for $\supseteq$ in the second part of the conjunction.

Ad transitivity: We have to show that for $\xi_1, \xi_2, \xi_3 \in \Xi$ the following holds: $\xi_1 \prec \xi_2 \wedge \xi_2 \prec \xi_3 \Rightarrow \xi_1 \prec \xi_3$. This is also true because $\mathbf{S}_1 \subseteq \mathbf{S}_2$ and $\mathbf{S}_2 \subseteq \mathbf{S}_3$ implies $\mathbf{S}_1 \subseteq \mathbf{S}_3$ for any sets $\mathbf{S}_1$, $\mathbf{S}_2$, $\mathbf{S}_3$. Correspondingly, we can show it for $\supseteq$ in the second part of the conjunction.

□

Theorem 5: $(\Xi, \prec)$ is a complete partial order.

Proof: In Lemma 4, we have proven that $\prec$ is a reflexive, antisymmetric and transitive relation on $\Xi \times \Xi$. Now, we have to show that

1. There is a smallest element (labeled $\bot$) in the set Ξ such that $\bot \prec \xi$ for any $\xi \in \Xi$.
2. For any string $\Upsilon \subseteq \Xi$ there is a supremum $\sqcup \Upsilon \in \Xi$.

ad 1: We can find the smallest element as an element $\bot$ for which

$$\psi(\bot) = \emptyset \quad \wedge \quad \rho(\bot) = \mathcal{M}$$

ad 2: Since there are only finite number of elements in our relation, also any string consists of finite number of elements. The highest (most defined) element in the string is then the wanted supremum.

□

Theorem 6: $(\Xi, \prec)$ forms a lattice.

Proof: We need to show that any subset of elements has an infimum and a

supremum. This can be achieved in such a way that for any $\xi_1, \xi_2, ...\xi_n \in \Xi$ we can find the wanted supremum $\top$ as such an element for which

$$\psi(\top) = \bigcup \psi(\xi_1), ..., \psi(\xi_n) \quad \wedge \quad \rho(\top) = \bigcap \rho(\xi_1), ..., \rho(\xi_n)$$

and the wanted infimum $\bot$ as such an element for which

$$\psi(\bot) = \bigcap \psi(\xi_1), ..., \psi(\xi_n) \quad \wedge \quad \rho(\bot) = \bigcup \rho(\xi_1), ..., \rho(\xi_n)$$

□

Corollary 7: The Theorem 6 implies that there is a supremum $\top$ of all frames which can be found as

$$\psi(\top) = \mathcal{M} \quad \wedge \quad \rho(\top) = \emptyset$$

Taken ad absurdum, we can have only one component that can substitute all other components. Practically this, of course, does not make sense because then no components are needed and the intrinsic idea of the component-based approach is sublated. Therefore in real situations, components with a suitable weight are to be used (cf. subsection 2.3.5).

However, the lattice describes a structure of components from the viewpoint of substitutions. Such an idea can be practically useful within a particular architecture for substituting existing components for components with e.g. enhanced functionality or when we need to replace an existing component deployed at a node that is to be shut down or unsafe, etc. or a component with expired license, etc. with "similar" conforming one elsewhere on the SOFAnet.

7.1.3 Roles of behavior protocols in substitutability

Since the introduction of CDL in [Mencl–98], where the simple subtype relation from the previous subsection was informally mentioned, the language has been going through a lot of changes that have to be taken into consideration for the good solution of the substitutability. As many times mentioned, one of the most important change was the introduction of the notion of behavior protocols in [Visnov–99].

Thus the substitutability of frames can be defined from the behavior protocols viewpoint using the notion of *protocol conformance*. Using this notion, we have a particular frame protocol and any frame, all interfaces' interface protocols of which *conform* to the frame protocol, can be used in a component in specification of which this frame protocol appears. See subsection 4.3 or [PlaVis–02] for details. Such frames can be used (in the component with a particular frame protocol), interfaces of which do not violate the behavior determined by the frame behavior protocol. Intuitively, a frame protocol can allow at most the same communication on its *requires* interface instances (the output in communication ties) as respective interface protocols allow (so that a counterpart in a tie was able to fulfill it) and at least the same communication on its *provides* interface instances as the respective interface protocols allow (so that it could fulfill all communication required by a counterpart in a tie).

7.2 Inheritance of frames

The previous chapter is devoted to interfaces and particularly to establishing interface inheritance rules. Now, we will try to establish inheritance rules for the higher-level notion — frames.

To be able to do that, we have to understand the role of frames in the component creation. So let us recall that component frames can be understood as black-box views of components and serve for defining sets of services that are both provided by the component to the environment (recall that components are units of independent deployment) or required from the environment so that the component was able to provide the promised services. This defining is done by instantiating interfaces in the *provides* section respectively *requires* section of frames. Also recall the end of the previous section in which we discussed the notion of frame behavior protocol and its conformance to the interface protocols.

Before proposing any solution, we will discuss several approaches to the frame inheritance and estimate assets and troubles each of the approaches brings.

7.2.1 Inheritance as a pure frame composition — initial reasoning

When we discussed interfaces, we stressed that interfaces are intended to specify small, well-designed services and that combination of such services is the task of a component designer. Frames are just the place the service composition is done.

The question arises, how many services a component should provide (and proportionally require). This question is a subject of discussions. This fact influences also the number of components an application is composed of. For instance, Clemens Szyperski in [Szyper–00] speaks about *component introversion* (in cases an application is composed of one or two self-contained components) and *component extroversion* (in cases an application is composed of many components) — confer section 2.3 for more information about this topic. One of possible strategies is that components should be as small as possible (i.e. they should provide one or two services and require only services that those provided services demand). This strategy is good for some cases and brings its advantages (e.g. finer capability of updating), but on the other hand, in some cases the communication overhead would be unbearable.

Therefore, the natural idea of solving the granularity problem is to use inheritance. In such a case, a composition (or mixing) of smaller components (as a whole) is suitable.

However, SOFA component model — in fact, most component models — already has a different mechanism that at least partially solves this problem. A component template is a pair $< Frame, Architecture >$. Each single frame can be associated with a lot of architectures, namely compound architectures that aggregate (instantiate) subcomponents. We should realize the fact that frame provisions do not determine frame requirements because in different architectures various subcomponents can be instantiated and those subcomponents can have different requirements (which can be fulfilled using various combinations of subsume/bind ties).

Therefore, we will try to find if there is a solution that can be used even more frequently than the pure frame combination to maximize the price/value ration of the inheritance.

7.2.2 Frame modification and its impact on architectures

Components are composed and then modified quite often, therefore, as for the value, it will be maximized if we allow composition of frames and their subsequent modification (i.e. adding and defeating interfaces).

Let's have $T \equiv< F, A >$ a component template with a frame F and an architecture A. What happens if we create a frame F' by adding another interface into the *provides* part of F? First observation is that this modification makes sense: we provided a component with one more ability requiring no additional services. The second observation is that if the architecture is not primitive, this modification inherently coerces modifications of the architecture A because the architecture must describe this provision. The only way how the modification can be done is delegating this provision to a subcomponent but this delegation can be done arbitrarily (instantiating a new subcomponent, utilizing an exempt interface in a subcomponent, rearranging all ties or delegating already tied interface (which is supported by connectors)).

As for adding an interface instance as a requirement, the situation is similar. This modification again coerces (for compound architectures) rearranging of the architecture because if the component originally worked, the additional requirement makes sense in cases a subcomponent that provided such a service is not available or a subcomponent must be replaced with a subcomponent with more requirements, etc.

Ergo, if we are given a component template, after a modification of its frame, we cannot unambiguously adjust the associated architecture to correspond to the new frame. Anyway, a new component is created but it can have various architectures.

However, since it is quite natural that from the black-box view a gray-box view cannot be foretold, it is OK to consider frame inheritance without architectures on mind, even though it would be advantageous in many cases.

7.2.3 Frame inheritance with independent provisions and requirements handling

If we consider the objection against the pure frame combination from subsection 7.2.1 that requirements are not always dependent on provisions, the natural idea is to allow to inherit provisions and requirements separately. We could use multiple inheritance which would work as a union of interface types (more instances of the same interface types are unnecessary because connectors allow multiple ties). To avoid interface instance name collisions, a rule can be established that all interface instances are suffixed with names of their original frames and in cases of the same interface types in more ancestor frames, the instance name of the interface from the frame which is declared earlier in the descendant, is used.

