

Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Lukáš Hermann

IDE-supported development of component-based applications

Department of Distributed and Dependable Systems

Supervisor of the master thesis: RNDr. Tomáš Bureš Ph.D.

Study programme: Informatics

Specialization: Software Systems

Prague 2011

I would like to thank my supervisor Tomáš Bureš for his valuable suggestions and observations, my colleague Jaroslav Kezníkl for his introduction to the SOFA 2 component system, my friend František Adamec for spelling corrections, my friend Lukáš Nádvorník and my mother for their continual motivation and support in my life and studies and my father, in memoriam, for everything.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, August 1, 2011

Lukáš Hermann

Název práce: Vývoj komponentově založených aplikací podporovaný integrovanými vývojovými prostředími

Autor: Lukáš Hermann

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: RNDr. Tomáš Bureš Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Na rozdíl od komerčních komponentových systémů nemají ty akademické dostatečnou podporu v integrovaných vývojových prostředích. Tato práce analyzuje vývoj komponentově založených aplikací na komponentovém systému SOFA 2 a zjišťuje, že hlavním problémem je nedostatečné propojení mezi procesy obecného návrhu aplikace a vytváření konkrétních komponent. Na základě této analýzy definuje podmnožinu UML, univerzálního jazyka pro návrh aplikací, a její sémantiku vzhledem k entitám komponentového systému SOFA 2. Dále vytváří nástroj integrovaný do vývojového prostředí Eclipse, který umožňuje automatické generování těchto entit z UML komponentového modelu, stejně jako propojení tohoto modelu s již existujícími entitami, umožňující jejich automatickou opravu v případě změny modelu. Tento nástroj je navržen modulárně tak, aby bylo možné jednoduše změnit sémantiku modelu nebo ho použít na jiné modely. Nakonec tato práce analyzuje možnosti rozšíření tohoto nástroje na další komponentové systémy, generování kódu a verifikaci chování komponent.

Klíčová slova: komponenty, vývoj software, UML, transformace modelů

Title: IDE-supported development of component-based applications

Author: Lukáš Hermann

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Tomáš Bureš Ph.D., Department of Distributed and Dependable Systems

Abstract: Unlike many proprietary component systems, the academic ones do not have sufficient support in integrated development environments. This thesis analyzes development of component-based applications in terms of the SOFA 2 component system and it finds out that the main issue is an insufficient connection between processes of common application design and creation of particular components. Based on this analysis, it defines a subset of the UML, a universal language for application design, and its semantics regarding entities of the SOFA 2 component system. Furthermore, it creates a tool integrated to the Eclipse IDE, which enables a developer to automatically generate these entities from a UML component model as well as to connect this model with already existing entities enabling their automatic correction in case of model changing. This tool is designed modularly so that it is possible to easily change semantics of the model or using it for other models. Finally, this thesis analyzes possibilities of extensions of this tool for other component systems, code generation and component behaviour verification.

Keywords: components, software development, UML, model transformation

Contents

Introduction	2
1 SOFA 2 component system	4
1.1 Component model	5
1.2 Application development	7
2 Analysis and general strategy	11
2.1 Analysis	11
2.2 General strategy	15
3 Defining the UML subset	18
3.1 Brief introduction to the UML	18
3.2 List of possible definitions	20
3.3 Analysis of the chosen option	20
3.4 Semantics of UML elements	22
4 Mapping UML elements to SOFA 2 entities	28
4.1 Mapping model	29
4.2 Preparation model	30
4.3 SOFA 2 entities generation and compatibility	34
5 SOFA 2 UML tool implementation	40
5.1 Editor of the mapping model	40
5.2 Implementation of model transformations	44
5.3 Integration with SOFA 2 IDE	47
6 Evaluation and discussion	50
6.1 Meeting the requirements	50
6.2 Other source and target models	51
6.3 Further model extensions	52
7 Related work	54
Conclusion	56
Bibliography	58
List of Figures	61
List of Abbreviations	62
A Content of the enclosed CD-ROM	63
B Additional resources	64
B.1 Complete UML model constraints	64
B.2 Complete UML to preparation model transformation	66

Introduction

In last decades, the software development advanced from creating simple applications by several programmers to developing complex systems by large teams. As the complexity of the applications grew, more and more errors occurred. When the situation was unmaintainable, several approaches of the software design and development were standardized as branches of the software engineering, a newly established discipline that studies how to design and implement software to be of a higher quality.

One of these approaches, which has become well-understood and widely used for developing of any kind of applications from small single-purposed tools to large enterprise systems, is the component-based development (CBD). The main characteristic of this technique is that applications are built by composing components, which encapsulate semantically related data and functions. There is a common approach declaring that the term *component* means a black-box entity with well-defined interfaces and behaviour. Only through its interface, it can provide some services to other components and from the opposite side, it can require other ones. The main advantage of this concept is that implementation of the provided services can be easily substituted without breaking the system in which the component operates.

Another advantage of the CBD is that a new application can be created by composing previously developed components, which may be reused as they are or easily adapted, which minimizes code writing. Unfortunately, this property of the CBD leads to significant differences from the common development process of software systems, because it separates the development process of components from the development process of systems. For the component development, the main effort is to design components to be reusable, while the system development focuses on finding the desired ones. Since both processes require different points of view to the application development, it might be tough to cooperate them.

There are two different ways how to build an application. In a top-down approach, a whole application architecture is designed at first, then components are developed to fit this design rather than for reuse. In a bottom-up approach, highly reusable components are developed, then various applications can be assembled from them. Since it is difficult to make both general and effective components, the first approach is widely spread as well as a combination of both [5].

A framework for developing, composing, and running components is called a component system. Each component system has its own abstract definition of a component called a component model. It specifies all involved entities with their semantics, i.e. how components are built, composed, deployed etc.

Nowadays, there exist many proprietary component systems as well as the academic ones. Members of the first group are, for example, CORBA Component Model (CCM) [17] from the Object Management Group (OMG), Component Object Model (COM) [16] from Microsoft, Enterprise Java Beans (EJB) [23] from Oracle etc. In the second group, we can find, for example, SOFA 2 [13] and Fractal [24] component systems. The academic component systems offer many advanced features in addition to features of the proprietary ones, on the other hand, they do not have as wide tool support of the development process.

Among main advanced features of the academic component systems, there is a hierarchical component model, a component repository and support for multiple communication styles. The hierarchical component model makes possible that every component might be implemented by a set of subcomponents that are connected together and can require same services as the parent component that can, on the other hand, delegate its provided interfaces to them. The component repository stores all components together with their source code and other resources. It often supports versioning, team development and application distribution. The support of multiple communication styles is implemented by replacement of simple interfaces between components by more general connectors, which can be generated from templates during the application deployment.

Both component and system development processes cover many steps and involve a number of stakeholder roles, so component systems should provide a big set of development supporting tools and infrastructure. The academic component systems have some tools, which provide a textual or graphical user interface for performing various component-related tasks, but there are not covered parts of the development process, especially those connecting both processes together.

While the component development process is tightly connected with the used component model specification, the system development process is more independent of the current implementation. Nevertheless, it can be also well-defined, since it can use some standards from the model-driven development (MDD), a software development methodology which focuses on a productivity increase through creating and exploiting domain models [36]. This leads us to another useful concept of the MDD that is automatic code generation from a model.

SOFA 2 [13] is an academic component system with all of mentioned advanced features. It is a complete framework supporting all stages of the application life cycle from development to execution and most of the development process is supported by several tools. Nevertheless, a connection between the component and system development processes is not well-defined nor tool-supported.

In this thesis, we analyze CBD processes in terms of the SOFA 2 component system from the point of view of chosen stakeholder roles. We identify those parts of the processes that lack of tool support with a focus on the connection between the component and system development processes. Based on this analysis, we precisely define all models necessary to build a tool connecting the design view represented by the UML component model and the implementation view represented by the SOFA 2 component model. This tool, whose implementation we describe, enables developers to automatically generate some parts of SOFA 2 applications from UML models.

To achieve this goal, the thesis is structured as follows. Chapter 1 briefly introduces the SOFA 2 component model and the development of SOFA 2 applications. In chapter 2, we analyze them, we find their unsupported parts and we propose a general strategy how to solve some of identified problems. As we find out that the main problem is in a mapping between the component and system development processes, first we precisely define how the system development is modeled in chapter 3, and then we define the mapping itself in chapter 4. Chapter 5 describes an architecture of implementation of a new supporting tool and its integration to the current environment. In chapter 6, we evaluate and discuss the described solution and in chapter 7, we compare it with other related works.

1. SOFA 2 component system

SOFA 2 is a component system supporting advanced concepts of the CBD such as a hierarchical component model, ADL-based design, extensible controllers, a component repository, connectors, behaviour specification and verification etc. [3].

The SOFA 2 component system uses a hierarchical component model defined by the MetaObject Facility (MOF) meta-model. Each component in the SOFA 2 component model is a well defined isolated entity encapsulating some business logic and it is connected with other components by well defined communication endpoints called interfaces. We distinguish three parts of the component – a border part, a content part and a control part (see figure 1.1). The border part, known as a component *frame*, defines possibilities of interaction with other components in the system. It specifies provided and required interfaces of the component and, optionally, its behaviour by behaviour protocol definitions (see more in [7]). The content part, known as a component *architecture*, specifies how the behaviour defined by the frame is implemented. Since this component model is hierarchical, the component architecture is either primitive (specified directly by code), or composite (specified by composition of connected subcomponents). The component’s control part manages its life cycle, instantiation etc. It is by default provided by the SOFA 2 environment but it can be easily extended by component aspects defined in the micro-component model (see more in [6]).

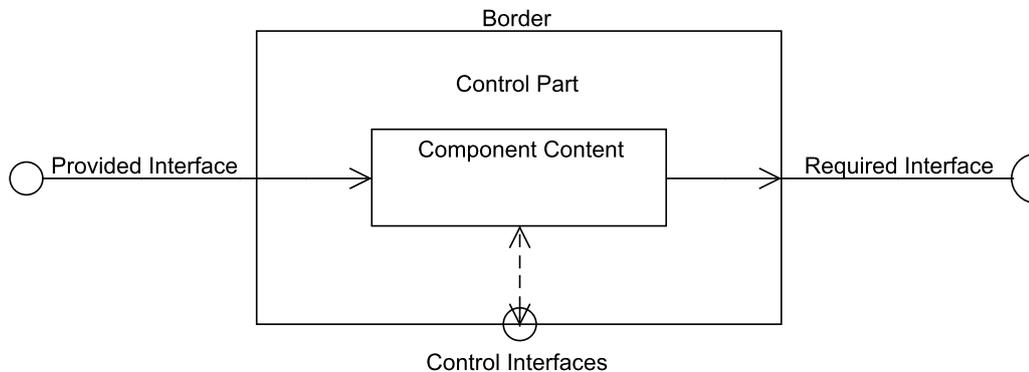


Figure 1.1: Component abstraction

When one component provides the same type of interface that another one requires (they share the same *interface type*), they can be connected, so the second component can demand the declared service on the first one. This connection can be realized simply by a method invocation, but in distributed systems, it is more complicated, since the communication can proceed through heterogeneous environments, so the concept of connectors is introduced. Various connectors can be developed for various kinds of situations, then a developer can specify which communication style is used in a particular interface (see more in [2]).

When an application is going to be deployed so that it can be launched, a top-level component must be chosen, which represents the application and does not have any provided and required interfaces. This component is called an

assembly. Furthermore, each component can operate on a different virtual or physical machine, known as a node, which is specified in a *deployment plan*.

Each of the mentioned SOFA 2 entities (see an example in figure 1.2) such as a frame, architecture, interface type, assembly, deployment plan etc. has its own definition in the Architecture description language (ADL) stored in a form of an XML file. These files are stored together with code in a directory of the SOFA 2 application. To be able to deploy and run the application or share its entities, they must be uploaded to the SOFA 2 repository. The repository serves as a storage for SOFA 2 entities that can be easily downloaded for reusing in other applications or updated, since it supports different versions.

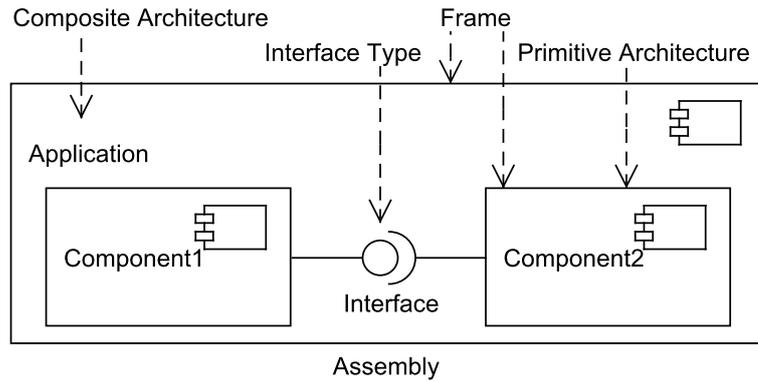


Figure 1.2: Example of SOFA 2 entities

In the following sections, we more deeply describe properties of chosen entities from the SOFA 2 component model related to our further work, which enables us to show how applications in the SOFA 2 component system are being developed.

1.1 Component model

Since the whole SOFA 2 component model is too complex from the point of view of this thesis, we introduce only its simplified version shown in figure 1.3. Therefore, the list of described model entities and also the lists of their properties are not complete.

InterfaceType: As we said, components can be connected by interfaces of the same interface type. This is a SOFA 2 entity that has its *name* and that points to a *signature* which serves as a definition of the declared service. The signature can be, for example, an interface in Java or an abstract class in C. An instance of an *InterfaceType* is called an *Interface*. It has its own *name* and it can specify its own communication style (default is a method invocation).

Frame: This is an entity representing the black-box view of a component. It owns two collections of *Interface* entities. The first collection contains the interfaces that the component *provides*, while the second one contains those that the component *requires*. These interfaces are only communication channels to other

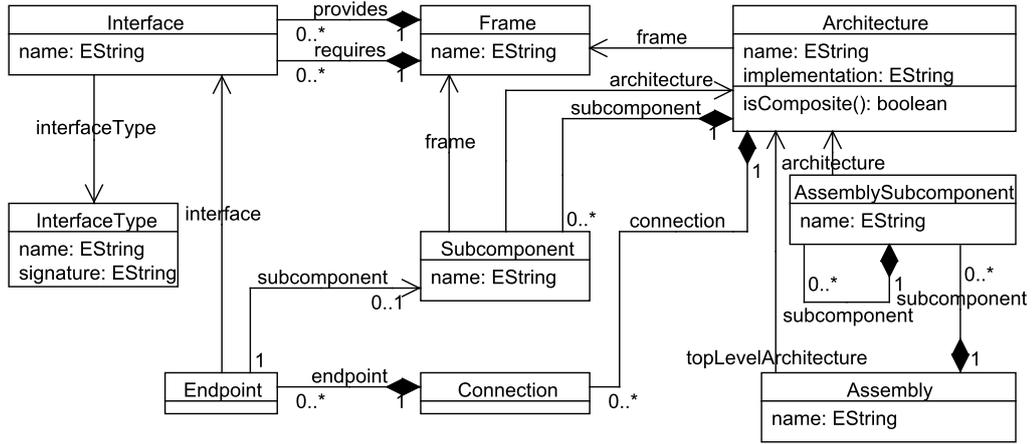


Figure 1.3: Simplified SOFA 2 component meta-model

components. Components with the same frame look similarly from outside but they can have different implementation. In figure 1.4, there is an example of a component frame.

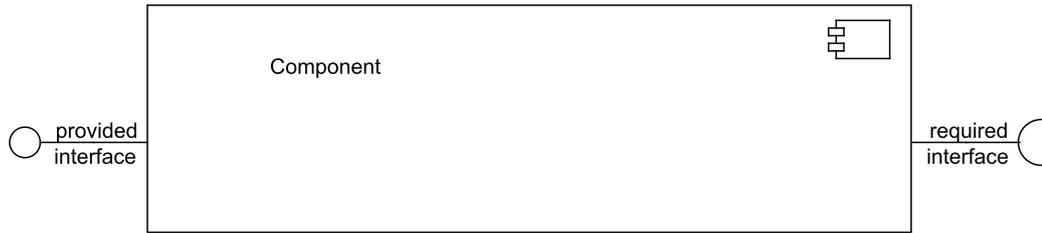


Figure 1.4: Example of a SOFA 2 *Frame*

Architecture: This entity defines implementation of a *Frame*, so it specifies the way how the services of the provided interfaces are implemented with using the services of the required ones. It can be done either directly by code (a primitive architecture), or indirectly by connecting the interfaces to subcomponents (a composite architecture). In case of a composite architecture, we distinguish three types of connections. The first one is a connection between two subcomponents, it is called a connector. The other two are connections between the parent component and one of its subcomponents. When the connection corresponds to a provided interface, we call it subsumption, when it corresponds to a required interface, we call it delegation. In figure 1.5, there is an example of a composite architecture.

Every architecture points to a *frame*. A primitive architecture specifies its *implementation* by a fully qualified name of a Java or C class. A composite architecture has a collection of *subcomponents*, where each of them contains a reference to a *frame* or *architecture* it has to implement. It has also a collection of *connections*, where each of them has two *endpoints*. Every endpoint is defined

by a *subcomponent* and some of its *interfaces*, or by an *interface* of a parent component, when the connection binds an outer interface to a subcomponent.

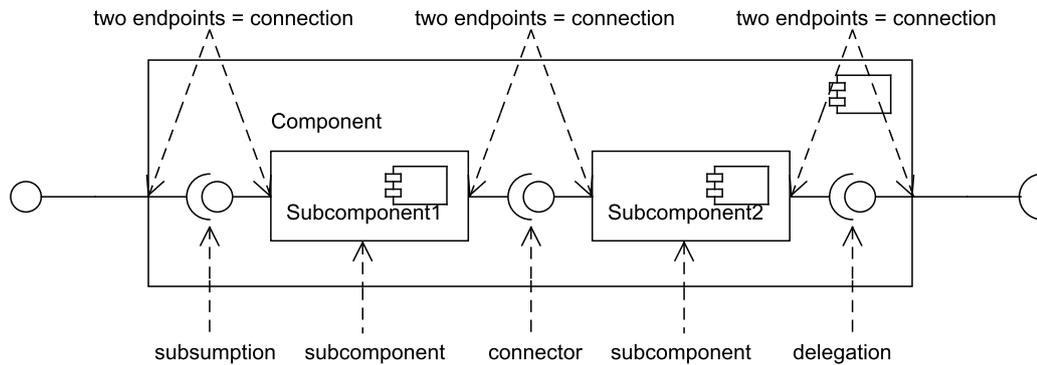


Figure 1.5: Example of a composite SOFA 2 Architecture

Assembly: This is a top-level component without any provided and required interfaces. It mostly has a composite architecture and since subcomponents in a composite architecture can be defined only by their frames, it can specify architectures of all of its direct or indirect subcomponents. Every assembly point to its *top-level architecture* and to a tree of its *subcomponents*.

There are other entities in the SOFA 2 component model such as a deployment plan or a code bundle, but they are not important for this thesis.

1.2 Application development

Development of a SOFA 2 application is divided into three separated processes. Two of them, the system and component development processes, are also present in the general CBD [5], while the third, the connector development process, is SOFA 2 specific.

The system development process begins by designing an overall architecture of the application. Someone must decide which components can be reused from previous projects and which ones must be newly developed. If a new component is required, the component development process is initiated. It is also possible that the application must be deployed to an environment requiring connectors that have not been created yet. In this case, the connector development process is launched to produce them. When all necessary components and connectors are developed, the whole application is assembled and deployed, which includes uploading all created SOFA 2 entities to the repository. Since each component can be independently deployed, someone should specify on which nodes each of them will run. The whole process is displayed in figure 1.6.

