Master Thesis

Matej Polák

UML 2.0 Components

Department of Software Engineering
Advisor: Vladimír Mencl, Ph.D.
Prague, 2005
I would like to thank my advisor Dr. Vladimír Mencl for his valuable comments, advices and help with this work. I also thank to Prof. František Plášil for providing me an Enterprise Architect licence and to Mgr. Petr Hnětynka and Dr. Tomáš Bureš for their advices about SOFA. Finally I would like to thank to all people from Distributed System Research Group and ObjectWeb for creating SOFA and Fractal which gave a lot of new ideas to me.

I declare that I wrote this work by myself and listed all used resources.
I agree with making the thesis available to the public.

Prague, June 8, 2005

Matej Polák
Název práce: UML 2.0 Components
Autor: Matej Polák
Katedra: Katedra softwarového inženýrství
Vedoucí diplomové práce: Vladimír Mencl, Ph.D.
e-mail vedoucího: mencl@nunya.ms.mff.cuni.cz
Cílem této práce je analyzovat komponentový model poskytovaný připravovanou verzí UML 2.0. Možnosti a omezení komponentového modelu UML 2.0 jsou demonstrovány na mapování do dvou vědeckých komponentových modelů: SOFA a Fractal. Součástí mapování je popis různých problémů které vyvstaly z odlišnosti komponentového modelu UML 2.0, SOFA a Fractal a analyzuje různé možnosti řešení.
Součástí práce je vytvoření pluginu do některého vybraného UML CASE nástroje pro generování zdrojových kódů pro SOFA a Fractal. Jelikož UML specifikace 2.0 v době psaní této práce ještě nebyla dokončena, bylo nutné prozkoumat podporu designování komponent v různých CASE nástrojích. Součástí práce je porovnání 6 vybraných UML CASE nástrojů.
Klíčová slova: UML, components, CASE tools, visual development

Title: UML 2.0 Components
Author: Matej Polák
Department: Department of Software Engineering
Supervisor: Vladimír Mencl, Ph.D.
Supervisor's e-mail address: mencl@nunya.ms.mff.cuni.cz
Abstract: Unified Modeling Language - UML is the most used modelling language for designing and analysis of object oriented software systems. The forthcoming version 2.0 is supposed to provide constructs for modeling software components. Software components are building blocks with well defined access interfaces.
The goal of this thesis is to analyze the UML 2.0 component model. Mapping UML 2.0 to SOFA and Fractal component models is provided in order to demonstrate its possibilities and restrictions. Mapping describes various problems that arise from differences of component models of UML, SOFA and Fractal and provides the best way to utilize UML constructs for designing components.
A prototype implementation as plugin for chosen UML CASE tool to generate SOFA and Fractal source code is part of this work. Since UML 2.0 specification was not finished in time of writing this thesis, it was necessary to research the support of designing components in various CASE tools. The thesis comprises comparison of 6 chosen UML CASE tools.
Keywords: UML, components, CASE tools, visual development
# Table of Contents

1. Introduction .......................................................................................................................... 3  
   1.1. Components .................................................................................................................... 3  
   1.2. UML ............................................................................................................................... 3  
   1.3. The Goals of this Thesis ............................................................................................... 4  
   1.4. Structure of the Text ..................................................................................................... 4  
2. UML ....................................................................................................................................... 5  
   2.1. Basics ............................................................................................................................. 5  
   2.2. Extensibility of UML ..................................................................................................... 7  
3. Software Component Models ............................................................................................... 9  
   3.1. Introduction to Software Components ........................................................................... 9  
      3.1.1. Reusability and Maintainability of Components ...................................................... 9  
      3.1.2. Provisions, Requirements and Behavior Protocol .................................................. 9  
      3.1.3. Hierarchical Composition ...................................................................................... 9  
      3.1.4. Interconnections of Components ........................................................................... 10  
   3.2. Industrial Component Models ....................................................................................... 10  
      3.2.1. CORBA CM ............................................................................................................ 10  
      3.2.2. COM, DCOM, COM+ ....................................................................................... 11  
      3.2.3. .NET Framework and Web Services .................................................................... 12  
      3.2.4. Enterprise Java Beans .......................................................................................... 12  
   3.3. Research Component Models ....................................................................................... 13  
      3.3.1. SOFA .................................................................................................................... 14  
      3.3.2. Fractal ................................................................................................................... 15  
4. UML 2.0 Component Model: an Analysis ........................................................................... 18  
   4.1. Composite Structures ................................................................................................. 18  
      4.1.1. Internal Structures ................................................................................................ 18  
      4.1.2. Ports ....................................................................................................................... 19  
      4.1.3. Structured Classes ................................................................................................ 19  
      4.1.4. Collaborations ...................................................................................................... 19  
   4.2. Components .................................................................................................................. 20  
      4.2.1. Provisions and Requirements .............................................................................. 21  
      4.2.2. Attributes and Methods ...................................................................................... 22  
      4.2.3. Subcomponents ..................................................................................................... 22  
      4.2.4. Connectors ............................................................................................................ 24  
      4.2.5. Realization ............................................................................................................. 25  
      4.2.6. Inheritance ............................................................................................................ 26  
5. Mapping UML 2.0 Component Model to Existing Component Models .............................. 27  
   5.1. SOFA ............................................................................................................................ 27  
      5.1.1. Two Kinds of Components in SOFA .................................................................... 27  
      5.1.2. Subcomponents .................................................................................................... 28  
      5.1.3. Properties and Constants ..................................................................................... 28  
      5.1.4. Provisions and Requirements .............................................................................. 28  
      5.1.5. Three Kinds of Connectors .................................................................................... 29  
      5.1.6. Component Inheritance ........................................................................................ 29  
      5.1.7. Realization of Primitive Architectures ................................................................... 30
5.1.8. Behavior Protocol ........................................................................................................ 30
5.2. Fractal .......................................................................................................................... 31
  5.2.1. Components, Subcomponents and Shared Components ........................................ 31
  5.2.2. Attributes .............................................................................................................. 32
  5.2.3. Provided and Required Interfaces ......................................................................... 32
  5.2.4. Bindings ................................................................................................................ 32
  5.2.5. Component Inheritance ......................................................................................... 33
  5.2.6. Realization of Primitive Components .................................................................... 33
  5.2.7. Behavior Protocol .................................................................................................. 33
5.3. UML Profiles .................................................................................................................. 34
6. UML Tools Overview ...................................................................................................... 35
  6.1. Borland Together Designer ......................................................................................... 35
  6.2. Enterprise Architect .................................................................................................. 37
  6.3. Magic Draw ................................................................................................................ 40
  6.4. Poseidon for UML ..................................................................................................... 43
  6.5. Rational Rose XDE for Visual Studio .NET ............................................................... 45
  6.6. Visual Paradigm for UML ......................................................................................... 47
  6.7. Conclusion ................................................................................................................ 49
7. Prototype Implementation ............................................................................................... 50
  7.1. Requirements ............................................................................................................. 50
  7.2. Installation Guide ...................................................................................................... 50
  7.3. User's Guide ............................................................................................................. 50
    7.3.1. Plugin Menu ......................................................................................................... 50
    7.3.2. Main Window ...................................................................................................... 51
  7.4. Programmer's Guide .................................................................................................. 52
    7.4.1. Plugin Principles in Enterprise Architect ............................................................ 52
    7.4.2. Prototype Component Architecture .................................................................... 52
    7.4.3. Simplified UML MetaModel .............................................................................. 55
    7.4.4. Installation details ............................................................................................... 56
  7.5. Evaluation and Examples .......................................................................................... 56
8. Related Work and Future Path ......................................................................................... 58
9. Conclusion ........................................................................................................................ 59
References ............................................................................................................................ 61
A. Example ............................................................................................................................ A.1
B. SOFA Validations ............................................................................................................ B.1
C. Fractal Validations .......................................................................................................... C.1
D. Structure of Templates ................................................................................................... D.1
  D.1. SOFA Template ........................................................................................................ D.1
    D.1.1. Architecture ......................................................................................................... D.1
    D.1.2. Frame ................................................................................................................ D.2
    D.1.3. Interface .............................................................................................................. D.2
    D.1.4. Package .............................................................................................................. D.3
    D.1.5. UML Diagrams .................................................................................................. D.3
  D.2. Fractal Template ........................................................................................................ D.7
    D.2.1. Class .................................................................................................................. D.7
    D.2.2. Component ......................................................................................................... D.7
    D.2.3. Implementation Class ....................................................................................... D.8
    D.2.4. Interface and AttributesInterface .................................................................... D.9
    D.2.5. Subcomponent ................................................................................................... D.10
    D.2.6. UML Diagrams .................................................................................................. D.11

UML 2.0 Components
1. Introduction

First two sections of this section approximate the two worlds that are to be brought together in this thesis. Goals of this thesis are described in the third section and finally the last section of this section describes structure of this text.

1.1. Components

Software components have become an integral part of almost every software project. There are various reasons: reusability, interchangeability, interconnection of incompatible hardware platforms and distributed deployment.

The basic idea of a components oriented programming is to split the code into various independent parts with well known access points called interfaces. Components usually have two kinds of interfaces: provided and required. Provided interfaces specify services that a component provides to its environment. Required interfaces on the other side specify services that a component requires by its environment. Component's interfaces are abstractions describing services provided or required by the component.

In order to invoke services specified by provided interfaces of one component from another component, both components must be interconnected. Interconnections between components are often implemented by a concrete component framework which enables interconnection of components at a deployment time and allows components to work and communicate together without a knowledge of interconnection details. A developer of a distributed application therefore does not need to write a special code to enable communication of components for example through network since the communication is covered by a component framework.

Hierarchical composition is a concept of creating components. The main idea of hierarchical composition is to build components from other components called subcomponents. Components built from other components are called composite components. Composite components use their subcomponents to implement services defined by their provided interfaces. A hierarchical composition may be applied to arbitrary depth, which means that subcomponents may be also composed from other components.

Components that do not contain any subcomponents that would implement their provisions must implement these provisions by them selves. Such components are called primitive components.

A hierarchical composition creates a tree called assembly tree where tree nodes represent composite components and tree leaves represent primitive components.

1.2. UML

Since it's first version, UML - Unified Modeling Language became a standard of visual design of software applications. The first version of UML was created by “three amigos” - Grady Booch, Ivar Jacobson, and Jim Rumbaugh from Rational Software, later a non profit consortium for maintain computer industry specifications OMG - Object Management Group [21] has taken an administration of UML.

UML is used for a visual development where some steps of a development process are performed by designing diagrams representing application parts. The main advantage of visual development is easier understanding and orientation in designed diagrams, then studying a source code of an application.

For purpose of visual development various UML CASE - Computer-Aided Software Engineering tools exist. The main features of UML CASE tools are design UML diagrams, generate source code from diagrams and reverse engineering which is the contrary process to source code generation, when UML diagrams are created from a source code.
1.3. The Goals of this Thesis

Visual design using UML became an integral part of software development. Current version UML 1.5 [24], which turn to be very popular in past years, however does not comfortably support design of software components. The forthcoming version of UML - 2.0 provides support to design software components. The first goal of this thesis is to analyze UML 2.0 specification from a component design view considering common component characteristics. The analysis should provide background of UML 2.0 component model and provide various ways of how UML 2.0 constructs for modeling components may be mapped to different component characteristics.

The second goal is to provide mapping from UML 2.0 to SOFA and Fractal component models. The mapping should analyze various possibilities of mapping component concepts from UML 2.0 to research component models SOFA and Fractal. The best mapping of component concepts for both SOFA and Fractal should be chosen. UML 2.0 profile for SOFA and Fractal comprising defined mappings should be created. A prototype implementation to generate SOFA and Fractal source code from UML model should be developed to demonstrate chosen mappings.

The prototype should be implemented as a plug-in for an existing UML CASE tools. Therefore various CASE tools should be researched and analyzed their support of UML 2.0.

1.4. Structure of the Text

The Section 2 provides introduction to UML 2.0 in general, comprising four layer metamodel hierarchy of UML 2.0 and extensibility of UML 2.0.

Introduction to software components is described in Section 3. The section also contains description of various existing industrial and research component models.

The results achieved in this work are described in Section 4, Section 5 and Section 6. The Section 4 provides an analysis of UML 2.0 component model on an abstract level with no focus on a concrete component model. The usage of UML 2.0 component model is presented in Section 5 by defining mappings to SOFA and Fractal component models. The section describes various ways of designing various component concepts of SOFA and Fractal in UML 2.0. The Section 6 provides an analysis and comparison of various UML CASE tools with consideration for component design support.

The prototype implementation is described in Section 7. The section describes requirements and installation of the prototype implementation and also a user's and a programmer's guide.

Related work, future path of the work and enhancements of the prototype are described in Section 8 and finally the conclusion with accomplishments of goals is presented in Section 9.
2. UML

The Section 2.1 provides introduction to forthcoming version 2.0 of Unified Modeling Language and related languages. Two ways of extending UML semantics are described in Section 2.2.

2.1. Basics

The Unified Modeling Language - UML - is a modeling language for designing and analysis of application structure, behavior, business process and data structure of object oriented software systems. It is maintained by Object Management Group - OMG. At present OMG is preparing a new version 2.0 of UML [25], [26], [27], [28]. One of its major extension against version 1.5 [24] is ability to model Software Components.

UML 2.0 is a metamodel that makes a part of a four layer metamodel. The main idea of layered metamodel is that every layer is an instance of the layer above (except the first layer), adds a new functionality and refines semantics against the layer above.

The top most layer - meta-metamodeling layer referred as M3 defines basic constructs - a language for use in next layers. Metamodel Meta Object Facility - MOF is an example of this layer. MOF is also a base layer for UML layer.

The next layer referred as metamodeling layer - M2 is an instance of M3. Its the layer where UML is located in the four layer metamodel hierarchy, in particular UML is an instance of MOF. The M2 layer is more complex than the M3 layer. Its main role is to define semantics for how elements from a user model (layer M1) that are instances of elements from the UML layer (layer M2), get instantiated in the run-time layer - M0.

Example: On Figure 1 a class Person with an attribute name are instances of UML elements Class and Attribute. The UML semantics defines characteristics of an object JohnLennon, which is an instance of the class Person.

![Figure 1. Four Layer Metamodel](image)
The M1 layer is a layer where user models are designed. Instances of elements from layer M1 are running applications in layer M0. 

*Infrastructure Library* is a base for various metamodels including MOF an UML. It defines the package *Core* which consists of four packages:

- **Primitive Types** defines a set of basic types used for defining syntax of metamodels. There are four basic types: *Integer, Boolean, String, UnlimitedNatural*.
- **Abstractions** consists of various packages, that define basic elements and associations used in metalanguages, e.g. *Element, Classifier, Comment, Generalization* and others.
- **Basics** provides a minimal modeling language from which more complex languages like MOF are built. Defines constructs for specifying data types, types of elements, packages and class-based modeling.
- **Constructs** package defines various constructs, e.g. package merge, expressions, constraints and others.

*Infrastructure Library* defines a language - *metalanguage* - used by metamodels. The metalanguage is used in metamodels to specify their characteristics e.g. in UML is used to specify semantics of a user model. *Infrastructure Library* is used by both MOF and UML metamodels.

![Figure 2. Part of UML Metaclass Hierarchy](image_url)

UML consists of a set of *elements* organized in a hierarchy using various associations in between like generalization and composition. The UML elements referred as *metaclasses* are instances of elements from MOF. UML uses inheritance specified by generalization to define new elements with more precise semantics for
a user model. Constraints, attributes of UML metaclasses and associations are used to refine the semantics. Inheritance comprises inheriting attributes, constraints and associations of UML metaclasses and replaceability of metaclasses with their descendants. On Figure 2 is shown a small part of UML 2.0 class hierarchy. The part shows metaclasses related to UML component model.

UML defines semantics of user model rather more lax and more using natural language then with exact language. Although Object Constraint Language - OCL - is heavily used within a specification still a lot of semantics is specified using a natural language. This makes for various inexactitudes and logical errors in UML specification.

When talking about elements, it is necessary to determinate which layer context the talking involves. In the following text a sans-serif style font will used for UML elements that will be referred as UML metaclasses to distinguish against elements from a user model from level M1. To simplify the following text, the elements from user model will not be referred as instances of metaclasses, the name of metaclass will be used instead. For example “class” will be used instead of using “instance of the metaclass Class”.

In order to distinguish attributes of UML metaclasses from user attributes of a class or an interface that are instances of metaclass Attribute, UML attributes will be referred as meta-attributes and a sans-serif style font will be used for them.

2.2. Extensibility of UML

Extensibility of UML is necessary when modeling user model that contains special elements, which semantics is not offered by UML. UML 2.0 brings up two possibilities of extensibility. The first one is based on hierarchical model of UML metaclasses by defining a new metaclass inherited from an existing UML metaclass, with added constraints, meta-attributes and associations to specify desired semantics required by elements from a user model. The added metaclass becomes a part of UML therefore is treated as other UML metaclasses. However, this approach has three big drawbacks:

- a user model designed with UML without a new metaclass must be redrawn to use a new defined metaclasses in order to profit from new semantics specified by the new metaclass. This is often almost impossible due to size of user model and no support from modeling CASE Tools to provide an automatic replacement of elements.
- portability of a user model that makes use of the new metaclasses is limited. It can be viewed only in tools with UML extended with the new metaclass.
- the last problem arises from no support of extending UML metaclass hierarchy in CASE Tools. None of UML CASE Tools discussed in Section 6 supports adding new metaclasses to UML hierarchy defined by UML specification.

Due to restriction of UML metamodel, that is considered as read only it is not possible to modify existing UML metaclasses and therefore a new base metaclass can not be specified for existing UML metaclasses. This approach is therefore suitable for extending not for modifying UML metamodel.

Profiles [7], [28] is the second way of UML extensibility. Profiles are defined in a package Profiles from Infrastructure Library. The main purpose of package profiles is to extend existing metaclasses for different purposes with no need to define a new metaclass. The extension mechanisms are stereotypes, tagged values and constraints.

User defined profile is in four layer metamodel hierarchy on the same level as a user model. Elements are instances of metaclasses Class and Stereotype from package Profiles.

Stereotypes provide a way to define new virtual metaclasses with additional semantics from existing UML metaclasses. Stereotype does not define a new metaclass, it defines a new kind of an existing UML metaclass
instead. Therefore a stereotype must always be associated with a metaclass which it is extending. A stereotype
can not be instantiated within a user model it may be only applied to an existing element in order to create an
instance of a virtual metaclass defined by the stereotype.

Constraints are frequently used to define restrictions of semantics for a new metaclass. Constraints associated
with a stereotype are applied to elements when applying profile to a whole user model or when applying a
stereotype to elements.

Tagged values are an equivalent of meta-attributes from metaclasses. They are attached to a stereotype and
specify properties of a new virtual metaclass. These properties are appended to elements when applying a
stereotype to an element.

**Example:** Figure 3 shows a definition of two stereotypes SOFAFrame and SOFAArchitecture that may be
applied to a metaclass Component. Stereotype SOFAArchitecture has defined two tagged values: version and isPrimitive.

![UML Profile Definition Example](image)

A profile may be *applied* to a user model which means that stereotypes from profile may be applied to elements
that are instances of metaclasses, which the stereotypes extend. Applying a stereotype to an element means
apply constraints on the element and attach tagged values to the element as new meta-attributes. Various profiles
may be applied on a user model to refine semantics as long as they do not have conflict constraints. A profile
may be removed from a user model at will. Removal of an applied profile should not remove elements from a
user model, it may only remove constraints and tagged values attached to elements when applying profile to a
model or stereotypes to elements.

**Example:** Figure 4 show application of stereotypes SOFAFrame and SOFAArchitecture to components
Bank and BankImpl.

![UML Profile Application Example](image)
3. Software Component Models

In Section 3.1 is provided introduction to various characteristics of components. Section 3.2 describes existing industrial component models and whereas Section 3.3 introduces two research component models SOFA and Fractal. The descriptions in both sections are focused on component's characteristics described in Section 3.1.