Moreover separate interfaces could be independently added to both provisions and requirements. Thus, we get a *selective* — or better *semi-selective* — inheritance mechanism (cf. subsection 3.4.5) because we can select parts of the frame we inherit, albeit with a very coarse granularity — only three choices:

requirements, provisions or both; not single interfaces — that is why we termed it semi-selective.

Advantages of such a solution are obvious: frame provisions form usually a functional unit that can be inherited as a whole, frame requirements may be inherited from the same frame as well (thus we obtain a classical frame combination) or may not and we can add some requirements manually. Notice that the interface-level granularity makes no sense because it is the same as the manual addition of interfaces.

We have even proposed a syntax for this inheritance mechanism:

```
frame ComponentName
{  provides:
     inherited InheritedComponentName1, InheritedComponentName2, ....;

     InterfaceType1 interfaceInstance1;
     InterfaceType2 interfaceInstance2;
                    .....................................
   requires:
     inherited InheritedComponentName3, InheritedComponentName4, ....;

     InterfaceType3 interfaceInstance3;
     InterfaceType4 interfaceInstance4;
                    .....................................

   protocol:
       the definition of protocol goes here;
}
```

We can use a very simple demonstrational example of case of use of such an inheritance mechanism. This example is based on the CDL example from section 4.6.

We will consider the frames Teler and SuperIntendent which provide interfaces needed for a typical bank teller activity and an interface for a superintendent activity, respectively. Let us assume that there is a need for merging these two activities in order to equip the superintendent with the common teller capabilities (or vice versa). A frame for such a component can be created from the scratch or inheriting both provisions and manually adding requirements, or inheriting both provisions and both requirements (using name conflicts solving by union) or inheriting requirements from only a single frame. We will choose the last case:

```
frame SuperTeller
{
  provides:
    inherited Teller, Superintendent;
  requires:
    inherited Superintendent;

  protocol:
```

```
    // protocol goes here
}
```

This inheritance mechanism is undoubtedly the most flexible mechanism that could be proposed, however this fact brings a lot of drawbacks. First, inheritance should be preferably used only for justifiable purposes, especially for conceptual modeling, not for pure convenience. Also, it should be considered how the inheritance affects other involved abstractions. Moreover, these facts should be considered:

1. Since a frame should be created after a profound analysis, we will want to keep even the requirements in most cases, despite the fact that generally requirements can be arbitrary.

2. Frame protocols cannot be generally automatically modified and must be created from the scratch in the descendant again.

The latter fact is quite serious. From all these reasons, we reject this form of inheritance too.

7.2.4 Frame inheritance and frame protocols

After rejecting the previous suggestion of frame inheritance, we will examine the role of behavior protocols in the inheritance. Frame protocols tie provisions and requirements of the frame together by — as defined in [PlaVis–02] — specifying the acceptable interplay of method invocations on *provides* interfaces and reactions on *requires* interfaces.

Inheritance can address several (antagonistic) issues:

1. Similarly as in the case of interfaces, we may want to use inheritance to modify protocols (keeping the same provisions and requirements).

2. We may want to use inheritance to create frames with the same protocol but with some of the interface types replaced with compatible ones (i.e. the frame protocol must conform to the interface protocols of the new interfaces).

3. We may want to use a functional unit combination (using a frame combination) and, having ensured that interface instances are univocally named, automatically combine frame protocols of ancestors.

4. We may want to add individual interfaces. This addition coerces changes in the frame protocol.

However, unlike the interface protocol, the frame protocol modifications (required in the cases one and four) are quite complicated because events (method names suffixed by the interface instance names) can often repeat in the protocol and generally cannot be modified uniformly. That is why the complete protocol replacement must be utilized, which is the same case as in the previous subsection that we rejected.

Case two keeps the frame protocol intact and replaces only interfaces. Note, that this to be true, we must preserve even the interface instance names and

change only the interface types. Otherwise a mechanism must exist which renames the event in the protocol. This use of inheritance may be quite useful.

As for the case three, there may appear some problems inherent to the frame combination, especially naming problems. If two interface instances have the same name (but different type) and we want to automatically combine original protocols into one protocol joining the distinct protocols by a binary operator, we have to use both such event names referencing its own type. However, since it is a problem of the frame combination as such, it has to be solved by using some renaming techniques in the descendant.

Otherwise, this case seems quite useful, albeit it still has some drawbacks. Some of them were mentioned in subsection 7.2.1. We are allowed to use only whole existing frames without possibilities of their modification (such a modification would coerce the protocol modification). This is useful for composition of components and creating frames architecture of which may use some ancestor frames as subcomponents. We will elaborate this case in the final proposal for the inheritance of frames.

7.2.5 Mixin inheritance and frames

Mixin inheritance may seem to several issues including the issue of naming conflicts and granularity. Mixin inheritance should be as for its effect very similar to the classic frame combination but mixins are supposed to have even finer granularity than frames, albeit we said that frames should be composed from the most simple functional units as well.

Let us recall from subsection 3.4.6 that we recognize three types of abstractions concerning mixins: base parent classes (common frames in our case), mixins as small functional units that are combined with a parent frame and the result of this combination. This is very confusing per se, not to mention their coexistence with a lot of other abstractions in the SOFA CDL. However, this fact may help to solve naming conflict problems.

Of course, a very important question is how to define a mixin in this case, especially how this mixin should differ from a classic frame. To justify the addition of two other abstractions to the CDL, mixins must bring a lot of advantages. The challenge is to create mixins in such a way that they are able to adjust the parent frame protocols automatically and enable the communication between them and the original frame. However, even in such a case, the drawbacks caused by introducing these abstractions into the CDL will be very significant. That is why this type of inheritance may be a subject of further research, however, we will reject it for now as well.

7.2.6 The final proposal for the frame inheritance mechanism

If we look at what we have already gone through, we can see that there is no optimal solution — all solutions have their significant pros and their significant cons. Since the requirement for the automatic modification of frame protocols seems crucial, we will consider only cases two and three from the enumeration in subsection 7.2.4. This means that we allow the interface type replacement for a compatible one and we return to the original idea of combining frames, which

is a reasonable use of inheritance. But we want to find a unified view for both types of uses of inheritance using a sole inheritance mechanism.

The notation should allow to specify interface instance names, types of which are to be changed. The inheritance should be of the concatenation type (see subsection 3.4.2) because frames have no direct implementation consequences (unlike interfaces) and they are used solely for specification purposes.

Now, we look at the problems concerning name collisions. Three collisions may appear:

1. Name collisions of interface instances of the same name and same type in various ancestors.

2. Name collisions of interface instances of the same name and different types in various ancestors.

3. Name collisions of interface instances of the same name in the opposite frame blocks of various ancestors regardless of types (one interface is instantiated as a requirement and the other is instantiated with the same instance name as a provision in another ancestor).

There are several approaches to this problem. One is to suppose that these cases should not happen and reject them (consider them as compile-time errors), second is to rename the interface instances or a different approach can be applied to each of the name collision types. The renaming as such is quite easy because the inheritance is of the concatenation type. Interface instances can be renamed selectively or globally.

The final decision should be made after the CDL implementation is examined, however, for now we tend to the global renaming of interface instance names as the safest solution. All interface instance names should be suffixed by the name of the ancestor frame from which the respective interfaces come. Note that we forbid the case of direct inheriting more frames of the same name. Since the inheritance is not supposed to be too deep, the names should have a reasonable length. This solution also ensures that all interfaces from the ancestors are part of the descendant, which may become handy in some cases. This renaming should be also internally reflected in the protocols and externally when using these interface instance names in architecture ties.

Thus, we allow multiple inheritance under conditions described in the previous paragraph. All interface instance names are renamed by suffixing with the original component names. Moreover, we allow the replacement of interfaces for the compatible ones. Thus, a compile-time check must be done in cases of replacing interfaces, whether the frame protocol really conforms to the interface protocols of the new interfaces.

To summarize it, the final frame will consist of provisions given by the union of renamed provisions of the ancestor frames, requirements given by the union of renamed requirements of the ancestor frames and a protocol which is created from the protocols of ancestors combined by explicitly specified operators. Moreover, optionally, the frame can also consist of some interfaces substituted with compatible ones (such a compatible interface does not require modifications of protocols).