The component development cycle includes these steps:

1. First, all *InterfaceTypes* that the component will use must be defined or selected. When a new *InterfaceType* is created, a Java interface or a C abstract class must be assigned to its *signature* property. Its methods define a contract between two components connected via an instance of this type.

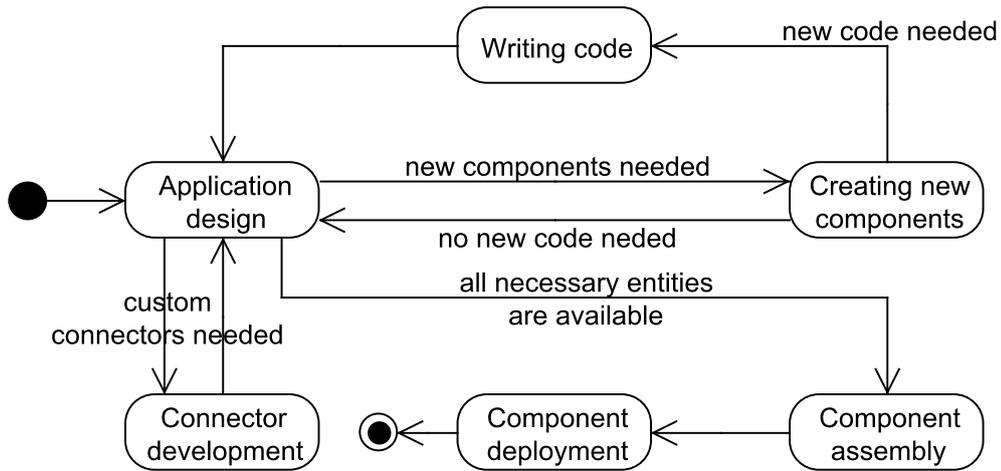


Figure 1.6: SOFA 2 system development process

2. Then, the component *Frame* should be created. Its provided and required interfaces of desired *InterfaceTypes* are specified.
3. Finally, the component *Architecture* implements its *Frame* either directly by code, or indirectly by delegation to subcomponents. In case of a primitive architecture, a Java or C class must be assigned to its *implementation* property. This class must implement all provided interfaces of architecture's frame and can define attributes of types of all of its required interfaces. When an architecture is composite, all of its *subcomponents* and their *connections* must be specified. For each subcomponent, its frame or architecture is reused or newly developed, which means further iterations of this cycle.

The connector development process is not widely described here, since it is not related to work in this thesis (see more in [2]).

Since every entity must be uploaded to the repository before an application can be launched and since deployment of an application is a little bit complicated due to various connector specifications, it is necessary to have a tool that automates these tasks. For these purposes, *Cushion* was created, which is a command-line tool for batch working with the SOFA 2 repository. It supports checking-out of entities from the repository, editing them and committing them back as well as creation of new ones. It also supports working with versions and automatic deploying of an application from its deployment plan. However, content of entities (e.g. a definition of frame) must be specified manually in a text editor or by another tool.

Let's have a look on a small sample of a *Cushion* script with a description how to map it to the application development:

1. Application design: Let's have an application (**Application**) containing two components (**Component1** and **Component2**) connected with one interface (**Interface**) as in figure 1.2 on page 5. *Cushion* does not support this phase.

2. Creating new components: This phase contains creation of all interface types, frames and architectures. *Cushion* can create these entities in the repository and commit their ADL descriptions, which must be specified manually:

```
# create new interface types in the repository
new interface initial sample.IInterface
# create new frames in the repository
new frame initial sample.FComponent1
new frame initial sample.FComponent2
new frame initial sample.FApplication
# new architectures in the repository
new architecture initial sample.AComponent1
new architecture initial sample.AComponent2
new architecture initial sample.AApplication
# before further actions a developer must specify
# ADL descriptions of all entities
# commit all ADL descriptions
commit
```

3. Writing code: In this phase, code of all interface types and primitive architectures is written, compiled and uploaded to the repository. *Cushion* supports automatic compilation and uploading of code:

```
# compile and upload code sources
compile sample.IInterface
upload sample.IInterface
compile sample.AComponent1
upload sample.AComponent1
compile sample.AComponent2
upload sample.AComponent2
```

4. Component assembly: When all components are ready, an assembly is created from the top-level component. This is similar for *Cushion* as creation of other entities:

```
# create an application assembly in the repository
assembly initial sample.Application sample.AApplication
# before a commitment the ADL description must be specified
commit sample.Application
```

5. Component deployment: After assembling, a deployment plan is specified and the application is deployed according to it. The deployment plan must be specified manually, but *Cushion* supports its automatic deployment:

```
# create a deployment plan in the repository
deplplan initial sample.DeplPlan sample.Application
# before a deployment the ADL description must be specified
deploy sample.DeplPlan
```

Since creation of ADL descriptions of SOFA 2 entities in a text editor is a little bit uncomfortable, *SOFA 2 IDE* was created. *SOFA 2 IDE* is a GUI tool based on Eclipse [27] for interactive creating and editing of SOFA 2 entities from interface types to assemblies. It introduces a SOFA 2 application as an Eclipse project, which is connected with the SOFA 2 repository. However, it can organize entities only in a list and it offers only a one level view to a component architecture, so a hierarchical structure of an application is hidden. The application development cycle is similar to that in Cushion, though some parts such as creation of entities and writing code can be mixed.

SOFA 2 IDE supports the first phases of the development cycle, for the second one, *MConsole* is used. *MConsole* is a GUI tool for managing SOFA 2 environment and launching SOFA 2 applications. It supports creation of SOFA 2 nodes and deployment plans and it can deploy an application according to them. In addition, it simplifies launching and stopping of SOFA 2 applications.

We can summarize coverage of the CBD process of SOFA 2 applications by available tools in figure 1.7. Note that not all parts of the CBD process are supported.

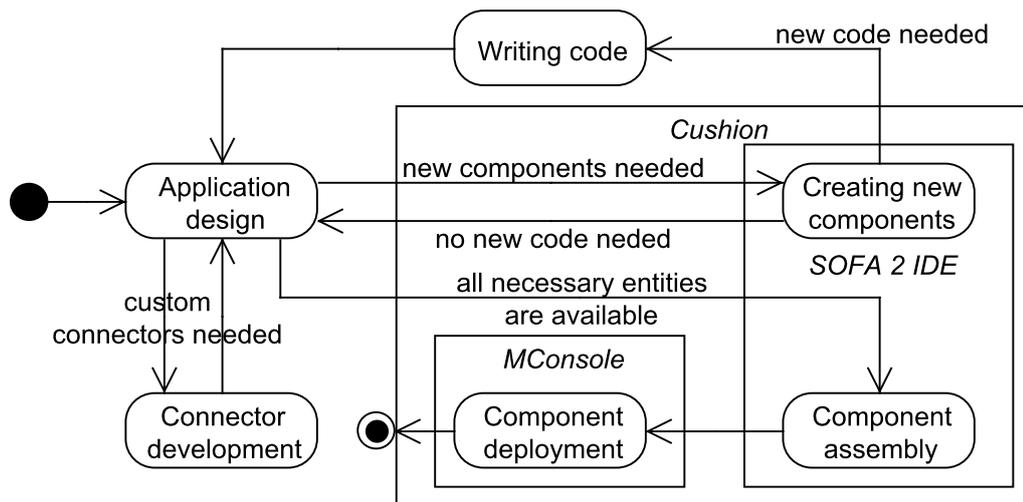


Figure 1.7: Covering the development process with available tools

2. Analysis and general strategy

After the introduction to the CBD process, we can analyze how its SOFA 2 implementation can be improved. Since this process is relatively complex, it involves a number of stakeholder roles with different approaches. In order to deeply understand it, we should analyze how people that fulfill these roles enter the process, what their input and output are and how they can utilize available tools. Based on this analysis, we should identify parts of the CBD process that have no tool support or current tools do not follow a developer in its particular development process. We should also think about compatibility between output of one part of the process and input of the following part. Is a current form of selected output ideal? Is it possible to propose a better one? From all of these issues, we choose one that most complicates the CBD process of SOFA 2 applications.

When we know what should be improved, we propose a general strategy that leads us in a process of finding how it should be done. In this strategy, we outline the main issues that are solved in next chapters of this thesis and we specify requirements that the solution must fulfill in order to really improve the CBD process.

2.1 Analysis

We can identify four base stakeholder roles that participate on the CBD process (see figure 2.1):

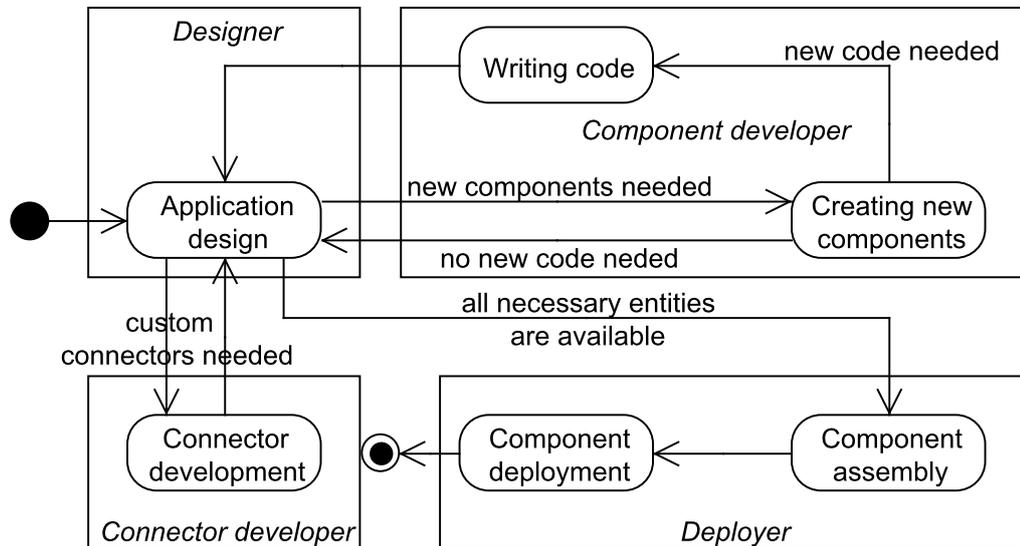


Figure 2.1: Participation of stakeholder roles on the component-based development process

1. *Designer*: He designs an overall architecture of an application. His input is a requirement specification with a list of use-cases and decisions about

used technologies. He transforms this input to a hierarchical model of components and he decides which components can be reused and which ones need to be newly defined. He also finds whether all necessary connectors are accessible considering proposed technologies. Therefore first, he gives instructions to a component developer what components he must create and to a connector developer what kind of connectors are needed. When he receives implementation of a new component, he puts it to the model that can be in this phase corrected due to issues in the component development. In this case, he also corrects its instructions to a component developer. When all necessary components and connectors are developed, he has a prepared component model that he delivers to a deployer for the last part of the development process. This process is unfortunately not covered by any of available SOFA 2 tools, but since it is relatively independent of a chosen component framework, it can be solved by several common tools such as UML editors.

2. *Component developer*: He develops new components, which includes creation of new SOFA 2 entities, specifying their properties and writing code for primitive architectures and interface types. His input is a request from a designer, in which it is specified what the component should offer and what services it can use. It is also recommended there, whether implementation of the component should be direct using code, or indirect using other existing or even non-existing components. Based on this demand, he creates or selects all necessary entities connected with the component such as interface types, frames and architectures. Then, if an interface type or a primitive architecture must be specified, he writes corresponding code. Optionally, he writes a behaviour specification for some frames. Finally, he commits all entities to the repository for further processing by a designer. For creation and commitment of SOFA 2 entities, a component developer can use *Cushion* or *SOFA 2 IDE* tools. Since writing code is a general task, a developer can use any of compilers suitable for the used programming language. The most useful might be *Eclipse IDE* because *SOFA 2 IDE* is based on Eclipse.
3. *Connector developer*: He develops new connectors for various technologies. His input includes a description of middleware technologies that components in the application can use for their communication. He creates templates that proxy classes will be generated from when the application is being deployed. These templates are also his output. A connector developer solves a very specific and also general task, thus *Eclipse IDE* might be a sufficient supporting tool.
4. *Deployer*: He assembles and deploys the application. His input are all SOFA 2 entities necessary for the application deployment. He chooses one top-level component as an assembly and defines used architectures of its subcomponents if needed. Then, he creates nodes, which are abstractions of physical or virtual machines where SOFA 2 components can be placed, and decides on which node each component will run when the application will be launched, which he specifies in a deployment plan. Finally, he deploys

the application according to the deployment plan. Therefore, his output is the application ready for launching. For all of these tasks, a deployer can use *Cushion* or *MConsole*, which fully support it.

Let's also analyze transitions between those parts of the CBD process that are solved by different stakeholder roles (see figure 2.1) or that are supported by different tools (see figure 1.7) because they can be somehow problematic:

- *Component assembly* \rightarrow *Component deployment*: Since both tasks are solved by a deployer and *SOFA 2 IDE* and *MConsole* are well connected or *Cushion* can be used, this transition is now enough managed.
- *Application design* \leftrightarrow *Connector development*: As we mentioned before, the connector development is a very specific and also general task, which is very rarely solved, thus it is not necessary to widely analyze this transition.
- *Creating new components* \rightarrow *Writing code*: When a new *SOFA 2 InterfaceType* or a new *SOFA 2 primitive Architecture* are created, a component developer must also manually create an interface or a class in the used programming language, although some parts of code (e.g. interface implementation for provided interfaces or private attributes for required ones) can be automatically generated from known *SOFA 2* entity properties. This tool might be handy but it is not so crucial because these tasks are managed by one stakeholder role.
- *Application design* \leftrightarrow *Creating new components*: This transition is the most problematic because not only that leads between different stakeholders, but also between two separated development processes, and in addition it is bidirectional. It is obvious that this should be widely analyzed.

During the preparation of the whole application architecture, a designer thinks in terms of components and interfaces (a design view) rather than frames, architectures, assemblies and interface types (an implementation view), which is common for a component developer. The problem is that there is a lot of communication between these two people that is not standardized. This can lead to misunderstanding or work duplication. A designer is also not forced to keep the application design up to date with the current implementation, because notation of mapping between a model and implementation is also not well-defined and there are no direct advantages of doing that. Once he needs a new component, he sends instructions to a component developer, who must manually transform the design view into the implementation one. However, these views share a lot of information (e.g. frame or composite architecture definitions), so it should be possible that this transformation will be done automatically.

Let's have a small example. A designer has analyzed requirements of an application and he has decided that he will solve them by a component (**Application**) with two subcomponents (**Component1** and **Component2**) connected with an interface (**Interface**). Since he does not find any components that fulfill the requirements in the repository, he creates a model for this application displayed in figure 2.2 and sends it with some comments, what these components should do and how they should communicate, to a component developer.

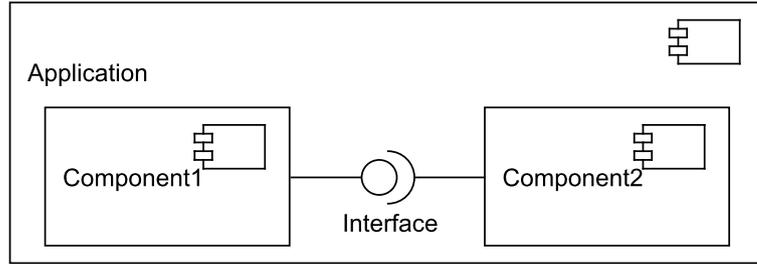


Figure 2.2: UML model of the example application

The component developer looks at the sent model diagram and finds out he needs to create one interface type for the **Interface** and three frames and architectures for those three components. He can do it by the similar *Cushion* script as that one in section 1.2 or by filling several forms in *SOFA 2 IDE*. He must manually fill all provided and required interfaces of the frames and all subcomponents and their connections of the composite architecture. When he starts to write classes for primitive architectures of those subcomponents, he finds out that the similar work as the **Component1** does is already done by another existing component that must be only adapted to the new interface. Therefore, he creates a composite architecture with this component and a delegating component as subcomponents and he also creates all resulting SOFA 2 entities. Finally, he sends all these entities back to the designer. The designer notes to its model with which SOFA 2 entities are the designed components and interfaces implemented and he should change the model according to the new implementation.

As we can see from this example, a lot of work such as defining composite architecture is somehow duplicated and the designer has no motivation to update the model. Let's think how this process can be improved on this example. A designer creates the same model as before. After that, he executes a tool that generates all necessary SOFA 2 entities that had to be created manually before. He sends instructions to a component developer, how the interface should be defined and how the primitive architectures should be implemented. The component developer creates classes for the interface type and for one of the subcomponents, then he also finds out that the **Component1** can be implemented alternatively. Therefore, he writes code of the delegating component and uploads all classes to the repository. The designer changes the implementation of the component in the model, assigns an already existing frame and architecture to the existing component by the tool and generates the rest entities.

This improved process leads to continual synchronization between a model and implementation and it automatizes a lot of actions. It is obvious that advantages of this approach will rise in larger applications. In order to realize this improvement, we must define and implement a connection between design and implementation views, which should be possible, since the manual transformation is really straightforward.

As we analyzed all parts of SOFA 2 implementation of the CBD process, we realize that it can be the best improved between the application design and the component creating parts. Moreover, this improvement will be a good practice

of the model driven development, since we can generate concrete implementation from an abstract model and we can synchronize changes between them. The rest of this thesis analyzes what kind of its realization is the best and how it can be implemented.

2.2 General strategy

If we want to define a connection between design and implementation views, we must first exactly specify them. The first half of the work is already done since the SOFA 2 meta-model is well defined (see section 1.1), so only the specification of the design view remains.

De facto standard in the field of object-oriented software engineering not only for modelling of component systems is the Unified Modeling Language (UML), managed by the Object Management Group (OMG) [21]. There are a lot of tools implementing UML component diagrams, but since *SOFA 2 IDE* is realized under the Eclipse platform, it is natural to use the implementation for this platform [31]. There are a lot of graphical editors for the UML in the Eclipse platform [29] (e.g. Topcased UML2 Editors [32]), so it has no sense to build a new one. It is reasonable that our specification will be independent of a used editor.

Two serious issues raise, when we want to use the UML as is. The first one is that the UML has a large-scale meta-model with many elements. It would be a hard and useless work to deal with all of them, since only few of them may have any significance regarding to our topic. The second one is that the UML in itself does not carry any semantics. Therefore, when a developer uses an element in a model, we must define what it means for further processing. On the other hand, there are a lot of situations whose definition is clear. For example, when a developer puts a component into another component, we can state that the second component will be a subcomponent of the first one. Since we are going to create a tool that automatically interprets semantics of a model, we need a very precise definition of which elements carry semantics and also how these elements can be connected so that it makes a sense. Based on these considerations, we can specify constrains that a UML model must fulfill to be usable for its transformation.

When we finish the specification of the design view, we can start assigning concrete semantics to each of selected elements in the UML component model and their relations. Since a model reflects an application design where some components and interfaces can be contained more than once, while SOFA 2 entities are rather related to component and interface types, we must specify a mapping between all components of the same type from the model and a SOFA 2 frame and architecture and between all interfaces of the same type and a SOFA 2 interface type (see figure 2.3). This mapping should be created automatically when a developer wants to generate new entities according to the model, but also it should be possible to edit it manually for easy reusing of already created entities. This mapping should also serve for changing of corresponding entities in case of changes in the model.