3.1. Introduction to Software Components

A software component, or shortly component, is usually defined as a piece of code with well defined functionality, that may be accessed through access points. A component appears to a user of the component as a black-box therefore its usually referred as a black-box entity. The user view on a component is usually referred as a black-box view. An access point of a component is referred as a component interface.

The following sections describe characteristics of components common to various component models.

3.1.1. Reusability and Maintainability of Components

Reusability of components is one of major benefits of component-oriented programming. It is accomplished thanks to well defined access points and functionality. Advantage of components reusability is that a component developed because of use in any application may be easily reused within another application. Well defined access points and functionality allow use of a component by a programmer with no need to study internals of the component. Corba CM [29] and COM [14] component models allow even reuse of components written in one programming language within other programming language.

A black-box view of components is a reason of easy maintain of components. Internals of components that are hidden before a user may be modified with no need to modify its environment unless access points of components change.

3.1.2. Provisions, Requirements and Behavior Protocol

Provisions is a set of services provided by a component. On the other side requirements is a set of services that a component requires to be provided by its environment. Both provisions and requirements are usually specified by provided and required interfaces and invocations of services are implemented as functional calls. Provisions and requirements form only structure of services - signature. They do not specify a valid sequence of service calls - semantics of interfaces.

To describe semantics of component's interfaces a behavior protocol may be used [31]. Behavior protocol is a regular-like expression generating a set of traces - valid communication sequences. A behavior protocol specified to an interface or a component may be used to validate calling sequences of methods of the interface or services of the component.

3.1.3. Hierarchical Composition

The basic idea behind hierarchical composition is composing components from another components. Components that compose a component are called subcomponents of the component. The process of composing components from another components may be applied also to subcomponents to arbitrary depth.

Components within a hierarchical composition are usually split into two kinds: primitive components and composite components. A primitive component is a component that contains only a code and does not contain
any subcomponents. A composite component contains only subcomponents and does not contain a code. These components kinds are usually exclusive, therefore components with a code do not contain subcomponents and vice versa. Every component - composed and primitive - may be a subcomponent of an arbitrary composite component except itself. A “self-owning” is not allowed.

### 3.1.4. Interconnections of Components

Interconnections between components serve to invoke services of one component from another component. The most common way of interconnection of components is a simple functional call on a source code level. Usually the function call may be local in the same address space or between two processes or even via a network. A big disadvantage of the functional call on a source code level is that call properties are hard coded in a source code. This complicates reusability of components because change of a type of a call e.g. from calling method in the same address space to a remote procedure call, requires a modification and recompilation of a component.

A connector is an abstraction of communication between components. Its major advantage is a separation of communication and technical details from a business logic of components. In component models connectors usually interconnect provisions and requirements of components following rules:

- requirements to provisions: this connection is common on the same level of hierarchical composition of components. Usually only interconnections between subcomponents of the same component are allowed.
- requirements to requirements connection is used to pass requests from a subcomponent through owning component to an environment of the owning component.
- provisions to provisions connection passes requests arrived on a provided interface of a component to a provided interface of its subcomponent.

### 3.2. Industrial Component Models

This section provides description of various industrial component models, namely: Corba CM (Section 3.2.1), MS COM (Section 3.2.2), .NET and Web Services (Section 3.2.3) and Enterprise Java Beans (Section 3.2.4).

#### 3.2.1. CORBA CM

The *Common Object Request Broker Architecture - Corba* [23] specified by OMG is widely used standard for a middleware and distributed computing. A part of Corba specification called Corba Object Model provides support for a remote procedure calls independently on communication protocol, programming language, operating system and hardware platform. OMG also provides a set of Corba Object Services that involves naming service, trading service, transactions, security, persistence and others.

Corba uses *Interface Definition Language - IDL* to describe procedures and functions that may be remotely invoked. Mappings of IDL to various programming languages like C++ or Java are defined; that implies possibility to use Corba with various programming languages.

The version 3.0 of Corba specification introduces *Corba Component Model - CORBA CM* [29], based on Corba Object Model, which covers a set of software component features. Also extends Corba Object Services to provide component specific services for managing, configuration, deploying and interconnecting of components.

Corba CM defines *Component IDL - CIDL* which is an extension of Corba IDL for defining components. CIDL defines two kinds of components: *basic components* and *extended components*. Basic components serve to simply encapsulate existing Corba objects within a component. They can not inherit from other components, nor specify provisions and requirements. Only attributes are allowed to be specified for component configuration purpose.
Extended components provide a rich set of component functionality. Extended components provide two kinds of provisions and requirements for synchronous and asynchronous invocations. Provisions for synchronous invocations are called facets and requirements receptacles. Asynchronous provisions and requirements are called event sources and event sinks. Corba CM also provides a possibility to define attributes of extended components which are named values, primarily intended for a configuration purpose.

Corba CM is a flat model though it does not support hierarchical composition of components. Hierarchical composition may be substituted by exposing requirements, that represent subcomponent's interfaces, from a component and connecting them to a component that is supposed to be a subcomponent.

Corba Component Implementation Framework - CIF is a framework for constructing component implementations. The CIF uses CIDL source files to generate skeletons that contain implementations of various mandatory behaviors of components. Mapping to a programming language is performed in two steps: mapping from CIDL to IDL and then mapping from IDL to a chosen programming language. Implemented and compiled component is packed along with a generated component descriptor file and default properties file into a single zip file and deployed on a server.

There are two possible ways how components may be instantiated and interconnected. The first one is statically through an assembly descriptor which is a XML file providing necessary information about instantiating and interconnecting of components. The second way is dynamically at a run-time either from a source code using Object Request Broker - ORB services or using a Corba scripting tool. For this purpose Corba CM framework provides variety of introspecting services to determine all provisions and requirements of a component.

3.2.2. COM, DCOM, COM+

Component Object Model - COM [14] is a component model developed by Microsoft. COM component model is designated to run components within different processes on the same computer. A distributed version DCOM - Distributed COM extends functionality of COM to run components over network. The last COM version COM+ extends COM with Microsoft Transaction Server - MTS to use transactions, Message Queue Server - MSMQ for asynchronous invocations and other services for improving performance and security. COM specification is not restricted on any platform, however the mainly supported COM platform is Microsoft Windows.

COM components are not restricted on one programming language, they may be written in any programming language which compiler is able to compile into a binary file with an internal structure including virtual tables and function calling conventions as specified in COM specification.

COM uses Object Remote Procedure Call - ORPC which is built on top of DCE/RPC [38]. To define components a Microsoft IDL - MIDL which is an extension of CORBA IDL may be used. MIDL is not directly used by COM, its used to pre-generate source code with MIDL compiler.

COM component model allows to specify only provisions of components in form of provided interfaces. COM provides capabilities for introspection of provisions of components. Requirements must be obtained programmatically from a source code.

When a provided interface is defined it should not be changed, new methods should not be added, modified or removed. This restriction is not enforced however is highly recommended. Following this rule potential version incompatibility may be removed. A new functionality should added by adding a new interface instead of modifying existing interfaces.

COM supports two approaches in hierarchical composition:

- **containment**: an owning component reimplements part or all provided interfaces of subcomponents. Reimplementation of interfaces may pass calls to a subcomponent, which interfaces are reimplemented.
- **aggregation**: when a client is obtaining an interface from a component in order to invoke component's
services a subcomponent's interface may be returned and therefore a client works directly with a subcomponent.

COM hierarchical composition is not defined at a component definition level but at source code level. It is actually only a design pattern how to create composed components.

COM allows to specify attributes of components which serve to configure components. Values of attributes may be specified at a compile time or at a run time.

In order to use a COM component, it must be registered in a windows registry database. A Globally Unique Identifier - GUID, which is a 128 bit key is used to identify COM components within the registry database. Windows registry database also contains a table to convert a class name to a GUID identifier. COM components are shared within a system therefore any application can use any of registered components. A big drawback of shared components is that a replacement of a COM component with a newer version because of needs of one application, may cause compatibility problems in other applications that also use the replaced COM component.

### 3.2.3. .NET Framework and Web Services

.NET Framework is a new component model developed by Microsoft. It provides completely new approach in order to create and deploy component against COM. A specification of .NET framework may be found at [15].

Components in .NET are called Assemblies. Assemblies consist of compiled classes and a manifest file, that contains description of published data types and dependencies. .NET uses Microsoft Intermediate Language - MSIL, which is an interpreted byte code just like Java byte code. On a target machine a Common Language Runtime - CLR which is an equivalent of Java Virtual Machine must be installed.

Provisions of .NET components are described in a manifest file which is a part of an assembly. .NET framework provides a functionality to introspect component's provisions. Requirements of .NET components are not explicitly defined, they must be obtained programatically from a source code instead.

Hierarchical composition of components is not supported by .NET framework. It may be achieved only explicitly from a source code.

.NET framework is not incorporated with any concrete communication protocol for RPC although Simple Object Access Protocol - SOAP is the preferred and the most supported one.

Web services - WS standard [40], maintained by World Wide Web Consortium [41], is not a “real” component model. In fact WS is a standard that defines language to describe services that a client may invoke on a server, common XML based communication protocol SOAP which runs over HTTP and various supporting services (e.g. UDDI).

WS services are described using XML based Web Service Definition Language - WSDL. Services defined in WSDL may be implemented in any programming language that supports XML and HTTP communication handling. WSDL allows to specify provisions only. Requirements of components that implement services described in WSDL must be obtained programatically from source code of those components.

Web services standard also defines Universal Discovery, Description, and Integration - UDDI for web services discovering purposes. UDDI is divided in three parts: white pages to provide informations about company that developed a concrete web service, yellow pages to list web services and green pages to discover WSDL descriptions of a concrete web service.

### 3.2.4. Enterprise Java Beans

Enterprise Java Beans - EJB is component model developed by Sun Microsystems with actual version 2.1 [35]. Version 3.0 is being prepared at the time of writing this thesis, however it is not released yet. EJB is primarily
used for a client-server model of distributed computing, where clients connect to a server in order to access services provided by the server with an emphasis to access relational database. EJB components are limited to the Java programming language, however they may be invoked from various programming languages e.g. C++, C#, Visual Basic .NET.

EJB specification introduces three kinds of components called beans: Entity beans, Session beans and Message-driven beans. Communication between client and beans and also between beans is performed using Remote Method Invocation - RMI which is a Java implementation of Remote Procedure Call - RPC.

The main purpose of entity bean is to access remotely over network data stored in a database or another permanent storage. Each entity bean represents an object view on one record from a database, and is therefore identified by a primary key. Due to permanent storage background, entity beans are statefull. Entity beans may be shared between multiple users, that may use a primary keys to access a concrete bean. Invocations are performed synchronously.

Session beans are an opposite to entity beans: from their nature are not permanent and have no primary key since are not backed by a database or other form of permanent storage. Session beans are not shareable in general. However persistency and shareability may be achieved by explicit access to a database and use of beans handle. Invocations of session beans are synchronous. Session beans may be statefull and stateless. Statefull bean maintains its state across various method calls. It is intended to be used by one remote client in an instant. On the other side stateless bean does not hold its state and may be pooled and used by various remote clients in an instant.

Message-driven beans such as session beans do not represent any data directly, however they may access any shared data in an underlying database. Message-driven beans execute when a message from a client receives on a server, so their invocation is asynchronous. Message-driven beans are in contrast to entity beans relatively short-lived.

Beans expose two kinds of interfaces:

- remote interface which represents provisions of a bean. Provides an access point for a client to access services of a bean and must be implemented by a developer of a bean.
- home interface provides methods for creating and finding beans. Home interface is automatically provided by an EJB container.

Both kinds of EJB interfaces are provided interfaces. EJB does not support required interfaces of beans. Only requirements in form of co-operating EJB may be specified within a Deployment descriptor. A reference to related EJB must be however obtained programmatically within a code of a bean.

EJB Container is and application server for running beans. Beans are deployed together with Deployment descriptor which is a single XML file. EJB server usually provides various services similar to Corba Object Services: naming and trading service, transaction service and others.

### 3.3. Research Component Models

Research component model involve various characteristics that are not supported at all or that are supported partially by actual industrial component models.

- explicit specification of components requirements within a component definition in form of required interfaces.
- definition of nested subcomponents in a component definition, i.e. specifying the gray view of a component.
- dynamic component replacement and configuration.
- behavior protocol.
This section describes two representatives of research component models: SOFA developed at Department of Software Engineering at Charles University in Prague and Fractal developed by ObjectWeb Consortium.

3.3.1. SOFA

**SOFTWARE APPLIANCES - SOFA** is an academic project developed by the Distributed System Research Group at Charles University in Prague [9].

SOFA is a complex system allowing applications to be build out of a set of dynamically updateable components [12], [30], [9]. The main design issues of SOFA are: dynamic component update - DCUP, component reusability and hierarchical composition, design and run-time protocol validation, distributed deployment and versioning.

**Components and Subcomponents**

**SOFA Component Model uses Component Definition Language - CDL** which is an extension of Corba Interface Definition Language. The very first version of CDL is described in [12]. Other informations about CDL may be found in [9]. SOFA uses two kinds of components: frame and architecture.

Frame represents a component template. It defines a black-box view of a component which consists of provisions and requirements. Optionally also a behavior protocol and properties may be specified. Behavior protocol is used to formally specify communication among SOFA frame. The purpose of properties is to parametrize the component. SOFA frames do not support inheritance.

SOFA architecture is an implementation of a frame. Architecture may be either composed or primitive. Composed architecture is built of frames as subcomponents. It defines a gray-box view of a component. A primitive component contains no subcomponents, only a code implementing the component's functionality specified through its provisions. SOFA architectures also do not support inheritance.

Frames as subcomponents of an architecture are defined at a design time. At a deployment time, architectures that implement the subcomponents get instantiated instead of frames specified as subcomponents at a design time. Architectures are chosen according to an *assembly tree* that is stored in a XML based file called *assembly descriptor*.

The two SOFA components are one of its big advantages. The reason is a separation of component's external view, by defining provisions and requirements in a frame, from component's implementation in an architecture. The separation allows use of component types (frames) at a compile time and specifying of implementations of those component types (architectures) at a deployment time. The separation into frame and architecture is an equivalent to interfaces and classes in object oriented programming.

**Interfaces and Behavior Protocol**

Provisions and requirements of frames are specified in SOFA using provided and required interfaces or user defined types. Both provisions and requirements are specified with a type and with an identifier that must be unique within a frame. The type may be either an already defined interface or a user defined type. SOFA interfaces are based on CORBA IDL's interfaces. The extension against CORBA IDL interfaces involves a possibility to specify a version and a behavior protocol of interfaces.

Likewise in IDL, user define types may be defined in SOFA CDL. The user defined types may be then used to specify provisions and requirements. The advantage is that the usage of user defined types enables specifying complex provisions and requirements in form structures or arrays of interfaces.

Behavior protocol allows to formally specify valid communication sequences of frames or interfaces. Behavior protocol of an architecture is dynamically generated by CDL compiler from a behavior protocol of its
subcomponent's and a frame that the architecture implements. The behavior protocol is then used to check the conformance of behavior protocols of architectures, frames and interfaces of frames.

**Connectors**

Interconnections between components are specified in SOFA through connectors. Connectors define interconnection semantics and separate deployment dependent details from an application logic contained in components. Connectors are first-class entities like component thus solve *deployment anomaly problem* described in [2], [3] and [4]. Connector type specifies semantics and implementation of a concrete connection type. Connector types similarly to components are specified by a *connector frame* and a *connector architecture*. Connector frame specifies the type of a connector by describing services provided by a connector. Connector architecture contains an implementation of a connector. Similarly to component architecture there are two kinds of connector architecture: *compound* connector architecture containing other connectors or components and *simple* directly implementing a connector frame. Three connector types are predefined in SOFA: *CSProcCall* for synchronous calls, *EventDelivery* for asynchronous calls and *DataStream* for data streaming.

In addition to a connector type, SOFA specifies three kinds of connectors depending on entities the connector is connecting:

- **bind** - connects required interface of a subcomponent to a provided interface of another subcomponent. Both subcomponents must be situated within the same owning component.
- **delegate** is used to forward requests received on a provided interface of a component to a provided interface of its subcomponent. Delegate connector interconnects provided interface of an owning component with a provided interface of a subcomponent.
- **subsume** passes requests originating in a subcomponent through required interfaces to an owning component. An owning component passes these requests through its required interface and a connector connected to this interface to its environment. Subsume connector therefore connects required interface of a subcomponent to a required interface of an owning component.

**SOFA\text{Node} and DCUP**

*SOFA\text{Node}* is an environment for developing, distributing and running SOFA applications. Consists of five logical parts: *Template repository - TR* containing CDL descriptions and implementations of components, three parts *CDL compiler, Template Information Repository - TIR* and *Code generator* for application development and *RUN* which is an environment for running applications. SOFA\text{Node} may be distributed over network on various hosts. Several SOFA\text{Nodes} may be interconnected forming a *SOFA\text{Net}.*

*Dynamic Component Update - DCUP* enables safe component updating and replacing at a runtime. DCUP components, which are extended SOFA components consist of two parts: *permanent* and *replaceable*. The permanent, not replaceable part is controlled by a *Component manager - CM* that controls the runtime life cycle of a component. The replaceable part of a component is controlled by *Component builder - CB* that also builds the part's internals.

**3.3.2. Fractal**

Fractal component model is developed by *ObjectWeb Consortium* [19]. It is designed to be used in variety different software branches, e.g. middleware, operating systems, information systems, graphical user interface libraries. The main Fractal's design principles are: composite components, shared components to model resources, introspection capabilities to monitor a running system, configuration and dynamic reconfiguration capabilities.
Fractal [6] component model consists of a framework for instantiation of components and a set of specifications what a component should or should not implement depending on what control capabilities a component developer wants to offer to a user of the component. The control interfaces are special provided interfaces with predefined names organized in levels of control with gradually increasing reflective and introspection capabilities of components:

- lowest level components have no control capabilities, only their methods may be invoked. These components serve only as a component embedding of existing objects.
- the next level provides introspection capabilities of components through a standard interface. This interface allows a user of a component to discover all external interfaces of the component.
- the last level, also called configuration level provides control interfaces to introspect and modify content of a component that consist of subcomponents interconnected with bindings.

Modular and extensible organization allows use of Fractal in different situations from highly optimized and hardly configurable [32] to less optimized and heavily configurable and dynamic applications.

Components and Subcomponents

Fractal components consist of two parts: a controller and a content. The controller is a set of interfaces designated to control behavior, functional and non-functional aspects of component like introspection, configuration, security, transactions. Controller interfaces may be internal which are accessible from component's subcomponents or external which are accessible from outside of the component. Further controller interfaces are divided in functional interfaces and control interfaces. The functional interfaces are provided or required interfaces, representing functional aspects of a component: its provisions and requirements. Functional interfaces are equivalent to remote interface from EJB described in Section 3.2.4. Control interfaces that are equivalent to EJB's home interfaces are provided interfaces that correspond to non-functional aspect of components such as introspection or configuration. Fractal defines various predefined controllers: attribute controller, binding controller, content controller and life cycle controller.

The content represents internals of a component. Fractal defines three kinds of components depending on its internals and exposed control interfaces:

- composite component that is a component that exposes content controller in order to add or remove its subcomponents.
- primitive component that does not expose its content controller, but has at least one control interface.
- base component that does not expose any control interface.

Fractal allows a component to be owned by various distinct components. Such components are called shared components. They are usually used to represent shared resources.