This unifies the frame composition with the interface substitutability because in the case when we want solely a replacement of some interfaces, we use this

inheritance with the original frame as the only ancestor. Thus we get the unified view.

The proposed syntax is as follows:

```
frame ComponentName
 inherits InheritedComponentName1, InheritedCompoentName2, ....;

 changes InterfaceInstnance1:: OriginalInterfaceType1 ⟹ NewInterfaceType1 ,
         InterfaceInstnance2:: OriginalInterfaceType2 ⟹ NewInterfaceType2 ,
    .....
 protocol:
  InheritedComponentName1 binary_operator InheritedComponentName1 ..... ;
```

By allowing the option of choosing a binary operator (from operators shown in subsection 4.3.1), we give the option to better control the way how the frame will be used. Most used binary operators will be the alternative operator, the and-parallel operator or the sequence operator.

As for the example of a typical usage of the proposed frame inheritance mechanism, see Chapter 9 which is devoted to a case study in which proposed inheritance mechanisms are used.

Chapter 8

Architecture-level problems

Now, we are reaching the most intricate part of the component specification in the CDL — the gray-box view of components. This specification level shows the particular design of the topmost level of components by instantiating subcomponent frames and specifying communication. Or an architecture can be primitive, i.e. flat, implemented in an underlying implementation language. See subsection 4.2.5 on page 29 for details.

8.1 Summarization of issues

As for the architectures, generally two quite orthogonal issues have to be distinguished: the substitutability and the inheritance.

8.1.1 Substitutability

Substitutability can be dealt with on two related levels:

1. **Connectivity** — on what conditions the requirement-provision, requirement-requirement and provision-provision ties can be established.
2. **Frame substitutability** — on what conditions a subcomponent of an architecture can be replaced with another one.

As for the connectivity, it seems appropriate to set rules for ties between interfaces of different types. Here, we summarize the possible situations:

- ties of two interfaces that have the same interface operation signatures except one can contain some additional ones; in other words, we are interested in situations, where one interface is a subset of the other (let's call it *subinterface*); this case can be split into two subcases:
  - ◇ the subinterface is instantiated in the *requires* part of a frame
  - ◇ the subinterface is instantiated in the *provides* part of a frame
- ties of two interfaces that have the same operation names except for that their signatures might differ; let's split it again into four subcases:

◇ interfaces have the same operations except for one interface contains an operation with less parameters than the corresponding operation in the other interface

◇ interfaces have the same operation except for one interface contains an operation with a parameter of a type that is different from the type of the corresponding parameter of the corresponding operation in the other interface

◇ the same case as the previous one except for one of the different types is a subtype of the other

◇ interfaces have the same operations except one operation's parameter differs in `in/out/inout` modifiers

We should consider these cases separately for each of the three ties (subsumption, delegation and the internal tie between subcomponents). Moreover, arbitrary combinations of these (sub)cases can occur in real situations. The way of handling occurrences of such cases is then the logical conjunction of restrictions of the individual cases.

As for the replaceability of frames, we have partially discussed it in section 7.1 on page 59 while dealing with frames. However, we will elaborate it a bit and apply conclusions from the connectivity aspects.

8.1.2 Inheritance

The main goal, however, is to propose an inheritance mechanism for architectures. This includes discussion of possibilities of combining two or more architectures together or constituting an architecture as an enrichment of another architecture. This requires classification of architectures and careful deliberation of possibilities.

8.2 Connectivity and conformance of ties

Let's discuss the connectivity problems first. This terminology is used in [LuVeMe–00] but we will use terms *interface conformance* and *frame conformance* which are closer to the terminology used in the SOFA/DCUP component model.

As to the author's knowledge, so far, no paper describing SOFA/DCUP component model has explicitly mentioned what kind of interfaces can be tied and, generally, it has been considered that two interfaces of equal type are to be tied (albeit protocols may allow more). Since this topic concerns the main goal of this chapter (the inheritance) only indirectly, we will present only a limited reasoning about this topic as an illustration, using the case of interfaces in strong suptype relation.

8.2.1 Ties between interfaces in a strong subtype relation

In this subsection, we will go one step beyond the requirement that a tie can be established between two interfaces of identical types only. We will consider

the simple subtype relation that was informally presented in section 6.2 when discussing interface inheritance. Let's remind it: an interface I is a subtype of another interface J iff it contains all method names and signatures plus possibly some more. We will name this relation *strong subtype interface relation* because the conditions under which two interfaces can be considered as subtypes are quite stict. This strong subtype relation fully conforms to the subtype definition used in subsection 5.2.4.

Now, we consider a situation where an interface I is a subtype of J and four frames:

- *FSubP* — contains an instance of I in the *provides* part of the frame
- *FSubR* — contains an instance of I in the *requires* part of the frame
- *FSupP* — contains an instance of J in the *provides* part of the frame
- *FSupR* — contains an instance of J in the *requires* part of the frame

Let's discuss various ties in various architectures these interfaces can be part of:

- *FSubP* and *FSupR* are instantiated in an architecture and we want to establish a tie between the instance of I and the instance of J using the *bind* between the subcomponents *FSubP* and *FSupR*; this is possible because all methods required by the J are supplied.
- in the opposite case (*FSupP* and *FSubR*), it is not possible to create a tie between their interface instances of types I and J because the provided interface of type J does not supply all methods required by the interface of type I
- an architecture of a component based on *FSubP* which contains an instance of *FSupP* — the instance of the type I cannot delegate accomplishment of its methods on the instance of J because it contains only a subset of them, thus the delegation tie cannot be established
- in the opposite case (arcihtecture based on *FSupP* which contains an instance of *FSubP*), the delegation tie can be established
- an architecture of a component based on *FSupR* which contains an instance of *FSubR* — the instance of the type I cannot subsume requirements for accomplishment of its methods to the instance of J because J contains only a subset of them
- in the opposite case (architecture based on *FSubR* which contains an instance of *FSupR*), the delegation tie can be established

Now, it is interesting to consider a tie as a type of $A * B$, where A and B represent interface types and $*$ can be one of *subs* for a subsumption tie, *deleg* for a delegation, *bnd* for a tie which binds a requirement of one subcomponent to a provision of another, and to discuss the subtype relation for them:

- *subs* is as such contravariant on its first projection and covariant on its second projection because if $A1$ is a subtype of $A2$ and $B1$ is a supertype of $B2$, and $A1$ *subs* $B1$ is a valid tie type, then $A2$ *subs* $B2$ is also a valid

tie type and, moreover, it is a subtype of the type $A1$ *subs* $B1$ in the sense that it can replace it; however, since it is not generally possible to replace a component with another with more requirements, the *subs* should be considered invariant

- the same goes for *deleg* (due to its opposite direction); again, it should be considered invariant because generally a component cannot be replaced with a component with fewer provisions
- *bnd* is contravariant on its first projection and covariant on its second projection because if $A1$ is a subtype of $A2$, $B1$ is a supertype of $B2$ and $A1$ *bnd* $B1$ is a valid tie type, then $A2$ *bnd* $B2$ is also a valid tie type and, moreover, it is a subtype of the type $A1$ *bnd* $B1$ in the sense that it can replace it

However, despite it is possible to find subcomponents, whose replacement suits the third case, most subcomponent replacements would also require modifications of their subsumption and provision ties, that's why it is not apt to try to set a rule for subcomponent replacements based on subtype relation of ties.

8.3 Inheritance of SOFA compound architectures

In Chapter 3, we presented the two main purposes for which inheritance is being used in OOPLs: essential use of conceptual modeling or accidental use for convenience. Now, we have to discuss why we want to use inheritance in the case of architectures and what possibilities we have.

8.3.1 Architecture of SOFA architectures

We know from Chapter 4 that a component is an instance of a template frame which consists of a frame and an architecture. Every compound architecture has at least one[1] subcomponent (as an instantiated frame). In general, there can be arbitrary number of subcomponents solving arbitrary tasks provided each of the component's provisions is delegated to a subcomponent's provision and each of the subcomponent's requirements is fulfilled either subsuming it in the component's requirements or binding it to a subcomponent provision. Thus, taken ad absurdum, provided above mentioned conditions are satisfied, nothing prevents us from instantiating e.g. subcomponents solving astronomic computations in a database component. That's why very little can be told about the gray-box view of components, provided we know the black-box view; a component template with a particular frame can have significantly differing architectures.