Note that both the design view and the mapping definitions described in the following chapters are not something given, since it is only one of many variants how semantics can be added to the UML meta-model. It can be changed to

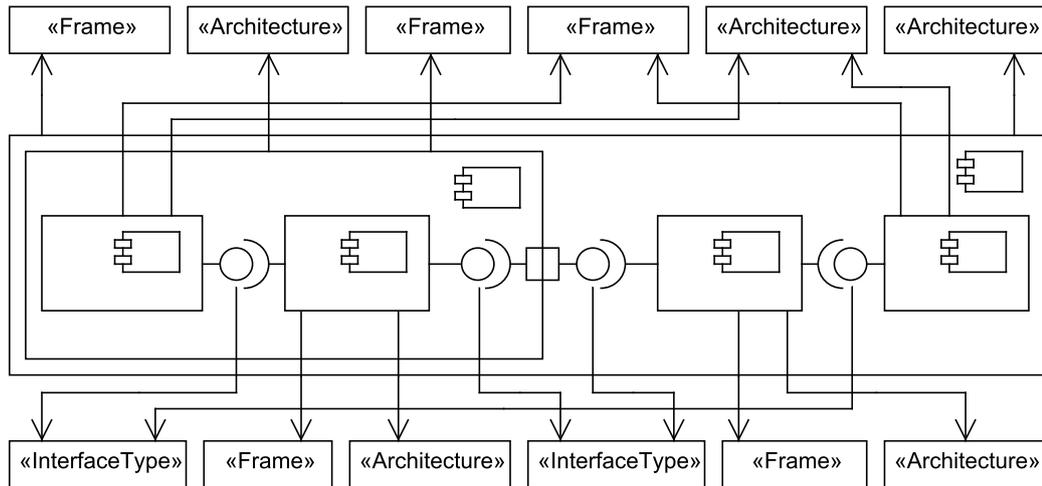


Figure 2.3: More than one UML element can be mapped onto one SOFA 2 entity

suit other requirements, so it is necessary to design a tool architecture for easy support of modifications.

Once the mapping is defined, a tool which covers all necessary processes can be developed. Since both a source UML model and SOFA 2 entities can be created and edited in the Eclipse environment and since the tool should cooperate with them, it is the best solution to develop it also as a set of plug-ins to the Eclipse platform.

From the previous analysis, we identify several requirements that this tool should satisfy:

1. *Fully automatic generation:* If a developer does not need any special properties of generation, a deployable application should be generated from a valid UML model fully automatically by few clicks.
2. *Form editor for manual generation:* If a developer wants to manage generation of entities, for example if he wants to map an element to an existing entity rather than to generate a new one, a form editor for these purposes should be available.
3. *Smart synchronization of changes:* When a UML model is changed, it should be possible to invoke an automatic correction of affected entities.
4. *Connection with other tools:* Since other tools already exist in the development process, the new tool should be fully compatible with them and should not duplicate their work. The main tool for a component developer, *SOFA 2 IDE*, is based on Eclipse, so this tool should be also fully integrated to the Eclipse platform.
5. *Sufficient information about errors:* When a user wants to transform an invalid model or something in a system goes wrong, the tool should fully inform him about arisen issues.

6. *Modular and platform-independent design:* If a redefinition of the mapping is needed, some well specified places in a system to change should exist. The mapping should be defined utmost platform-independent for an easy transformation for supporting other source models and other component systems.

3. Defining the UML subset

In section 2.2, we decided that the application design will be specified in the UML component model using the implementation for the Eclipse platform. We stated that it is impossible to give semantics to each UML element, so in this chapter, we somehow define a subset of UML elements together with restrictions to their relations according to the current topic. First, we bring a brief introduction to the UML, then we summarize possibilities how the specification can be done and finally we choose and further analyze one of them.

3.1 Brief introduction to the UML

The UML is a standardized visual modelling language used for analyzing, designing and implementing software systems as well as modelling processes. It offers a structural view of a system using objects, attributes, operations and relationships (a class diagram, a component diagram, ...) as well as a behaviour view by showing collaboration among objects and changes to their internal states (a state machine diagram, an activity diagram, ...). It is not completely visual since some information can (or sometimes must) be stored only in the model but not in the diagram itself. UML models can be easily exchanged among tools supporting them by using XMI interchange format [22] and may be automatically transformed to other representations by means of QVT-like transformation languages [20]. UML elements can have their relations constrained by the OCL [19] or they can be extended by UML profiles [34].

UML2 is an implementation of the UML 2.x meta-model for the Eclipse platform based on the Eclipse modeling framework (EMF) [28]. It does not provide UML modelling tools themselves, but there are a lot of tools that are compatible with this implementation or that even use it as their native format [29].

All UML elements are defined in the `org.eclipse.uml2.uml` package in a form of a multi-inheritance hierarchy of Java interfaces. Some interfaces represent only a collection of features that are used by other interfaces (for example, *NamedElement* adds the name property to elements and *Relationship* adds the related elements property), other ones represent complete elements visible in diagrams (for example, *Component* represents a component in the component diagram and it also implements the *NamedElement* interface). So each element has some features such as `Name` or `OwnedElement` that introduce some methods such as `getName()` or `allOwnedElements()` and represent its attribute or relation to other elements.

Let's look again at diagram 2.2 on page 14 of our sample application model and let's identify UML elements (see figure 3.1) and their important features in it. There are three instances of the *Component* element and one instance of the *Interface* element in the model. Their `Name` properties are set to `Application`, `Component1`, `Component2` and `Interface`. The `OwnedElement` property of the `Application` component is set to a list with other two components and the interface as its members, while they have this component in their `Owner` property. These two properties are defined as opposite to each other, so it suffices to define only one of them, the other one is set automatically. There are also

two instances of elements implementing the *Dependency* interface in the model (concretely instances of *InterfaceRealization* and *Usage* elements), which introduces the **Supplier** and **Client** features. In this case, the **Supplier** property in both elements is set to the element named **Interface**, while the **Client** property is set to the element named **Component1** or **Component2**.

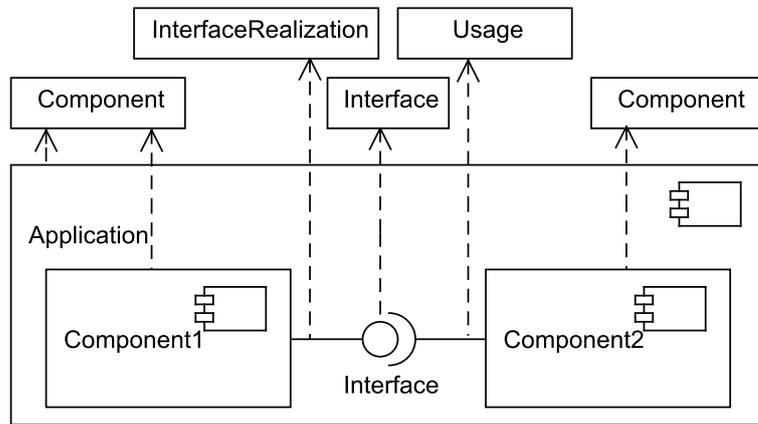


Figure 3.1: Example of UML elements

When someone looks at the diagram properly, he can think those dependencies are owned also by the **Application** component. However, they can also be owned by those subcomponents. The true is that it is unidentifiable from the diagram but from the model, it is. Fortunately in this case, it does not matter since the **Supplier** and **Client** features carry semantics. There are also elements that are not visible in the diagram but still exist in the model. The element that exists in each model is the *Model* element representing the model itself. Every other element in the model is directly or indirectly owned by this element, so it is always a root element. Another invisible but important element is the *Package* one, which enables us to separate elements into a hierarchy of packages.

Relations among elements can be constrained by some rules. Some constrains are defined internally. For example, the *InterfaceRealization* element can have its contract only with the *Interface* element, because its **Contract** property is of the *Interface* type. But we may want to specify other constraints which are useful in our semantics, such as a client of the *InterfaceRealization* element should be only one *Component* element. For this purpose, the Object constraint language (OCL) [19] is used to declaratively describe constraint rules. For example, this constraint is written as follows:

```
context InterfaceRealization
  inv: self.client->forAll(c: NamedElement |
    c.ocIsTypeOf(Component)) and self.client->size() = 1
```

It exactly means that for each instance of the *InterfaceRealization* element holds that all of its clients are components and it has precisely one client. As we can see, the OCL supports the propositional and predicate calculus with all features of all elements as well as queries on the element type. It also supports set operations, function and local variable definitions, if-else clauses etc.

When someone wants to customize UML meta-model for a particular domain or platform, he can use a UML profile [34] as a generic extension mechanism by defining custom stereotypes, tagged values, and constraints applicable to specific model elements. It is not possible to remove any internal meta-model constraint, but it is possible to add new ones.

3.2 List of possible definitions

After the introduction to some UML concepts, we can list possibilities how a useful subset of the UML meta-model can be specified:

1. *Definition of a new meta-model:* This option means a strict definition of possible elements and their relations in the EMF and creation of a new diagram editor for this meta-model. An advantage is full control over the models during their creation, an disadvantage is impossibility to use other than the newly created diagram editor.
2. *Definition of a UML profile:* This option means creation of a UML profile that will specify stereotypes and constraints applied to specific model elements. An advantage is that a designer will be constrained during creation of a model, an disadvantage is that any model without the profile will not be valid.
3. *Definition of UML model post-processing:* This option means that any UML model can be used, but only some elements will carry semantics for further processing and the rest will be ignored. Constraints of their relations will be applied when a model is going to be transformed. An advantage is that a designer is constrained only in relations of the key elements and the rest of the model is on his decision and that any diagram editor can be used, an disadvantage is that standard properties of elements may not be sufficient for required semantics.

As we can see, the main criterion in what these alternatives differ is freedom of a designer and a variety of valid models. The concepts go from strict definitions of models to the weaker ones. In our case, it is more important to support a wider variety of models and designer views rather than exactly lead a designer and have precisely defined semantics. Therefore, we will further analyze the third option.

3.3 Analysis of the chosen option

In the chosen option, we must define which types of UML elements are important for the transformation to SOFA 2 entities and which are going to be ignored. For the included elements, we must also specify semantics of some of their features.

Let's look again at diagram 3.1 on page 19, where some UML elements are already used for a description of a SOFA 2 application. It is meaningful that the *Component* element represents a component and the *Interface* element represents an interface. When a developer wants to specify that a component provides or requires an interface, he connects the component with the interface through

a relation. When the component provides the interface, it realizes it, so the best relation in the UML is the *InterfaceRealization* one. Again when the component requires the interface, it wants to use it, so the best relation in the UML is the *Usage* one.

We have defined how to specify a component frame in the UML, so a specification of a component architecture remains. It is obvious that a subcomponent will be also represented by the *Component* element. Therefore, we must only think about a representation of three kinds of connections between components (a connector, delegation and subsumption). When a subcomponent provides an interface by an *InterfaceRealization* relation and another subcomponent requires the same interface by a *Usage* relation, a connection between them can be simply realized such that both will be connected with the same instance of the *Interface* element.

In case of delegation, a subcomponent has a *Usage* relation to an *Interface* element inside its parent component, and the parent component has a *Usage* relation to another *Interface* element of the same type outside it. In this situation, we want to express by a relation that the inner request of the interface is delegated to the outer request. There are two issues to solve – what kind of a relation element should be used and how to declare which inner and outer requests are connected in case of multiple requests of the same type.

There are various possibilities of solving the first issue because this is not standardized. Some sources use the dependency relation, some of them use the association relation with or without a direction and some of them add the <<delegate>> stereotype to the relation [32, 15, 33, 35]. Therefore, we can choose the relation that fits the best to our demand. As we can see in figure 1.5 on page 7, we choose the *InterfaceRealization* relation because it is understandable and we do not introduce another UML element for processing. Therefore in the opposite situation, in case of subsumption, we use the *Usage* relation.

For the second issue, we can use the *Port* element which often groups relations of its component. Unfortunately, some editors of UML diagrams do not save information about relations connected to a port into a model [32], so it is not possible to find out which relations are connected together from the model. Moreover, since the SOFA 2 *Interface* entity, which is a member of provided and required interface collections in the SOFA 2 *Frame* entity, needs a name to specify, we can use this name as a reference which requests should be connected. Therefore, the *InterfaceRealization* relation connecting the parent component with the inner interface should have the same name as the *Usage* relation connecting the parent component with the outer interface to make delegation, and vice-versa to make subsumption (see figure 3.2).

Since more than one UML element in a model can be an instance of one component or interface type (see figure 2.3 on page 16), a developer must have a possibility to somehow declare that two components or two interfaces are of the same type. The simplest way to do it is to make their names the same. Therefore, each component or interface of the same name are considered as of the same type. Since the elements can be separated to a hierarchy of the *Package* elements, we can use this hierarchy for namespaces. Hence, we correct the definition, so only elements with the same name in the same package are considered as the same type.

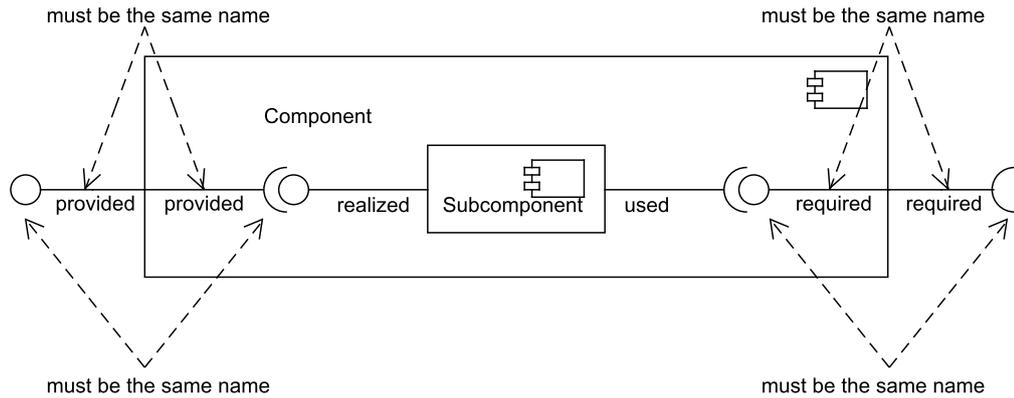


Figure 3.2: Delegation and subsumption connections must have the same name as the frame interfaces

There are two more issues to solve – assemblies and inheritance. How does a developer specify that a component should serve as an assembly? Since we do not support new stereotypes, we must use one of the features of the *Component* element available. Since the **Active** feature has no important semantics and it is boolean, we state that when this feature is set to **true** and other conditions are met, the component is considered as an assembly. And what about the inheritance relation between two components? Since it is senseless in case of a composite architecture and since the tool will not generate code in case of a primitive architecture, we ignore this relation.

3.4 Semantics of UML elements

Now, we can summarize all UML elements and their features with their semantics for further processing (note that opposite features such as **Owner** are not listed):

- *Model*: This is always a root element of all UML models, thus it is the same in our semantics.

Name: It is ignored and considered empty for further definitions.

Owned Element: It gives an access to all elements in the model.

- *Package*: All elements are organized in a hierarchy of packages. We use this element to define a namespace of owned elements. We define a full name for each named element as its name prefixed by the full name of its nearest package plus **.(dot)**. For example, the full name of the component named **Component** in the sub-package **Subpackage** in the package **Package** is **Package.Subpackage.Component**. Elements with the same full names are considered same for the further transformation.

Name: It is used for the above definition of the full name.

Owned Element: It gives an access to all elements in the package.

- *Component*: This element represents an instance of a component, hence it could own other components as its subcomponents in the architecture point of view and also other elements such as interfaces and relations.

Name: It is used for a unique identification among other components in the same package, for generation of corresponding SOFA 2 entities' names and as a name of a subcomponent in a composite architecture.

Owned Element: It gives an access to all elements owned by the component and all components in this list are considered as its subcomponents.

Active: Since a developer should have a possibility to somehow specify that he wants to generate also a SOFA 2 *Assembly* from a component directly in the model, this feature is used to this task. When it is set to `true` and other conditions are met, an assembly is generated.

- *Interface*: This element corresponds to an instance of an interface type. It cannot exist alone but only in a connection with neighbour components.

Name: It is used for a unique identification among other interfaces in the same package and for generation of the corresponding SOFA 2 *InterfaceType*'s name.

- *Interface realization*: This relation connects a component with an interface and it means that the component realizes that interface, so it is the provided interface.

Name: It is used for a name of the corresponding SOFA 2 *Interface*. An interface relation connected to a component from inside of the same name as a usage connected to the same component from outside is considered as delegation.

Client: This is the component that realizes the interface.

Supplier: This is the interface that is realized by the component.

Contract: This is a redundant feature with the same semantics as the **Supplier** one.

- *Usage*: This relation also connects a component with an interface and it similarly means that the component wants to use that interface, so it is the required interface.

Name: It is used for a name of the corresponding SOFA 2 *Interface*. A usage connected to a component from inside of the same name as an interface realization connected to the same component from outside is considered as subsumption.

Client: This is the component that uses the interface.

Supplier: This is the interface that is used by the component.

Once we list the elements that interest us, we should think over constraints of their relations that can be divided to the ownership relations and the other ones.

A definition of the ownership relation constraints is really straightforward. The top element is a *Model* that can own any count of *Components* directly or indirectly via a hierarchy of *Packages*. Since the component model is hierarchical,

these *Components* can own another components, interfaces, interface realizations and usages. As all top-level components owned directly by one package can have external relations with neighbour interfaces that will be therefore mixed although they logically do not relate, it can be handy to have an element that owns a top-level component and all its related elements and that will be then directly owned by the package. Since some UML editors [32] use the *Component* as the diagram element for this approach, we will use it as well, although it will not have any semantic sense. This hierarchy, displayed in figure 3.3, is so natural that it can be hard-coded without any trouble.

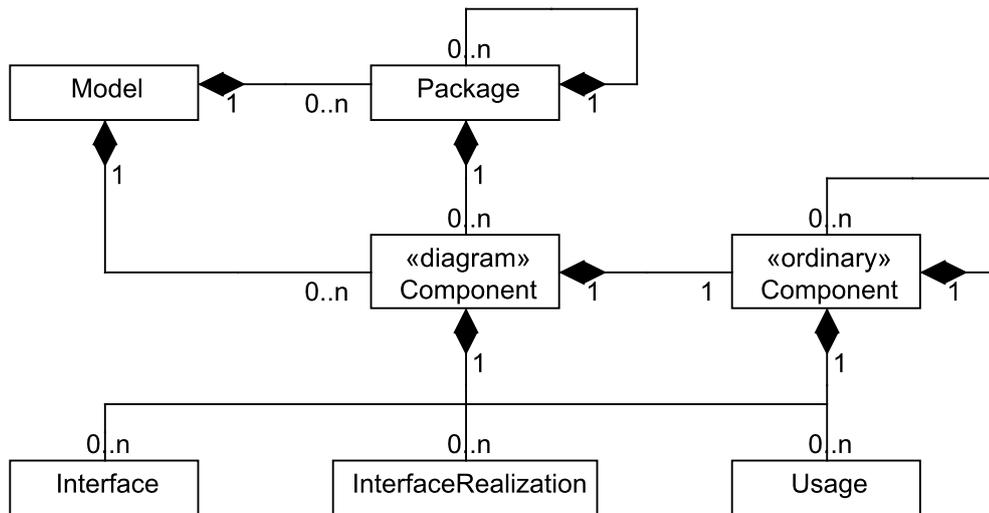


Figure 3.3: Ownership hierarchy of semantically important UML elements

It is a little bit complicated to define constraints of the other relations because there are many proper ways how to do it. Therefore, it often depends on concrete implementation, but we can find some constraints that are not implementation specific. For example, it is obvious that each interface realization or usage must connect exactly one component with one interface. Nevertheless, we list here some constraints that a model should meet to be valid that are meaningful for further processing. These constraints should be fully changeable in tool implementation. We use a verbal description as well as OCL statements because the OCL is a natural constraint language for the UML.