Interfaces, Interconnections and Behavior Protocol

Fractal allows to define provided and required functional interfaces of a component. All interfaces of a component must have an unique name. Provided interfaces are called server interfaces and required are called client interfaces. Interfaces of fractal components have two characteristics: contingency and cardinality. The contingency of an interface indicates availability of a functionality specified by the interface while a component is running. Contingency may have two values:

- mandatory which means that the functionality provided by the interface must be available while a component is running.
- optional indicating that the interface's functionality is not guaranteed to be available while a component which owns the interface is running.
The cardinality indicates the multiplicity of an interface, i.e. how many instances of the interface a component may have:

- **singleton** specifies, that the component has only one instance of the interface.
- **collection** cardinality specifies, that the component may have arbitrary instances of the interface.

Bindings in Fractal are used to interconnect components. Fractal defines two kinds of bindings:

- **primitive** bindings to interconnect one client and one server interface. Primitive bindings can not cross component boundaries, i.e. bindings may be defined only between two components having the same owner or between component and its subcomponents and vice versa. The server interface must be a sub type of the client interface. In other words, the server interface must accept all invocations, that the client interface can emit.

- **composite** bindings to interconnect arbitrary number of component interfaces and language types. Composite bindings are not a special type since Fractal has no connectors like SOFA and has no special support for them. They are built out of a set of binding components (stubs, skeletons, adapters, etc.) which are regular components interconnected with primitive bindings.

Fractal does not support behavior protocol at the time of writing this thesis. However in cooperation with Distributed System Research Group at Charles University in Prague and Academy of Sciences of the Czech Republic work on support of behavior protocol in Fractal is in progress at the time of writing this thesis.

**Fractal ADL**

Since Fractal is a set of specifications and an run-time framework, a Fractal Architecture Description Language - ADL was created in order to define Fractal components. Fractal ADL is an open, extensible XML based language. Fractal ADL hides some implementation details, like implementation of attribute, binding or content controllers.

Fractal ADL defines two kinds of components: primitive and composite. Primitive components specify only provisions and requirements in form of provided and required interfaces and a content which is a name of an implementation class. Composite components contain nested subcomponents that may be interconnected with bindings. Fractal ADL allows two ways of specifying subcomponents:

- specifying a subcomponent as a named instance of an already defined component. The definition of a subcomponent contains an identifier of the subcomponent and a name of the subcomponent's type, which is a component elsewhere. This method is more reusable since one definition of a component may be used to define various subcomponents.

- as an embedded definition of a subcomponent's type within an owning component. This method is not recommended due to its less reusability because the whole definition of a subcomponent must be copied every time when a new subcomponent of the same type is defined.

The nesting of components may be done to an arbitrary depth.

In Fractal ADL instead of specifying a component to be shared, a reference to another component may be specified. The referenced component is considered to be a shared component.

A definition of components and also subcomponents in Fractal ADL allows multiple inheritance. Inheritance is simply an extension of mechanism, that allows adding and overriding component's interfaces, subcomponents, bindings, attributes and implementation class definitions. Conflicts resulting from multiple inheritance are solved by linearizing the inheritance graph.
4. UML 2.0 Component Model: an Analysis

This section will provide an analysis of a UML 2.0 component model [25], [26], [27], [28]. Section 4.1 describes UML's package Composite structures that provides basic constructs for defining components whereas Section 4.2 describes metaclass Component and provides analysis of UML 2.0 component model.

As already mentioned in Section 2 a sans-serif style font is used for UML metaclasses in order to distinguish UML metaclasses from elements from a user model.

4.1. Composite Structures

In package Composite Structures UML 2.0 introduces constructs for specifying internal structures, interconnections and collaborations of elements within a containing classifier. According to UML 2.0 specification [28] on page 68 a Classifier means “a classification of instances”. Internal elements, interconnections and collaborations defined within a classifier define characteristics of run-time instances of the classifier. The metaclass Classifier is an abstract metaclass and therefore can not be instantiated. To simplify the following text, an instance of an abstract metaclass which can not be instantiated within a user model means an instance of one of its instantiatable descendants, e.g. “instance of a Classifier” will be used instead of “instance of instantiatable descendant of a Classifier”.

Constructs specified in a package Composite Structures are used by a package Components in order to define characteristics of a metaclass Component.

Composite Structures consist of four packages: Internal Structures, Ports, Structured Classes and Collaborations all described in following sections.

4.1.1. Internal Structures

Package Internal Structures defines constructs for specifying an internal structure within instances of StructuredClassifier, which is an abstract descendant of a Classifier. Internal sub elements of a structured classifier are defined using instances of metaclass Property. Elements within a structured classifier are called parts. Parts have a name and a type which are specified with meta-attributes name and type. Part's name specifies a role that the part plays within its owning classifier and part's type specifies a classifier that will be instantiated within an instance of the part's owning classifier. Part declares that instances of an owning structured classifier will contain instances of a classifier specified with the type of the part.

![Figure 5. A Class with Interconnected Parts](image)

**Example:** On Figure 5 a class GuiProgram contains four parts with types: Menu having role menu, MainWindow with a role main and two Toolbars with roles edit and standard. Within an instance of the GuiProgram four instances of its parts will be contained, namely one instance of Menu, one instance of MainWindow and two instances of Toolbars.
Elements within a structured classifier may interconnected with connectors. Metaclass Connector is an abstraction, that is used to define which instances will be interconnected, but it does not specify a type and semantics of the connection (e.g. a simple pointer, a function call or a network connection). Connectors may have a name in order to describe the purpose of a connection.

Class GuiProgram on Figure 5 specifies that within its instances four instances of its parts will be created and interconnected. Parts Toolbar and Menu will be linked with a connector representing synchronization of elements from menu and toolbar.

4.1.2. Ports

Package Ports introduces metaclass Port as a communication point of a Classifier. Every communication between internals and environment of an instance of a Classifier passes through this communication point.

Port specifies the communication with provided and required interfaces. Provided interfaces represent services that the classifier offers to it's environment whereas required interfaces represent services, that the classifier requires by its environment.

Example: Figure 6 presents a component DatabaseEngine with two ports: da with one provided interface IDataAccess and logger with one required interface ILoggerAccess.

![Diagram of Ports with Provided and Required Interfaces and a Multiplicity](image)

UML 2.0 allows to define a multiplicity and a meta-attribute isService of a port. The multiplicity of a port indicates how many instances of the port will be created within an instance of an owning element. The meta-attribute isService of a port is rather ambiguous. One of its possible use is to specify whether a port is mandatory thus must be connected to another port or must not.

Example: Two ports da and logger are shown on Figure 6. Port da has a multiplicity 0..*.

4.1.3. Structured Classes

The very simple package StructuredClasses adds an ability to a metaclass Class to have an internal structure and own ports. In contrast to instances of Class from package Classes that can not contain any sub elements nor ports, instances of Class from package StructuredClasses may contain sub elements playing specified roles and ports as communication points.

4.1.4. Collaborations

Package Collaborations specifies how elements within a structured classifier collaborate. The package also contains constructs to define a collaboration template that may be applied on a structured classifier to explain how the structured classifier and its internals work.

Example: On Figure 7 a collaboration Sale with roles Seller and Buyer is shown. The collaboration is used by a class InternetShop which internal elements play roles specified by the collaboration.

Since collaborations are not important for an analysis of UML 2.0 component model, more detailed description is not provided. More details about collaborations may be found in [28].
4.2. Components

Metaclass Component defined in package Components represent “a modular part of a system that encapsulates its contents and whose manifestation is replaceable within its environment.” in [28] on page 168. Component provides a black-box view, since it hides its internals from its environment and specifies access points through which component communicates with its environment.

Figure 8. A segment of UML 2.0 Class Hierarchy showing UML Metaclasseis Involved in UML 2.0 Component Model.
Component is a descendant of a metaclass Class from package StructuredClasses which is described in Section 4.1. The inheritance allows components from a user model have attributes and methods and enables participation of components in various associations and generalization.

The main purpose of a UML 2.0 component is to serve as a type that specifies a part of an application typed by its provisions and requirements. When instantiating an application, an element that realizes the component is used instead of the component. This enables specifying the realizing element of a component at a deployment time.

Example: Figure 8 shows a position and inheritance of metaclass Component from metaclass Class from package StructuredClasses in a part of UML 2.0 class hierarchy.

The following sections describe characteristics of UML 2.0 metaclass Component and provide analysis of UML 2.0 component model. In the following sections a “component” means an instance of UML 2.0 metaclass Component if not specified other.

4.2.1. Provisions and Requirements

Provisions of a component specify services provided by the component to its environment. Requirements on the other side specify services that a component requires by its environment. UML 2.0 provides two ways of specifying provisions and requirements:

- direct provided and required interfaces attached directly to a component. Direct provided and required interfaces specify anonymous provisions and requirements of a component, since they have no identifier.
- using ports with provided and required interfaces. Since ports may have a name, provided and required interfaces of a port which is attached to a component therefore specify a named set of provisions and requirements of the component.

Example: Figure 9 shows both ways how to specify components in UML 2.0. Component DataStore has direct provided interface IDataStoreAccess and direct required interface ITMAccess whereas component TransactionManager has provided interface ITMAccess attached via port tma and required interface ILoggerAccess attached via port lg.

Using of ports brings up various advantages:

- a multiplicity of provision and requirements may be specified through a multiplicity of a port. The port's multiplicity specifies, how many instances of the port will created within an instance of an element which realizes a component.
- a meta-attribute isService of metaclass Port may be used to specify a behavior of a port more exactly. As mentioned in Section 4.1.2 UML 2.0 does not give exact meaning of this meta-attribute. One of its possible usage is to specify contingency of provisions and requirements as described in Section 5.2.3.

![Figure 9. Provisions and Requirements of Components](image-url)
4.2.2. Attributes and Methods

The inheritance of metaclass Component from metaclass Class as shown on Figure 8 allows components from
a user model to own attributes and methods. Attributes of components are usually used for configuration
purposes. Methods a component may be used to implement services provided by the component.

Example: Figure 10 shows a diagram of a component Logger with one attribute m_format and two methods
logMessage and clearLog.

![Diagram of Logger component]

Figure 10. A Component with Methods and Attributes

4.2.3. Subcomponents

Subcomponents are integral part of a Hierarchical Composition of components which is used to define internal
structure of components. UML 2.0 is very open in specifying internal structure of components. Not only a
component but even interface, class, instance of component or class may be internal elements of a component.
In UML 2.0 there are two ways of specifying subcomponents, using instances of metaclass Property or
instances of metaclass PackageableElement.

Subcomponents using Metaclass Property

Because a metaclass Component is a descendant of a metaclass StructuredClassifier its instances may have
an internal structure in form of parts as described in Section 4.1.1.

The use of properties brings up two advantages:

- multiplicity: UML 2.0 allows to specify a multiplicity of properties. That means that within an instance of
an owning component as many instances of its parts will be created as specified with their multiplicity.
- reference: Reference property means that an instance of a property will not be an integral part of an
owning component. A reference to an instance of a part within another component will be created instead.
Although UML allows specifying reference properties it does not define how to specify the reference end.
We have discovered two possibilities of specifying the reference end. The first one uses association
between the reference property and an element the property is referencing. Problem with this solution is
that the reference property does not inherit neither ports nor provisions and requirements from the
referenced element because association does not support inheritance of element's features. Generalization
can not be used instead of association, because property can not participate generalization.

The second way is to specify the referenced element through a type of a property. This method allows use
of ports, provided and required interfaces of a classifier which is a type of the property. A small
disadvantage is that only instances of Classifier may be used as a property type. Therefore instances of
InstanceSpecification can not be used as the reference end and therefore specify an instance of a
component as referenced element. The metaclass InstanceSpecification is described in the following
section.
Example: Figure 11 shows a component DatabaseEngine with two parts named ds and tm with types DataStore and TransactionManager. Part ds has a multiplicity with value 1..*.

Figure 11. Parts as Subcomponents.

Subcomponents using Metaclass PackageableElement

According to UML 2.0 specification instances of metaclass Component may contain instances of descendants of metaclass PackageableElement. PackageableElement is a base metaclass for various UML 2.0 metaclasses as shown on Figure 8 including Class, Component and Interface. Therefore a component may contain other components, interfaces or classes as subcomponents.

When using a component as a subcomponent, it is possible to define a hierarchical composition of a component and its nested subcomponents to arbitrary depth at one place with no need to define internals of subcomponents outside the owning component. Such a definition of subcomponents is called embedded definition of components.

Figure 12. A Component as a Subcomponent.

Example: Figure 12 contains an example of a component DatabaseEngine with two subcomponents DataStore and TransactionManager which has a subcomponent FileLogger. Internals of component TransactionManager are declared within a component DatabaseEngine with no need to be declared elsewhere.

Metaclass InstanceSpecification represents “an instance in a modeled system” [28] page 101. Type of an instance is specified by an meta-attribute classifier, that specifies a classifier that will be instantiated when instantiating an application. InstanceSpecification is an descendant of a PackageableElement as shown on Figure 8 and therefore its instances may be used as subcomponents.

Since Class, Component and other metaclasses are descendants of metaclass Classifier, the InstanceSpecification represents instances of instances of these metaclasses. The word “instances” is twice because instance specification represents instances of elements from user model that are instances of UML metaclasses. InstanceSpecification therefore allows a component to own instances of other components.

Example: Figure 13 shows a component DatabaseEngine with two subcomponents ds and tm. Subcomponent ds represents an instance of a component DataStore and a subcomponent tm represents an instance of a component TransactionManager.
4.2.4. Connectors

Connections between components specify links through which components communicate. Realization of these links may be arbitrary complex, though it may be a simple pointer or even a secure network communication channel. The UML 2.0 metaclass Connector is designated to specify connections.

Connectors are used to interconnect provisions and requirements of components. Since provisions and requirements may be specified using ports or direct provided and required interfaces, connectors may be attached to either ports, interfaces, components or subcomponents. Various possibilities of attaching connectors are shown on Figure 14.

When using instance of a Property or instance of an InstanceSpecification as a subcomponent a connector may be attached to it, although neither property nor instance specification may have ports nor provided or required interfaces. The reason is that both property and instance specification represent classifiers that will be instantiated when instantiating their owning component and classifiers might own ports, provided and required interfaces.

UML 2.0 specifies two kinds of connectors:

- an assembly connector that connects either port with required interfaces or required interface of a component to a port with provided interface or to a provided interface of another component. Simply an assembly connector connects requirements of one component to provisions of another component. An
assembly connector is usually used by various component models to interconnect subcomponents having the same parent. However this constraint is not required by UML 2.0 specification. An assembly connector should be designed as “a lollipop in a slot”.

**Example:** Figure 15 shows an assembly connector connecting port tma of component ds to port tma of component tm.

- a delegate connector connecting provisions to provisions or requirements to requirements of components.
  A delegate connector is usually used to connect component’s provisions to its subcomponent’s provisions or subcomponent’s requirements to its owning component’s requirements. In the first case a component uses delegate connector to pass requests invoked on its provided interface to its subcomponent that implements services specified by component’s provided interfaces. In the second case a component uses delegate connector to pass subcomponent’s requirements to environment of the component. The described usage of a delegate connector is not prescribed by UML 2.0. It is a usual usage of a delegate connector. A delegate connector should be designed as a connector with an open arrowhead and optional stereotype «delegate».

**Example:** On figure Figure 15 both usual usages of delegate connector are shown. The first connects port da of a component DatabaseEngine to a port dsa of a subcomponent ds. The other delegate connector connects port lg of a subcomponent tn to a port lg of its owning component.

![Diagram](image-url)

**Figure 15. A Component With Two Subcomponents, Two Delegate and One Assembly Connector.**

### 4.2.5. Realization

Realization of a component means realization of services provided by a component through its provided interfaces or provided interfaces of its ports. Realization of component’s provisions may be specified in three ways using either component’s methods or component’s subcomponents or using a realizing classifier. The first two methods were described in Section 4.2.3 and Section 4.2.2.

In order to specify a realizing classifier that realizes component’s provisions an UML 2.0 metaclass Realization may be used. Realization is a Dependency that specifies a classifier that realizes the component’s provisions. Usually a class is used as a realizing classifier. Realization is designed as a dashed arrow with a closed head directing from an realizing element to a realized element.

**Example:** On Figure 16 class DataStoreImpl realizes provided interface IDataStoreAccess of component DataStore.
4.2.6. Inheritance

Inheritance of metaclass Component from metaclass Class allows components to participate generalization. Generalization in UML is used to specify inheritance of elements. Inheritance in UML 2.0 may be specified between two arbitrary classifiers. Since Component, Class, Interface and others are descendants of Classifier instances of all of them may participate inheritance in a user model.

When a base classifier of a component is a class, the component may inherit all attributes and methods from the base class. When a base classifier of a component is a component, the descendant component may inherit from the base component all provisions and requirements in addition to methods and attributes.

Example: Figure 17 show two components FileLogger and NetLogger both descendants of a component Logger. Both components inherit methods logMessage and clearLog and port lg.
5. Mapping UML 2.0 Component Model to Existing Component Models

This section describes mappings of UML 2.0 component model to SOFA in Section 5.1 and to Fractal in Section 5.2.

5.1. SOFA

Designing SOFA Components with UML 2.0 brings up various problems resulting from a strict syntax of SOFA and universality of UML 2.0 specification. Therefore there must be created a set of constraints that refine UML 2.0 component model semantics to be suitable to design SOFA Components.

Following sections provide analysis of various ways of designing SOFA components in UML 2.0.

5.1.1. Two Kinds of Components in SOFA

SOFA introduces two kinds of components: frame and architecture. Frame serves as an abstract component type that defines provisions and requirements. A frame provides a black box view of a component.

An architecture is an realization of a frame in context of provided and required interfaces. There are two kinds of architectures which differ from each other the way how they realize a frame: a primitive architecture and a composite architecture.

A primitive architecture realizes provided interfaces of a frame by itself, which means that the architecture defines a class that implements all services provided by a frame through its provided interfaces.

A composite architecture uses other components to implement provisions of a frame. An architecture describes its internal structure on the first level of a hierarchical composition by defining its subcomponents and connections between them and itself.

![Figure 18. Both SOFA Components.](image)

UML 2.0 component model provides only one kind of components. For needs of SOFA it is necessary to distinguish between two kinds of components. There are two solutions:

- extend UML class hierarchy by adding two descendants of metaclass Component whose instances represent SOFA frame and architecture components within a user model.
- use UML Profiles that are described in Section 2.2 and define two virtual metaclasses with stereotypes «SOFAArchitecture» and «SOFAFrame» that may be applied on components within a user model and therefore distinguish between both kinds of components.

The latter approach has two major benefits. It does not require modifying existing model because UML profile may be applied to already designed model. The second one is more technical since none of UML CASE tools
discussed in Section 6 supports adding new metaclasses to metaclass hierarchy defined by UML 2.0 specification.

We have chosen the later approach in a prototype implementation. To indicate that an architecture is implementing a concrete frame a realization dependency may be used.

**Example:** Figure 18 shows a usage of both stereotypes «SOFAArchitecture» and «SOFAFrame». Architectures DataStoreFile and DataStoreDB implement frame DataStore.

### 5.1.2. Subcomponents

An internal structure of a component also called a gray box view of a component specifies elements contained by the component on the first level of a hierarchical composition and interconnections between them. Internal elements are usually referred as subcomponents. Such a component is called composite component.

In SOFA only an architecture may have an internal structure. Subcomponents of an architecture at a design time may be frames only. When instantiating an architecture a suitable architecture that implements a frame is chosen according to an assembly descriptor.

When mapping UML 2.0 to SOFA it is necessary to constrain subcomponents to be owned only by an architecture and allow only two methods of specifying subcomponents using instances of:

- **Property** as a part of a component, which meta-attribute **type** must be set to a frame component already defined within a user model.
- **InstanceSpecification**, which meta-attribute **classifier** must be nonempty and must be set to a frame component already defined.

Metaclass **Component** is not allowed to specify subcomponents because SOFA does not allow embedded definition of components which means that a component is defined within another component.

### 5.1.3. Properties and Constants

Properties in SOFA serve for configuration purposes of SOFA components. Frame and architecture are designed as instances of metaclass **Components** which is a descendant of metaclass **Class**. Therefore properties of frames and architectures may be designed as attributes of components. Since SOFA does not support default values of properties a default value of an attribute should not be specified otherwise it will be ignored.