8.3.2 Semiformal notation for architectures

Although we have already informally introduced some terms concerning architectures (e.g. tie types), we will introduce a simple but slightly more formal notation for the SOFA architectures in this subsection. Note that in further

[1] you can notice that having only one subcomponent will be a very rare case because all that subcomponent's requirements would have to correspond to component ones and that subcomponent would have to supply all component's provisions

text, we will often denote architectures by the letter A and a possible index instead of the component name and version.

SOFA component architectures are characterized by subcomponents and ties on their (and also the mother component's) interfaces. Nothing more can be involved in the inheritance.

We will consider the definition of the notion of frames from subsection 7.1.2 as well as the definitions of projections ψ and ρ. We denote $\mathcal{R}$ set of all architectures. For each architecture $a \in R$ we denote a set S_a as a set of all frame names (i.e. subcomponents) instantiated in the architecture a. A tie can be viewed as $\theta \in \mathcal{Q} \times \mathcal{Q}$, where Q is a set of qualified interface names (consisting of a frame name and a interface instance name).

Definition: We define the SOFA component architecture as a quadruplet

$$A = (S, DT, ST, BT)$$

where S is a set of subcomponents, DT is set of delegation-ties, ST is set of subsume-ties, BT is a set of bind-ties.

Any primitive architecture has all those four sets empty. For any compound architecture of a sound component, at least S and DT are non-empty.

8.3.3 Architecture inheritance concepts dichotomy

First and quite a cardinal question is, if we want to consider architecture inheritance independently on frames or not. If the former was the case, we could use the inheritance relation across frames (i.e. components). For example, a descendant could be created by adding a new subcomponent (which provides new interfaces) to an existing architecture and, provided original ties do not change and the provisions are not exempt, this descendant would represent an architecture of another frame, i.e. another component (which has all provisions as the original one plus those that are provided by the subcomponent). This approach kind of contradicts the intuitive order of component design (first we determine what interfaces a component provides and what requires and then we describe the internal communication), but it allows broader and more natural inheritance.

If the latter was the case, the inheritance relation could relate only architectures of a single component, which may seem to better reflect the concept of component architectures but restricts possibilities of inheritance and makes it less intuitive.

We will discuss both possibilities in following subsections.

8.3.4 Inheritance of architectures of a single component

Now, we try to consider the latter case and to classify some possibilities which we might want to use inheritance for:

1. We want to replace a subcomponent with another without changing the structure of ties, i.e. the sets DT, ST and BT will be the same for both original and new architectures.

2. We want to the change structure of ties without changing subcomponents, i.e. at least the set S will be the same for both architectures.

3. We want change both subcomponents and ties.

The first case implies that the new subcomponent's frame protocol must conform to the architecture protocol, but it does not necessarily have to conform to the original subcomponent's frame protocol, because some of its provisions could be exempt and it is unnecessary for the new subcomponents to provide them as-well. All ties original subcomponent participated in have to be established by the new subcomponent as well. This type of architecture modifications may be quite handy when we need to preserve the logic of an existing component template.

The second case assumes that there are enough couples of interfaces that can be re-bound; for example, some originally exempt interfaces of some subcomponents become part of ties. However, in well-designed component architectures, such situations should occur very rarely.

The third case is most universal but also most complicated. If unrestricted, taken ad absurdum, all architectures (of component templates with the same frame) could be in the inheritance relation. This is, of course, undesirable because it neither models any sound relation nor it saves any designing effort. Thus, if we accept this case to be a serious candidate for the benefit of which the inheritance should be introduced, we have to find special cases for which it is reasonable.

8.3.5 Sound cases of architecture modifications

There are a few simple cases of architecture sound modification involving changes of both, the subcomponent set and any of the tie sets:

- a subcomponent is replaced with a subcomponent which has fewer requirements, thus some ties of type bind or subsume[2] are abolished
- a subcomponent is replaced with another subcomponent which has more provisions thus, if those provisions are compatible with some of existing subcomponents' provisions that form a tie, those ties (from DT or BT) may be replaced by a tie with this new subcomponent

But in most cases, several components will be involved in the modification. As an example, we will follow a case when several components are replaced with a single subcomponent. If we consider $B \subseteq S$ a subset of subcomponents, we can replace such subcomponents with a subcomponent that preserves interfaces which are part of ties that can be viewed as external from the B viewpoint. Internal ties can be abolished. This may be useful, for instance, when we want to change architecture from light-weighted architecture to more heavy-weighted one, e.g. in order to limit communication overhead in some practical cases. This modification does not change cardinality of the architecture's DT and ST.

The opposite case is also sound: we may want to replace a heavy-weighted subcomponent with several light-weighted subcomponents preserving its provides-interfaces and requires-interfaces which are divided among the new subcomponents with having possible additional provisions and requirements fulfilled by internal ties.

[2]but abolishing a subsume tie changes the component frame

It is obvious that to express these modifications using inheritance would be too intricate and thus not much useful. Not to mention cases when one subset of subcomponents is replaced with another set of subcomponents. Therefore, we will try to find another approach.

8.3.6 Architecture inheritance across components

This type of architecture modifications brings plenty of new forms of changes compared with the previous type. As we have already mentioned, allowing unrestricted addition might lead to the situation, where potentially all architectures are related by the inheritance relation, which would extremely degrade the inheritance relation value. Besides, from practical point of view, describing all the changes done in a descendant architecture compared with its ascendant architecture might took much more effort then creating the descendant from scratch. Thus, we have to propose a "sound" inheritance relations which would be valuable and also practically useful.

We can try to get inspired by the subtyping-based inheritance relations we have already proposed for the lower-level abstractions in the previous chapters. If we use a natural definition of supertype, it may be done in this way:

Definition: A_s is an ext-subtype architecture of an architecture type A_t iff

$$DT_{A_s} \supseteq DT_{A_t} \wedge ST_{A_s} \subseteq ST_{A_t}$$

In other words, a component which has an architecture that is an ext-subtype of another architecture type can be used in an all environments, where a component with the ext-supertype architecture can be used because it provides at least the services the component with the supertype architecture provides and it requires at most the services the component with the ext-supertype architecture requires. The number of subcomponents is irrelevant and also the structure of internal ties is irrelevant. However, we should notice that this definition follows the component point of view (external) and it is only a reformulation of the frame subtype relation in the terms of architectures. But it is a natural consequence of the facts that component architectures are not stand-alone abstractions but only a second approximations of components and that the first approximations (frames) provide information enough to decide whether subtypes preserve all the properties of supertypes (see the reasoning about subtypes in subsection 5.2.4) from the external point of view (see the component definition in subsection 2.3.1). That's why this definition of the architecture ext-subtype relation leads to the pure component substitutability and no inheritance relation reflecting the internal architecture structure is involved. If we wanted to take the internal view into consideration, this definition would not hold.

Therefore, in our quest for finding the optimal inheritance relation for architectures, we will use another approach that takes the internal point of view of component into consideration, in addition to the external point of view. A basic premise, we want to keep, is that a descendant must preserve all properties (i.e. all ties and subcomponents) from its ancestors.

Besides, we will examine the possibility to "synchronize" the architecture inheritance with the frame inheritance. Suppose we have component templates T_1 and T_2 consisting of frames C_1 and an architecture A_1 and C_2 and A_2, respectively. We create a descendant frame C by combining the two component

frames. There can be arbitrary number of component templates created upon the descendant frame C. However, in addition to creating new component frames using inheritance, we try to use also the architectures A_1 and A_2 to create an architecture A of the component template. This approach results in a complete component template inheritance.

8.3.7 Architecture combination-based inheritance concepts

To reflect the two conditions mentioned in the previous subsection and to obtain some other benefits, we propose to use the simplest possible solution: to combine architectures similarly like in the case of frames.