First, when we want to use the OCL, we must define some functions that help us with constraint specifications:

- We define a full name for each named element as its name prefixed by the full name of its nearest package plus *.(dot)* where the name of the model element is considered empty.

```

context NamedElement
def: fullName : String = if self.ocIsTypeOf(Model) then ''
  else if self.ocIsTypeOf(Package) then self.owner.
    ocIsTypeOf(NamedElement).fullName.concat(self.name).concat('.')
  else self.getNearestPackage().fullName.concat(self.name)
  
```

```
endif endif
```

- We define a function returning whether two dependencies have the same name of their supplier.

```
context Dependency
  def: sameSupplierName(other: Dependency) : Boolean =
    self.supplier.fullName = other.supplier.fullName
```

- We define component's frame usages/interface realizations as those ones that connect the component with an interface with the same owner as the component has (so the component and the interface are neighbours). We define component's delegation usages/interface realizations as those ones that connect the component with an interface directly owned by the component. We define a component as primitive when it does not have any subcomponents and as an assembly when its feature `Active` is set to `true`.

```
context Component
  def: frameUsages : Set(Usage) = Usage.allInstances()->select
    (client->includes(self) and supplier->forAll(s: NamedElement |
      client->exists(cl: NamedElement | s.owner = cl.owner)))
  def: frameInterfaceRealizations : Set(InterfaceRealization) =
    InterfaceRealization.allInstances()->select(client->includes(self)
      and supplier->forAll(s: NamedElement | client->exists
        (cl: NamedElement | s.owner = cl.owner)))
  def: primitive : Boolean = not self.allOwnedElements()->exists
    (e:Element | e.ocIsTypeOf(Component))
  def: delegationUsages: Set(Usage) = Usage.allInstances()->select
    (client->includes(self) and supplier->forAll(s: NamedElement |
      client->forAll(cl: NamedElement | s.owner = cl)))
  def: delegationInterfaceRealizations : Set(InterfaceRealization) =
    InterfaceRealization.allInstances()->select(client->includes(self)
      and supplier->forAll(s: NamedElement | client->forAll
        (cl: NamedElement | s.owner = cl)))
  def: assembly : Boolean = self.isActive
```

Afterwards, we can list all constraints. These constraints are listed with shortened comments in appendix B.1.

- Each interface realization and usage must be connected to exactly one component (the `Client` property) and one interface (the `Supplier` property).

```
context InterfaceRealization
  inv: self.contract->notEmpty() and self.supplier->size() = 1 and
    self.client->size() = 1 and self.supplier->forAll
      (c: NamedElement | c.ocIsTypeOf(Interface)) and
      self.client->forAll(c: NamedElement | c.ocIsTypeOf(Component))
context Usage
  inv: self.supplier->size() = 1 and self.client->size() = 1 and
    self.supplier->forAll(c: NamedElement | c.ocIsTypeOf(Interface))
    and self.client->forAll(c: NamedElement | c.ocIsTypeOf(Component))
```

- Each interface must be connected by at most one interface realization and/or some usages. This means that many components can call a method of the interface but only one can handle it.

```
context Interface
  inv: InterfaceRealization.allInstances()->exists(contract = self) or
      Usage.allInstances()->exists(u: Usage | u.supplier->includes(self))
  inv: InterfaceRealization.allInstances()->select
      (contract = self)->size() <= 1
```

- Each component does not own itself directly or indirectly and it is unique within its owner component. On the other hand, an instance of the same component can be in other places in the model.

```
context Component
  inv: not self.allOwnedElements()->exists(e: Element |
      e.oclassType(Component).name = self.name)
  inv: self.owner.ownedElement->select(e: Element |
      e.oclassType(Component).name = self.name)->size() = 1
```

- For each usage (except for that owned directly by a top-level diagram component) exists an interface realization on the supplier interface. This means that every interface inside a component that has a usage connection must also have an interface realization connection, so every call is properly handled.

```
context Usage
  inv: self.client->exists(owner.owner = self.getNearestPackage()) or
      InterfaceRealization.allInstances()->exists
      (self.supplier->includes(contract))
```

- Each usage/interface realization is either a frame usage/interface realization, or a delegation usage/interface realization of its client component. This means that its supplier interface is a neighbour of the client component, or it is directly owned by it. Therefore, it is not possible to have a connection from an outside interface directly to an inner subcomponent, a developer must use delegation/subsumption instead.

```
context InterfaceRealization
  inv: self.client->forAll(cl: NamedElement | cl.oclassType(Component).
      frameInterfaceRealizations->includes(self) or cl.oclassType
      (Component).delegationInterfaceRealizations->includes(self))
context Usage
  inv: self.client->forAll(cl: NamedElement | cl.oclassType(Component).
      frameUsages->includes(self) or cl.oclassType
      (Component).delegationUsages->includes(self))
```

- Two components with the same full name must have the same frame usages and interface realizations (the same full names and the same full names of supplier interfaces). This means that every instance of a component in the model must have the same outer connections.

```

context Component
  inv: let r1 : Set(Usage) = self.frameUsages in let p1 :
    Set(InterfaceRealization) = self.frameInterfaceRealizations in
  let prn : Bag(String) = r1.name->union(p1.name) in
  prn->size() = prn->asSet()->size() and
  Component.allInstances()->select(comp: Component | comp <> self
    and comp.fullName = self.fullName)->forall(comp: Component |
  let r2 : Set(Usage) = comp.frameUsages in let p2 :
    Set(InterfaceRealization) = comp.frameInterfaceRealizations in
  r1->forall(u1: Usage | r2->exists(u2: Usage |
    u1.fullName = u2.fullName and u1.sameSupplierName(u2))) and
  p1->forall(i1: InterfaceRealization | p2->exists
    (i2: InterfaceRealization | i1.fullName = i2.fullName and
    i1.sameSupplierName(i2)))

```

- Each component must not have any subcomponents, or must have properly connected each frame usage/interface realization with a corresponding (the same names and the same supplier interfaces) delegation interface realization/usage. This means that a component's architecture must be either primitive, or fully compatible with a component's frame.

```

context Component
  inv: self.primitive or let fi : Set(InterfaceRealization) =
    self.frameInterfaceRealizations in let fu : Set(Usage) =
    self.frameUsages in let di : Set(InterfaceRealization) =
    self.delegationInterfaceRealizations in
  let du : Set(Usage) = self.delegationUsages in
  fi->forall(i: InterfaceRealization | du->one(u: Usage |
    i.fullName = u.fullName and i.sameSupplierName(u))) and
  di->forall(i: InterfaceRealization | fu->exists(u: Usage |
    i.fullName = u.fullName and i.sameSupplierName(u))) and
  du->forall(u: Usage | fi->exists(i: InterfaceRealization |
    u.fullName = i.fullName and u.sameSupplierName(i)))

```

- Each component marked as an assembly (the `Active` property is set to `true`) must not have any frame usages or interface realizations. This means that an assembly is a top-level component that does not communicate outside. All instances of the same component must be either ordinary components, or assemblies.

```

context Component
  inv: not self.assembly or (self.frameInterfaceRealizations->isEmpty()
    and self.frameUsages->isEmpty())
  inv: Component.allInstances()->select(comp: Component | comp <> self
    and comp.fullName = self.fullName)->forall(comp: Component |
    comp.assembly = self.assembly)

```

Now, we have a complete definition of a valid UML model that can be used in further processing. We have implied semantics of some UML elements' features and we have also solved some semantical problems such as representing assemblies or connections in a composite architecture.

4. Mapping UML elements to SOFA 2 entities

Since the specification of the design view is compiled in section 3.4 and the implementation view is introduced in section 1.1, we can, in this chapter, define a mapping between them. As we mentioned in section 2.2, we need to somehow collect all components and interfaces from a UML model of the same type and to store a mapping of each of these types with corresponding SOFA 2 entities such as SOFA 2 *InterfaceType* in case of interfaces. For this purpose, the mapping meta-model is introduced primarily.

So that we can transform a UML model to a mapping model and then generate corresponding SOFA 2 entities, we must specify semantics of UML elements' features in detail according to properties of the SOFA 2 entities. Since a UML model can contain several instances of the same component or interface type and it represents a connection between a component and an interface as a UML relation element rather than a property of component's frame or architecture, it is inconvenient for direct generation of SOFA 2 entities. So a mapping model is, because it stores only the mapping between a unique component or interface type and corresponding SOFA 2 entities but no further information about how the component or the interface type looks like. Therefore, another model is created that prepares a UML model to a form that is better for generation of SOFA 2 entities. This model will be only temporary and also independent of a used component framework, because it does not provide any new information, it just offers another point of view to data in the UML model. This model is called a preparation model and it serves not only as a source model for generation of SOFA 2 entities but also as a reference model for finding whether an existing SOFA 2 entity is compatible with a component or an interface type found in the UML model. Finally, it can be easily transformed to a mapping model.

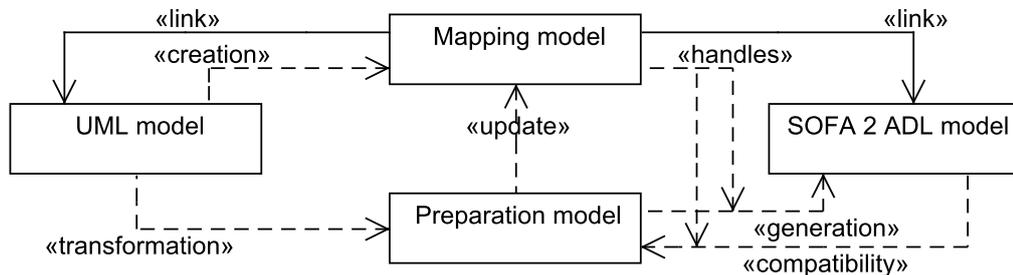


Figure 4.1: Relations among the used models

So after the introduction of the mapping meta-model, we introduce also the preparation meta-model. Then, we specify how a UML model is transformed to an instance of this meta-model and how this instance is transformed to an instance of the mapping meta-model. Finally, we define how new SOFA 2 entities are

generated according to an instance of the preparation meta-model and how the existing ones can be tested on compatibility with a component or an interface type. In figure 4.1, described relations among the used models are shown.

4.1 Mapping model

A mapping model should serve as a connection between elements in a UML model and SOFA 2 entities. First, let's do not solve how interfaces and components of common types will be found in a UML model and how this model will be created from it, but let's define how this model should look like.

Each unique interface or component type found in a UML model should be represented by an entity in a mapping model containing its name, which is useful for an easy manipulation with the entity by a user, which can be used for a name of automatically generated SOFA 2 entities and which serves as unique identification. Furthermore, an entity representing an interface type should store a mapped SOFA 2 *InterfaceType* and similarly, an entity representing a component type should store a mapped SOFA 2 *Frame* and SOFA 2 *Architecture*. Moreover, each component type marked as an assembly should store a mapped SOFA 2 *Assembly*. Finally, the whole mapping model should store a path to the source UML model and to a place where the SOFA 2 entities will be generated. For this purpose, a name of a file where the source UML model is stored and a name of a SOFA 2 project, which is a concept of *SOFA 2 IDE* where the SOFA 2 entities can be stored and manipulated locally outside the repository, can be used.

Since both the source UML meta-model and the SOFA 2 component model are based on the EMF, so the mapping meta-model is. Main advantages are easier model transformations and an automatically generated editor. Although the name of each model entity is generated from the source model and it is immutable, the mapping to SOFA 2 entities should be, on the other hand, fully and easily editable for reusing already existing entities.

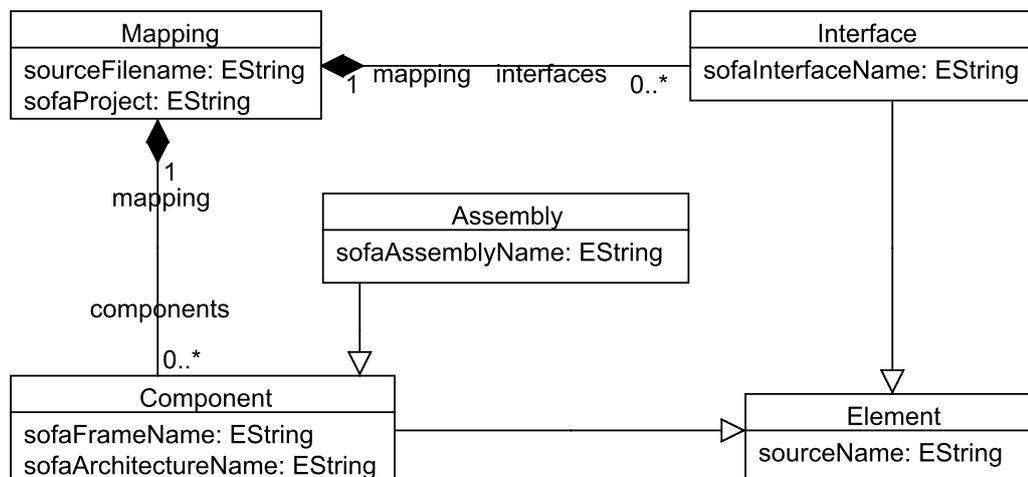


Figure 4.2: The mapping meta-model based on the EMF

The root element of the mapping meta-model is the *Mapping* element containing source UML model's filename and SOFA 2 project's name. This element owns any amount of interface and component types in two collections. The interface type is represented by the *Interface* element, while the component type is represented by the *Component* one. Both they have a common base class, the *Element* class, since it stores element's name. They also contain their specific mapping to SOFA 2 entities in a form of entities' names. Moreover, the *Assembly* element is derived from the *Component* one because in addition, it stores a name of a mapped SOFA 2 *Assembly*. In figure 4.2, there is the whole mapping meta-model.

4.2 Preparation model

The preparation meta-model should serve as a connection meta-model between the UML component model and the SOFA 2 component model for an easier transformation from the first one to the second one. Therefore, it should have common parts with both meta-models, although it should be independent of both of them.

Let's define a component concept for this meta-model. One element should represent one component type found in a UML model as the *Component* class in the mapping meta-model. On the other hand, it should not be divided to a frame and an architecture, so it should represent both of them as in the UML component model. Therefore, let's create the *Component* element, identified by its name.

A component element in a UML model is externally connected via relation elements to interface elements of some types that appear in a mapping model, while each SOFA 2 *Frame* owns its provided and required interfaces of some SOFA 2 *InterfaceTypes*, to which are the interface types from the mapping model mapped. Therefore, the *Component* element of the preparation meta-model should have also collections of provided and required interfaces. Their element is the *Interface* class that has its own name as the UML relation element or the SOFA 2 *Interface* have and that points to its *InterfaceType* corresponding to the *Interface* class in the mapping model and to the SOFA 2 *InterfaceType*.

A component architecture can be either primitive, or composite. In case of a composite architecture, a component has subcomponents of particular names, which are also instances of a component type. It is the same both in UML and SOFA 2 models, so we create the *Subcomponent* element identified by its name with an association to the *Component* element as its type. In the SOFA 2 component model, an architecture has also a collection of connections represented by two endpoints (see figure 1.5 on page 7). Each endpoint stores a subcomponent which the connection links (or none in case of a connection with its parent component) and one of its interfaces. In the UML component model, this is represented by relation and interface elements as external connections (see figure 3.2 on page 22). For easier generation of SOFA 2 *Architecture*, it should be represented in the first way in the preparation meta-model, so the *Component* class contains also a collection of the *Connection* class which is also a collection of the *Endpoint* class containing a name of a subcomponent and an interface.

Since a root element is needed, we create the *Preparation* element, which has

collections of interfaces, components and assemblies (an assembly has the same type as a component, they are distinguished only by their parent collection). The only constraint for elements of this model is that each component must not directly or indirectly own itself. In figure 4.3, there is the whole preparation meta-model.

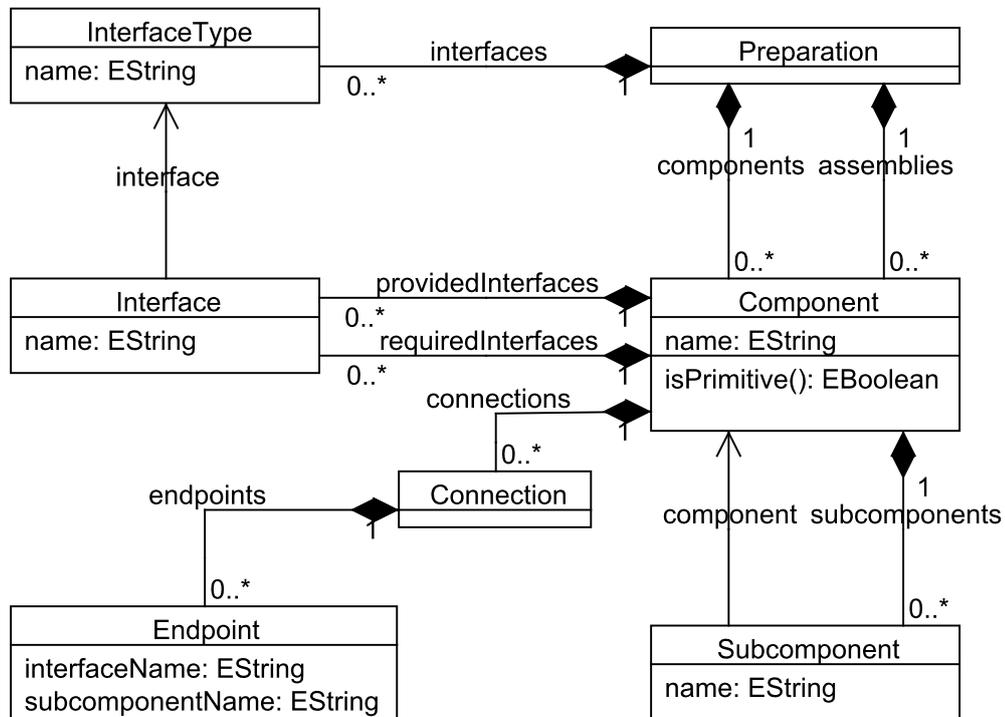


Figure 4.3: The preparation meta-model based on the EMF

Once we have defined the preparation meta-model, we can specify how its instance is created from a valid UML model from section 3.4. For better understanding of some transformation definitions, we also introduce some parts of QVT Operational [20] code that can realize them. For a complete definition of the transformation in QVT Operation code, see appendix B.2.

First, we map all UML *Interface* elements and all UML *Component* elements with the same full name (see section 3.4 for its definition) to one preparation *InterfaceType* class and one preparation *Component* class. The **name** property of these classes is set to that full name. For further definitions, we assume that if an instance of a component type in a UML model has a composite architecture, component type’s architecture will be defined from it (other instances have primitive architectures or the same composite architecture, since the model is valid).