Constants of frames may be specified in UML 2.0 as attributes with meta-attribute **isReadOnly** set to **true** and with default value specified.

It is important to specify constraints on data types of attributes. A data type of UML 2.0 attributes may be specified either by a simple data type or by a classifier that must be defined in a model. SOFA allows only primitive data types of properties therefore data types specified by a classifier should not be allowed. SOFA has following primitive types: short and long (both signed and unsigned), char, wide char, boolean, floating, string, wide string fixed and octet. More detail may be found in [12] and [9].

### 5.1.4. Provisions and Requirements

Provisions and requirements of SOFA frames may be specified either by interfaces or by user defined types. Except for some special cases, user defined types as specified in SOFA are not possible to be designed in UML in general. Therefore we have decided to use only provisions and requirements specified by interfaces.

Provisions and requirements using user defined types are only a by-product of a source code generator when using provisions and requirements with multiplicity.
Provided and required interfaces of a SOFA frame must have an identifier that identifies individual provisions or requirements and is used in an architecture to specify interconnections of provisions and requirements of components.

UML 2.0 supports two ways of specifying provided and required interfaces: direct provided and required interfaces or via ports. More detailed description is provided in Section 4.2.1.

The first way allows to specify only types of provisions and requirements and does not allow to specify identifiers of neither provided nor required interfaces. A solution is to create these identifiers dynamically in a source code generator.

When using the second way the name of a port may be used as a name of provided or required interface of a SOFA frame. Only one interface is allowed to be connected to a port. This constraint is due to SOFA bindings, that may interconnect only two interfaces. More interfaces attached to a port could cause ambiguities because when interconnecting two ports that have more than one interface, it is ambiguous to choose which two interfaces of ports should be interconnected. One solution is to generate SOFA bindings to every allowed combination of intersection of interfaces of ports. However this solution is undesirable because it modifies semantics of generated source code against a designed model, making more SOFA bindings out of one UML connector.

The use of ports has also one big advantage, it allows to specify a multiplicity of provisions and requirements. Provisions and requirements with multiplicity in form of ports with either provided or required interface and multiplicity different from 1 are transformed to new SOFA types which are arrays of interfaces. This types are then used as provisions or requirements of frames. When a multiplicity does not contain both upper and lower bounds as numbers, a concrete values must be completed by a user in a generated source code.

Provided and required interfaces of SOFA may have behavior protocol specified that indicates legal calling sequences of interface's methods. A manner of specifying behavior protocol to SOFA interfaces is described in Section 5.1.8

5.1.5. Three Kinds of Connectors

SOFA uses three kinds of connectors for interconnecting components:

- **bind** for connecting requirements to provisions of two subcomponents having the same owner.
- **delegate** for connecting provisions of a component to provisions of its subcomponent.
- **subsume** for connecting requirements of a subcomponent to requirements of its owner.

UML 2.0 specifies only two kinds of connectors: assembly and delegate. An assembly defined by UML 2.0 has the same nature as a bind connector defined by SOFA. Therefore SOFA bind connector may be designed with UML 2.0 assembly connector.

UML 2.0 delegate connector compounds both surplus kinds defined by SOFA: delegate and subsume. Both SOFA connector kinds - delegate and subsume may be unambiguously discriminated from UML connector delegate according to interfaces to which a connector is connected. In case of connector between required interfaces of a subcomponent and a component, a connector is a subsume connector. When a connector is between provided interfaces of a component and a subcomponent then a connector is a delegate. Other cases are invalid and not allowed by SOFA.

5.1.6. Component Inheritance

As described in Section 4.2.6, UML provides constructs for designing inheritance of components. In addition to a class inheritance ports, provided and required interfaces may be inherited.
SOFA does not allow *component inheritance* at all. There are three ways to solve component inheritance in designed user model:

- prohibit usage of component inheritance in a user model. This is the most easy way, however it does not utilize advantages of inheritance.
- extend SOFA CDL to support inheritance. This solution overlaps scope of this thesis and is not its objective, therefore it will not be discussed here. Extending SOFA with inheritance is described in [20].
- solve component inheritance on a *source code generation* level. This solution represents generation of SOFA components with inherited elements e.g. from a designed component with a base component a SOFA component will be generated containing also provisions, requirements and properties of the base component.

We have chosen the last solution as the most suitable one in a prototype implementation. Following section describes the chosen solution with more details.

**Inheritance of SOFA Components Processed in a Source Code Generator**

Component inheritance principles described in this section are based on [20] and simplified in some aspect.

Inheritance of SOFA components must be divided in two parts: *frame inheritance* and *architecture inheritance*. These inheritances can not be mixed so that a frame may inherit from frames and alike an architecture may inherit from architectures only.

Frame inheritance involves inheritance of properties, provisions and requirements. All provisions, requirements and properties may be inherited from a base component therefore a descendant owns all properties and provisions and requirements of its base components. In case names ambiguities of interfaces or properties, an inheritance is not allowed. Operator *rename* defined in [20] is not supported due to no support in UML 2.0.

Architecture inheritance is supported in composite architectures only. Primitive architecture specifies a class that implements its provisions, therefore there is nothing to inherit. Architecture inheritance involves inheritance of provided and required interfaces, ports, properties, subcomponents and connectors. Since UML 2.0 does not support neither inheritance of subcomponents nor connectors, neither inherited subcomponents nor connectors may be modified within a descendant component.

Behavior protocol of an inherited frame must be completely rewritten. More complex solution overlaps the scope of this work and is not its objective.

**5.1.7. Realization of Primitive Architectures**

SOFA uses *architecture* as an implementation of a *component type* defined by a frame. An architecture may implement frame in two ways: through its subcomponents in case of a composite architecture or directly with a code in case of a primitive architecture. In the later case a class implementing the architecture must be created.

A skeleton of an implementation class is generated by SOFA TIR therefore there is no need to specify implementation class in a user model.

**5.1.8. Behavior Protocol**

Behavior protocol is one of major advantages of SOFA. We have discovered two ways how to specify behavior protocol of frames and interfaces in a user model:

- using *Port State Machines* described in [13].
- define a tagged value that could be used to enter a textual form of a behavior protocol.
The first possibility overlaps the scope of this thesis, therefore we have chosen the second possibility to be implemented in a prototype implementation; the name of the tagged value is SOFABehaviorProtocol

5.2. Fractal

Fractal is a “modular and extensible component model that can be used with various programming languages” [6]. Fractal is therefore not incorporated with any concrete programming language like for example SOFA CDL. Fractal ADL is a XML based language that may be used to develop Fractal components.

Fractal ADL due to its open specification fits much easier to a component model provided by UML 2.0. Although there are some problems that had to be solved to fit Fractal ADL to UML 2.0 component model: subcomponents and shared components, properties and constants, provided and required interfaces, one kinds of connectors, component inheritance, realization of primitive components, behavior protocol

5.2.1. Components, Subcomponents and Shared Components

Fractal ADL in contrast to SOFA does not distinguish between a component type and a component implementation like frame and architecture from SOFA. Fractal ADL specifies similarly to SOFA's architecture two kinds of components:

- primitive component defining provided and required interfaces of a component and a class that implements its provisions.
- composite component which is a container for subcomponents that implement provisions specified by the owning component. Definition of a composite component contains definition of provisions and requirements, subcomponents and interconnections between owning component and subcomponents called bindings.

Primitive and composite components can be easily distinguished according to number of subcomponents. Therefore it is not necessary to distinguish between primitive and composite component explicitly using UML stereotypes. Though Fractal ADL component may be designed using an UML 2.0 metaclass Component.

Fractal ADL subcomponents may be defined in two ways:

- using an embedded definition of a subcomponent within a definition of its owning component.
- as a named link to a definition of a component defined outside the owning component.

Both possibilities have their equivalent in UML 2.0 specification. The first possibility may be designed by an embedded definition of a component, where a component is contained within another component. Mores detail are described in Section 4.2.3. The name of the embedded component which is the name of the component type may be used as subcomponent's name. This possibility is not recommended due to its less reusability, however it is supported by a prototype implementation.

For the second way of definition of subcomponents UML 2.0 provides metaclasses InstanceSpecification and Property. The usage of both metaclasses is described in Section 4.2.3. When using instances of metaclass InstanceSpecification to define Fractal's subcomponents, its meta-attribute Classifier must be set to an existing component, not a subcomponent and similarly when using instances of Property its meta-attribute Type must be set to an existing component.

Shared components in Fractal serve to model shared resources. A detailed description of designing shared components as reference parts is described in Section 4.2.3. We have decided to use the second way of defining shared components, since the use of association from the first way does not allow reference property to inherit ports nor provisions and requirements from referenced element and therefore the usage of reference property is limited since neither ports nor provisions and requirements of referenced element can be reused.
5.2.2. Attributes

The ability to specify attributes of components in UML 2.0 results from inheritance of metaclass Component from metaclass Class. Mapping UML attributes with default values to Fractal ADL attributes is rather straightforward. The only constraint is that data type of Fractal ADL attributes should conform to data types of a programming language of used fractal implementation. Currently the only implementation is written in Java, therefore the prototype implementation constrains data types of attributes to primitive Java data types.

5.2.3. Provided and Required Interfaces

Fractal ADL supports definition of provisions and requirements of components in form of interfaces with role attribute set to server for provided interfaces and client for required interfaces. Definition of both provided and required interfaces may contain:

- **name** specifying an identifier used to access the interface. The name is mandatory.
- **signature**, which is the name of the interface type.
- **contingency** indicating whether the functionality represented by this interface is guaranteed to be available or not.
- **cardinality** specifying the multiplicity of the interface.

The principals of mapping UML 2.0 provisions and requirements to Fractal ADL provisions and requirements are the same as in mapping to SOFA described in Section 5.1.4.

UML 2.0 allows two ways of specifying provided and required interfaces: direct or via ports as described in Section 4.2.1. Direct provided and required interfaces specify only types of provisions or requirements, and do not specify their names. Therefore the names of provided and required interfaces must be generated when generating Fractal ADL source code.

When defining provided and required interfaces through ports, a name of the port may be used to specify name of provisions or requirements. In Fractal ADL names of interfaces must be unique, therefore ports must have unique names and only one interface attached to a port is allowed. More interfaces of a port could modify semantics of generated sources when one connector connecting two ports with more interfaces transforms into more connectors each connecting two interfaces.

As mentioned in Section 4.1.2 meta-attribute isService of metaclass Port may be used to specify a contingency of provided or required interfaces. Its value true indicates that the interface is mandatory on the other side value false means that the interface is optional.

5.2.4. Bindings

Fractal ADL uses two kinds of bindings primitive bindings and composite bindings. A primitive binding interconnects two interfaces of two components. Interconnections may be specified either between provided and required interface of components having the same parent or between a component and its own subcomponent and vice versa.

Two UML 2.0 connectors - assembly and delegate may be used to design Fractal's primitive bindings. An assembly connector may be used to design bindings between provisions and requirements of components having the same parent and delegate connector may be used to design bindings between provisions or requirements of a component and provisions or requirements of its subcomponents.

Composite bindings are not a special type in Fractal. They are built out of primitive bindings and binding components. Binding components are components designated for communication purposes. Therefore Fractal's
composite bindings may be designed in UML 2.0 using assembly and delegate connectors in order to design primitive bindings and components in order to design binding components.

5.2.5. Component Inheritance

As described in Section 4.2.6 UML 2.0 allows component inheritance in a user model. Fractal ADL allows two kinds of component inheritance: component inheritance and subcomponent inheritance. UML 2.0 component inheritance is equivalent to the Fractal ADL component inheritance. The only constraint is, that a base component must be a Fractal component.

The subcomponent inheritance in Fractal is a little bit messy. The reason is that definition of a type of a subcomponent and a definition of a base component is mixed together. Therefore the use of subcomponent inheritance in a user model depends on the way how a subcomponent is defined. There are three ways of defining subcomponents: embedded definitions of components, as instances of components using metaclass InstanceSpecification or as parts using metaclass Property. When defining subcomponents as nested components (the first way), then subcomponent inheritance is the same as component inheritance since both participants of inheritance are components and therefore the inheritance is allowed.

When using the second and third way to define subcomponents as an instance or a part, the subcomponent inheritance is not allowed. The first reason is that UML 2.0 metaclasses InstanceSpecification and Property can not participate inheritance. The second reason is that the type of a subcomponent, specified by a meta-attribute classifier of a metaclass InstanceSpecification or by a meta-attribute type of a metaclass Property, is specified in Fractal ADL using the same XML attribute definition as specification of a base component, therefore a definition of a subcomponent's type would be mixed with a definition of base components of the subcomponent.

5.2.6. Realization of Primitive Components

UML 2.0 defines Realization connector in order to specify element that implements a component. More details about realization of components are described in Section 4.2.5.

In Fractal ADL only one implementation class is allowed. An implementation class may be specified in a user model when special characteristics like methods and inheritances need to be designed. Otherwise if an implementation class is not designed in a user model, a default skeleton may be generated along with generating Fractal ADL sources according to tagged value FractalGenerateContentClass of a designed component. It's values may be true which means that the skeleton will be generated if a component is primitive or false which means that the skeleton will not be generated.

In a prototype implementation generation of a skeleton of an implementation class has to deal with Fractal controllers. The support of Binding controller is generated since a component may be connected to other components. The support of Attribute controller is generated depending on use of attributes in a designed component. Other controllers are necessary for composite components, therefore their support is not generated in the skeleton of an implementation class since the skeleton is generated only for primitive components.

5.2.7. Behavior Protocol

Although Fractal does not support behavior protocol at the time of writing this thesis, a work to support behavior protocol in Fractal is in progress.

There are two ways as in mapping UML 2.0 to SOFA to specify behavior protocol within a user model:

- using Port State Machines, which overlaps the scope of this thesis
• using a tagged value FractalBehaviorProtocol to specify behavior protocol in a textual form.

The behavior protocol may be specified to components and also to interfaces.

The first possibility overlaps the scope of this thesis, therefore we have chosen the second possibility to be implemented in a prototype implementation.

5.3. UML Profiles

Mappings from UML 2.0 to SOFA or Fractal require refine UML semantics and create new metaclasses. As described in Section 2.2 the best way of extending UML for purpose of this work are UML Profiles. UML profile allows to create new metaclasses with new meta-attributes and constraints that specify their semantics. Constraints in UML are described using either natural language or Object Constraint Language - OCL.

Unfortunately the support of profiles in UML CASE tools is very weak. Although tools allow to define constraints using OCL, the constraints are not interpreted. Therefore profiles created in order to define constraints required by mappings from UML to SOFA and Fractal can not be used to validate user model.

The solution is to create UML profile only to define new UML metaclasses and implement constraints in a source code generator. We have chosen this way to use profiles only to create new metaclasses for SOFA and Fractal and implement constraints resulting from mappings UML to SOFA and Fractal in a prototype implementation.

Since constraints were left to be implemented in source code generator, profiles are very simple. Profile for SOFA includes:

• two new metaclasses SOFAFrame and SOFAArchitecture extending UML metaclass Component to define both SOFA components. SOFAFrame has two tagged values SOFABehaviorProtocol to define behavior protocol of a frame and SOFASubcomponentAuto to define whether a subcomponent is an auto subcomponent or not. More informations about auto subcomponents are described in [9].

• one new metaclasses SOFAInterface extending UML metaclass Interface to define SOFA interface. The new metaclass has a new tagged value SOFABehaviorProtocol to define behavior protocol of an interface.

• three new metaclasses SOFACSProcCall, SOFAEventDelivery and SOFADataStream extending Dependency metaclass in order to specify connector type. More about SOFA connector types is described in Section 3.3.1.

List of validations performed by a prototype implementation in order to validate SOFA model is described in Appendix B.

UML profile for Fractal includes:

• one new metaclasses FractalComponent extending UML metaclass Component to define Fractal component. Tagged value FractalBehaviorProtocol may be used to define behavior protocol of a frame.

• one new metaclasses FractalInterface extending UML metaclass Interface to define Fractal interface with tagged value FractalBehaviorProtocol to define behavior protocol of an interface.

List of validations performed by a prototype implementation in order to validate Fractal model is described in Appendix C.
6. UML Tools Overview

Even though that UML 2.0 specification is not finished at the time of writing this work, various CASE tools claim to support UML 2.0 specification. A complete analysis of all existing UML CASE tools overlaps the scope this thesis. Therefore six best-known CASE tools were chosen to analyze only their support of UML 2.0 components. The main reasons when choosing the CASE tools was support of modeling components using UML 2.0 constructs.

CASE tools were not analyzed completely since it is not purpose of this work. Only component design capabilities according to Section 4.2 were analyzed: provided and required interfaces attached to ports or direct provided or required interfaces, attributes and methods of components, subcomponents as part or instance and embedded component definition, connectors attached to components, interfaces of ports, realization of components, component inheritance and inheriting of ports, interfaces and port's interfaces. In addition to components design capabilities, also ability to write own plugins and import user defined profiles according to UML 2.0 profiles were researched.

Note: CASE tools are ordered alphabetically according to their names.

6.1. Borland Together Designer

The first analyzed tool is Borland Together developed by Borland Inc [5]. Current version at the time of writing this thesis is Version 2005 Build 5435.0. Borland Together Designer is written in Java, therefore it may run on almost every platform. Borland offers standalone version or either extension over various development environments including: Eclipise, NetBeans or MS Visual Studio .NET. A 15-day trial version may be downloaded from [5].

Figure 19. Borland Together Designer
**Provisions and requirements**

Provisions in Borland Together Designer are not designed according to UML 2.0 specification. A dependency realization is used instead to specify provided interface and a dependency usage is used to specify required interface. Although the design of provided and required interfaces is the same as in UML 2.0 specification, provided and required interfaces in Borland Together Designer are shared between various components in contrary to UML 2.0. Both direct interfaces and interfaces via ports may be designed and a multiplicity, type and isService meta-attribute of a port may be specified in Borland Together.

**Example:** Figure 20 shows components TransactionManager and DataStore with provided and required interfaces ITMAccess, ILoggerAccess and IDatastoreAccess; the first one is shared between both components. All shown interfaces are attached to components through ports.

![Figure 20. Borland Together Designer: Provisions and Requirements](image)

**Attributes and Methods of Components**

Neither attributes nor methods of components may be specified in Borland Together.

**Subcomponents**

Subcomponents may be designed in Borland Together as embedded definitions of components, as parts, and as instances of components. Embedded definition of a components is supported according to UML 2.0.

Parts may be used as subcomponents, however parts can not be designed in a component diagram, they must be designed in composite structures diagram. Ports specified in a component which represents a type of a part are not shown in the part and also new ports can not be attached to the part. Therefore connectors must be attached to the part, instead of ports. Multiplicity of a part and also a reference part may be specified.

**Example:** Figure 21 shows a component DatabaseEngine having two subcomponents as parts: ds with a multiplicity 0..* and tm.

Metaclass InstanceSpecification may be also used to specify subcomponents of components. Neither ports nor interfaces of a classifier are not shown. Therefore connectors must be attached to the subcomponent and can not be attached to ports.

**Example:** Figure 19 shows a component DatabaseEngine with both kinds of subcomponents: as parts (the lower components ds and tm) and as instances of components (the upper components ds and tm).
Connectors and Realization

A delegate connector is supported according to UML 2.0. However an assembly connector is not present, an association must be used instead. Both kinds of connectors may be attached to a component, its subcomponents, component's interfaces and ports.

Example: Figure 21 shows a component with two subcomponents interconnected with two delegate and one assembly connector.

![Diagram showing Connectors and Realization](image)

**Figure 21. Borland Together Designer: Subcomponents and Connectors**

A realization connector is present in Borland Together Designer, however it can not connect a class to a component, in order to design that the class is implementing the component. Only the other direction is allowed that the component implements the class. Therefore an implementation class of a component can not be designed.