In a more detailed view this means that if we have architectures $A_1 = (S_1, DT_1, ST_1, BT_1)$ and $A_2 = (S_2, DT_2, ST_2, BT_2)$, the resulting descendant looks like this

$$A = A_1 \cup A_2 = (S_1 \cup S_2, DT_1 \cup DT_2, ST_1 \cup ST_2, BT_1 \cup BT_2)$$

This solution has a lot of advantages. Let's itemize some of them:

- the original architecture structures are preserved
- the architecture inheritance can be synchronized with the corresponding frame inheritance (the correspondence between delegation ties and provisions specified in a corresponding frame and subsumption ties and requirements specified in a corresponding frame respectively, can be found) due to the fact that the original tie sets were not modified in the descendant
- the complete component template inheritance is made possible as a direct consequence of the item above and the fact that component templates consist of frames and architectures
- a simple notation can be used because there are no modifications to describe
- descendant's protocol can be automatically obtained from its ancestors
- this approach follows the bottom-up concept — components are built from small functional units (some simple contracts) and if a more heavy-weighted component is required, the inheritance can be of help
- the language clarity is preserved — no complex modifications (as mentioned in previous subsections) are necessary

Of course, as any other solution, this solution has (or at least seems to have) also some drawbacks. As the main one seems the expressive power — we cannot simply take an existing architecture and obtain a new architecture as a result of its modification. But as reasoned in previous chapters, such an approach would result in quite a complex description with many negative consequences, including the fact that the effort spent on such a modification might in many cases exceed the effort necessary to create a completely new architecture.

Another important question is if the proposed solution works in all cases. The proposed architecture inheritance assumes union to be disjoint. This must

be solved using a renaming mechanism. We can notice that the names of subcomponents are the main cause of ambiguity. Thus, we propose a rule that all subcomponents in the descendant should be named (before they are inherited and the union is applied):
subcomponent_name_in_the_ancestor-name_of_the_ancestor_architecture
thus all subcomponents from ancestors will be contained in the descendant.

The names of the component's interfaces (i.e. the interface instances as declared in component frames) represent another cause of ambiguity. Now, the issue is here again whether the architecture inheritance should be considered as independent on frames or not. If the architecture inheritance was independent, we could rename these interfaces similarly as the subcomponents and the frame that this new architecture determines should be composed using this renamed interface instance names. If frames were supposed to be created first, then interface instances referenced in architectures would have to be renamed according to the names in frames, i.e. suffixed with a component name.

Again, this issue should be a subject of further discussions. However, for now we propose to rename the interface instances by suffixing them with the component name (not with the architecture version as proposed previously for subcomponents). This restricts the inheritance a bit (more architectures of the same component cannot be directly combined) but it will synchronize the architecture inheritance with the frame inheritance. In this situation, renaming subcomponents by suffixing with the ancestor architecture version is unnecessary, the component name is sufficient.

8.3.8 Architecture combination-based inheritance proposal

We propose the architecture inheritance in the spirit discussed in the previous subsection. The proposed syntax is the following:

```
architecture ComponentName version versionNumber
  inherits InheritedArchitecture1, InheritedArchitecture2, ....;
```

The semantics of such a notation would be the following: A new descendant architecture ComponentName version versionNumber is created in such a way, that it contains all subcomponents from the ancestor architectures but renamed and also the interface instances of ancestors, also renamed. All the ties are preserved but named according to the new names. No other changes are made. Thus, we can use inheritance to create new architectures as a composition of existing architectures.

8.3.9 Architecture enrichment-based inheritance concepts

The architecture inheritance based on architecture combination proposed in the previous subsection is perfectly usable in cases we want to merge whole existing architecture structures.

However, we may want to create a new architecture based upon an existing one. We have already shown that modifying existing ties is not a good way. Thus, keeping in mind previous subsections, we propose an inheritance mechanism

that preserves the subtype relation between ancestors and descendants and in which the descendants are created as independent enrichments of ancestors. This means that all four sets of structures in the new architecture (S_n, DT_n, BT_n, ST_n) become supersets of the original architecture (S, DT, BT, ST). In practice, it means that we create a descendant by adding new subcomponents (at least one) and describe ties of these subcomponents.

Keep in mind that these new subcomponents cannot be bound to the original subcomponents, thus the interfaces of the new subcomponents can be bound to corresponding interfaces of any of the new subcomponent and corresponding interfaces of the component itself (delegation and subsumption ties). Notice that this type of inheritance is not synchronized with the frame inheritance and it is assumed that a component designer has already designed the frame of the new component (including the frame behavior protocol), thus interfaces of the new component are known and can be bound to interfaces of subcomponents. Enriching architectures of the same component (without adding new interfaces to the component) makes no sense in most cases.

The problem of naming conflicts should not happen (the architecture designer can instantiate the subcomponent under an arbitrary name and, thus, he is supposed to choose a name that is not in use).

8.3.10 The final proposal for the architecture inheritance mechanism

We have suggested two inheritance mechanisms. Both of them have their reasons, each of them has its special cases when its using is more beneficial than using the other. However, having two separate inheritance mechanisms for a single abstraction would be too confusing and cases can be found when it is suitable to use both of these inheritance mechanisms at once. Since those inheritance mechanisms should not interfere, they should be unified without problems.

This unified view can be naturally transformed to one of those two original archetypal cases of inheritance just by simple instantiating no subcomponents, respective inheriting only a single architecture. Therefore, renaming is involved only if more than one architecture is inherited.

Notice, however, that using the architecture enrichment-based inheritance parts implies the loss of the synchronization with the frame inheritance and the whole component frame (including the anticipation of renaming of component interface instances) must be written by the component designer.

The proposed syntax is as follows:

```
architecture ArchitectureName version versionNumber
 inherits InheritedArchitecture1, InheritedArchitecture2, ....;
{
 inst FrameName1 frameInst1;
 inst FrameName2 frameInst2;
  ....
 bind NewSubComp1Inst: interfInstReq to NewSubComp2Inst: interfInstProv;
  ....
 delegate ComponentProvIntInst to NewSubCompInst: interfInstProv;
```

```
  ....
  subsume NewSubCompInst: InterfInstReq to componentReqIntInst;
  ....
}
```

The semantics of this notation is obvious: a new architecture is based on a combination of existing architectures plus some new subcomponents. These new subcomponents can communicate via newly established ties but they cannot interfere with the subcomponents from the original architectures.

Chapter 9

Case study: components for passive electronic banking

Now, having discussed all topics and proposed an inheritance mechanism for each of the discussed abstractions, we should, as a proof of the concept, present an example of a typical possible usage of these mechanisms. We will get inspired by the example from section 4.6, however, we will consider a somewhat more specific case and use a new methodology that is possible due to the new inheritance mechanisms.

Since this case study should serve for demonstrational purposes only, we will omit some definitions that are not essential for what we want to demonstrate, and we will use the saved space for commenting on the code. From the same reasons, the abstractions will be rather schematic and, in real applications, more complex definitions should be necessary.

9.1 Passive banking components example

9.1.1 The situation to be described

Suppose we have to design some components for a bank application. Our example will aim at the designing of components for providing passive remote electronic banking services to the bank customers. We will focus on two services: a phone banking and a GSM banking.

9.1.2 Some necessary interfaces

Interfaces define the base services involved. As we have already stated, we will define some essential interfaces only (no interfaces of subcomponents in architectures are presented here) and their methods will be rather schematic.

We present here an example of interface inheritance (in the case of customer--related interfaces). First, we define an interface `ICustomer` which is composed of general customer-related methods, subsequently the descendant `IPBCustomer` and `IGSMCustomer` interfaces which add methods specific for the phone banking, resp. GSM SMS banking, are designed. Also note that the new method in each of the descendants has the same name in both cases but different headers and

supposed functionality. These two methods could not have been added to a single interface but both interfaces created in such a way can be added to a single frame (a frame can offer both of these interface types). This is a typical illustration why we said that the frames were basic units of the functionality combination and we did not allow the interface combination, but the frame combination we did.

```
interface IPhoneBanking
{ bool  AcceptCall(in string password, out int choiceNo,
                   out string customerID, out int accountNo);
  bool  ProcessChoice(in int choiceNo, in string customerID,
                      in int accountNo, out string results);
  void  Answer(in string answerText);

protocol: (AcceptCall; ProcessChoice; Answer)*

}
```

```
interface IGSMBanking
{ bool AcceptSMS(in string GSMNumber, out string requestID,
                 out string customerID, out int accountNo);
  bool ProcessRequest(in string requestID, in string customerID,
                      in int accountNo, out string results);
  bool SendReply(in string GSMNumber, in string replyText);

protocol: (AcceptSMS; ProcessRequest; SendReply)*

}
```

```
interface ICustomer
{ bool  GetAccounts(in string customerID, out string accounts);
  bool  GetAddress(in string customerID, out string address);
  bool  GetDisposRights(in string customerID,
                        in string account, out string rights);

protocol: (GetAccounts* || GetAddress* || GetDisposRights*)

}
```

```
interface IPBCustomer
  inherits ICustomer
{
  bool GetCustomerID(in string password, out string customerID);

     GetDisposRights* ⇝ GetDisposRights* || GetCustomerID*

}
```

```
interface IGSMBCustomer
  inherits ICustomer
{
  bool GetCustomerID(in string GSMNumber, out string customerID);

     GetDisposRights* ⇝ GetDisposRights* || GetCustomerID*

}
```

```
interface IAccount
{ bool  CheckAccountExistence(in string accountID);
  void  CheckBalance(in string accountID, out currency balance);
  void  GetTransactionHistory(in string accountID,
                              out string transactionHistory);

protocol: (CheckAccountExistence* || CheckBalance* || GetTransactionHistory*)


}
```