Since the *InterfaceType* has no other properties, let’s define relations of the *Component* class. First, we define content of provided and required interfaces’ collections. For provided interfaces’ collection, we select all UML *InterfaceRealization* elements that have a corresponding instance of the *Component* element as a **client** and their *Interface* **supplier** has the same **owner** as that *Component* element (so the relation leads from the component to an external interface). For each of them, we create an instance of the *Interface* class from the preparation

meta-model, whose `name` is set to the name of the *InterfaceRealization* element and whose `interface` is set to the preparation *InterfaceType* corresponding to the supplier UML *Interface* element. For required interfaces' collection, the process is similar, only the *InterfaceRealization* element is replaced by the *Usage* element. Let's resume it in this piece of QVT Operational code:

```
// maps a UML InterfaceType or Usage to a preparation Interface
mapping UML::Dependency::dependency2Interface() : Preparation::Interface {
    name := self.name;
    interface := self.supplier->asSequence()->first().oclAsType(Interface).
        map(Preparation::InterfaceType);
}
// maps a UML Component to a preparation Component
mapping UML::Component::component2ComponentType() : Preparation::Component {
    ...
    providedInterfaces := self.getSourceDirectedRelationships()
        [UML::InterfaceRealization]->select(ir | ir.supplier->
            exists(owner = self.owner))->map dependency2Interface();
    requiredInterfaces := self.getSourceDirectedRelationships()
        [UML::Usage]->select(us | us.supplier->
            exists(owner = self.owner))->map dependency2Interface();
    ...
}
```

Afterwards, we define content of subcomponents' collection. We select all *Component* elements directly owned (`ownedElement`) by the corresponding instance of the *Component* element. For each of them, we create an instance of the *Subcomponent* class from the preparation meta-model, whose `name` is set to the full name of the source element and whose `component` is set to the preparation *Component* corresponding to the source element. Let's resume it in this piece of QVT Operational code:

```
// maps a UML component to a preparation Subcomponent
mapping UML::Component::component2Subcomponent() : Preparation::Subcomponent {
    name := self.fullName();
    component := self.late resolveone(Preparation::Component);
}
// maps a UML component to a preparation Subcomponent
mapping UML::Component::component2ComponentType() : Preparation::Component {
    ...
    subcomponents := self.ownedElement[UML::Component]->
        map component2Subcomponent();
    ...
}
```

Finally, we define content of connections' collection. As we know, we distinguish three types of connections – a connector, delegation and subsumption (see figure 1.5 on page 7). We get all connectors and all cases of delegation of a component when we select all *Usage* elements leading from one of its subcomponents to the *Interface* element owned directly by the component. Since the model is valid, there is exactly one *InterfaceRealization* element for each *Usage* element, leading to the same *Interface* element from another subcomponent (a connector) or from the component itself (delegation). We get all cases of subsumption of a component when we select all *Usage* elements leading from itself to one of its

directly owned *Interface* elements. Similarly, there is exactly one *InterfaceRealization* element for each *Usage* element, leading to the same *Interface* element from one of its subcomponents. It is obvious how an endpoint of a created connection looks like. Its `interfaceName` is set to the name of the corresponding *Usage* or *InterfaceRealization* element and its `subcomponentName` is set to the full name of the corresponding subcomponent, or to an empty string if it connects the component itself. Let's resume it in this piece of QVT Operational code:

```
// maps a UML Usage to a preparation Connection
mapping UML::Usage::usage2Connection(parent : UML::Component)
  : Preparation::Connection {
  let comp : UML::Component = self.client->asSequence()->first() in
  let interface : UML::Interface = self.supplier->asSequence()->first() in
  let otherRel : UML::InterfaceRealization =
    interface.getTargetDirectedRelationships() [UML::InterfaceRealization]->
    asSequence()->first() in
  let otherComp : UML::Component = otherRel.client->asSequence()->first() in
  endpoints := Set{
    object Endpoint {
      interfaceName := self.name;
      subcomponentName := if (comp != parent) then comp.fullName()
        else null endif; },
    object Endpoint {
      interfaceName := otherRel.name;
      subcomponentName := if (otherComp != parent) then otherComp.fullName()
        else null endif; } }
}
// maps a UML Component to a preparation Component
mapping UML::Component::component2ComponentType() : Preparation::Component {
  ...
  connections := self.getSourceDirectedRelationships() [UML::Usage]->select
    (us | us.supplier->exists(owner = self))->map usage2Connection(self)->
    union(self->ownedElement[UML::Component].getSourceDirectedRelationships()
    [UML::Usage]->select(us | us.supplier->exists(owner = self))->
    map usage2Connection(self));
  ...
}
```

Now, we have completely defined the transformation from a UML model to a preparation model, so a description of a transformation from a preparation model to a mapping model remains. A representation of a component and interface type is similar in both models. Only assemblies are represented a little bit differently but it is clear how to map a membership in a different collection to an instantiation of a different class. Moreover, there is only the `name` property that is mapped onto the `sourceName` property, other properties of the preparation classes are not mapped anywhere and other properties of the mapping classes are left blank because they are filled by a developer or on SOFA 2 entities generation. Therefore, the transformation is really straightforward.

The only issue to solve is updating an existing mapping model from a changed preparation model so that mapping to SOFA 2 entities from unchanged component and interface types will preserve. This can be solved by the following simple algorithm:

1. Create empty reference lists of interface, component and assembly names. They will serve as lists of objects found in the preparation model.

2. For each interface, component and assembly in the mapping collections:
 - (a) If an object of the same name is found in the corresponding preparation collection, add its name to its reference list.
 - (b) Else, remove it from its mapping collection, because it was deleted in the source model.
3. For each interface, component and assembly in the preparation collections:
 - (a) If its name is not found in its reference list, create the corresponding mapping class and add it to its mapping collection.
 - (b) Else, do nothing, because this object already exists in the mapping collection.

After described transformations, a mapping model is filled with all interface, component and assembly types found in its source UML model, so a developer can now manually assign existing SOFA 2 entities to them. Note that the names of these objects correspond to the names of objects in its preparation model, so they can be connected any time for getting necessary properties for further processing.

4.3 SOFA 2 entities generation and compatibility

Although a developer can manually assign existing SOFA 2 entities to a mapping model, the main benefit of a preparation model is their automatic generation. Since some SOFA 2 entities have pointers to other ones (e.g. a SOFA 2 *Frame* has provided and required interfaces of a SOFA 2 *InterfaceType*), data from a mapping model are also necessary for the generation. Moreover, since a developer can launch the generation for only one interface, component or assembly and since he works with the mapping model to which the generated entities will be assigned, the mapping model will manage the generation and will take necessary data from the preparation model. To remind how the target SOFA 2 entities look like, see figure 1.3 on page 6.

Let's start with generation of a SOFA 2 *InterfaceType* from a mapping *Interface*. This is quite easy since the target entity has only **name** and **signature** properties. It is obvious that both of them can have the same value which is get from the source **sofaInterfaceName** property if specified, or which is somehow generated from the source **sourceName** property if the previous one is left blank. Note that the preparation model is not used in this case. In figure 4.4, a connection among discussed properties is displayed.

Generation of a SOFA 2 *Frame* from a mapping *Component* is a little bit complicated because it must use the preparation model and also the mapping model for mapping of interface types. Therefore, the corresponding preparation *Component* is found first, then the frame **name** property is set or generated according to the **sofaFrameName** property. Finally, for each preparation *Interface* in component's collections of provided and required interfaces, a SOFA 2 *Interface* is created and added to frame's collection of provided or required interfaces. Its **name** property is set to the **name** property of the preparation *Interface*. Its

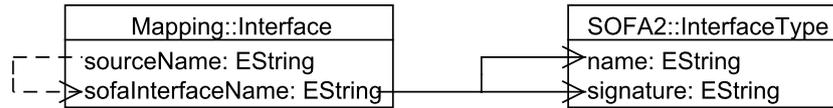


Figure 4.4: Generation of a SOFA 2 *InterfaceType* (a solid line represents the same value setting, a dash line represents value generation)

`interfaceTypeName` property, from which the `interfaceType` property is generated during a deployment, is get from the `sofaInterfaceName` property of the mapping *Interface* of the same name as the preparation *InterfaceType* got from the `interface` property of the preparation *Interface*. If the found mapping *Interface* has not assigned its SOFA 2 *InterfaceType* yet, it is generated as in the previous paragraph. In figure 4.5, a connection among discussed properties is displayed.

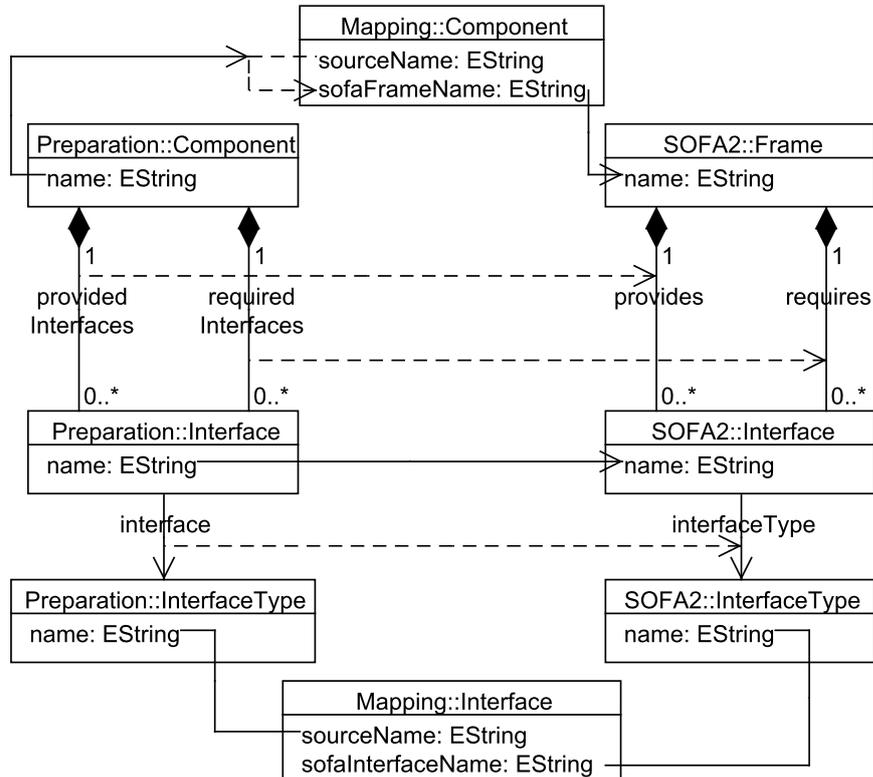


Figure 4.5: Generation of a SOFA 2 *Frame* (a solid line represents the same value setting / checking, a dash line represents value generation)

A SOFA 2 *Architecture* is also generated from a mapping *Component*. Its `name` property is set or generated according to the `sofaArchitectureName` property, its `frameName` property is set to the `sofaFrameName` property. If this property is blank, a SOFA 2 *Frame* is generated from the mapping *Component* as in the previous paragraph. If the preparation *Component* has a primitive architecture,

so the SOFA 2 *Architecture* will be and its `implementation` property will be the same as its `name` property. In case of a composite architecture, the `subcomponent` and `connection` collections must be filled. For each preparation *Subcomponent* in the collection of `subcomponents`, a SOFA 2 *Subcomponent* is created and added to the `subcomponent` collection. Its `name` property is set to the `name` property of the preparation *Subcomponent*. Then, the mapping *Component* with the same `name` as the preparation *Component* from the `component` property is found. If this component has not assigned a frame or an architecture yet, they will be generated. Finally, the `frameName` and `architectureName` properties of the created *Subcomponent* are set to the `sofaFrameName` and `sofaArchitectureName` of the found mapping *Component*. For each preparation *Connection* in the collection of `connections`, a SOFA 2 *Connection* is created and added to the `connection` collection. It is similar for each preparation *Endpoint* in the collection of `endpoints` and a SOFA 2 *Endpoint* in the `endpoint` collection. Their `interfaceName` and `subcomponentName` properties are connected to each other. In figure 4.6, a connection among discussed properties is displayed.

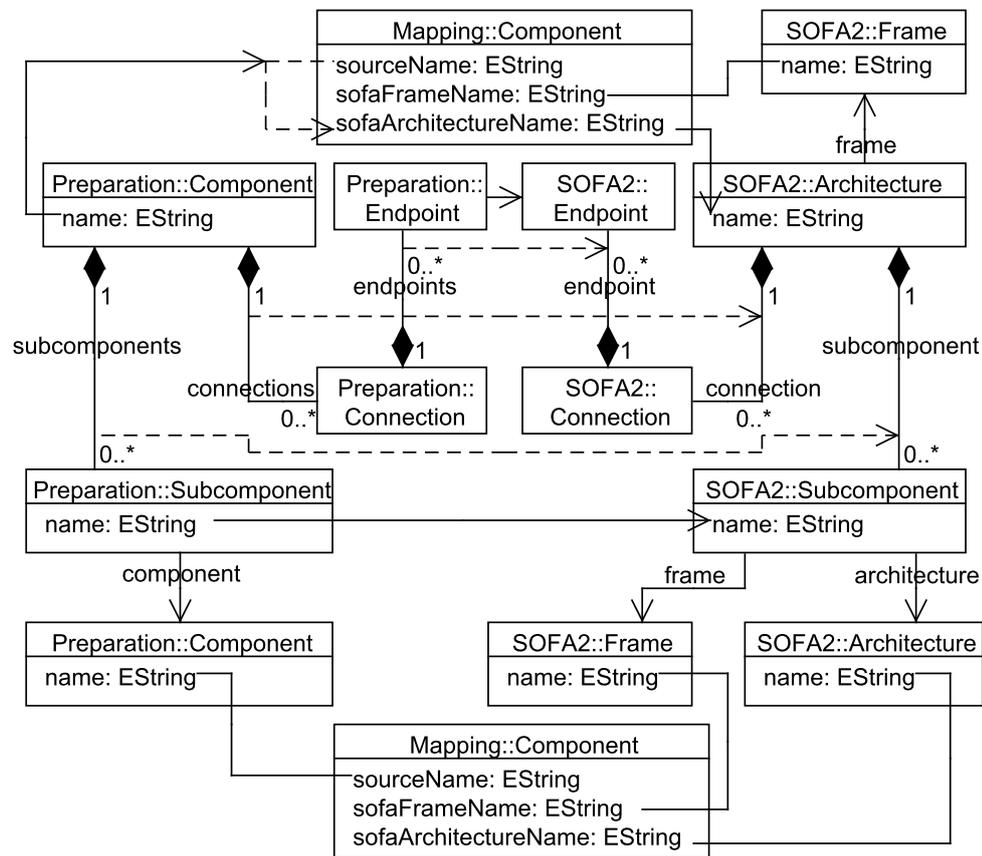


Figure 4.6: Generation of a SOFA 2 *Architecture* (a solid line represents the same value setting / checking, a dash line represents value generation)

A SOFA 2 *Assembly* is generated from a mapping *Assembly*. Its `name` property is set or generated according to the `sofaAssemblyName` property, while its `topLevelArchitectureName` property is set to the `sofaArchitectureName` property. If this property is blank, a SOFA 2 *Architecture* is generated from the

mapping *Assembly* as in the previous paragraph. The *subcomponent* collection is recursively filled by all subcomponents. For each preparation *Subcomponent* in the collection of *subcomponents*, a SOFA 2 *AssemblySubcomponent* is created and added to the *subcomponent* collection. Its *name* property is set to the *name* property of the preparation *Subcomponent*. Then, the mapping *Component* with the same *name* as the preparation *Component* from the *component* property is found. The *architectureName* property of the created *AssemblySubcomponent* is set to the *sofaArchitectureName* of the found mapping *Component*. Finally, the *subcomponent* property is filled recursively as described. In figure 4.7, a connection among discussed properties is displayed.

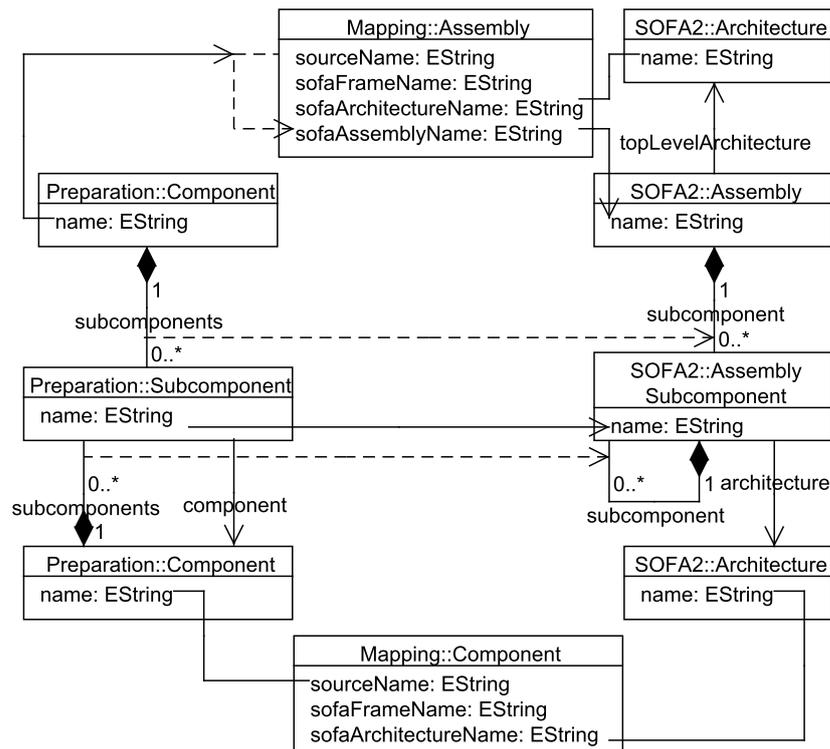


Figure 4.7: Generation of a SOFA 2 *Assembly* (a solid line represents the same value setting / checking, a dash line represents value generation)

Now, we have defined the generation of all SOFA 2 entities from the mapping and preparation models. Since a developer can change the source model after the generation of entities, some of them may become incompatible with the changed model. It would be uncomfortable to generate new entities because some properties of the old ones could be set manually, so a developer would have to set them again. Therefore, it should be possible to somehow repair existing entities to fit the model. Let's briefly think about repairing of all mentioned entities.

There is no property to repair in a SOFA 2 *InterfaceType*. Some provided and required interfaces in a SOFA 2 *Frame* may be added, changed or deleted as well as subcomponents in a SOFA 2 *Architecture* and both the *Interface* and *Subcomponent* classes have properties that can be set manually. All of these collections can be repaired by the following algorithm:

1. Create a set for unused interface / subcomponent names and fill it with all names of preparation collection's members.
2. For each interface / subcomponent in the SOFA 2 collection:
 - (a) If its name is not contained in the set of unused interface / subcomponent names, remove it from the SOFA 2 collection.
 - (b) Else, remove its name from the set of unused interface / subcomponent names.
3. For each remaining name in the set of unused interface / subcomponent names:
 - (a) Create a SOFA 2 interface / subcomponent according to the preparation one of the same name.
 - (b) Add it to the SOFA 2 collection.

Note that the SOFA 2 *Connection* and *Endpoint* classes have no properties to set manually, so the `connection` collection of the SOFA 2 *Architecture* does not need to be repaired, only its `frame` is generated if no one is set. The similar situation occurs in case of a SOFA 2 *Assembly* and its `topLevelArchitecture`. For the `subcomponent` property of a SOFA 2 *Assembly*, a modification of the previous algorithm is used that recursively calls itself in step 2.(b).

To be able to detect whether a SOFA 2 entity is incompatible with the source model, so it needs to be repaired, we must define what means that it is compatible. Moreover, when a developer wants to assign an existing SOFA 2 entity to the mapping model, we also need to know if it can be assigned. Let's define compatibility of all mentioned entities.