Example: The class `DataStoreImpl` on Figure 19 shows the wrong direction of realizing connector.

Inheritance

Borland Together Designer enables designing an inheritance hierarchy of components. However inheritance can not be designed in a component diagram, only in a class diagram, that makes the designing uncomfortable. Neither ports nor interfaces nor interfaces of ports are inherited from a base component although it should be inherited according to UML 2.0 specification.

Plugins and profiles

Plugins may be written in Java programming language. Plugin has an access to a Borland Together functionality (e.g. generate source code) and to a loaded model. The interface to access model is readonly, therefore the model can not be modified.

Profiles may be applied on a model, however only various predefined proprietary profiles may be applied. Borland Together Designer has no option to import user defined profiles.

6.2. Enterprise Architect

Enterprise Architect is CASE tool developed by Australian company Sparx Systems [33]. The tested version is 4.51.748. A 15 days trial version may be downloaded from [33]. The supported platforms are MS Windows and Linux.
Provisions and Requirements

Enterprise Architect allows to specify direct provided and required interfaces of components and also provided and required interfaces via ports according to UML 2.0 specification. Multiplicity, type and meta-attribute isService of a port may be specified.

Example: On Figure 23 both kinds of provisions and requirements are shown. The component DataStore specifies provisions and requirements through ports whereas the component TransactionManager has direct provided and required interfaces.

Attributes and Methods of Components

Attributes and methods of a component may be specified in Enterprise Architect. Default values of attributes and constant attributes may also be defined.
Example: An example of a component with attributes and methods is shown on Figure 24. A component DataStore has two attributes connectionString and pooledConnectionsCount and two methods: saveString and readString.

![Diagram of DataStore component]

Figure 24. Enterprise Architect: Attributes and Methods of a Component

Subcomponents

Enterprise Architect supports all three types of defining subcomponents using metaclasses Property, InstanceSpecification and Component. Diagram of a part shows ports and interfaces acquired from a component which is a type of the part. Similarly diagram of an instance specification show ports and interfaces acquired from its classifier. However a small disadvantage is that neither part nor instance specification shows interfaces of acquired ports.

Example: On Figure 25 is depicted a component with four subcomponents. Two subcomponents (ds and tm) are specified as instances of metaclass InstanceSpecification and other two subcomponents (dsPart and tmPart) are specified as instances of metaclass Property.

Connectors and Realization

Both delegate and assembly connector are supported by Enterprise Architect with appearance according to UML 2.0 specification. Both connectors may be attached to a component, component's ports, subcomponent, subcomponent's ports and provided and required interfaces.

![Diagram of subcomponents and connectors]

Figure 25. Enterprise Architect: Subcomponents and Connectors

Example: Both kinds of connectors are show on Figure 25 - two assembly connectors between ports tma and four delegate connectors between ports da and ds and between ports lg.

Realization of a component may be specified with an implementation class using realization connector directing from a class to a component it is realizing.
Example: A class `DataStoreImpl` shown on Figure 26 is an example of a class implementing a component.

Inheritance

Component inheritance is fully supported in Enterprise Architect. Methods and ports may be inherited from a base component; interfaces of ports are not inherited.

Example: On Figure 26 a component `ExtendedDataStore` inherits both ports and required interface from a base component `DataStore`.

![Inheritance Diagram](image)

Figure 26. Enterprise Architect: Component's Inheritance and Realization

Plugins

Plugins may be written in any programming language that supports writing MS COM components. A designed model and all Enterprise Architect functionality may be accessed from a plugin.

User defined profiles may be written in a proprietary XML based file, imported and applied to a user model.

6.3. Magic Draw

The analyzed version is Magic Draw Enterprise Edition 9.0 [18]. The program is written in Java which enables run on various platforms and operating systems. A 6 month trial version and a free Community Edition may be downloaded from [18].

![Provisions and Requirements Diagram](image)

Figure 27. Magic Draw: Provisions and Requirements

Provisions and Requirements

Neither provided nor required interfaces are depicted according to UML 2.0. Provided interfaces are designed as `realization` dependency from components or ports to interfaces. Required interfaces are designed as `usage`
dependency directing from components or ports to interfaces. Interfaces may be viewed using a circle notation as a lollipop or using a rectangular notation both shown on Figure 27.

Provided and required interfaces in contrary to UML 2.0 specification, may be attached only to ports with a type. Ports without type can not have neither provided nor required interfaces in Magic Draw. Multiplicity, type and also value of meta-attribute isService may be specified to the port.

Figure 28. Magic Draw

Attributes and Methods of Components

Magic Draw does not support neither attributes nor methods of components.

Subcomponents

Subcomponents as parts are not supported in Magic Draw. The other two ways of specifying subcomponents using metaclasses InstanceSpecification and Component are allowed. Interfaces and ports from a component which is a classifier of an instance specification are not shown, therefore connectors may be attached only to a subcomponent. Also an embedded definition of a component as a subcomponent is allowed.

Figure 29. Magic Draw: Subcomponents and Connectors
Example: Both kinds of subcomponents are shown on Figure 29. Subcomponent MySubComponent of component DatabaseEngine is an example of an embedded definition of a component. Subcomponents ds and tm are examples of subcomponents defined as instances of components DataStore and TransactionManager.

Connectors and Realization

Both delegation and assembly connector are present in Magic Draw. An appearance of both connectors is slightly different than specified in UML 2.0 specification. Both connectors may be attached to a component, component's ports, a subcomponent and subcomponent's ports. In contrary to UML 2.0, connectors can not be attached to an interface.

Example: Figure 29 shows three delegate connectors between component DatabaseEngine and subcomponents MySubComponent, ds and tm and one assembly connector between subcomponents ds and tm.

A realization of a component may be designed using implementation class attached to the component via realization connector.

Example: On Figure 30 a class DataStoreImpl realizes component DataStore.

![Diagram](image)

Figure 30. Magic Draw: Component inheritance and realization of a component example

Inheritance

Magic Draw enables design of a component inheritance hierarchy. Neither ports nor interfaces nor interfaces of ports are inherited.

Example: A component inheritance example is shown on Figure 30. Component ExtendedDataStore is a descendant of component DataStore.

Plugins and Profiles

Plugins to Magic Draw may be written in Java. A designed model may be accessed from a plugin. Magic Draw uses an internal representation of a designed model based on UML 1.4. In addition to plugins also scripts may be written in JPython language.

New profiles may be written, exported, imported and applied to a user model. Existing UML elements may be extended with a new appearance, constraints and tagged values.
6.4. Poseidon for UML

Poseidon for UML developed by Gentleware AG [10] is UML CASE tool based on open source project ArgoUML written in Java [1]. The tested version is Poseidon for UML Community Edition 3.0.1 which may be freely downloaded from [10]. Also 15 days trial version of other editions may be downloaded from a Gentleware home page [10].

Figure 31. Poseidon for UML

Provisions and Requirements

Provided and required interfaces of components are designed according to UML 2.0 specification, in case of provided interfaces as a lollipop in case of required interfaces as a slot. However interfaces are shared between components which is in contrast to UML 2.0 specification. A rectangular notation may be also used for interfaces.

Example: On Figure 32 both circular (interfaces IDataStoreAccess and ILoggerAccess) and rectangular (interface ITMAccess) notations are shown.

Provided and required interfaces of ports may be also specified, however existing interfaces can not be used to specify neither provided nor required interfaces of ports. A new interface is always created along with creating new provided or required interface of a port. Multiplicity, type and meta-attribute isService of ports may be specified.
Attributes and Methods of Components

Both attributes and methods of a component may be specified, however none of them is shown in a diagram of a component. The default value of an attribute may be also specified.

Subcomponents

Poseidon does not support parts at all. Therefore only metaclasses Component and InstanceSpecification may be used to design subcomponents of components. An embedded definition of a subcomponent using metaclass Component is supported according to UML 2.0 specification. Use of InstanceSpecification is limited since neither ports nor interfaces of an instance specification's classifier are shown in a diagram of the instance specification.

Connectors and Realization

Connectors are not explicitly separated into a delegate connector and an assembly connector. Only a dependency with suitable stereotype may be used. All connectors may be attached to a component, component's ports and interfaces, a subcomponent and its ports and interfaces.

Example: Figure 33 shows a component with two subcomponents interconnected with two delegate and one assembly connector. Both connector kinds are denoted with corresponding stereotype.

An implementation class may be specified using a realization connector targeting from a class to a component.

Inheritance

Inheritance of components is not supported in Poseidon for UML.
Plugins and Profiles

Poseidon for UML supports user defined plugins written in Java. Plugins may access a user model and a Poseidon functionality e.g. source code generation.

User defined profiles are not supported.

6.5. Rational Rose XDE for Visual Studio .NET

Rational Rose is a famous CASE tool originally developed by Rational Software. Its three developers Grady Booch, Ivar Jacobson, and Jim Rumbaugh known as “three amigos” have create the very first version of UML standard. At the time of writing this thesis Rational Rose is owned by IBM [11]. IBM offers it as a standalone application and also as an extension to various development environments like Eclipse and MS Visual Studio. The tested version is Rational Rose XDE 2003.06.12. A 15 days trial version as a plugin for Microsoft Visual Studio .NET and may be downloaded from [11].

Figure 34. Rational Rose

Provisions and requirements

Rational Rose does not support ports therefore only direct interfaces of components may be designed. Provided and required interfaces of components are not designed according to UML 2.0 specifications, only as a dependency between component and interfaces. Concretely a provided interface may be designed as a realization dependency and a required interface as an usage dependency between component and interface.

Example: Figure 35 shows realization dependency between component DataStore and interface IDataAdapterAccess and between TransactionManager and ITMAccess in order to specify provided interfaces of components. Usage dependency is also shown between component DataStore and interface ITMAccess and between component TransactionManager and interface ILoggerAccess.
Attributes and Methods of Components

In the tested version of Rational Rose only attributes of components may be specified. Default value and also constant attributes are supported. Methods of components are not supported.

Example: Figure 36 shows a component DataStore with one attribute sourceString.

Subcomponents

Parts as subcomponents are not supported in Rational Rose at all. Metaclass InstanceSpecification is supported, however it is not allowed to use a component as a classifier of an instance specification. The third way of specifying subcomponents, using embedded definition of a component is fully supported to arbitrary depth.

Connectors and Realization

Rational Rose does not support neither delegate nor assembly connector, only general association may be used to interconnect components and subcomponents.

A realization of a component may be specified using a realization connector with a class on a client side and a component on a supplier side.

Example: Figure 36 shows a class DataStoreImpl that realizes a component DataStore.

Inheritance

Inheritance of components may be designed in Rational Rose. However neither ports nor interfaces are inherited from a base components.
Example: Component inheritance is shown on Figure 36. A component ExtendedDataStore inherits from a component DataStore.

Plugins and profiles

Plugins to Rational Rose may be written as ActiveX components. Plugin are allowed to access and modify edited model.
New UML profiles may be created. To create a new profile a new plugin which implements the profile's functionality must be created.

6.6. Visual Paradigm for UML

The tested version is Visual Paradigm for UML Enterprise Edition 4.1 [39]. The CASE tool is written in Java. A 30 days evaluation version may be downloaded from [39].

Figure 37. Visual Paradigm

Provisions and Requirements

Direct provided and required interfaces of components may be designed, however the manner differs from UML 2.0 specification. They may be designed only as dependency in both cases - the provided and required interfaces.

Figure 38. Visual Paradigm: Provisions and Requirements
**Example:** Figure 38 show definition of provided and required interfaces of components in Visual Paradigm. Provisions and requirements of components through ports are not possible to design because ports may be attached only to classes from composite structures and not to components.

**Attributes and Methods of Components**

Neither attributes nor methods of components are supported in Visual Paradigm for UML.

**Subcomponents**

Subcomponents may be designed in Visual Paradigm for UML only as instances of components using metaclass InstanceSpecification or as embedded definitions of subcomponents using metaclass Component. Use of InstanceSpecification is limited since neither name of a classifier nor ports nor interfaces from a classifier are shown. Parts can not be used to design subcomponents of components.

**Example:** Two subcomponents ds and tm as instance specifications are show on Figure 39.

![Subcomponents and Connectors](image)

**Figure 39. Visual Paradigm: Subcomponents and Connectors**

**Connectors and Realization**

Both kinds of connectors - delegate and assembly are supported. However its appearance differs from UML 2.0 specification. There are also some problems with a delegate connector. The delegate connector directing from a component to its subcomponent may be designed but it does not appear on a diagram. The delegate connector directing from a subcomponent to its parent component can not be even designed.

**Example:** An assembly connector is shown on Figure 39 between components ds and tm.

A realization connector is supported, however a class realizing a component can not be specified because class can not be designed in a component diagram and a component can not be designed in a class diagram which means that a class and a component can not be designed in the same diagram and therefore interconnected with a realization connector.

**Inheritance**

Inheritance of components is not supported in Visual Paradigm for UML.

**Plugins and Profiles**

User plugins that extend functionality of Visual Paradigm may be written in Java. User model may be accessed from the plugin.

Visual Paradigm for UML documentation does not provide information about user defined profiles.
6.7. Conclusion

Since UML 2.0 specification is not released at the time of writing this thesis the support of constructs for modeling components is rather weak in tested tools. The only CASE tool supporting vast majority of UML 2.0 component modeling constructs is Enterprise Architect (Section 6.2). The most common problems in other tools are: weak support for provided and required interfaces, ports and component inheritance. None of tested tools, except for Enterprise Architect, supports inheritance of ports, required and provided interfaces.

Due to described errors and problems with support of UML 2.0 Component Model in tested CASE tools, we have chosen to implement prototype as a plugin to Enterprise Architect.
7. Prototype Implementation

This section provides a description of a prototype implementation which implements mappings of UML 2.0 component model to SOFA and Fractal component models as described in Section 5. Section 7.1 and Section 7.2 describe requirements and installation of the prototype. Section 7.3 provides user's guide and Section 7.4 provides programmer's guide. Evaluation and description of examples is provided in Section 7.5.

7.1. Requirements

Microsoft Windows 2000 or XP and a .NET Framework version 1.0 or 1.1 [15], [17] are required to be installed. The prototype may run as a standalone application or as a plugin in Enterprise Architect version 4.5. To run as a plugin an Enterprise Architect version 4.5 is required to be installed.

7.2. Installation Guide

To install the prototype, follow the instructions bellow:

1. Install the Enterprise Architect 4.5
2. Install Microsoft .NET Framework version 1.0 or 1.1
3. Unpack ZIP archive to a target directory (typically: “C:\Program Files\EAAdlPlugin 1.0\”)
4. Run install.bat from the installation directory

To uninstall prototype, follow the instructions bellow:

1. Run uninstall.bat located in the prototype's installation directory
2. Remove prototype's installation directory with all it's content

7.3. User's Guide

The prototype may run as a standalone application or as an Enterprise Architect plugin. The difference is that the standalone application uses XML file whereas plugin uses an Enterprise Architect as a data source.

7.3.1. Plugin Menu

Plugin menu is created as a submenu with name EAAAdlPlugin 1.0 of menu item Addins in Enterprise Architect. It has three items:

- Generate Fractal sources to generate Fractal source from currently open model.
- Generate SOFA sources to generate SOFA source from currently open model.
- Generate XML to generate a XML file from currently open model that may be loaded from a prototype when running as standalone application.

Figure 40. Plugin Menu
7.3.2. Main Window

The main window (shown on Figure 41) of the prototype shows tree of a loaded model, properties of a selected element, log box and structured messages. The xml file name may be entered only when using the prototype as a standalone application. Accepted are XML files generated from plugin menu Generate XML. Output file name in case of SOFA or output directory in case of Fractal, specifies the file name or a directory where source codes will be generated.

Model tree shows loaded elements with their sub elements. When selecting any element in a model tree, element's properties are shown in a properties section. Properties of a selected element are divided in sections according to a UML metaclass to which a property belongs.

Errors and notices from loading, validating model and generating source code are shown in log box at a bottom of a window and in Structured messages box.

Buttons >> and << serve to navigate through history of selection in loaded model tree.

![Figure 41. Main Window](image)

UML 2.0 Components
7.4. Programmer's Guide

The prototype implementation is written in C# using a .NET framework and implemented in Microsoft Visual Studio .NET 2002 [16].

7.4.1. Plugin Principles in Enterprise Architect

Enterprise Architect provides a support for other applications to access its functionality and designed model using Windows OLE Automation. Plugin, also called add-in, must be a COM component that exposes defined interface, used by Enterprise Architect to integrate plugin and invoke its functionalities. A plugin may be written in any language that supports creating COM components.

7.4.2. Prototype Component Architecture

From logical view, the prototype implementation is composed of various components with well defined provided and required interfaces in order to achieve the prototype's modularity. The basic separation of the prototype's functional part is into three components Loader, ModelValidator and SourceGenerator. This section provides description of functional part of the prototype implementation that covers loading and validating model and generating source code. Other components together with complete UML component diagrams of the prototype are described in Appendix A.

An implementation of the prototype's component model does not use any component framework. It is based on class hierarchy instead. Classes are used instead of components, provided interfaces of classes are interfaces which are specified as interfaces that the class implements. Required interfaces are specified as class members. Interconnections between components are not created dynamically, must be hard coded within the source code.

Loader

Loader is an abstract class which descendants should implement interface ILoaderAccess. Its purpose is to load model from any data source and create an internal representation of loaded model. The class LoaderViaCache loads data from an external data source to a cache and then from elements in cache creates UML model.

From a logical view a LoaderViaCache is a component composed of three components: LoaderToCache, CacheCreator and LoaderFromCache. The separation is important due to reusability of a code. LoaderToCache has for example two implementations to load data from XML file or directly from Enterprise Architect. Replacement of loader from XML for a loader from Enterprise Architect does not require to modify other components.

Following sections describe all three subcomponents of LoaderViaCache.

LoaderToCache

An abstract class LoaderToCache with one required interface ICacheWriteAccess is connected to CacheCreator which has a provided interface ICacheWriteAccess. Its purpose is to load data from a data source (e.g. XML file, Enterprise Architect) to an internal cache. The prototype implementation includes two implementations of LoaderToCache: EALoaderToCache to load data directly from Enterprise Architect using Windows OLE Automation and XmlLoaderToCache to load data from a XML file.
CacheCreator

CacheCreator is used to filter loaded elements to load only elements that will be processed by a validator and a source code generator. The prototype implementation provides two implementations of CacheCreator: SOFACacheCreator and FractalCacheCreator.

SOFACacheCreator loads following elements: classes, interfaces, operations, attributes, connectors, packages, ports, properties and components with stereotypes SOFAFrame and SOFAArchitecture. Other elements are ignored, therefore not loaded.

FractalCacheCreator loads following elements: classes, interfaces, operations, attributes, connectors, packages, ports, properties and components with stereotype FractalComponent. Other elements are ignored.

LoaderFromCache

LoaderFromCache creates an internal representation of a model from elements in cache. The model is based on simplified UML metaclass hierarchy described in Section 7.4.3.

ModelValidator

ModelValidator serves to validate loaded model with regard to constraints resulting from a chosen component model. The prototype implementation contains two implementations of ModelValidator: SOFAValidator and FractalValidator.

Validations of elements from a loaded model are performed per element type, which means that there is a set of constraints for element types that all elements of that type must fulfill. Element type means an UML metaclass from which an element is instance. Validations are specified only for UML metaclasses that are not abstract and therefore are instantiable. Validations of abstract metaclasses make part of validations of their descendants which are not abstract.

Validations are hardcoded in a plugin as a set of "validate..." (e.g. validateComponent, validateInterface) methods of classes SOFAValidator and FractalValidator

SOFAValidator

SOFAValidator validates model with regard to SOFA language as described in Section 5.1. List of validations performed by a prototype implementation is described in Appendix B.

FractalValidator

SOFAValidator validates model with regard to SOFA language as described in Section 5.2. List of validations performed by a prototype implementation is described in Appendix C.