9.1.3 Frames for basic passive banking components

Now, we will consider the black-box views of the components for two basic passive banking services. A phone banking component should provide the functionality of phone banking (in a real-life application, the phone banking functionality should be structured into more interfaces) and require phone banking-specific information about customers and information about accounts.

Of course, as mentioned when reasoning about the inheritance questions earlier in this thesis, the requirements are dependent on the subcomponents rather than the pure provided functionality, therefore there can be a lot of components that provide the phone banking functionality but have different requirements.

The same goes for the GSM banking component which provides the GSM SMS banking functionality and requires GSM banking specific information about customers and information about accounts.

```
frame PhoneBanking                        frame GSMBanking
{ provides:                               { provides:
   IPhoneBanking    iPB;                      IGSMBanking       iGSMB;
  requires:                                 requires:
    IPBCustomer     iPBC;                     IGSMBCustomer     iGSMBC;
    IAccount        iAcc;                     IAccount          iAcc;

 protocol:                                  protocol:
  (?iPB.AcceptCall                          (?iGSMB.AcceptSMS
      {!iPBC.GetCustomerID};                    {!iGSMBC.GetCustomerID};
   ?iPB.ProcessChoice                        ?iGSMB.ProcessRequest
      {!iPBC.GetAccounts;                       {!iGSMC.GetAccounts;
       !iAcc.CheckAccountExistence;              !iAcc.CheckAccountExistence;
       !iPBC.GetDisposRights;                    !iGSMC.GetDisposRights;
       !iAcc.CheckBalance};                      !iAcc.CheckBalance};
   ?iPB.Answer                               ?iGSMB.SendReply

   )*                                        )*

}                                         }
```

9.1.4 Using inheritance for combining the services of the passive banking components

Let us suppose that we have a bank that provides both, phone banking and GSM banking services and that a more heavy weighted single component that provides these services is needed. Using the inheritance mechanism proposed, we obtain a component which provides functionality of both types of passive banking forms. The names of the interface instances will be suffixed by the ancestor component

names in the descendant. This ensures that the `iAcc` account interface which is the same in both ancestor components, will appear twice in the descendant as expected.

Of course, this is not the only possible design of such components. We can create a completely different architecture, e.g. with subcomponents which provide specific services based on the information obtained from the component's `iAcc` interface. Such a component does not need the `IAccount` as a requirement. The same goes for customer-related interfaces.

```
frame PassiveRemoteBanking
inherits PhoneBanking, GSMBanking;
protocol: (PhoneBanking + GSMBanking)*
```

The fact that the frame protocol is derived from the original protocols using the alternative operator means that the component is supposed to process either the phone banking services or GSM banking services at a time, but not both. If we had wanted to create a component which is supposed to process both types of services at a time, we would have used the or-parallel operator instead.

9.1.5 Architecture descriptions of the passive banking components

Now, we focus on the gray-box view of components for the phone banking and the GSM banking. The architecture for the phone banking consists of three subcomponents: The first one is the core which implements the phone banking logic. The other two components should provide the core subcomponent with pre-processed customer resp. account info. Those components need external information from DBs of customers resp. accounts, obtained via respective interfaces. The GSM Banking architecture is similar but the core logic is, of course, different. Note that both architectures use the same subcomponent for customer management but its different interface. To save space, we will not specify the exempt interfaces. The subcomponent frames are not specified as well.

```
architecture PhoneBanking version v1
{ inst PBCore                sciPBCore;
  inst CustomerManagement    sciCustomer;
  inst AccountManagement     sciAccount;

  delegate iPB to PBCore: iPB;

  subsume  sciCustomer: iPBCustomer to iPBC;
  subsume  sciAccount:  iAccount    to iAcc;

  bind sciPBCore: iSpecCustInfo  to sciCustomer: iTheSpecCustInfo;
  bind sciPBCore: iSpecAccntInfo to sciAccount:  iTheSpecAccntInfo;
}



architecture GSMBanking version v1
{ inst GSMBCore              sciGSMBCore;
  inst CustomerManagement    sciCustomer;
  inst AccountManagement     sciAccount;

  delegate iGSMB to sciGSMBCore: iGSMB;
```

```
  subsume  sciCustomer: iGSMCustomer to iGSMBC;
  subsume  sciAccount:  iAccount     to iAcc;

  bind sciGSMBCore: iSpecCustInfo  to sciCustomer: iTheSpecCustInfo;
  bind sciGSMBCore: iSpecAccntInfo to sciAccount:  iTheSpecAccntInfo;
}
```

9.1.6 Using inheritance for combining the architecture descriptions of the passive banking components

Now, we can use the proposed inheritance mechanism (see subsections 8.3.7 and bellow) to create an architecture of a component that provides functionality of both forms of passive banking services.

```
architecture PasiveRemoteBanking version v1
 inherits PhoneBanking version v1, GSMBanking version v1;
```

The resulting architecture `PassiveRemoteBanking version v1` will contain structures of both ancestors. The subcomponents will be internally renamed, thus the `PassiveRemoteBanking version v1` architecture will contain six distinct subcomponent which will make independent architectures of subcomponents possible (i.e., for example, when creating the application by recursively binding nested component frames to concrete architectures, each of the original `sciAccount` frames can be associated with a different architecture). Also the component interfaces are renamed in such a way that they are suffixed by their respective ancestor component name.

9.1.7 Component templates and the inheritance

In this example we also demonstrate the possibility of the "synchronized" inheritance as mentioned when reasoning about architecture inheritance. Suppose the frames `PhoneBanking` and `GSMBanking` and architectures `PhoneBanking version v1` and `GSMBanking version v1` are descriptions of component templates

```
ctPhoneBankingComponent =
     (PhoneBanking, PhoneBanking version v1);
```

resp.

```
ctGSMBankingComponent =
     (GSMBanking, GSMBanking version v1);
```

When we want to create a new component template based on the `ctPhoneBankingComponent` and `ctGSMBankingComponent` component templates, we simply inherit both constituents (as shown in the previous subsections) and then we get the component template

```
ctPassiveRemoteBankingComponent =
     (PassiveRemoteBanking, PasiveRemoteBanking version v1);
```

Chapter 10

Evaluation and conclusion

10.1 Related work

This thesis has quite a large scope and we could divide related works into many categories (inheritance, formal description of components, types in component systems, etc.). This work drew from a lot of papers and, since most of them were mentioned throughout the thesis, we will focus here on very brief descriptions of some other interface and architecture description languages and highlight how their abstractions' abilities of incremental refinements are solved (the term "inheritance" is quite rarely used in cases of component systems). Apart from CORBA IDL and Microsoft COM which were mentioned in previous chapters, lets mention four classic representatives of ADLs and one experimental ADL based on XML:

Wright ([AlaGar–94]) — is designed to support the formal description of the architectural structure of software systems. It permits both, description of individual systems and families of systems. Wright is built around the basic architectural abstractions of *components*, *connectors* and *configurations*. Components have two important parts — an interface and a computation. The interface consists of several ports. Each port represents an interaction. The computation provides a more complete description of what is done. No inheritance is used but connectors are considered as composition patterns among components. A collection of interface instances combined via connectors is called a configuration.