Every SOFA 2 *InterfaceType* is compatible with every mapping *Interface*. A SOFA 2 *Frame* is compatible with a mapping *Component*, if and only if the corresponding preparation collections of provided and required interfaces are compatible with those SOFA 2 ones. A SOFA 2 *Architecture* is compatible with a mapping *Component*, if and only if its `frame` is also compatible with it and the corresponding preparation collections of subcomponents and connections are compatible with those SOFA 2 ones. A SOFA 2 *Assembly* is compatible with a mapping *Assembly*, if and only if its `topLevelArchitecture` is also compatible with it and the preparation collection of subcomponents is compatible with that SOFA 2 one.

Let's introduce an algorithm resolving whether preparation and SOFA 2 collections of some entities are compatible. Note that interfaces and subcomponents are identified by their names while connections and endpoints are identified by their indexes in their collections, so we must distinguish these two cases in two little different algorithms. The algorithm with names looks as follows:

1. Create a set for unused interface / subcomponent names and fill it with all names of preparation collection's members.
2. For each interface / subcomponent in the SOFA 2 collection:
 - (a) If its name is not contained in the set of unused interface / subcomponent names, return `false`.

- (b) Remove its name from the set of unused interface / subcomponent names.
 - (c) Find the corresponding interface type / component in the mapping model.
 - (d) If its interface type / frame or architecture name does not equal to the corresponding name in the mapping model, return **false**.
3. Return **true** if the set of unused interface / subcomponent names is empty, or **false** if it is not.

In case of a SOFA 2 *AssemblySubcomponent* collection, we recursively call this algorithm after step 2.(d) and an empty SOFA 2 collection is compatible with every preparation collection because a SOFA 2 *Assembly* does not have to be defined in a whole subcomponent hierarchy. The algorithm with indexes looks as follows:

1. Create a set for unused connection / endpoint indexes and fill it with all indexes of preparation collection's members.
2. For each connection / endpoint in the SOFA 2 collection:
 - (a) For each index in the set of unused connection / endpoint indexes, try whether the connection / endpoint of this index in the preparation collection is compatible with a SOFA 2 connection / endpoint.
 - (b) If a compatible connection / endpoint is found, remove its index from the set of unused connection / endpoint indexes.
 - (c) Else, return **false**.
3. Return **true** if the set of unused connection / endpoint indexes is empty, or **false** if it is not.

A SOFA 2 *Connection* is compatible with a preparation *Connection*, if and only if their collections of endpoints are compatible. A SOFA 2 *Endpoint* is compatible with a preparation *Endpoint*, if and only if their subcomponent and interface names equal.

Now, we have a complete definition of all models and transitions among them necessary for mapping UML elements to SOFA 2 entities. We have introduced two meta-models that help us with this approach – the mapping meta-model, which instance stores current assignment of UML elements to SOFA 2 entities, and the preparation meta-model, which serves as a platform-independent interlink between the source and target models. We have defined how to automatically generate SOFA 2 entities from a UML model and repair them in case of source model changing. We are able to determine whether an existing SOFA 2 entity is compatible with a UML element, so it can be assigned to it. Therefore, we have laid a theoretical base for implementation of a tool providing all of demanded features.

5. SOFA 2 UML tool implementation

In this chapter, we analyze how to implement the *SOFA 2 UML* tool which satisfies all requirements listed in section 2.2. We have decided that the tool should be developed as a set of plug-ins to the Eclipse platform. However, we should analyze a user point of view on this tool first, then we proceed to a division of functionality into Eclipse plug-ins. The main part of user work with this tool is a specification of mapping between UML elements and SOFA 2 entities. Therefore, we first describe the editor of a mapping model both from a user and tool developer perspective. During this description, we find out that some features such as the model transformations should be externalized to other plug-ins that are introduced in the next section. Finally, we focus on integration of this tool with *SOFA 2 IDE*.

5.1 Editor of the mapping model

A user starts to use the tool when he has finished a UML model and when he wants to generate SOFA 2 entities from it. Since SOFA 2 entities are managed in a SOFA 2 project, the tool should provide an extension to the pop-up menu opened above UML model files, which enables us to create a new SOFA 2 project with a new mapping model connected to the source UML file or to bind this new mapping model to an existing project. The created mapping model contains all interfaces, components and assemblies found in the UML model but there is no mapping to SOFA 2 entities. Therefore, the form editor of the mapping model is opened providing a manual specification of mapping as well as automatic generation of them.

The editor of the mapping model should look like the editors of SOFA 2 entities so that a user has a common way of working with them. Therefore, we divide the editor to five pages. The first page is an overview page so it provides editing of properties of the whole mapping model such as the source UML model file or the target SOFA 2 project. It also contains possibilities to refresh the mapping model in case of source model changing and to automatic generation and/or repair of all SOFA 2 entities. The following three pages contain a list of interfaces, components or assemblies found in the source model. When a user chooses an element from the list, names of its mapped SOFA 2 entities are shown. A user can edit the name manually and then he can generate or repair the entity of the specified name, or he can choose a compatible entity from a list to assign. He can also open the corresponding editor of an existing entity. Moreover, there is a possibility to show only those elements in the list that have none or erroneous mapping. In the last page, there is a text editor for editing the file with the mapping model directly. In figure 5.1, the coverage of the mapping model by this editor is displayed.

After the brief description of the mapping editor from the user point of view, let's describe it from the tool developer point of view. The mapping model is created using the EMF tools first. The final diagram resembles that one in figure

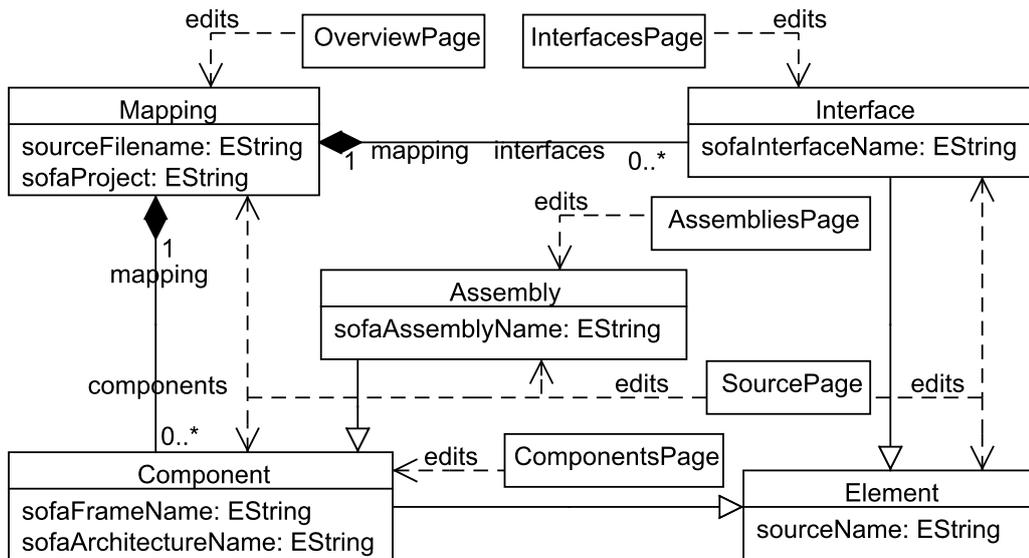


Figure 5.1: Coverage of the mapping model by the editor

5.1. Then, model, edit and editor code is generated, so three cooperating Eclipse plug-ins are created. Although the default editor is generated for this model, it does not satisfy our requirements so another editor described above must be created. The new editor is represented by the `MappingFormEditor` class deriving from the `org.eclipse.ui.forms.editor.FormEditor` class which is a base class of multi-page form editors. Its content is similar to the generated editor class and the form editor of SOFA 2 ADL files [13]. It differs mainly in the `addPages()` method that creates and adds pages of the editor.

There is a common base class for the first four pages called `MappingFormPage` providing an access to common classes related to the EMF infrastructure and to the extensions (see section 5.2). All its descendants create a part of the form using SWT widgets [30]. Therefore, the list of source elements or SOFA 2 entities is represented by the `Table` widget, the name editor consists of the `Label` and `Text` widgets and the action leading to refreshing the model or to automatic generation of entities is shown as the `Button` widget. For grouping of related widgets, the `Section` widget is used.

The main advantage of using the SWT is its support of data binding between its widgets and properties in EMF models. When a widget is bound to a model property, its content is automatically filled with a value of the property and also changing of its content is propagated back to the property. Let's have an example of the interface type name binding. First, we fill the interface list by interfaces from the model:

1. We specify the feature of the mapping model representing the mapping collection of interfaces. This constant is generated during the model code generation:

```
EStructuralFeature feature = MappingPackage.Literals.MAPPING__INTERFACES;
```
2. We create an EMF list property from this feature and the current editing

domain initialized in the `MappingFormPage` class:

```
IEMFEditListProperty listProp = EMFEditProperties.list(domain, feature);
```

3. We create an observable list that observes the list property on the current instance of the `mapping` model:

```
IObservableList list = listProp.observe(mapping);
```

4. We set this list as input of the previously initialized viewer that shows the interface list:

```
viewer.setInput(list);
```

Then, we observe the current selection of the interface list:

```
IObservableValue selection = ViewersObservables.observeSingleSelection(viewer);
```

Finally, we observe its interface type name property and we bind it to the corresponding `Text` widget:

1. We specify the feature of the mapping model representing the interface type name property:

```
feature = MappingPackage.Literals.INTERFACE__SOFA_INTERFACE_NAME;
```

2. We create an EMF value property from this feature:

```
IEMFEditValueProperty prop = EMFEditProperties.value(domain, feature);
```

3. We create an observable value that observes the property on the current selection:

```
IObservableValue value1 = prop.observeDetail(selection);
```

4. We create a text widget value property:

```
IWidgetValueProperty textProp = WidgetProperties.text(SWT.Modify);
```

5. We create an observable value that observes the property on the corresponding widget:

```
IObservableValue value2 = textProp.observe(interfaceTypeNameField);
```

6. We bind these two values in the current `context` initialized in the base class:

```
context.bindValue(value2, value1);
```

It is not necessary to have the binding values of the same type. For example, we can bind the `visible` property of a widget to a textual property of a model element if we specify a converter class with a method that returns `true` when the text is not an empty string.

Another useful feature of SWT list widgets is that their content got from the model can be filtered by any condition without reloading source data. For example, we have a list to which all found SOFA 2 *Frames* are being continually loaded, but we want to show only those ones that are compatible with the selected component. Therefore, we add a selection change listener to the component list viewer that removes all list filters first. Then, it adds a filter class with a method that is called on each member of the list and that returns `true` when the currently selected component is compatible with the checked SOFA 2 *Frame*:

```

componentsViewer.addSelectionChangedListener(new ISelectionChangedListener() {
    @Override
    public void selectionChanged(SelectionChangedEvent event) {
        final Component selectedComp = (Component)
            ((IStructuredSelection)event.getSelection()).getFirstElement();
        framesViewer.resetFilters();
        if (selectedComp != null) {
            framesViewer.addFilter(new ViewerFilter() {
                @Override
                public boolean select(Viewer viewer, Object parent, Object element) {
                    return compatibility != null ? compatibility.isCompatible
                        (selectedComp, (Frame)element) : true;
                }
            });
        }
    }
});
}
}
});

```

The last page of the editor shows textual content of the mapping model file with a text color distinction of comments, elements and texts and XML scanner for propagation of changes back to the model. This editor is taken from the SOFA 2 ADL form editor implementation [13]. In figure 5.2, there is a diagram of classes related to the mapping model form editor.

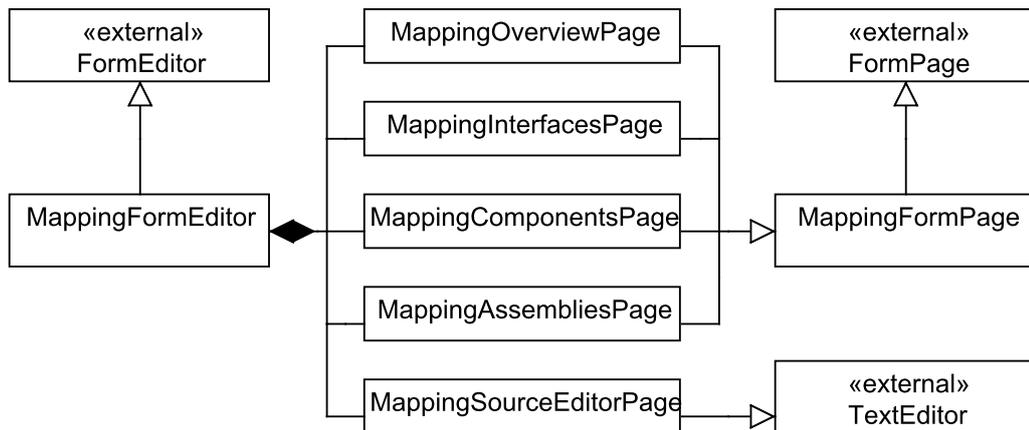


Figure 5.2: Diagram of classes related to the mapping model form editor

There are several undefined places in the mapping editor yet. What happens when a user invokes refreshing of the mapping model or generation of SOFA 2 entities? Where are the lists of SOFA 2 entities loaded from and how are they filtered? As we note in section 2.2, these questions can be answered in many ways so implementation of these parts of the editor should be fully changeable. For this purpose, the concept of extension points exists in the Eclipse platform. Through its extension point, a plug-in can be extended by other plug-ins. It can ask on runtime which plug-ins is extended with. In the simplest case, an extension point is an interface which other plug-ins implement in one of their classes. Therefore, the extended plug-in can call methods implementing interface's methods.

Let's summarize which extension points are needed for the mapping editor. First, when a mapping model is refreshed, its source model must be validated

and then transformed to the mapping model. Then, when a user wants to generate or repair SOFA 2 entities from the source model, an extension must specify how it is done. Finally, there must be an extension that searches already existing SOFA 2 entities and another one that decides whether they are compatible with a selected source element. Since generation and repair of SOFA 2 entities are similar tasks, we can identify five extension points shown also together with their current extension plug-ins in figure 5.3 – *validator*, *transform*, *compatibility*, *generator* and *searcher*. The first three extension points are related to the model transformation, so they are specified in section 5.2, while the last one is related to *SOFA 2 IDE*, so it is specified in section 5.3. The fourth one is related to both subjects, so it is specified in both sections. All current extensions can be obtained on runtime via the `Provider` class.

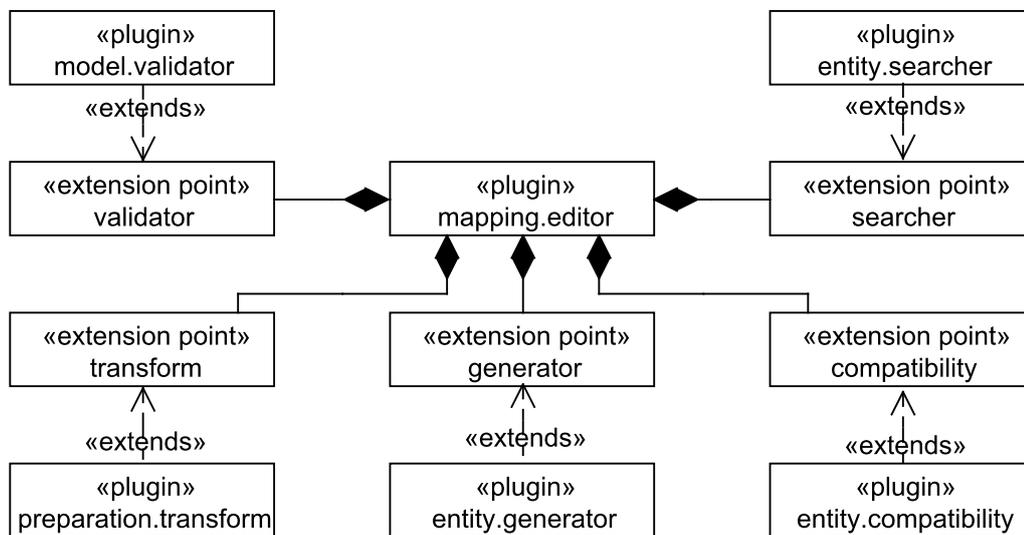


Figure 5.3: Extension points of the mapping model form editor with their current implementation

5.2 Implementation of model transformations

Let's define the *validator* extension point first. It can be extended by any amount of plug-ins with a class implementing the `IValidator` interface. It requires two more properties to specify – *obligatory* and *type*. The first boolean property determines whether the whole validation fails, when the validation done by this plug-in fails. The second string property indicates which extension of source model files it supports. The `IValidator` interface is simple since it contains only one method that requires a source model file and returns whether it is valid or not:

```

public interface IValidator {
    boolean validate(IFile sourceFile);
}

```

The only currently developed plug-in extending the *validator* extension point is called `model.validator`, it is obligatory and it validates UML model files. Its implementation consists of checking the OCL constraints introduced in section 3.4 on each corresponding element in the UML model. It uses the OCL implementation from the `org.eclipse.oc1.uml` plug-in and it loads the OCL constraints from a plug-in file so they can be easily changed without any code writing. In case of failing of a constraint, it marks the source file with a corresponding error message taken from another plug-in file for each inconsistency.

The second extension point *transform* can be also extended by any amount of plug-ins with a class implementing the `ITransform` interface for a change. It requires a specification of one property, `types`, indicating which extensions of model files (separated by commas) it transforms. The `ITransform` interface also contains only one method that requires a source model file and a root object of a target mapping model and returns whether the transformation is successful:

```
public interface ITransform {
    boolean transform(IFile sourceFile, Mapping mapping);
}
```

The `preparation.transform` plug-in extends this extension point. It uses the preparation meta-model and its generation from the UML component model and its transformation to the mapping meta-model described in section 4.2. First, it generates a preparation model from a source model and then, it updates a mapping model according to it. It supports all source models that the preparation model does.

Since the preparation meta-model is used also in other tasks like SOFA 2 entities generation and compatibility, it is cached and regenerated only when an instance of the mapping meta-model that uses it is refreshed. The preparation meta-model is also created using EMF tools as the mapping meta-model according to figure 4.3 on page 31, but only model code is generated. Its constraint is checked after its generation from a source model that is also solved by an extension point. It is called *generation* and it can be extended by any amount of plug-ins with a class implementing the `IPreparationGeneration` interface. It requires a specification of the property `type` representing an extension of source model files from which it generates a preparation model. The `IPreparationGeneration` interface contains one method requiring a source model file and returning a generated preparation model if successful:

```
public interface IPreparationGeneration {
    Preparation generate(IFile sourceFile);
}
```

The only currently supported source model is the UML component model in the extension plug-in `model.transform`. Although this model transformation is described in QVT Operational language in section 4.2, the used implementation is written in Java code because it is more effective. Note that the implementation in QVT is also accessible (see appendix A). The Java implementation is based on recursive scanning of the model, packages, diagrams and components for UML elements summarized in section 3.4. When some of these elements is found, the preparation model is updated according to relations to other elements. In figure

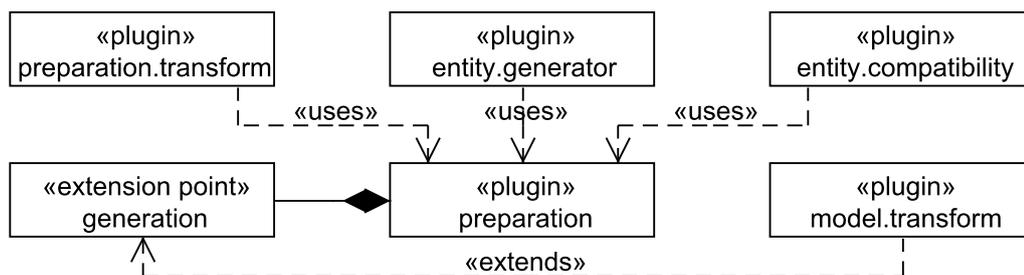


Figure 5.4: Extension point of the preparation meta-model plug-in with its current implementation

5.4, there is a diagram of relations among mentioned plug-ins and extension points.