SourceGenerator

SourceGenerator serves for two purposes: as an abstraction of classes SOFASourceGenerator and FractalSourceGenerator and to provide basic functionalities to enable use of templates to generate source code.

SOFASourceGenerator and FractalSourceGenerator provide generation of source code with regard to SOFA or Fractal ADL language. Both use templates to specify the formatting of a result source code.
Templates

Templates of source code generator are XML based files used to precisely specify formatting of the result source code. The most important XML elements of templates are: template and list

XML element template defines a named piece of a code, its structure and formatting. A template may contain other templates, a source code, parameters and lists. The source code is written to an output file with various modifications:

- end of lines are replaced with a single space
- all spaces and tabulators are replaced with one space
- escape sequences \\
  and \\	 are replaced with a new line, space and tabulator
- \i is replaced with an indent character specified by an XML element indent.
- parameter specifications are replaced by the parameter's values. The parameter specification has a form:
  \[\#\text{PARAM\_NAME}\#\] where a PARAM\_NAME is a name of a parameter.

Parameters of templates must be specified using XML tag param. The tag has two mandatory attributes: name specifying the name of parameter and type specifying the type of parameter. Only two types of parameters are allowed: string and bool. For parameters with type string next two attributes may be specified: format used to specify formatting of a value of the parameter and formatEmpty used to specify value that is used when value of parameter is an empty string. Within value of attribute format a sequence \{0\} may be used to specify value of the parameter. If no format is specified then only not formatted value of parameter will be used. An empty value of attribute format is equivalent to value "\(\{0\}\)".

For a parameter with type bool more three attributes may be specified: defaultValue specifying the default value of the parameter. Its values may be true or false. The next two attributes are valueTrue specifying the value that will be used when the value of parameter is true and valueFalse specifying the value that will be used when the value of parameter is false. When attributes valueTrue or valueFalse are omitted then the text true is used in case of true value of parameter and text false is used in other case.

XML element list may be used to specify a set of templates that are used by a prototype to create an ordered list. In addition to XML element template that may be contained in a list, next three XML elements may be specified in order to customize formatting of the generated source code of the list. XML elements header and footer specify code that is inserted in front and at the end of the list. XML element itemSeparator specifies code that is inserted between elements; it is not inserted before the first element and after the last element.

When generating a source code of an element from a loaded model, parameters of templates or lists which represent an element type or its parts are filled with values from processed element and then a source code from templates with specifications of parameters replaced with values of parameters is written to an file. A suitable template or list to an element type of processed element is found by the name. If more XML elements list with the same name are found, all of them are processed and filled with the same data. In case of nested templates, the nested templates are processed as first and generated source code becomes a part of their owners.

Detailed structure of templates used by the prototype implementation is described in Appendix D.

Example: Figure 42 shows a simple template of a class named class with an empty body and with a list of base classes with a name inheritanceSpec.

```xml
<template>
    <indent string="\t" />
    <!--
    ******************************************************************************
    Class
```
**Figure 42. Template example**

**Generated Source Code**

From loaded SOFA model the prototype implementation generates cdl file containing descriptions of frames and architectures. Skeletons of implementation classes are not generated since SOFA TIR may be used to generate the skeletons.

From loaded Fractal model the prototype implementation generates:

- Fractal ADL files containing definitions of components.
- interfaces defined in model.
- interfaces which serve as attribute controllers.
- skeletons of implementation classes.

**7.4.3. Simplified UML MetaModel**

The prototype implementation uses a simplified UML 2.0 metamodel for an internal representation of a processed model. A simplified metamodel is a subset of UML 2.0 metaclasses that are involved in UML 2.0 component model. The used metaclasses are: Attribute, Class, Classifier, Component, Connector, InstanceSpecification, Interface, Element, Operation, Package, PackageableElement, Port and Property. Figure 43 shows a class hierarchy of implemented metaclasses.
7.4.4. Installation details

The prototype implementation may run as a standalone application or as a plugin in Enterprise Architect 4.5. An installation is necessary only in the later case. An installation consists of two steps:

1. Registering the prototype as an COM object. Since the prototype is written as a .NET application which is not compatible with COM standard, a standard COM registration technique cannot be used. However .NET framework provides a support to run .NET components as COM components. It is based on a special COM component that transforms COM style of invocation of methods to .NET style of invocation of methods and vice versa. To register .NET components as COM components, .NET Framework provides a method RegisterAssembly from a class System.Runtime.InteropServices.RegistrationServices [17].

2. Registering the prototype as a plugin in Enterprise Architect. A list of installed plugins in Enterprise Architect is stored in Windows Registry in a key: HKEY_CURRENT_USER\Software\Sparx\Systems\EAAAddins as described in [34]. In order to install a new plugin a sub key with a project name must be created in the EA plugins key and a default value set to a class name.

7.5. Evaluation and Examples

In order to examine mappings defined in Section 5 the prototype implementation was used to generate source code for various designed models

- one Fractal ADL example - Helloworld which is a part of current Fractal ADL distribution version 2.1.3.
- four SOFA examples (cplayer, logdemo, pdemo and protodemo) that make part of SOFA distribution released on March 30th 2004.
- variations of a component model of the prototype implementation described in Appendix A. Since the
prototype implementation supports only Fractal and SOFA component models, the model described in Appendix A must be modified to be SOFA or Fractal compliant model. Both models are part of CD that makes an electronic part of this work in directories [CD]/examples/sofa/plugin for SOFA version and [CD]/examples/fractal/plugin for Fractal version.

Since the prototype is implemented in C# and current implementations of SOFA and Fractal support only Java programming language, the primitive components are not implemented because it would mean to rewrite the whole plugin to Java programming language.

All examples and also tested distributions of Fractal and SOFA may be found in directories [CD]/examples/, [CD]/fractal/ and [CD]/sofa/ on CD which is an electronic part of this work. More information about CD is provided in file [CD]/readme.txt located on the CD.

Source code generated by the prototype implementation may not be compiled and executed without modifications in general since designed UML model does not contain enough information to create runnable source code an therefore the source code must be edited by user in order to run, e.g. implement empty methods of generated skeletons of implementation classes.

Generated source code for SOFA contain descriptions of frames and architectures and may be compiled and stored to TIR. In order to run components, implementation classes of primitive components must be created and installed to TIR. Skeletons of implementation classes may be generated by TIR.

Source code generated for Fractal contain definitions of components, skeletons of implementation classes of primitive components and attribute controllers for attributes of components. The skeletons must be edited and user code filled in empty methods which implement component's provisions. CD which makes an electronic part of this work contains in directory [CD]/examples/fractal/helloworld/src/ generated runnable source code for Fractal with implemented implementation classes.
8. Related Work and Future Path

Because the previous version of UML - 1.5 does not support modeling component architectures and components are used only for design of deployment diagrams and because a final version of UML 2.0 specification was released at the time of writing this thesis there is not a lot of publications comprising both areas described in this work.

The Object Management Group has released a Request For Proposal - RFP that solicits proposal for Corba or Corba CM profile for UML 2.0 [22]. The task of RFP relates to a part of this thesis, where mapping UML 2.0 to SOFA and Fractal is defined.

Visual development is the main purpose of [8]. In contrary to this work which analyzes modeling concepts defined in UML 2.0 the mentioned work specializes in common concepts of visual development environment.

Future Path

Project Evolution and Validation Environment - EVE [37], [36] represents a new view on modeling software. The main idea is to create a central storage for models that enables re-use of designed models in the same manner as re-use of software components. The central storage is intended to provides various services e.g. validations of models. Validations of SOFA and Fractal models that are performed in the prototype implementation might be added to EVE in order to enhance its support of SOFA and Fractal.

XML Metadata Interchange - XMI is a standard for exchanging UML models. A support of XMI may be added to the prototype. The main advantage of the usage of XMI is no need to reimplement prototype as a plugin to other CASE tools. Since the prototype may run as a standalone application a support to load XMI would enhance the prototype to cooperate with every CASE tool that supports XMI export.

Port state machines [13] may be used to design behavior protocol of interfaces or components. Since support of port state machines overlaps scope of this thesis, the support was not implemented in the prototype. Adding support of port state machines to the prototype implementation could simplify creation of behavior protocol of components or interfaces.

SOFA TIR serves to store definitions of SOFA component types. Current implementation of SOFA exposes its services through Java RMI. Current implementation of the prototype generates CDL files that are parsed by SOFA CDL compiler and definitions of components are stored in SOFA TIR. Enhancing the prototype implementation to communicate with TIR through RMI may be a subject for future work of this thesis.

Another field of enhancing this thesis is to define constraints required by mappings from UML to SOFA and Fractal in OCL. Profiles described in Section 5.3 do not contain constraints in OCL because none of tested UML CASE tools supports OCL. Constraints written in OCL will be useful in a CASE tool supporting OCL.

Reverse engineering of SOFA and Fractal source might be added to the prototype implementation in order to support creating diagrams from SOFA or Fractal source code. In case of SOFA the process of reverse engineering might have two entry points: from CDL files or from SOFA TIR where definitions of components are stored.
9. Conclusion

The thesis provides an analysis of UML 2.0 component model addressing the first goal outlined in Section 1.3. The analysis is provided on an abstract level with no focus on concrete component model characteristics and based on abstractions known from both research and industrial component models. The analysis comprises descriptions of UML 2.0 constructs to design software components and provides various ways of use of UML 2.0 component constructs.

Mappings of various component concepts from UML 2.0 component model to SOFA and Fractal component models are provided in this thesis in order to demonstrate UML 2.0 component model capabilities and address the second goal of this thesis. Various possibilities of refining UML 2.0 semantics for the purpose of mappings needed to be analyzed. The major problems that have arisen when defining mapping of UML 2.0 to SOFA were the two kinds of components - Frame and Architecture, named provisions and requirements, component inheritance that is not supported in SOFA and behavior protocol. The major problems within mapping UML 2.0 to Fractal ADL were named provisions and requirements, shared subcomponents and behavior protocol. The best ways of mappings that we have chosen were implemented in the prototype implementation.

UML profiles have been created for both SOFA and Fractal in order to refine UML 2.0 semantics. Unfortunately the support of profiles in UML CASE tools is very weak and tools do not allow to define constraints required by refinement of UML 2.0 semantics. Therefore the created profiles are used only to define new kinds of UML metaclasses with new attributes and the constraints are implemented as a part of the prototype.

As proposed in the goals if this thesis, the prototype had to be implemented as a plugin for an existing UML CASE tool. During the initial studies, the need to provide more complex research of various UML CASE tools has arisen. The reason is that support of UML 2.0 components in UML CASE tools is rather weak and not supported very well. Therefore various tools had to be analyzed in order to find the tool with satisfactory support of UML 2.0 components. The result of analysis of chosen six CASE tools is a tool with a best support of UML 2.0 component model and a review of UML 2.0 component model support in the chosen tools.
References

[27] OMG: UML 2.0 OCL Specification. ptc/03-10-14, 2004


[41] World Wide Web Consortium: [http://www.w3.org](http://www.w3.org), April 2005
Appendix A. Example

As described in Section 7.4.2 the prototype implementation is composed from various components. This appendix provides UML component diagrams describing components used to build prototype implementation.

Diagram on Figure A.1 describes two components Plugin and Appl that represent two different entry points of an application: as a plugin from Enterprise Architect or as a standalone application. Both components call MainWindow component. MainWindow provides one interface IMainWindow that enables components Plugin and Appl to run MainWindow component.

![Figure A.1. Two Entry Points of Prototype Implementation](image)

The next diagram shown on Figure A.2 describes the MainWindow component interconnected with three other components Loader, ModelValidator and SourceGenerator. MainWindow calls services of all three components in sequence. If one component fails, e.g. when model is not valid and ModelValidator returns false, services of the next component are not invoked.

![Figure A.2. The Main Functionality of Prototype Implementation](image)

Figure A.3 shows component LoaderViaCache which is a descendant of component Loader. Also internals of component LoaderViaCache are shown. More detail description of both components is provided in Section 7.4.2.
Figure A.3. A Component Diagram of Components Designated to Load Model

The last two diagrams on Figure A.4 and Figure A.5 show components Validator and SourceGenerator with their descendants SofaValidator, FractalValidator, SofaSourceGenerator and FractalSourceGenerator that provide validation and generation of source code dependent on chosen language. More detailed description of components is provided in Section 7.4.2

Figure A.4. A Component Diagram of Components Designated to Validate Model
Figure A.5. A Component Diagram of Components Designated to Generate Source Code
# Appendix B. SOFA Validations

This appendix describes set of validations performed by a prototype implementation in order to validate loaded SOFA model.

The first column of Table B.1 - Validation describes validation. The next three columns Type, Number and Message describe error, warning and notice messages reported by a prototype implementation. The list of messages is not complete, since the table contains only messages reported from model validator. Messages resulting from loading model or generating source code are not described here.

<table>
<thead>
<tr>
<th>Validation</th>
<th>Type</th>
<th>Number</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly Connector</td>
<td>Error</td>
<td>167</td>
<td>A client end of an assembly connector must be an instance of a component or a property.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>168</td>
<td>A supplier end of an assembly connector must be an instance of a component or a property.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>169</td>
<td>Supplier and client of an assembly connector must have the same parent component.</td>
</tr>
<tr>
<td>Assembly connector must interconnect subcomponents that have the same parent.</td>
<td>Error</td>
<td>170</td>
<td>Components connected with an Assembly connector must have the opposite interface(s).</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>172</td>
<td>Components on a supplier end of an Assembly connector must have opposite interface as a port connected on a client end.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>173</td>
<td>Ports on client and supplier end of an Assembly connector must have opposite interface.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>171</td>
<td>Components on a client end of an Assembly connector must have opposite interface as a port connected on a supplier end.</td>
</tr>
<tr>
<td>Attribute</td>
<td>Error</td>
<td>112</td>
<td>An attribute must have a name.</td>
</tr>
<tr>
<td></td>
<td>Notice</td>
<td>113</td>
<td>Attribute type converted: int/integer-&gt;long.</td>
</tr>
<tr>
<td></td>
<td>Warning</td>
<td>114</td>
<td>Invalid attribute data type.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>115</td>
<td>Invalid classifier of an Attribute. Only a Class and Instance are allowed.</td>
</tr>
<tr>
<td></td>
<td>Notice</td>
<td>116</td>
<td>Default value should be specified for a constant attribute.</td>
</tr>
<tr>
<td></td>
<td>Notice</td>
<td>117</td>
<td>Default value of attribute is not allowed in Sofa. Value will be ignored.</td>
</tr>
<tr>
<td>Architecture</td>
<td>Error</td>
<td>134</td>
<td>An Architecture must have a name.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>135</td>
<td>Only a Package can be a parent of an Architecture.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>136</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>142</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>137</td>
<td>A base of an Architecture must be another Architecture.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>143</td>
<td>Only an Architecture can inherit from an Architecture.</td>
</tr>
<tr>
<td>Validation</td>
<td>Type</td>
<td>Number</td>
<td>Message</td>
</tr>
<tr>
<td>------------</td>
<td>------------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>Architecture must implement exactly one frame.</td>
<td>Error</td>
<td>138</td>
<td>A realized element of an Architecture must be a Frame.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>140</td>
<td>An Architecture must implement a Frame.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>141</td>
<td>An Architecture can't implement more then one Frame.</td>
</tr>
<tr>
<td>Only assembly, delegate generalization and realization dependency may be attached to an architecture.</td>
<td>Error</td>
<td>139</td>
<td>A connector can't be connected to an Architecture on a client side.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>146</td>
<td>A connector can't be connected to an Architecture on a supplier side.</td>
</tr>
<tr>
<td>Only a class may realize an architecture. However skeleton of implementation class is generated by TIR therefore implementation class is ignored.</td>
<td>Error</td>
<td>144</td>
<td>An Architecture can be realized only by a Class.</td>
</tr>
<tr>
<td>Notice</td>
<td>145</td>
<td>Implementation class will be ignored. Implementation classes are generated by Sofa's TIR.</td>
<td></td>
</tr>
<tr>
<td>Methods of architectures are not allowed.</td>
<td>Warning</td>
<td>147</td>
<td>An Architecture can't have any methods. All methods will be ignored.</td>
</tr>
<tr>
<td>Only an instance specification and a property may be subcomponents of an architecture.</td>
<td>Error</td>
<td>148</td>
<td>Element can't be a subcomponent.</td>
</tr>
<tr>
<td>Ports of designed frame and its realizing architecture must be the same.</td>
<td>Error</td>
<td>149</td>
<td>An architecture must have the same ports as a Frame that is realizing.</td>
</tr>
<tr>
<td>Assembly connectors must connect required to provided interface.</td>
<td>Error</td>
<td>150</td>
<td>An assembly must have required interface on a client side.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>151</td>
<td>An assembly must have provided interface on a supplier side.</td>
</tr>
<tr>
<td>Delegate connectors must connect provided to provided interface.</td>
<td>Error</td>
<td>152</td>
<td>A delegate must have provided interface on a client side.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>153</td>
<td>A delegate must have provided interface on a supplier side.</td>
</tr>
<tr>
<td>Subsume connectors must connect required to required interface.</td>
<td>Error</td>
<td>154</td>
<td>A subsume must have required interface on a client side.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>155</td>
<td>A subsume must have required interface on a supplier side.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class</th>
<th>Type</th>
<th>Number</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>A class must have a name.</td>
<td>Error</td>
<td>228</td>
<td>A Class must have a name.</td>
</tr>
<tr>
<td>A class may be contained only by a package.</td>
<td>Error</td>
<td>229</td>
<td>Only a Package can be a parent of a Class.</td>
</tr>
<tr>
<td>Both client and supplier end of connectors must be specified.</td>
<td>Error</td>
<td>230</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>235</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>Class inheritance is allowed only between classes.</td>
<td>Error</td>
<td>231</td>
<td>A base of a Class must be another Class.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>236</td>
<td>Only a Class can inherit from a Class.</td>
</tr>
<tr>
<td>Only a class may realize an architecture and an architecture may be realized only by a class.</td>
<td>Error</td>
<td>232</td>
<td>A Class can realize only an Architecture.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>233</td>
<td>A Class can't realize the Element.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>237</td>
<td>A Class can't be realized.</td>
</tr>
<tr>
<td>Only realization and generalization dependency may be connected to a class.</td>
<td>Error</td>
<td>234</td>
<td>A connector can't be connected to a Class on a client side.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>238</td>
<td>A connector can't be connected to a Class on a supplier side.</td>
</tr>
<tr>
<td>Validation</td>
<td>Type</td>
<td>Number</td>
<td>Message</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
<td>--------</td>
<td>---------</td>
</tr>
<tr>
<td>A class can not have neither subelements nor ports.</td>
<td>Error</td>
<td>239</td>
<td>A class can’t have any elements.</td>
</tr>
<tr>
<td>A class can't have any elements.</td>
<td>Error</td>
<td>240</td>
<td>A class can’t have any ports.</td>
</tr>
</tbody>
</table>

**Delegate Connector**

| A delegate connector must be attached to instance specification or component or property. | Error | 156 | A supplier end of a delegate connector must be an instance of a component or a property. |
| A delegate connector must connect component to its subcomponent. | Error | 157 | A client end of a delegate connector must be a component, that owns a connector. |
| A subsume connector must be attached to instance specification or component or property. | Error | 158 | A supplier end of a delegate connector must be a subcomponent, of a component that owns a connector. |
| A subsume connector must connect subcomponent to its owner. | Error | 159 | A client end of a subsume connector must be an instance of a component or a property. |
| A subsume connector must connect subcomponent to its owner. | Error | 160 | A supplier end of a subsume connector must be a component, that owns a connector. |
| A delegate/subsume connector must interconnect component and its subcomponent with the same provisions/requirements. | Error | 161 | A client end of a subsume connector must be a subcomponent, of a component that owns a connector. |
| A delegate/subsume connector must interconnect component and its subcomponent with the same provisions/requirements. | Error | 162 | A Delegate/Subsume connector must be between a component and a subcomponent. |
| A delegate/subsume connector must interconnect component and its subcomponent with the same provisions/requirements. | Error | 163 | Components connected with a Delegate/Subsume connector must have the same interface(s). |
| A delegate/subsume connector must interconnect component and its subcomponent with the same provisions/requirements. | Error | 164 | Components on a client end of a Delegate/Subsume connector must have the same interface as a port connected on a supplier end. |
| A delegate/subsume connector must interconnect component and its subcomponent with the same provisions/requirements. | Error | 165 | Components on a supplier end of a Delegate/Subsume connector must have the same interface as a port connected on a client end. |
| A delegate/subsume connector must interconnect component and its subcomponent with the same provisions/requirements. | Error | 166 | Ports on client and supplier end of a Delegate/Subsume connector must have the same interface. |