C2 ([TMAWRN–96]) — is a component- and message-based style designed to support particular needs of graphical applications. A canonical form of a C2 component contains three distinct building blocks: a dialog, an internal object and an optional domain translator. C2 allows creation of subtypes of such a component by subtyping from any or all of the internal blocks. C2 also supports conformance checking mechanisms. It even allows subtyping from several types, potentially using different subtyping mechanisms due to multiple conformance mechanisms.

Rapide ([LucVer–95]) — is a computer language for defining and executing models of system architectures. The result of executing a Rapide model is a set of events that occurred during the execution, together with causal

and timing relationships. Rapide is a set of five languages: Types, Patterns, Architecture, Constraint and Executable Module. In addition to defining interfaces and function types, the Types language also allows deriving new interface type definitions by inheritance from existing ones, including the capability of dynamic substituting of subtypes for supertypes. However, at the higher levels (Architecture, ...), inheritance is not supported.

Darwin ([MaDEK–95]) — is a language for describing software structures very similar to SOFA CDL which has been around, in various syntactic guises, since 1991. Components hide their behavior behind well-defined interfaces and programs are constructed by creating instances of component types and binding their interfaces together. These compositions are considered as types as well, which leads to a hierarchical composition. Interfaces in Darwin can be parameterized and derived by inheritance from one or more base interface types. Also, component types can be defined explicitly or fully or partially typed from an existing component type (a partial component declaration).

xADL 2.0 ([DaHoT–01]) — is a highly extensible XML-based ADL. It is based on xArch, a core representation for basic architectural elements, that uses the XML schema extension method for extending this core. XML schemas provide only a single inheritance model for the type extension. Thus, it is not possible for two independent extensions of the same XML tag to co-exist in an XML document. Therefore xADL 2.0 introduces some artificial dependencies among the xADL 2.0 schemas. If the multiple inheritance is added to the XML schemas by the XML community, the xADL 2.0 schemas will be revised keeping only the necessary conceptual dependencies.

10.2 Future work

This thesis has been conceived rather comprehensively, thus it includes a great number of concepts and ideas — not always tightly related — work on which has been initiated only and they appear in this thesis either without a deep elaboration or done in depth only as an illustration for a selected part of the whole issue.

Since the subject of this thesis is not a self-contained issue but a contribution to the project of the SOFA/DCUP Component Model run by the Distributed Systems Research Group at the Charles University in Prague, further elaboration and fine tuning of the ideas and proposals sketched in this thesis have to reflect both, the general conception of that project and the progress in that project. Thus, it can be done only with a tight cooperation with the founders of that project and other contributors to it.

Two main directions in the further research can be distinguished:
First, the inheritance mechanisms for individual CDL abstractions that have been proposed in this thesis have to be brought to life. But before this can be done, it is necessary to finish some open issues that remained in the solution. Especially issues concerning solving naming conflicts and the details of inheritance synchronization should be given a final form, even the non-combinational form of inheritance of frames should be considered. Then, it is necessary to

closely inspect the CDL implementation and modify it by implementing the inheritance of interfaces, component template frames and component template architectures into the CDL compiler. After the close inspection of the CDL implementation and some consultations with the project contributors, some minor revisions and closer specifications of the concepts proposed might be done, so that the results of this thesis were better compatible with the current SOFA CDL implementation and the current state of the SOFA/DCUP project.

The second direction concerns formal approaches to the component conception and their reflection in the CDL. Component systems generally — and SOFA/DCUP in particular — introduce some entirely new conceptions and abstractions which have to be formally described and theoretical aspects of which have to be subjects of further research. This thesis has already touched some of these issues (let's mention the preliminary examples of ways how to seize the formalization of the notions like template frames and template architectures, problems of types and subtyping for various abstractions in CDL, etc.), however, it is necessary to elaborate a more complete and more consistent notation and to describe types using the lambda calculus or another formal approach.

Also, some other issues related to the CDL (especially issues related to architectures) that are not elaborated in detail in this thesis from the reasons that they were secondary to this work and too complex to be solved without a closer cooperation with other contributors to the SOFA/DCUP project, require closer elaboration in the future.

10.3 Conclusion

The main objectives of this thesis as specified in section 5.1 have been accomplished. Inheritance issues for all the three abstractions of the SOFA CDL were quite thoroughly analyzed from many viewpoints. In all cases, we discussed the assets of using inheritance within the given abstraction first, then several inheritance models and usage scenarios were presented and discussed and the solution that proved to be the best from the viewpoints studied was subsequently elaborated to the final mechanism with initial proposals for its basic syntax and semantics. On purpose, the syntax issues were not specified in too much detail because they were supposed to be precisely specified at the time of implementing the inheritance mechanisms into the latest revision of the CDL.

As for the results that arose from the solution, inheritance proved to be useful in some cases, however, we have shown that in cases of component specification languages, the inheritance is generally less important than in cases of full-fledged object oriented languages. We have also shown that inheritance mechanisms that allow more types of modifications (i.e. changing or defeating attributes of the examined abstractions), which fact theoretically implies that such an inheritance can be used used in more cases, cannot counterbalance the loss of the language clarity and possible improper inheritance usage. However, as for the frames, we allowed replacement of interfaces in descendants for conforming ones. Therefore, we can say that the proposed mechanisms are quite strict. We always tried to propose an inheritance mechanism that preserved a form of subtype relation in the inheritance hierarchy. As anticipated, multiple inheritance brought more power and more possibilities but also its inherent naming conflicts. Since components have the combinational nature which can

be expressed by multiple inheritance mechanisms quite well, we decided to use it in cases of components. We did not use it in the case of interfaces because interfaces do not have the combinational nature. Naming conflicts were solved by renaming because concatenation form of inheritance allows this technique. However, due to the component dichotomy (frames and architectures), we had to propose inheritance mechanisms for components twice (for each of the two abstractions separately) which brought some additional complications (especially when we wanted to synchronize the proposed inheritance mechanisms). All these facts resulted in quite a cumbersome naming technique. Mixin inheritance was also briefly considered but we did not see any place for the trichotomy of its abstractions in the SOFA/DCUP component framework.

In general, we can say that each of the considered inheritance mechanisms had its advantages and disadvantages and therefore often the criteria for our final proposals resulted from preferring one point of view over another. Therefore, we think that inheritance of SOFA components is still an open issue and many improvements of these foundations will be introduced. For example, after quite a long deliberation we decided to add a non-combinational form of inheritance to architectures. The same thing might be considered even for frames.

Apart from these primary results, this thesis brings a significant number of remarks and observations from the fields of object oriented programming, component systems and theory of programming languages in general. They are contained in both, the first chapters that do not bring directly original results, however, they provide readers with a compact consistent overviews of three research fields that are important for successful solution of the goals of this thesis and bring some observations that may be useful for our further work (for example assertions as complements to behavior protocols), and also in chapters that are devoted the the solution of the main goals. These chapters contain, in addition to that, a lot of partial or initial results that might be useful in further research. Especially those concerning formalization of SOFA CDL concepts should be highlighted.

Writing of this thesis brought a lot of direct and indirect benefits also to me, as the author. As for the direct benefits, I have gained a lot of knowledge of the latest results of the research in such exciting fields of software engineering as component technologies and object oriented programming. As for the indirect benefits, thanks to my supervisor, I have familiarized myself with the methods of cutting-edge research, I have discovered and examined a lot of papers and information sources, etc., which results in my better readiness for further activities in these research fields.

We believe that, despite the fact that some effort still needs to be done in order the results of this work to become practically useful, this master thesis hopefully represents a good introduction to the problems of inheritance in SOFA CDL and component systems in general, and that it can be helpful in future research in these disciplines.