We have defined the *validator* and *transform* extension points, so we can specify what happens when a user invokes refreshing of a model in the mapping form editor. First, all markers on its source file are removed so the extensions can add the current ones. Then, all *validator* extensions for the current source file type are found and their `validate()` method is invoked. If an extension with the `obligatory` property set to `true` returns `false`, the refresh procedure is stopped with an error message informing that the source model is not valid and the source file is marked with detail error messages. Otherwise, all *transform* extensions for the current source file type are found and their `transform()` method is invoked. If one method returns `false`, another error message raises.

The third extension point *compatibility* can be extended only by one plug-in with a class implementing the `ICompatible` interface. This interface contains four methods that require a mapping *Interface*, *Component* or *Assembly*, and a corresponding SOFA 2 entity (an *InterfaceType*, *Frame*, *Architecture* or *Assembly*) and return whether they are compatible:

```

public interface ICompatible {
    boolean isCompatible(Interface mapIface, InterfaceType sofaIface);
    boolean isCompatible(Component mapComp, Frame sofaFrame);
    boolean isCompatible(Component mapComp, Architecture sofaArch);
    boolean isCompatible(Assembly mapAssembly,
        org.objectweb.dsrp.sofa.adl.Assembly sofaAssembly);
}

```

This extension point is used as in the example in section 5.1 and it is extended by the *entity.compatibility* plug-in. It uses a preparation model generated from a source model because an element from the preparation model and a SOFA 2 entity can be easily checked for their compatibility according to the algorithm described in section 4.3.

The fourth extension point *generator* can be extended also only by one plug-in with a class implementing the `IGenerator` interface. This interface contains four generation methods that require a mapping *Interface*, *Component* or *Assembly*, and a name, and return a corresponding new SOFA 2 entity with the specified name and properties according to a source model. It also contains four repair

methods requiring a mapping model element and a corresponding SOFA 2 entity which properties are repaired according to a source model. Moreover, its method `getEntityFile()` should return a file where a SOFA 2 entity with a specified name in a specified mapping model is/will be stored. Let's list all of its methods here:

```
public interface IGenerator {
    InterfaceType generateInterfaceType(Interface mapIface, String name);
    Frame generateFrame(Component mapComp, String name);
    Architecture generateArchitecture(Component mapComp, String name);
    org.objectweb.dsrg.sofa.adl.Assembly generateAssembly
        (Assembly mapAssembly, String name);
    void repairInterfaceType(Interface mapIface, InterfaceType sofaIface);
    void repairFrame(Component mapComp, Frame sofaFrame);
    void repairArchitecture(Component mapComp, Architecture sofaArch);
    void repairAssembly(Assembly mapAssembly,
        org.objectweb.dsrg.sofa.adl.Assembly sofaAssembly);
    IFile getEntityFile(Mapping mapping, String entityName);
}
```

The `entity.generator` plug-in extends this extension point and it also uses a generated preparation model and the algorithms introduced in section 4.3. There are two implementation problems to solve. The first one is SOFA 2 entity name generation when a user does not provide any own name. It is solved according to the naming convention used in the SOFA Shop example [13]. We just add the letter **I** / **F** / **A** in case of a SOFA 2 *InterfaceType* / *Frame* / *Architecture* behind the last dot (or at the beginning if no dot is found) of the name of the mapping element which the entity is generated from. In case of an *Assembly*, the original name is left. For example, when we generate a *Frame*, an *Architecture* and an *Assembly* for the mapping *Assembly* named `sample.Application`, their names will be `sample.FApplication`, `sample.AApplication` and `sample.Application` in the same order. The second problem is SOFA 2 entity resource creation. Since this problem is related to *SOFA 2 IDE*, we leave its solution along with implementation of the `getEntityFile()` method to section 5.3.

Now, we can generate a new SOFA 2 entity from a mapping model element, or we can repair an existing one if it is not compatible with a source model. When a user wants to generate all SOFA 2 entities at once, he pushes the **Generate** button from the `MappingOverviewPage` class. First for each mapping *Interface*, its SOFA 2 *InterfaceType* is searched. If no one is found, it is generated, otherwise, it is repaired. Then for each mapping *Component* and *Assembly*, its SOFA 2 *Frame*, *Architecture* and, in the second case, also its SOFA 2 *Assembly* are searched and generated or repaired in this order. If an error occurs during the generation, it is stopped and a user is informed what happens.

5.3 Integration with SOFA 2 IDE

The *SOFA 2 UML* tool is integrated with the *SOFA 2 IDE* tool by several ways. It uses its concept of storing of SOFA 2 entities outside the repository, so for generation of new entities or for searching of existing ones, it uses its classes and their methods. It also uses its form editors of ADL files for SOFA 2 entities editing.

Let's describe the used *SOFA 2 IDE* classes first. A SOFA 2 project is an Eclipse platform project with the *SOFA2ProjectNature* nature and it is implemented by the *SOFA2Project* class. It contains a list of its SOFA 2 entities in the XML file `.sofa2` accessed by the *SOFA2ProjectConfiguration* class and stored in its folder. Each entity has its name, version and type there and it is represented by a subdirectory of the same name. This subdirectory, represented by the *SOFA2Folder* class implementing the *ISOFA2LocalResource* interface, contains the `adl.xml` file where the SOFA 2 ADL representation of the entity is stored. Content of this file is represented by the *SOFA2ADL* class from which the particular SOFA 2 entities can be taken. Each SOFA 2 entity and this covering class can be created by the *SOFA2ADLFactory* class. When the SOFA 2 ADL representation of a new entity is going to be created locally, this entity should be also created in the SOFA 2 repository. For this purpose, the *SOFA2NewCommand* class serves, which creates the entity in both places according to the provided instance of the *SOFA2Resource* class storing entity's name, version and type.

When a user wants to create a mapping model from a UML model file, he must also create or bind a SOFA 2 project. For creation of a new SOFA 2 project, the *SOFA2ProjectWizard* wizard is used, which leads a user through setting of all information necessary for project creation. For binding of an existing SOFA 2 project, the *WorkspaceResourceDialog* dialog is used with the *Sofa-ProjectViewerFilter* filter applied, which enables a user selection of a project in a current workspace. Both actions are accessible from the pop-up menu opened on a UML model file thanks to deriving from the *IObjectActionDelegate* interface and extending of the `org.eclipse.ui.popupMenus` extension point by the `model.generator` plug-in where they are stored.

From section 5.2, two issues remain to solve in the `entity.generator` plug-in. Since we have described the structure of the SOFA 2 project folder, implementation of the `getEntityFile()` method is straightforward. We just get a workspace root and we find a folder of a current SOFA 2 project stored in a mapping model there. In this folder, we find its subdirectory of the same name as the searched entity and we return contained `adl.xml` file. When we want to create a new SOFA 2 entity, we first create an instance of the *SOFA2Resource* class with its name, type and an empty version. We use this instance for creation of an instance of the *SOFA2NewCommand* command, where the current project is also needed, and we execute it. If no error occurs, we create an instance of the *SOFA2ADL* class to which we put the particular SOFA 2 entity class. Finally, we create and save an XML resource with this instance in a file returned from the `getEntityFile()` method.

The last extension point of the mapping model form editor, *searcher*, can be extended only by one plug-in with a class implementing the *ISearcher* interface. This interface contains four methods for getting a list of available SOFA 2 entities of a certain type for a specified mapping model and four methods for getting a specific SOFA 2 entity of a given name and type. Moreover, there is a method returning whether the source of these entities is valid:

```
public interface ISearcher {
    List<InterfaceType> getAvailableInterfaceTypes(Mapping mapping);
    List<Frame> getAvailableFrames(Mapping mapping);
    List<Architecture> getAvailableArchitectures(Mapping mapping);
}
```

```

List<Assembly> getAvailableAssemblies(Mapping mapping);
InterfaceType getInterfaceType(Mapping mapping, String name);
Frame getFrame(Mapping mapping, String name);
Architecture getArchitecture(Mapping mapping, String name);
Assembly getAssembly(Mapping mapping, String name);
boolean checkSource(Mapping mapping);
}

```

This extension point is now extended by the `entity.searcher` plug-in which searches for entities in a SOFA 2 project folder specified in a mapping model. The `getResources()` method of the *SOFA2Project* class returns all SOFA 2 entities in a form of classes implementing the *ISOFA2LocalResource* interface. From this list, resources of the demanded type are filtered and their entities are extracted and returned. When a SOFA 2 entity of a specific name should be found, the `lookupResource()` method of the *SOFA2Project* class is used and its result is casted to the demanded type. The `checkSource()` method returns `true` when the mapping model project exists and has the SOFA 2 project nature.

When a user wants to edit a SOFA 2 entity, the corresponding editor of ADL file should be opened. First, the ADL file of the entity is found by the `getEntityFile()` method of the *generator* extension point. Then, the SOFA 2 ADL form editor is opened with this file. It identifies entity's type and offers corresponding pages.

6. Evaluation and discussion

We have completed all steps specified in section 2.2 so in this chapter, we evaluate our solution regarding the requirements listed in the same section. The developed tool is primarily intended for the UML component model and the SOFA 2 component system. However, we analyze how it can be generalized for other source models and other component systems. During the analysis, some problems and possible improvements of the SOFA 2 CBD process are identified that are not solved by this thesis, so they are further discussed here as well as other concepts of academic component systems.

6.1 Meeting the requirements

The *fully automatic generation* of SOFA 2 entities composing a deployable application from a UML model file is done by a developer in few steps – selection of an item in a pop-up menu over a source model file, creation of a SOFA 2 project using a wizard or choosing an existing one from a list and clicking on one button in the opened mapping form editor. Therefore, the *first* requirement is fully satisfied by the tool implementation.

The *second* requirement is creation of a *form editor for manual generation* of SOFA 2 entities, providing a possibility for mapping of existing SOFA 2 entities to elements rather than creation of new ones. This editor is described in section 5.1 and it enables a user full control of which entity is generated and which one is reused. It even offers a list of existing entities compatible with a selected element. Therefore, this requirement is also satisfied.

When a UML model is changed, a user does not have to generate new entities instead of those that have become incompatible. He can just invoke automatic repair of affected entities as described in section 4.3, so the *third* requirement of the *smart synchronization of changes* is fulfilled.

Since this tool is a compound of Eclipse plug-ins using a concept of extension points, its GUI is created using SWT widgets compatible with the Eclipse framework and all models are designed using the EMF, we can state that it is fully integrated to the Eclipse platform. Furthermore, it is *connected with other tools* already presented in the development process, primarily with the main tool for a component developer, *SOFA 2 IDE*, which is widely described in section 5.3, thus the *fourth* requirement is satisfied.

When a user tries to transform an invalid UML model, the *validator* plug-in described in section 5.2 informs him in a form of markers on the model file which conditions are broken. Also in case of other errors in the system, such as a failed search, generation, repair or editing of SOFA 2 entities, error messages arisen in a form of a window or a console log. Therefore, the tool fulfills the *fifth* requirement of *sufficient information about errors*.

Since the source model validation, its transformation to a mapping model, generation, repair and compatibility of SOFA 2 entities are defined in changeable extensions to the mapping model editor, we can state that the first part of the *sixth* requirement is satisfied. The second part of the requirement of the *modular and platform-independent design* is fulfilled only partially since especially

a transformation of the mapping meta-model for another component system is not something simple. We discuss this issue in section 6.2.

6.2 Other source and target models

Although this thesis is focused on the mapping between the UML and SOFA 2 component models, we should analyze how its results can be used in case of other source or target meta-models related to hierarchical component systems.

We can state that the preparation meta-model described in section 4.2 is independent of a target meta-model because each hierarchical component meta-model has a concept of a component, its provided and required interfaces of some types and its subcomponents connected together and to their parent component. It is also independent of a source meta-model because it carries all necessary information for its further transformation and it does not link any source entities. The mapping meta-model described in section 4.1 is independent of a source meta-model too because it stores only names of source entities and a source model filename. On the other hand, it is not independent of a target meta-model because it must know to which kinds of entities are its elements mapped. In figure 6.1, general relations among various source and target meta-models and the mapping and preparation meta-models are shown.

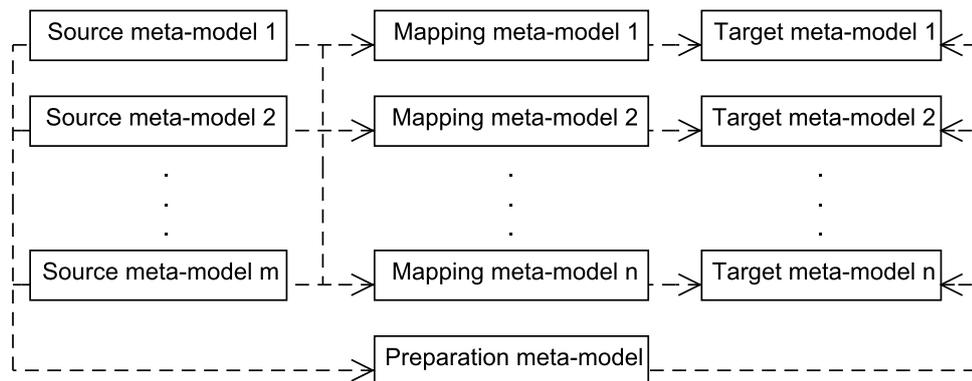


Figure 6.1: General relations among various source and target meta-models and the mapping and preparation meta-models

What do these statements mean in practice? Our tool can be developed to be easily extensible for support of other source meta-models, which already is, since a developer needs only to implement an own *validator* extension of the mapping form editor for a definition of a valid model and an own *generation* extension of the preparation meta-model plug-in where a transformation from a source meta-model to the preparation one is defined. On the other hand, a support of a different target meta-model, so a different component system, is much more complicated because it requires a definition of another mapping meta-model, therefore another editor must be created with its own extensions (*generation*, *compatibility*, *searcher*) and system integration according to its target component

system. Fortunately, most of these tasks are relatively straightforward and similar to those described in section 4.3.

For example, let's have a brief description how a support of the Fractal component model [24] can be achieved. First, a new mapping meta-model is created. It somehow differs from that one described in section 4.1 since the Fractal ADL [25] has not assemblies and it compounds a frame and an architecture together. Then, its editor is developed similarly as described in section 5.1. Finally, generation, compatibility and searching of Fractal entities is defined and implemented. Since the Fractal ADL has a well-defined XML structure, it should be no problem to generate, compare or scan it.

6.3 Further model extensions

As we note in section 2.1, some parts of code of interface types and primitive architectures can be also automatically generated from a source model. Furthermore, SOFA 2 and also other similar component systems offer many advanced concepts that can be well-defined by a source model too, such as a behaviour specification of components or component properties. Let's briefly discuss what must be extended in our solution to support these suggestions.

We analyze automatic code generation in two levels. First, we find out what can be generated without changing of the models, and then we propose suitable changes to the models that allow better code generation. With the current state of the models, we have enough information to generate only a header of a Java interface or a C abstract class without any methods for a SOFA 2 *InterfaceType* and for a primitive SOFA 2 *Architecture*, it is possible to generate a Java or C class implementing all provided interfaces and containing attributes of all required interfaces of its frame. For proper initialization of these attributes, an assignment method (i.e. the `setRequired()` method of the implemented *SOFA-Client* interface) or Java annotations can be also generated.

It will save much effort if methods of an interface type can be also generated to its interface and also to all classes implementing an architecture providing it. To make it possible, we would have to extend the used UML subset defined in section 3.4 and the preparation meta-model defined in section 4.2 as well as their transformations by a concept of operations owned by interface elements. In the UML meta-model, there is the *Operation* element used for this purpose. It stores among others a name, parameters and a return type of an operation. These attributes can be extracted from a UML model and stored in a preparation model in a new element type owned in any amount by the *Interface* element. From this model, code for a SOFA 2 *InterfaceType* and *Architecture* can be easily generated, repaired and checked for compatibility.

Behaviour of a component in the SOFA 2 component system is specified in its *Frame* definition by a behaviour protocol or its extension. It takes a form of an expression representing a set of a sequence of method calls and returns from them appearing on component interfaces [14]. For example, `!IInterface.method` means a call of the `method` method on the *IInterface* interface, while its acceptance is expressed as follows: `?IInterface.method`. Furthermore, method calls can be finitely repeated (*), sequenced (;), alternated (+) or parallelized (|). Its extension offers a customizing of method reactions by their parameters and local

variables for storing a current component state. Behaviour of an implementing architecture can be then checked by a *Cushion* command.

The described behaviour protocol and its extension can be modeled by a UML state machine diagram which would be owned by the UML *Component* element. The wide analysis of used UML elements and their semantics and transformations should be done in future work. It can use the analysis of this issue for the Fractal component system [1]. Its result would be also an extension of the preparation meta-model and related transformations as in the previous case.

It is sometimes useful to parametrize a component so its behaviour can be changed by a variation in attributes just before an application run rather than by modifying its code. For this purpose, each SOFA 2 *Frame* and *Architecture* can store names and types of their properties, which values can be specified in a *deployment plan*. These properties are then accessible in code via implementation of the *SOFAParametrized* interface in a form of the Java *Properties* class.

The concept of properties can be also easily modeled in the UML because it is possible to assign the *Property* element containing information about its name and type to the *Component* element as its owned attribute. These attributes can be extracted from a UML model and stored in a preparation model in a new element type owned in any amount by the *Component* element. Finally, generation, repair and checking compatibility methods are extended to map UML properties onto those in SOFA 2 entities.

These three cases serve as examples how other concepts can be included to the mapping between the UML and SOFA 2 component models described in chapter 4. It shows that this solution is extensible without changing of its fundamental design.

7. Related work

There already exist several works that define and sometimes implement mapping between the UML meta-model and a component system. In this chapter, we introduce their approach to the problem solved by this thesis.

First, we analyze mapping between the UML meta-model and proprietary component systems, such as CCM [17], COM [16] and EJB [23]. Since the component models of these systems are flat, their source UML model, known as a platform independent model (PIM), is in a form of class diagrams containing classes with attributes, operations and references. For each system, its platform specific model (PSM) is defined by its own UML profile [34] introducing a set of stereotypes and constraints applied to specific UML elements, such as `<<CORBAInterface>>`, `<<COMInterface>>` or `<<EJBRemoteInterface>>`. Furthermore, a transformation from the PIM to each PSM is defined, so for an element in the PIM, several elements with different stereotypes in the PSM can be generated.

For CCM, a document defining its UML profile is introduced by the OMG [11]. For example, the CORBA component is represented in the PSM as a *Class* element with the `<<CORBAComponent>>` stereotype with several constraints. For COM, some parts of its UML profile and a transformation from the PIM is defined [9]. For EJB, a document defining its UML profile is introduced as a public draft [10] with stereotypes such as `<<EJBEntityBean>>` extending a *Subsystem* element and representing an EJB entity bean, and a transformation from the PIM is implemented as a built-in transformation in the Enterprise Architect tool [26].