**Frame**

| A frame must have a name. | Error | 121 | A Frame must have a name. |
| A frame may be contained only by a package. | Error | 122 | Only a Package can be a parent of a Frame. |
| Invalid connector attached to a frame. Both client and supplier end must be specified. | Error | 123 | Invalid connector. |
| A frame inheritance is allowed only between frames. | Error | 124 | A base of a Frame must be another Frame. |
| A frame can not realize other elements. Only an architecture may realize frame. | Error | 125 | A Frame can’t be on a client end of a realization. |
| A frame can not realize other elements. Only an architecture may realize frame. | Error | 128 | Only a Frame can inherit from a Frame. |
| Only generalization and realization dependency may be attached to a frame. | Error | 129 | A realization of a Frame must be an Architecture. |
| Only generalization and realization dependency may be attached to a frame. | Error | 126 | A connector can’t be connected to a Frame on a client side. |
| Only generalization and realization dependency may be attached to a frame. | Error | 130 | A connector can’t be connected to a Frame on a supplier side. |
| Methods of frames are not allowed. | Error | 131 | A Frame can’t have any methods. |
| A frame can not have an internal structure. | Error | 132 | A Frame can’t have any subcomponents. |
| A frame can not have an internal structure. | Error | 133 | A Frame can’t have any bindings. |

**InstanceSpecification**

<p>| An instance must have a name. | Error | 174 | An InstanceSpecification must have a name. |</p>
<table>
<thead>
<tr>
<th>Validation</th>
<th>Type</th>
<th>Number</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>An instance may form an internal structure of architectures only.</td>
<td>Error</td>
<td>175</td>
<td>Only an Architecture can be a parent of an InstanceSpecification.</td>
</tr>
<tr>
<td>Invalid connector attached to an instance. Both client and supplier end</td>
<td>Error</td>
<td>176</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>must be specified.</td>
<td>Error</td>
<td>180</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>An instance can't participate generalization nor realization.</td>
<td>Error</td>
<td>177</td>
<td>An InstanceSpecification can't inherit from any Element.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>178</td>
<td>An InstanceSpecification can't realize any Element.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>181</td>
<td>Elements can't inherit from an InstanceSpecification.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>182</td>
<td>Elements can't realize an InstanceSpecification.</td>
</tr>
<tr>
<td>Only assembly and delegate dependency may be attached to an instance.</td>
<td>Error</td>
<td>179</td>
<td>A connector can't be connected to an InstanceSpecification on a client side.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>183</td>
<td>A connector can't be connected to an InstanceSpecification on a supplier side.</td>
</tr>
<tr>
<td>A classifier of an instance must be an existing frame.</td>
<td>Error</td>
<td>184</td>
<td>A Classifier of an InstanceSpecification must be specified.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>185</td>
<td>A Classifier of an InstanceSpecification must be a component.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>186</td>
<td>A Classifier of an InstanceSpecification must be a SOFAFrame component.</td>
</tr>
<tr>
<td>An instance must have the same ports as its classifier.</td>
<td>Error</td>
<td>187</td>
<td>InstanceSpecification and a Classifier must have the same ports.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>188</td>
<td>InstanceSpecification can't have extra ports against a Classifier.</td>
</tr>
<tr>
<td><strong>Interface</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>An interface must have a name.</td>
<td>Error</td>
<td>219</td>
<td>An Interface must have a name.</td>
</tr>
<tr>
<td>An interface may be contained only by a package.</td>
<td>Error</td>
<td>220</td>
<td>Only a Package can be a parent of an Interface.</td>
</tr>
<tr>
<td>Invalid connector attached to an interface. Both client and supplier end</td>
<td>Error</td>
<td>221</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>must be specified.</td>
<td>Error</td>
<td>225</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>Only an interface may participate interface inheritance hierarchy.</td>
<td>Error</td>
<td>222</td>
<td>A base of an Interface must be another Interface.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>226</td>
<td>Only an Interface can inherit from an Interface.</td>
</tr>
<tr>
<td>An interface can not realize other elements.</td>
<td>Error</td>
<td>223</td>
<td>An Interface can't be on a client end of a realization.</td>
</tr>
<tr>
<td>Only generalization and realization dependency may be attached to an</td>
<td>Error</td>
<td>224</td>
<td>A connector can't be connected to a Interface on a client side.</td>
</tr>
<tr>
<td>interface.</td>
<td>Error</td>
<td>227</td>
<td>A connector can't be connected to a Interface on a supplier side.</td>
</tr>
<tr>
<td><strong>Multiplicity</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A multiplicity with only upper bound specified is allowed.</td>
<td>Error</td>
<td>241</td>
<td>Invalid multiplicity of an Element.</td>
</tr>
<tr>
<td></td>
<td>Notice</td>
<td>242</td>
<td>Lower bound of multiplicity will be set to 0 because SOFA arrays start from 0.</td>
</tr>
<tr>
<td><strong>Operation</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>An operation must have a name.</td>
<td>Error</td>
<td>118</td>
<td>An operation must have a name.</td>
</tr>
<tr>
<td>Conversion of return type int or integer to long.</td>
<td>Notice</td>
<td>119</td>
<td>Operation return type converted: int/integer-&gt;long.</td>
</tr>
<tr>
<td>Return type of an operation is should be any SOFA primitive data</td>
<td>Warning</td>
<td>120</td>
<td>Invalid operation return type.</td>
</tr>
<tr>
<td>Validation</td>
<td>Type</td>
<td>Number</td>
<td>Message</td>
</tr>
<tr>
<td>------------------------------------</td>
<td>------</td>
<td>--------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>type.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Port</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A port must have a name.</td>
<td>Error</td>
<td>204</td>
<td>A Port must have a name.</td>
</tr>
<tr>
<td>A port may be owned only by</td>
<td>Error</td>
<td>205</td>
<td>Only an Architecture of a Frame can be a parent of a Port.</td>
</tr>
<tr>
<td>architecture, frame, property or</td>
<td>Error</td>
<td>206</td>
<td>An Element can't be a parent of a Port.</td>
</tr>
<tr>
<td>instance specification.</td>
<td>Error</td>
<td>207</td>
<td>A Port must be have a Parent.</td>
</tr>
<tr>
<td>Invalid connector attached to a</td>
<td>Error</td>
<td>208</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>port. Both client and supplier end</td>
<td>Error</td>
<td>212</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>must be specified.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A port can't participate neither</td>
<td>Error</td>
<td>209</td>
<td>A Port can't inherit from any Element.</td>
</tr>
<tr>
<td>generalization nor realization.</td>
<td>Error</td>
<td>210</td>
<td>A Port can't realize any Element.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>213</td>
<td>Elements can't inherit from a Port.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>214</td>
<td>Elements can't realize a Port.</td>
</tr>
<tr>
<td>Only <strong>assembly</strong> and <strong>delegate</strong></td>
<td>Error</td>
<td>211</td>
<td>A connector can't be connected to a Port.</td>
</tr>
<tr>
<td>dependency may be attached to a</td>
<td>Error</td>
<td>215</td>
<td>A connector can't be connected to a Port.</td>
</tr>
<tr>
<td>port.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A type of a port is ignored.</td>
<td>Notice</td>
<td>216</td>
<td>A Type of a Port will be ignored.</td>
</tr>
<tr>
<td>Ports can't have an internal</td>
<td>Error</td>
<td>217</td>
<td>A Port can't have subports.</td>
</tr>
<tr>
<td>structure.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Only one either provided or required</td>
<td>Error</td>
<td>218</td>
<td>A Port may have only one provided or required interface.</td>
</tr>
<tr>
<td>interface may be owned by a port.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Property</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Properties must have a name.</td>
<td></td>
<td>189</td>
<td>A Property must have a name.</td>
</tr>
<tr>
<td>Properties may form an internal</td>
<td>Error</td>
<td>190</td>
<td>Only an Architecture can be a parent of a Property.</td>
</tr>
<tr>
<td>structure of a architectures only.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Invalid connector attached to a</td>
<td>Error</td>
<td>191</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>property. Both client and supplier</td>
<td>Error</td>
<td>195</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>end must be specified.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Properties can not participate</td>
<td>Error</td>
<td>192</td>
<td>A Property can't inherit from any Element.</td>
</tr>
<tr>
<td>inheritance nor realization.</td>
<td>Error</td>
<td>193</td>
<td>A Property can't realize any Element.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>196</td>
<td>Elements can't inherit from a Property.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>197</td>
<td>Elements can't realize a Property.</td>
</tr>
<tr>
<td>Only <strong>assembly</strong> and <strong>delegate</strong></td>
<td>Error</td>
<td>194</td>
<td>A connector can't be connected to a Property on a client side.</td>
</tr>
<tr>
<td>dependency may be attached to an</td>
<td>Error</td>
<td>198</td>
<td>A connector can't be connected to a Property on a supplier side.</td>
</tr>
<tr>
<td>instance.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validation</td>
<td>Type</td>
<td>Number</td>
<td>Message</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>-------</td>
<td>--------</td>
<td>-----------------------------------------------------------</td>
</tr>
<tr>
<td>A type of a property must be an existing frame.</td>
<td>Error</td>
<td>199</td>
<td>A Type of a Property must be specified.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>200</td>
<td>A Type of an Property must be a component.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>201</td>
<td>A Type of an Property must be a SOFAFrame component.</td>
</tr>
<tr>
<td>Properties must have the same ports as their types.</td>
<td>Error</td>
<td>202</td>
<td>Property and a Type must have the same ports.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>203</td>
<td>Property can't have extra ports against a Type.</td>
</tr>
</tbody>
</table>

Table B.1. SOFA Validations
# Appendix C. Fractal Validations

This appendix describes set of validations performed by a prototype implementation in order to validate loaded Fractal model. The first column of Table C.1 - Validation describes validation. The next three columns Type, Number and Message describe error, warning and notice messages reported by a prototype implementation. The list of messages is not complete, since the table contains only messages reported from model validator. Messages resulting from loading model or generating source code are not described here.

<table>
<thead>
<tr>
<th>Validation</th>
<th>Type</th>
<th>Number</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Assembly Connector</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>An assembly connector must be attached to instance specification or component or property.</td>
<td>Error</td>
<td>442</td>
<td>A client end of an assembly connector must be an instance of a component or a property or a component.</td>
</tr>
<tr>
<td>Assembly connector must interconnect subcomponents having the same parent.</td>
<td>Error</td>
<td>444</td>
<td>Supplier and client of an assembly connector must have the same parent component.</td>
</tr>
<tr>
<td>Assembly connector must connect required to provided interface.</td>
<td>Error</td>
<td>445</td>
<td>Components connected with an Assembly connector must have the same interface(s).</td>
</tr>
<tr>
<td><strong>Attribute</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Attributes must have a name.</td>
<td>Error</td>
<td>405</td>
<td>An attribute must have a name.</td>
</tr>
<tr>
<td>Conversion of attribute data type string to String.</td>
<td>Notice</td>
<td>406</td>
<td>Attribute type converted: string-&gt;String.</td>
</tr>
<tr>
<td><strong>Class</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A class must have a name.</td>
<td>Error</td>
<td>503</td>
<td>A Class must have a name.</td>
</tr>
<tr>
<td>Class may be nested only in a package.</td>
<td>Error</td>
<td>504</td>
<td>Only a Package can be a parent of a Class.</td>
</tr>
<tr>
<td>Invalid connector attached to an architecture. Both client and supplier end must be specified.</td>
<td>Error</td>
<td>505</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>Class inheritance is allowed only between classes.</td>
<td>Error</td>
<td>506</td>
<td>A base of a Class must be another Class.</td>
</tr>
<tr>
<td><strong>Error</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>442 - A client end of an assembly connector must be an instance of a component or a property or a component.</td>
<td>Error</td>
<td>442</td>
<td></td>
</tr>
<tr>
<td>443 - A supplier end of an assembly connector must be an instance of a component or a property or a component.</td>
<td>Error</td>
<td>443</td>
<td></td>
</tr>
<tr>
<td>444 - Supplier and client of an assembly connector must have the same parent component.</td>
<td>Error</td>
<td>444</td>
<td></td>
</tr>
<tr>
<td>445 - Components connected with an Assembly connector must have the same interface(s).</td>
<td>Error</td>
<td>445</td>
<td></td>
</tr>
<tr>
<td>446 - Components on a client end of an Assembly connector must have the same interface as a port connected on a supplier end.</td>
<td>Error</td>
<td>446</td>
<td></td>
</tr>
<tr>
<td>447 - Components on a supplier end of an Assembly connector must have the same interface as a port connected on a client end.</td>
<td>Error</td>
<td>447</td>
<td></td>
</tr>
<tr>
<td>448 - Ports on client and supplier end of an Assembly connector must have the same interface.</td>
<td>Error</td>
<td>448</td>
<td></td>
</tr>
<tr>
<td>449 - Client and supplier of an assembly must have the same parent.</td>
<td>Error</td>
<td>449</td>
<td></td>
</tr>
<tr>
<td>405 - An attribute must have a name.</td>
<td>Error</td>
<td>405</td>
<td></td>
</tr>
<tr>
<td>406 - Attribute type converted: string-&gt;String.</td>
<td>Notice</td>
<td>406</td>
<td></td>
</tr>
<tr>
<td>407 - Invalid attribute data type.</td>
<td>Warning</td>
<td>407</td>
<td></td>
</tr>
<tr>
<td>408 - Invalid classifier of an Attribute. Only a Class and Instance are allowed.</td>
<td>Error</td>
<td>408</td>
<td></td>
</tr>
<tr>
<td>409 - Default value should be specified for a constant attribute.</td>
<td>Notice</td>
<td>409</td>
<td></td>
</tr>
<tr>
<td>Validation</td>
<td>Type</td>
<td>Number</td>
<td>Message</td>
</tr>
<tr>
<td>---------------------------------------------------------------------------</td>
<td>-------</td>
<td>--------</td>
<td>-------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Only &quot;class realize component&quot; realization is allowed.</td>
<td>Error</td>
<td>507</td>
<td>A Class can't realize the Element.</td>
</tr>
<tr>
<td>Only realization and generalization dependency may be connected to a class.</td>
<td>Error</td>
<td>508</td>
<td>A connector can't be connected to a Interface on a client side.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>512</td>
<td>A connector can't be connected to a Interface on a supplier side.</td>
</tr>
<tr>
<td>A class can not have nor subelements nor ports.</td>
<td>Error</td>
<td>513</td>
<td>A class can't have any elements.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>514</td>
<td>A class can't have any ports.</td>
</tr>
<tr>
<td><strong>Component</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A component must have a name.</td>
<td>Error</td>
<td>413</td>
<td>A component must have a name.</td>
</tr>
<tr>
<td>Components may be nested only in a package.</td>
<td>Error</td>
<td>414</td>
<td>Only a Package or a Component can be a parent of a Component.</td>
</tr>
<tr>
<td>Invalid connector attached to a component. Both client and supplier end must be specified.</td>
<td>Error</td>
<td>415</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>419</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>A component may inherit only from another component.</td>
<td>Error</td>
<td>416</td>
<td>A base of a Component must be another Component.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>420</td>
<td>Only a Component can inherit from a Component.</td>
</tr>
<tr>
<td>Components can not realize other elements.</td>
<td>Error</td>
<td>417</td>
<td>A Component can't be on a client end of a realization.</td>
</tr>
<tr>
<td>Only assembly, delegate generalization and realization dependency may be attached to a component.</td>
<td>Error</td>
<td>418</td>
<td>A connector can't be connected to a Component on a client side.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>422</td>
<td>A connector can't be connected to a Component on a supplier side.</td>
</tr>
<tr>
<td>Only a class may realize a component</td>
<td>Error</td>
<td>421</td>
<td>Realization of a Component must be a Class.</td>
</tr>
<tr>
<td>Methods of components are not allowed.</td>
<td>Error</td>
<td>423</td>
<td>A Component can't have any methods.</td>
</tr>
<tr>
<td>Only an instance specification and a property may be subcomponent of a component.</td>
<td>Error</td>
<td>424</td>
<td>Element can't be a subcomponent.</td>
</tr>
<tr>
<td>An assembly connector must connect required to provided interface.</td>
<td>Error</td>
<td>425</td>
<td>An assembly must have required interface on a client side.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>426</td>
<td>An assembly must have provided interface on a supplier side.</td>
</tr>
<tr>
<td>A delegate connector must connect provided to provided interface.</td>
<td>Error</td>
<td>427</td>
<td>A delegate must have provided interface on a client side.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>428</td>
<td>A delegate must have provided interface on a supplier side.</td>
</tr>
<tr>
<td>A subsume connector must connect required to required interface.</td>
<td>Error</td>
<td>429</td>
<td>A subsume must have required interface on a client side.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>430</td>
<td>A subsume must have required interface on a supplier side.</td>
</tr>
<tr>
<td><strong>InstanceSpecification</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>An instance must have a name.</td>
<td>Error</td>
<td>451</td>
<td>An InstanceSpecification must have a name.</td>
</tr>
<tr>
<td>Instance may form an internal structure of components only.</td>
<td>Error</td>
<td>452</td>
<td>Only a Component can be a parent of an InstanceSpecification.</td>
</tr>
<tr>
<td>Invalid connector attached to an instance. Both client and supplier end must be specified.</td>
<td>Error</td>
<td>453</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>457</td>
<td>Invalid connector.</td>
</tr>
</tbody>
</table>