Bibliography

[PublicList] Distributed systems research group: *List of publications*; http://nenya.ms.mff.cuni.cz/publications.phtml

[PlaBaJ–98] Plášil F., Bálek D.,Janeček R.: *SOFA/DCUP: Architecture for Component Trading and Dynamic Updating*; Department of Software Engineering, Charles University, Prague; published in Proceedings of ICCDS'98, Annapolis, Maryland, USA, May 1998

[PlaViB–99] Plášil F., Višňovský S., Bešta M.: *Bounding Component Behavior via Protocols*; Department of Software Engineering, Charles University, Prague; TOOLS USA '99 Conference Santa Barbara, CA, Aug 1999

[PlaMik–97] Plášil F., Mikušík D.: *Inheriting Synchronization Protocols via Sound Enrichment Rules*; Department of Software Engineering, Charles University, Prague; published in Springer LNCS 1204, Berlin 1997

[PlaBal–01] Plášil F., Bálek D.: *Software Connectors And Their Role In Component Deployment*; Department of Software Engineering, Charles University, Prague; published in Proceedings of DAIS'01, Krakow 2001, Kluwer 2001

[PlaVis–02] Plášil F., Višňovský S.: *Behavior Protocols For Software Components*; Department of Software Engineering, Charles University, Prague; accepted for publication in Transactions on Software Engineering, IEEE, Jan 2002

[Kral–98] Král J.: *Informační systémy*; Science, Veletiny, 1998

[KraDem–91] Král J., Demner J.: *Softwarové inženýrství*; Academia, 1991

[Brada–99] Brada P.: *Component Change and Version Identification in SOFA*; published in Proceedings of SOFSEM'99, 1999

[Brada–00] Brada P.: *SOFA Component Revision Identification*; Technical report No. 2000/9; Department of Software Engineering, Charles University, Prague, 2000

[Mencl–98] Mencl V.: *Component Definition Language*; Master Thesis; Department of Software Engineering, Charles University, Prague, 1998

[Mencl–01] MENCL V.: *Considering Updates when Describing Software Components and Application Configuration*; Department of Software Engineering, Charles University Prague, 2001

[Visnov–99] VIŠŇOVSKÝ S.: *Checking Semantic Compatibility of SOFA/DCUP Components*; Master Thesis; Department of Software Engineering, Charles University, Prague, 1999

[Hnetyn–00] HNĚTYNKA P.: *Managing Type Information in an Evolving Environment*; Master Thesis; Department of Computer Science; Charles University, Prague, 2000

[MenHne–01] MENCL V., HNĚTYNKA P.: *Managing Evaluation of Components Specifications Using a Federation of Repositories*; Tech Report No. 2001/2; Department of Software Engineering, Charles University, Prague, 2001

[RiSoch–94] RICHTA K., SOCHOR J.: *Softwarové inženýrství*, Lecture Notes, Technical University CVUT, Prague, 1994

[Ochran–79] OCHRANOVÁ R.: *Úvod do programování*; Faculty of Science, University of J.E. Purkyně, Brno, 1979; published by Státní pedagogické nakladatelství, Prague, 1982

[OchKoz–93] OCHRANOVÁ R., KOZUBEK M.: *Objektově orientované programování v Turbo Pascalu*; 1^{st} edition; Department of Theory of Programming, Faculty of Science, Masaryk University, Brno, 1993

[Zlatus–93] ZLATUŠKA J.: *Lamdba-Kalkul*; 1^{st} edition; Department of Theory of Programming, Faculty of Science, Masaryk University, Brno, 1993

[Stanic–99] STANÍČEK Z.: *Řízení implementace IS*; lecture notes; Department of Software Systems & Communications, Faculty of Informatics, Masaryk University, Brno, 1998

[Skarva–99] ŠKARVADA L.: *Principy programovacích jazyků*; lecture notes; Department of Theory of Programming, Faculty of Informatics, Masaryk University, Brno, 1998

[Brim–00] BRIM L.: *Komunikace a paralelismus*; lecture notes; Department of Theory of programming, Faculty of Informatics, Masaryk University, Brno, 2000

[LeaSit–00] LEAVENS G.T., SITARAMAN M.: *Foundations of Component-Based Systems*; Cambridge University Press; Cambridge, UK, 2000

[CardAb–96] ABADI M., CARDELLI L.: *A Theory of Objects*; Springer-Verlag New York, Inc., 1996

[MowRuh–97] MOWBRAY T.J, RUH W.A: *Inside CORBA: Distributed Object Standards and Applications*; Addison-Wesley, 1997

[LuVeMe–00] Luckham D.C., Vera J., Meldal S.: *Key Concepts in Architecture Definition Languages*; published as the second chapter of [LeaSit–00]

[WinOck–00] Wing J.M, Ockerbloom J.: *Respectful Type Converters for Mutable Types* ; published as the eighth chapter of [LeaSit–00]

[OMTR–98] Oreizy P., Medvidovic N., Taylor R.N., Rosenblum D.S.: *Software Architecture and Component Technologies: Bridging the Gap*; University of California, Irvine, CA, USA; http://www.ics.uci.edu/pub/arch; published in the Proceedings of Workshop on Compositional Software Architectures, Monterey, California, 1998

[Mikhaj–98] Mikhajlova A.: *Consistent Extension of Components in Presence of Explicit Invariants*; Turku Centre for Computer Science, Åbo Akademi University, Turku, Finland; Anna.Mikhajlova@abo.fi; published in the Proceedings of W-COOP'98, 1998

[BJPW–99] Beugnard A., Jézéquel J.M., Plouzeau N., Watkins D.:*Making Components Contract Aware*; published in Computer July 1999, ACM Publishing

[StHuSc–99] Stets R.J., Hunt G.C., Scott M.L.: *Component-Based APIs for Versioning and Distributed Applications*; published in Computer July 1999, ACM Publishing

[CicRot–99] Cicalese C.D.T, Rotenstreich S.: *Behavioral Specification of Distributed Software Component Interfaces*; published in Computer July 1999, ACM Publishing

[SzyWec–96] Szyperski C.; Weck W.: *Do We Need Inheritance?*, published in Proceedings of ECOOP'96, 1996

[Szyper–00] Szyperski C.: *Component Software and the Way Ahead* published as the first chapter of [LeaSit–00]

[Weck–97] Weck W.: *Inheritance Using Contracts & Object Composition*; Turku Centre for Computer Science, Åbo Akademi University, Turku, Finland, 1997 published in the Proceedings of W-COOP'97, 1997

[Sakkin–89] Sakkinen M.: *Disciplined Inheritance*; Department of Computer Science, University of Jyväskylä, Finland; published in Proceedings of ECOOP'89, 1989

[Taival–96] Taivalsaari A.: *On the Notion of Inheritance*; Nokia Research Center; published in ACM Computing Surveys, Vol 28, No.3, pages 438-479, Sept 1996

[Eliens–00] Eliëns A.: *Principles of Object Oriented Software Development, 2^{nd} Edition*; Pearson Education Ltd., 2000

[Meyer–87] Meyer B.: *Design by Contract*, Technical Report TR-EI-12/CO, ISE Inc, 1987

[Meyer–97] Meyer B.: *Object-Oriented Software Construction, Second Edition*, Prentice Hall, 1997

[MeyJez–97] Meyer B., Jézéquel J.M.: *Design by Contract: The Lesson of Ariane* published in Computer vol. 30, no. 1, January 1997

[Meyer–95] Meyer B.: *Object Success: A Manager's Guide to Object Orientation, its Impact on the Corporation, and its Use for Reengineering the Software Process*; Prentice Hall, 1995

[EddEdd–99] Eddon G., Eddon H.: *Inside Microsoft COM+ Base Services*, published by Microsoft Press, A division of Microsoft Corporation, Redmond, Washington 98052-6399, 1999

[SecCai–00] Seco J.C., Caires L: *Parametric Typed Components*, Universidade Nova de Lisboa, Portugal, 2000

[AlaGar–94] Allen R., Garlan G.: *Formalizing Architectural Connection*, published in Proceedings of the Sixteenth Int. Conference on Software Engineering, Italy, May 1994

[MaDEK–95] Magee J., Dulay N., Eisenbach S., Kramer J.: *Specifying Distributed Software Architectures*, published in Proceedings of ESEC'95, Sept. 1995

[TMAWRN–96] Taylor R.N, Medvidovic N., Anderson K.M., Whitehead E.J., Robbins J.E., Nies K.A., Oreizy P., Dubrow D.:A Component- and Message-Based Architectural Style for GUI Software. published in IEEE Transactions on Software Engineering, June 1996

[LucVer–95] Duckham D.C., Vera J.: *An Event-Based Architecture Definition Language*, published in IEEE Transactions on Software Engineering, Sept. 1995

[DaHoT–01] Dashofy E.M., Hoek A., Taylor R.N.: *A Higly Extensible XML-Based Architecture Description Language*, published in Proceedings of the Working IEEE/IFIP Conference on Software Architectures, Amsterdam, Netherlands, 2001