UML 2.0 Components

Appendix C. Fractal Validations - 2
<table>
<thead>
<tr>
<th>Validation</th>
<th>Type</th>
<th>Number</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>An instance can't participate generalization nor realization.</td>
<td>Error</td>
<td>454</td>
<td>An InstanceSpecification can't inherit from any Element.</td>
</tr>
<tr>
<td>An instance can't participate generalization nor realization.</td>
<td>Error</td>
<td>455</td>
<td>An InstanceSpecification can't realize any Element.</td>
</tr>
<tr>
<td>An instance can't participate generalization nor realization.</td>
<td>Error</td>
<td>458</td>
<td>Elements can't inherit from an InstanceSpecification.</td>
</tr>
<tr>
<td>An instance can't participate generalization nor realization.</td>
<td>Error</td>
<td>459</td>
<td>Elements can't realize an InstanceSpecification.</td>
</tr>
<tr>
<td>Only assembly and delegate dependency may be attached to an instance.</td>
<td>Error</td>
<td>456</td>
<td>A connector can't be connected to an InstanceSpecification on a client side.</td>
</tr>
<tr>
<td>Only assembly and delegate dependency may be attached to an instance.</td>
<td>Error</td>
<td>460</td>
<td>A connector can't be connected to an InstanceSpecification on a supplier side.</td>
</tr>
<tr>
<td>A classifier of an instance must be an existing component.</td>
<td>Error</td>
<td>461</td>
<td>A Classifier of an InstanceSpecification must be specified.</td>
</tr>
<tr>
<td>A classifier of an instance must be an existing component.</td>
<td>Error</td>
<td>462</td>
<td>A Classifier of an InstanceSpecification must be a component.</td>
</tr>
<tr>
<td>An instance must have the same ports as its classifier.</td>
<td>Error</td>
<td>463</td>
<td>InstanceSpecification and a Classifier must have the same ports.</td>
</tr>
<tr>
<td>An instance must have the same ports as its classifier.</td>
<td>Error</td>
<td>464</td>
<td>InstanceSpecification can't have extra ports against a Classifier.</td>
</tr>
<tr>
<td>Interface</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>An Interface must have a name.</td>
<td>Error</td>
<td>494</td>
<td>An Interface must have a name.</td>
</tr>
<tr>
<td>An interface may be nested only in a package.</td>
<td>Error</td>
<td>495</td>
<td>Only a Package can be a parent of an Interface.</td>
</tr>
<tr>
<td>Invalid connector attached to an interface. Both client and supplier end must be specified.</td>
<td>Error</td>
<td>496</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>Invalid connector attached to an interface. Both client and supplier end must be specified.</td>
<td>Error</td>
<td>500</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>Only an interface may participate interface inheritance hierarchy.</td>
<td>Error</td>
<td>497</td>
<td>A base of an Interface must be another Interface.</td>
</tr>
<tr>
<td>Only an interface may participate interface inheritance hierarchy.</td>
<td>Error</td>
<td>501</td>
<td>Only an Interface can inherit from an Interface.</td>
</tr>
<tr>
<td>An interface can not realize other elements.</td>
<td>Error</td>
<td>498</td>
<td>An Interface can't be on a client end of a realization.</td>
</tr>
<tr>
<td>Only generalization and realization dependency may be attached to an interface.</td>
<td>Error</td>
<td>499</td>
<td>A connector can't be connected to a Interface.</td>
</tr>
<tr>
<td>Only generalization and realization dependency may be attached to an interface.</td>
<td>Error</td>
<td>502</td>
<td>A connector can't be connected to a Interface.</td>
</tr>
<tr>
<td>Multiplicity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Only multiplicity in following forms is allowed: 1, *, ?, ..? where ? means any number or identifier.</td>
<td>Error</td>
<td>465</td>
<td>Invalid multiplicity of an Element.</td>
</tr>
<tr>
<td>Operation</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>An operation must have a name.</td>
<td>Error</td>
<td>410</td>
<td>An operation must have a name.</td>
</tr>
<tr>
<td>Return type string is converted to String.</td>
<td>Notice</td>
<td>411</td>
<td>Attribute type converted: string-&gt;String.</td>
</tr>
<tr>
<td>Return type of an operation is should be any Java primitive data type.</td>
<td>Warning</td>
<td>412</td>
<td>Invalid operation return type.</td>
</tr>
<tr>
<td>Port</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A Port must have a name.</td>
<td>Error</td>
<td>480</td>
<td>A Port must have a name.</td>
</tr>
<tr>
<td>Ports must be owned only by components, properties or instance specifications.</td>
<td>Error</td>
<td>481</td>
<td>An Element can't be a parent of a Port.</td>
</tr>
<tr>
<td>Ports must be owned only by components, properties or instance specifications.</td>
<td>Error</td>
<td>482</td>
<td>A Port must be have a Parent.</td>
</tr>
</tbody>
</table>

UML 2.0 Components  Appendix C. Fractal Validations - 3
<table>
<thead>
<tr>
<th>Validation</th>
<th>Type</th>
<th>Number</th>
<th>Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>Invalid connector attached to a port. Both client and supplier end must be specified.</td>
<td>Error</td>
<td>483</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>487</td>
<td>Invalid connector.</td>
</tr>
<tr>
<td>A port can't participate neither generalization nor realization.</td>
<td>Error</td>
<td>484</td>
<td>A Port can't inherit from any Element.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>485</td>
<td>A Port can't realize any Element.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>488</td>
<td>Elements can't inherit from a Port.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>489</td>
<td>Elements can't realize a Port.</td>
</tr>
<tr>
<td>Only <em>assembly</em> and <em>delegate</em> dependency may be attached to a port.</td>
<td>Error</td>
<td>486</td>
<td>A connector can't be connected to a Port.</td>
</tr>
<tr>
<td></td>
<td>Error</td>
<td>490</td>
<td>A connector can't be connected to a Port.</td>
</tr>
<tr>
<td>A type of a port is ignored.</td>
<td>Notice</td>
<td>491</td>
<td>A Type of a Port will be ignored.</td>
</tr>
<tr>
<td>Ports can't have an internal structure.</td>
<td>Error</td>
<td>492</td>
<td>A Port can't have subports.</td>
</tr>
<tr>
<td>Only one either provided or required interface may be owned by a port.</td>
<td>Error</td>
<td>493</td>
<td>A Port must have exactly one provided or required interface.</td>
</tr>
</tbody>
</table>

**Property**

| Properties must have a name.                                             | Error  | 466    | A Property must have a name.                     |
| Properties may form an internal structure of components only.            | Error  | 467    | Only a Component can be a parent of a Property.  |
| Invalid connector attached to a property. Both client and supplier end must be specified. | Error  | 468    | Invalid connector.                               |
|                                                                          | Error  | 472    | Invalid connector.                               |
| Properties can not participate inheritance nor realization.             | Error  | 469    | A Property can’t inherit from any Element.       |
|                                                                          | Error  | 470    | A Property can’t realize any Element.            |
|                                                                          | Error  | 473    | Elements can’t inherit from a Property.          |
|                                                                          | Error  | 474    | Elements can’t realize a Property.               |
| Only *assembly* and *delegate* dependency may be attached to an instance. | Error  | 471    | A connector can't be connected to a Property on a client side. |
|                                                                          | Error  | 475    | A connector can't be connected to a Property on a supplier side. |
| A type of a property must be an existing component.                      | Error  | 476    | A Type of a Property must be specified.          |
|                                                                          | Error  | 477    | A Type of an Property must be a component.       |
| Properties must have the same ports as their types.                      | Error  | 478    | Property and a Type must have the same ports.    |
|                                                                          | Error  | 479    | Property can't have extra ports against a Type.  |

**Table C.1. Fractal Validations**
Appendix D. Structure of Templates

This appendix describes template used by the prototype implementation to generate SOFA and Fractal source code. Principals of templates are described in Section 7.4.

Templates are described as UML class elements with stereotypes «template» for templates and «list» for list. Parameters are designed as attributes of classes with stereotype «param».

Dependency composition is used between templates and lists in order to describe ownership between them. Multiplicity of a client of a composition shows how many client’s instances may be contained by its owner.

D.1. SOFA Template

This section describes templates used by the prototype implementation to generate SOFA source code.

D.1.1. Architecture

The prototype uses two templates for SOFA architecture: architecturePrimitive which represents a primitive architecture and architecture which represents a composed architecture. Both templates are shown on Figure D.2.

Attributes: scopedNameWithVersion - an identifier of an architecture; scopedNameWithVersionImpl - an identifier of a frame that the architecture implements; system - represents whether the architecture is system architecture or not; typedef - definitions of new types for subcomponents with multiplicity different then 1.

The template architecturePrimitive does not contain neither lists nor templates.

The template architecture contains in addition to template architecturePrimitive four lists:

- properties with one subtemplate properties that allows define properties of an architecture.
  Attributes: constType - a data type o a property; identifier - represents an identifier of a property.

- bindings that allows to define bindings within an architecture. The list contains three subtemplates: assembly, subsume and delegate that represent three kinds of SOFA bindings.
  Attributes: client - a subcomponent which is on a client end of a connector; clientInterface - an interface of a component or a subcomponent which is on a client side of a connector; clientInterfaceMultiplicity - a multiplicity of a client interface; clientMultiplicity - a multiplicity of a client; supplier - a subcomponent which is on a supplier end of a connector; supplierInterface - an interface of a component or a subcomponent which is on a supplier side of a connector; supplierInterfaceMultiplicity - a multiplicity of a supplier interface; supplierMultiplicity - a multiplicity of a supplier; using - a type of a connector.

- subcomponents with one subtemplate subcomponent represents subcomponents of an architecture.
  Attributes: auto - indicates whether a subcomponent is auto subcomponent or not; identifier - an identifier of a subcomponent; scopedNameWithVersion - a scoped identifier of a subcomponent's type.
• typedef with one subtemplate typeArray which is used to specify definitions of arrays, that are used as subcomponents with multiplicity.
  Attributes: identifier - identifier of a new type; upperBound - upper bound of defined array; scopedNameWithVersion - scoped identifier of interface which makes a part of new type.

D.1.2. Frame

Template for SOFA frame shown on Figure D.3 contains one subtemplate protocol used to specify behavior protocol of a frame and four lists:
• typedef with one subtemplate typeArray which is used to specify definitions of arrays, that are used as provisions or requirements with multiplicity.
  Attributes: identifier - identifier of a new type; upperBound - upper bound of defined array; scopedNameWithVersion - scoped identifier of interface which makes a part of new type.
• provides and requires with one subtemplate interface used to define provisions and requirements of a frame.
  Attributes: identifier - an identifier of provision or requirement; scopedNameWithVersion - identifier of an interface or defined type which is a type of provision/requirement.
• properties with one subtemplate properties that allows define properties of a frame.
  Attributes: constType - a data type of a property; identifier - represents an identifier of a property.

D.1.3. Interface

The prototype implementation uses template interface for SOFA interface. The template is shown on Figure D.4. Attributes: identifierWithVersion - an identifier of an interface.

The template contains two lists:
• inheritanceSpec with one subtemplate item used to define base interfaces of an interface.
  Attributes: scopedNameWithVersion - a scoped name of a base interface.
• export used to define internals of an interface. The list contains four subtemplates:
  • interfaceProtocolSpec used to specify a behavior protocol of an interface.
  • constDcl used to specify constants of an interface.
  • attrDcl used to specify attributes of an interface.
  • opDcl used to specify methods of an interface. The template contains list parametersDcls with one subtemplate param used to define parameters of methods.
  Attributes: paramAttribute - represent direction of a parameter (in, out or inout); paramTypeSpec - data type of a parameter; simpleDeclarator - identifier of a parameter;
Attributes: **iProtOr** - behavior protocol of an interface; **constType** - data type of a constant; **identifierWithVersion** - identifier of a constant; **constExpr** - general value of a constant; **constExprString** - string value of a constant; **constExprChar** - character value of a constant; **constExprBool** - boolean value of a constant; **readonly** - identifies whether an attribute is readonly or not; **paramTypeSpec** - data type of an attribute; **identifier** - identifier of an attribute or a method; **opTypeSpec** - return data type of a method.

### D.1.4. Package

The prototype uses one simple template for SOFA modules. The template is shown on Figure D.1

Attributes: **definition** - represents a code contained by a module; **identifier** - an identifier of a module.

### D.1.5. UML Diagrams

![Figure D.1. Package](image)

Figure D.1. Package
Figure D.2. Architecture

UML 2.0 Components
Figure D.3. Frame
Figure D.4. Interface
D.2. Fractal Template

This section describes templates used by the prototype implementation to generate Fractal source code.

D.2.1. Class

Template for Fractal class is shown on Figure D.6. Attributes: identifier - an identifier of a class; package - scoped name of package in which a class is located. The template contains five lists:

- **attributes** with one subtemplate attribute used to define attributes of a class. Attributes:
  - defaultValue - general default value of an attribute;
  - defaultValueBool - boolean default value of an attribute;
  - defaultValueChar - character default value of an attribute;
  - defaultValueString - string default value of an attribute;
  - identifier - identifier of an attribute;
  - paramTypeSpec - data type of an attribute;
  - readonly - specifies whether an attribute is readonly or not;
  - scope - scope of an attribute (private, public, protected, etc.).

- **inheritanceSpec** and **implementationSpec** for specifying base classes and interfaces implemented by a class. The templates contain one subtemplate item. Attributes: scopedName - a scoped identifier of a base class or an implemented interface.

- **operations** with one subtemplate operation which represent methods of a class. The template operation contains two lists:
  - **parametersDcls** with one subtemplate param which specify parameters of a method.
  - **raisesList** with a subtemplate item to specify list of exceptions.

Attributes:
- **identifier** - an identifier of a method;
- **isStatic** - indicates whether the method is static or not;
- **opTypeSpec** - return type of a method;
- **paramAttribute** - represents a direction of method's parameter (e.g. in, out or inout);
- **paramTypeSpec** - data type of a parameter;
- **scope** - scope of a method (private, public, protected, etc.).
- **scopedName** - a scoped identifier of an exception;
- **simpleDeclarator** - an identifier of a parameter.

- **interfacesImplementations** with one subtemplate item used to generate skeletons of implemented methods of interfaces implemented by an owning class. The template item contains list operations which is described in previous paragraph.

D.2.2. Component

Template for Fractal component is shown on Figure D.5. Attributes: scopedName - an scoped identifier of a component. The template contains two subtemplates:

- **content** used to specify a name of an implementation class of a primitive component. Attributes: scopedName - a scoped name of an implementation class.

- **protocol** used to specify behavior protocol of a component. Attributes: protOr - a behavior protocol of a component.

and six lists:

- **extends** with one subtemplate item used to specify base components of a component. Attributes: scopedName - a scoped identifier of a base component.
• subcomponents which contains two templates subcomponent and subcomponentEmbeded that are used to specify both ways of specifying subcomponents in Fractal: as a link to an existing component or as an embedded definition of a component.

Attributes: code - source code of an embedded subcomponent; identifier - an identifier of a subcomponent; scopedName - a scoped identifier that represents a component which is a type of a subcomponent.

• attributes with one subtemplate attribute. Both are used to specify attributes of a component.

Attributes: attributeSignature - a name of an attribute controller; identifier - an identifier of an attribute; value - a default value of an attribute.

• provides and requires with subtemplate interface used to generate source code of component's provisions and requirements.

Attributes: cardinality - cardinality of a provision or a requirement; contingency - contingency of a provision or a requirement; identifier - an identifier of a provision or a requirement; scopedName - a scoped identifier of an interface which is a type of a provision or a requirement.

• bindings with one subtemplate binding which represent bindings within a component.

Attributes: client - an identifier of a subcomponent which is on a client side of a binding; clientInterface - an identifier of an interface of a component or a subcomponent which is on a client side of a binding; supplier - an identifier of a subcomponent which is on a supplier side of a binding; supplierInterface - an identifier of an interface of a component or a subcomponent which is on a supplier side of a binding.

D.2.3. Implementation Class

Template for a class that implements a component is shown on Figure D.7. Attributes: identifier - an identifier of a class; componentIdentifier - an identifier of a component that the class implements; package - scoped name of package in which an interface is located. The template contains eight lists:

• inheritanceSpec and implementationSpec for specifying base classes and interfaces implemented by a class. The templates contain one subtemplate item.

Attributes: scopedName - a scoped identifier of a base class or an implemented interface.

• componentAttributesDefinitions with one subtemplate attribute used to define realization of attributes of a component which is realized by the class.

Attributes: dataType - data type of an attribute; defaultValue - general default value of an attribute; defaultValueBool - boolean default value of an attribute; defaultValueChar - character default value of an attribute; defaultValueString - string default value of an attribute; identifier - identifier of an attribute;

• operations with one subtemplate operation which represent methods of a class. The template operation contains two lists:

• parametersDcls with one subtemplate param which specify parameters of a method.

• raisesList with a subtemplate item to specify list of exceptions.

Attributes: identifier - an identifier of a method; opTypeSpec - return type of a method; paramAttribute - represents a direction of method's parameter (e.g. in, out or inout); paramTypeSpec - data type of a parameter; scope - scope of a method (private, public, protected, etc.). scopedName - a scoped identifier of an exception; simpleDeclarator - an identifier of a parameter.

• interfacesImplementations with one subtemplate item used to generate skeletons of implemented methods of interfaces implemented by the class. The template item contains list operations which is described in previous paragraph.
• componentAttributesMethods with one subtemplate attribute which are used to generate skeletons of getter/setter methods of attributes of a component which is realized by the implementation class.

• attributes with one subtemplate attribute used to define attributes of the class.
  Attributes: defaultValue - general default value of an attribute; defaultValueBool - boolean default value of an attribute; defaultValueChar - character default value of an attribute; defaultValueString - string default value of an attribute; identifier - identifier of an attribute; paramTypeSpec - data type of an attribute; readonly - specifies whether an attribute is readonly or not; scope - a scope of an attribute (private, public, protected, etc.).

• implClientInterfacesList with one subtemplate item used to define implementation of client (required) interfaces of implemented component.
  Attributes: name - an identifier of an interface; scopedName - a type of an interface.

D.2.4. Interface and AttributesInterface

The prototype uses two templates to generate interfaces: interface which represents a general interface and attributesInterface which represents an attribute controller. Both templates are shown on Figure D.8. Attributes: identifier - an identifier of an interface; package - scoped name of package in which an interface is located.

The template attributesInterface contains one list attributesMethods with one subtemplate attribute which represents getter/setter methods for an attribute. Attributes: dataType - data type of an attribute; identifier - an identifier of an attribute; identifierMethod - an identifier with big first character that is used in names of getter/setter methods.

The template interface contains two lists:
• inheritanceSpec with one subtemplate item which represent base interfaces.
• export which represents internals of an interface. The list contains four subtemplates:
  • constDcl used to specify constants of an interface.
  • attrDcl used to specify attributes of an interface.
  • opDcl used to specify methods of an interface. The template contains lists parametersDcl with one subtemplate param used to define parameters of methods and raisesList with one subtemplate item used to define list of exceptions that may be thrown out from the method.
    Attributes: paramAttribute - represents direction of a parameter (in, out or inout); paramTypeSpec - data type of a parameter; simpleDeclarator - identifier of a parameter;
  • interfaceProtocolSpec used to specify a behavior protocol of an interface.
    Attributes: iProtOr - behavior protocol of an interface; constType - data type of a constant; identifierWithVersion - identifier of a constant; constExpr - general value of a constant; constExprString - string value of a constant; constExprChar - character value of a constant; constExprBool - boolean value of a constant; readonly - identifies whether an attribute is readonly or not; paramTypeSpec - data type of an attribute; identifier - identifier of an attribute or a method; opTypeSpec - return data type of a method.
D.2.5. Subcomponent

Template for an embedded definition of a subcomponent is shown on Figure D.9. Attributes: name - an identifier of a subcomponent. The template contains two subtemplates:

- **content** used to specify a name of an implementation class of a primitive subcomponent. Attributes: scopedName - a scoped name of an implementation class.
- **protocol** used to specify behavior protocol of a subcomponent. Attributes: protocol - a behavior protocol of a subcomponent.

and six lists:

- **definition** with one subtemplate item used to specify base components of a component. Attributes: scopedName - a scoped identifier of a base component.
- **subcomponents** which contains two templates subcomponent and subcomponentEmbedded that are used to specify both ways of specifying subcomponents in Fractal: as a link to an existing component or as an embedded definition of a component. Attributes: code - source code of an embedded subcomponent; identifier - an identifier of a subcomponent; scopedName - a scoped identifier that represents a component which is a type of a subcomponent.
- **attributes2** with one subtemplate attribute. Both are used to specify attributes of a component. Attributes: attributeSignature - a name of an attribute controller; identifier - an identifier of an attribute; value - a default value of an attribute.
- **provides** and **requires** with subtemplate interface used to generate source code of component's provisions and requirements. Attributes: cardinality - cardinality of a provision or a requirement; contingency - contingency of a provision or a requirement; identifier - an identifier of a provision or a requirement; scopedName - a scoped identifier of an interface which is a type of a provision or a requirement.
- **bindings** with one subtemplate binding that represent bindings within a component. Attributes: client - an identifier of a subcomponent which is on a client side of a binding; clientInterface - an identifier of an interface of a component or a subcomponent which is on a client side of a binding; supplier - an identifier of a subcomponent which is on a supplier side of a binding; supplierInterface - an identifier of an interface of a component or a subcomponent which is on a supplier side of a binding.
D.2.6. UML Diagrams

Figure D.5. Component
Figure D.6. Class
Figure D.7. Implementation Class
Figure D.8. Interface
Figure D.9. Subcomponent