The UML meta-model mapping is analyzed not only in the proprietary component systems but also in the academic ones. A work most related to this thesis is the master thesis written by Matej Polák [8], because it introduces mapping to the previous version of the SOFA component system [13] and the Fractal project [24]. Its mapping definition and implementation differs in several important points. The SOFA *Frame* and *Architecture* are separated already in the design view. They are represented by the *Component* element with different stereotypes connected by the *Realization* dependency. Only one level of a subcomponent hierarchy can be modeled within an element representing the *Architecture*, so a model does not provide a full-scale view to an application. Its provided tool serves as a generator of entities and code from a UML model created in the Enterprise Architect tool, so it is not connected with currently used tools. It supports some concepts discussed in section 6.3, such as code generation and component properties, but it does not support reusability of SOFA entities and their corrections.

Another work related to this thesis is the master thesis written by Michael Cifka [4], because its subject is visual development of software components. The work introduces the CDB process, several component systems and various tools for developing. It implements a tool for the previous version of the SOFA component system as a plug-in to NetBeans IDE. This tool serves as an visual editor of SOFA applications. It supports a hierarchy view to an application via an assembly tree, reusability of SOFA entities and component properties, but it does

not support import from the UML meta-model or another standardized source meta-model. Therefore, a designer is forced to use this tool for modelling an application, which is also not connected with currently used tools.

We have already mentioned the work [1] that analyzes mapping between UML state machine diagrams and behaviour protocols for the Fractal project. It also introduces mapping between UML component diagrams and Fractal ADL based on the Polák's work [8]. It differs mainly in using ports for provided and required interfaces and no support of reusability. Other mapping of the UML is for the Architecture Analysis and Design Language (AADL) [12] that is designed for the component-based specification, analysis and automated integration of real-time performance-critical distributed computer systems. This mapping is in a form of a UML profile and its implementation is integrated in the Topcased platform [32].

Conclusion

The main goal of this master thesis was an analysis of CBD processes in terms of the SOFA 2 component system [13] and their improvement. We identified the main problem in the transition between the application design and creation of new components, because these parts of CBD processes are solved by different stakeholders that have considerably different views on the application development. We introduced the implementation view by a description of the SOFA 2 component model and its current development processes, we further analyzed the design view as a subset of the UML meta-model and we proposed the mapping between these views, i.e. between UML elements and SOFA 2 entities. Based on this analysis, we implemented a tool integrated to the Eclipse platform and cooperating with currently existing tools. We also proposed further improvements of this tool. Therefore, the main goal of this thesis was satisfied.

Several issues were found caused by a nonexistent definition of mapping between the design and implementation views, such as work duplication or bad synchronization of changes between a model and implementation. The communication between a designer and a component developer was described on a simple example, problematic parts were identified and the process improvement was proposed. This improvement requires not only a precise definition of the mapping but also a tool that automatizes some parts of the process. The general strategy to build this tool was introduced along with requirements on its implementation.

The UML meta-model [21] was chosen as a modelling language for the design view, more precisely its implementation for the Eclipse platform. Since the UML has a large-scale meta-model and it does not carry any semantics in itself, a subset of its elements from component diagrams was selected and for each element and its attributes, precise semantics was assigned and constraints on its relations were defined. As we want to constrain a developer as few as possible, we decided not to extend the UML meta-model by UML profiles or to create a new meta-model, hence every UML model not breaking constraints on selected elements is valid, these elements are further processed and the rest is ignored. All selected elements and their attributes with their semantics were listed as well as their constraints in the OCL [19].

Since the SOFA 2 component model is hierarchical, a designed UML model contains a full-scale view on an application component hierarchy rather than a specification of component and interface types. Therefore, some component and interface types can appear in the model more than once. Moreover, the UML meta-model represents a connection between a component and an interface as a relation rather than a property of the component. That is why the preparation meta-model was created as a connection model between the UML meta-model and the SOFA 2 component model. It contains a collection of component types with their provided and required interfaces of some types and subcomponents with connections among them. Another advantage of this approach is that this model is independent of its source and target meta-models, so both can be changed to support other models and systems. A transformation between the UML and preparation meta-models was specified, grouping elements of the same name to the same type, and algorithms for generation, repairing and compatibility checking

of SOFA 2 entities based on this meta-model were introduced.

One of the requirements was that a user could assign existing SOFA 2 entities to some source model elements rather than generate new ones. For this purpose, the mapping meta-model was introduced, which instance stores mapping between component and interface types found in its source model and their corresponding SOFA 2 entities. For this meta-model, a form editor was created as a plug-in to the Eclipse platform. The editor allows one-click automatic generation of all SOFA 2 entities, or manual setting for each component and interface type found in the source model. For each of them, a user can assign existing compatible entities, generate new ones, or repair previously assigned ones according to changes in the model. Therefore, the design model is always synchronized with the implementation. The editor is fully integrated with *SOFA 2 IDE*, an existing tool for managing SOFA 2 entities. Moreover, it was implemented modularly so the model transformations and the integration can be easily modified and it is possible to implement transformations from other source models. Unfortunately, implementation for another target component system is not as straightforward, but it was briefly described.

In the proposed solution, we implemented essential concepts of the SOFA 2 component system. However, this system has a lot of extended features, such as behaviour specification of components or component properties. Moreover, not only SOFA 2 entities can be generated, but some parts of corresponding code as well. All these improvements were discussed and they should be widely analyzed and implemented in future work. The main advantage of our solution is that its basic concepts would be extended rather than changed during implementation of improvements.

Bibliography

- [1] AHUMADA, Solange, et al. Specifying Fractal and GCM Components with UML. In *Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*. Washington, DC, USA : IEEE Computer Society, 2007. pp. 53–62. ISBN 0-7695-3017-6.
- [2] BÁLEK, Dusan; PLASIL, Frantisek. Software Connectors and their Role in Component Deployment. In *Proceedings of the IFIP TC6*. Deventer, The Netherlands : Kluwer, B.V.. 2001, pp. 69–84. ISBN 0-7923-7481-9.
- [3] BURES, Tomas; HNETYNKA, Petr; PLASIL, Frantisek. SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model. In *Proceedings of SERA 2006*. Seattle, USA : IEEE Computer Society, 2006. pp. 40–48. ISBN 0-7695-2656-X. Aug 2006
- [4] CÍFKA, Michael. *Visual Development of Software Components*. Prague, 2002. 54 p. Master Thesis. Charles University in Prague, Faculty of Mathematics and Physics, Department of Software Engineering. Web: <<http://d3s.mff.cuni.cz/publications/download/cifka-master.pdf>>.
- [5] CRNKOVIC, Ivica; CHAUDRON, Michel; LARSSON, Stig. Component-Based Development Process and Component Lifecycle. In *Proceedings of the International Conference on Software Engineering Advances*. Washington, DC, USA : IEEE Computer Society, 2006. pp. 44–. ISBN 0-7695-2703-5.
- [6] MENCL, Vladimir; BURES, Tomas. Microcomponent-Based Component Controllers: A Foundation for Component Aspects. In *Asia-Pacific Software Engineering Conference*. Los Alamitos, CA, USA : IEEE Computer Society, 2005. pp. 729–737. ISSN 1530-1362.
- [7] PARIZEK, Pavel; PLASIL, Frantisek. Modeling Environment for Component Model Checking from Hierarchical Architecture. In *Electron. Notes Theor. Comput. Sci.*. Amsterdam, The Netherlands : Elsevier Science Publishers B. V.. June 2007, 182, pp. 139–153. ISSN 1571-0661.
- [8] POLÁK, Matej. *UML 2.0 Components*. Prague, 2005. 62 p. Master Thesis. Charles University in Prague, Faculty of Mathematics and Physics, Department of Software Engineering. Web: <<http://d3s.mff.cuni.cz/publications/download/polak-masterthesis-uml20components.pdf>>.
- [9] *Component-Based Software Engineering : Lecture 22 — Model Transformations* [online]. Waterloo, Canada : University of Waterloo, 2007-02-28 [cit. 2011-06-21]. Web: <<http://www.stargroup.uwaterloo.ca/~ltahvild/courses/ECE493-T5/materials/notes/Lecture22-ModelTransformations-2up.pdf>>.
- [10] GREENFIELD, Jack. *UML Profile For EJB* [online]. Cupertino, CA, USA : Rational Software Corporation, 2001-05-25 [cit. 2011-06-21]. Web: <www.jeckle.de/files/UMLProfileForEJBPublicDraft.pdf>.

- [11] *UML Profile for CORBA and CORBA Components Specification* [online]. Needham, USA : Object Management Group, 2008-04-07 [cit. 2011-06-21]. Web: <<http://www.omg.org/spec/CCMP/1.0/PDF>>.
- [12] *AADL*. 2011. Web: <<http://www.aadl.info>>
- [13] *D3S, Charles University in Prague*. 2011. SOFA 2 component system. Web: <<http://sofa.ow2.org>>
- [14] *D3S, Charles University in Prague*. 2011. Formal Methods. Web: <<http://sofa.ow2.org/fm>>
- [15] *IBM*. 2004 [cit. 2011-05-24]. UML basics: The component diagram. Web: <<http://www.ibm.com/developerworks/rational/library/dec04/bell>>
- [16] *Microsoft*. 2011. Component Object Model. Web: <<http://www.microsoft.com/com>>
- [17] *Object Management Group*. 2011. CORBA Component Model. Web: <<http://www.omg.org/spec/CCM>>
- [18] *Object Management Group*. 2011. MetaObject Facility. Web: <<http://www.omg.org/mof>>
- [19] *Object Management Group*. 2011. Object Constraint Language. Web: <<http://www.omg.org/spec/OCL>>
- [20] *Object Management Group*. 2011. Query/View/Transformation. Web: <<http://www.omg.org/spec/QVT>>
- [21] *Object Management Group*. 2011. Unified Modeling Language 2.0. Web: <<http://www.uml.org>>
- [22] *Object Management Group*. 2011. XML Metadata Interchange. Web: <<http://www.omg.org/spec/XMI>>
- [23] *Oracle*. 2011. Enterprise Java Beans. Web: <<http://www.oracle.com/technetwork/java/javaee/ejb/index.html>>
- [24] *OW2 Consortium*. 2011. The Fractal Project. Web: <<http://fractal.ow2.org>>
- [25] *OW2 Consortium*. 2011. Fractal ADL Tutorial. Web: <<http://fractal.ow2.org/tutorials/adl>>
- [26] *Sparx Systems*. 2011 [cit. 2011-06-21]. Built-in Transformations – EJB Transformations. Web: <http://www.sparxsystems.com/resources/mda/ejb_transformations.html>
- [27] *The Eclipse Foundation*. 2011. Eclipse. Web: <<http://www.eclipse.org>>
- [28] *The Eclipse Foundation*. 2011. Eclipse Modeling Framework. Web: <<http://wiki.eclipse.org/EMF>>

- [29] *The Eclipse Foundation*. 2011. MDT-UML2 tool compatibility. Web: <<http://wiki.eclipse.org/MDT-UML2-Tool-Compatibility>>
- [30] *The Eclipse Foundation*. 2011. The Standard Widget Toolkit. Web: <<http://www.eclipse.org/swt>>
- [31] *The Eclipse Foundation*. 2011. UML2. Web: <<http://www.eclipse.org/uml2>>
- [32] *Topcased*. 2011. Web: <<http://www.topcased.org/>>
- [33] *UML Diagrams*. 2011 [cit. 2001-05-24]. Component Diagrams. Web: <<http://www.uml-diagrams.org/component-diagrams.html>>
- [34] *UML Diagrams*. 2011 [cit. 2001-05-21]. UML Profile Diagrams. Web: <<http://www.uml-diagrams.org/profile-diagrams.html>>
- [35] *Wikipedia*. 2011 [cit. 2011-05-24]. Component diagram. Web: <http://en.wikipedia.org/wiki/Component_diagram>
- [36] *Wikipedia*. 2011 [cit. 2011-05-13]. Model-driven engineering. Web: <http://en.wikipedia.org/wiki/Model_driven_development>

List of Figures

1.1	Component abstraction	4
1.2	Example of SOFA 2 entities	5
1.3	Simplified SOFA 2 component meta-model	6
1.4	Example of a SOFA 2 <i>Frame</i>	6
1.5	Example of a composite SOFA 2 <i>Architecture</i>	7
1.6	SOFA 2 system development process	8
1.7	Covering the development process with available tools	10
2.1	Participation of stakeholder roles on the CBD process	11
2.2	UML model of the example application	14
2.3	More than one UML element can be mapped onto one SOFA 2 entity	16
3.1	Example of UML elements	19
3.2	Delegation and subsumption connections must have the same name as the frame interfaces	22
3.3	Ownership hierarchy of semantically important UML elements . .	24
4.1	Relations among the used models	28
4.2	The mapping meta-model based on the EMF	29
4.3	The preparation meta-model based on the EMF	31
4.4	Generation of a SOFA 2 <i>InterfaceType</i>	35
4.5	Generation of a SOFA 2 <i>Frame</i>	35
4.6	Generation of a SOFA 2 <i>Architecture</i>	36
4.7	Generation of a SOFA 2 <i>Assembly</i>	37
5.1	Coverage of the mapping model by the editor	41
5.2	Diagram of classes related to the mapping model form editor . . .	43
5.3	Extension points of the mapping model form editor	44
5.4	Extension point of the preparation meta-model plug-in	46
6.1	General relations among various source and target meta-models and the mapping and preparation meta-models	51

List of Abbreviations

AADL	Architecture Analysis and Design Language
ADL	Architecture description language
CBD	Component-based development
CCM	CORBA Component Model
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
EJB	Enterprise Java Beans
EMF	Eclipse modeling framework
GUI	Graphic user interface
IDE	Integrated development environment
MDD	Model-driven development
MOF	MetaObject facility
OCL	Object constraint language
OMG	Object Management Group
PDF	Portable Document Format
PIM	Platform Independent Model
PSM	Platform Specific Model
QVT	Query/View/Transformation
SWT	Standard Widget Toolkit
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

A. Content of the enclosed CD-ROM

This thesis is enclosed by the CD-ROM containing its PDF version, source code and binaries of the described tool along with its prerequisite applications and its user and developer documentation. The CD-ROM is organized as follows:

Contact.txt Contact to the author of this thesis.

Content.txt Content of the CD-ROM.

Examples A sample UML component model with its generated SOFA 2 application.

Miscellaneous Other miscellaneous data, such as the unused QVT Operational transformation.

Prerequisites Applications necessary for the described tool, such as the SOFA 2 component system, Eclipse, Java SE 6 JDK, ...

Text This thesis and the user and developer documentation for the described tool in their PDF version.

Tool Source code, binaries and the update site for Eclipse of the described tool.

For installation information of the described tool, a tutorial providing step-by-step creation of a SOFA 2 application from a UML component model as well as a complete description of the mapping form editor, see the user documentation on the CD-ROM.

B. Additional resources

This chapter contains additional resources for this thesis. They can be found also as files in source directories of plug-ins to which correspond.

B.1 Complete UML model constraints

In this section, all OCL function and constraint definitions applied on each UML model are listed with brief comments. For more extensive comments, see section 3.4.

```
-- FUNCTION DEFINITIONS

-- a full name of an element
context NamedElement
  def: fullName : String = if self.ocIsTypeOf(Model) then ''
    else if self.ocIsTypeOf(Package) then self.owner.
      ocIsTypeOf(NamedElement).fullName.concat(self.name).concat('.')
    else self.getNearestPackage().fullName.concat(self.name)
  endif endif

-- the same dependency supplier name
context Dependency
  def: sameSupplierName(other: Dependency) : Boolean =
    self.supplier.fullName = other.supplier.fullName

-- all component frame/delegation usages and interface realizations,
  a primitive component, an assembly
context Component
  def: frameUsages : Set(Usage) = Usage.allInstances()->select
    (client->includes(self) and supplier->forall(s: NamedElement |
      client->exists(cl: NamedElement | s.owner = cl.owner)))
  def: frameInterfaceRealizations : Set(InterfaceRealization) =
    InterfaceRealization.allInstances()->select(client->includes(self)
      and supplier->forall(s: NamedElement | client->exists
        (cl: NamedElement | s.owner = cl.owner)))
  def: primitive : Boolean = not self.allOwnedElements()->exists
    (e:Element | e.ocIsTypeOf(Component))
  def: delegationUsages: Set(Usage) = Usage.allInstances()->select
    (client->includes(self) and supplier->forall(s: NamedElement |
      client->forall(cl: NamedElement | s.owner = cl)))
  def: delegationInterfaceRealizations : Set(InterfaceRealization) =
    InterfaceRealization.allInstances()->select(client->includes(self)
      and supplier->forall(s: NamedElement | client->forall
        (cl: NamedElement | s.owner = cl)))
  def: assembly : Boolean = self.isActive

-- INVARIANT DEFINITIONS

-- each component diagram must have only one top-level component
context Component
  inv one_toplevel_comp: self.owner.ocIsTypeOf(Package) implies
    self.ownedElement->one(e: Element | e.ocIsTypeOf(Component))

-- each interface realization and usage must be connected to
```

```

    a component and an interface
context InterfaceRealization
    inv connected_ir: self.contract->notEmpty() and self.supplier->size() = 1
        and self.client->size() = 1 and self.supplier->forAll
            (c: NamedElement | c.oclIsTypeOf(Interface)) and
            self.client->forAll(c: NamedElement | c.oclIsTypeOf(Component))
context Usage
    inv connected_us: self.supplier->size() = 1 and self.client->size() = 1 and
        self.supplier->forAll(c: NamedElement | c.oclIsTypeOf(Interface))
        and self.client->forAll(c: NamedElement | c.oclIsTypeOf(Component))

-- each interface must be connected with at most one interface realization
and/or some usages
context Interface
    inv connected_iface: InterfaceRealization.allInstances()->exists
        (contract = self) or Usage.allInstances()->exists
        (u: Usage | u.supplier->includes(self))
    inv one_ir: InterfaceRealization.allInstances()->select
        (contract=self)->size() <= 1

-- each component does not own itself and it is unique within its
owner component
context Component
    inv comp_own_itself: not self.allOwnedElements()->exists(e: Element |
        e.oclAsType(Component).name = self.name)
    inv comp_owner_unique: self.owner.ownedElement->select(e: Element |
        e.oclAsType(Component).name = self.name)->size() = 1

-- for each usage (except from that owned directly by a top-level diagram
component) exists an interface realization on the supplier interface
context Usage
    inv us_has_ir: self.client->exists(owner.owner = self.getNearestPackage())
        or InterfaceRealization.allInstances()->exists
        (self.supplier->includes(contract))

-- each usage/interface realization is either a frame usage/interface
realization, or a delegation one of its client component
context InterfaceRealization
    inv valid_ir: self.client->forAll(cl: NamedElement | cl.oclAsType
        (Component).frameInterfaceRealizations->includes(self) or cl.oclAsType
        (Component).delegationInterfaceRealizations->includes(self))
context Usage
    inv valid_us: self.client->forAll(cl: NamedElement | cl.oclAsType
        (Component).frameUsages->includes(self) or cl.oclAsType
        (Component).delegationUsages->includes(self))

-- each component has the same or less frame usages and interface
realizations with the same name and target interface as other
components with the same name
context Component
    inv same_valid_frames: let r1 : Set(Usage) = self.frameUsages in let p1 :
        Set(InterfaceRealization) = self.frameInterfaceRealizations in
        let prn : Bag(String) = r1.name->union(p1.name) in
            prn->size() = prn->asSet()->size() and
            Component.allInstances()->select(comp: Component | comp <> self
                and comp.fullName = self.fullName)->forAll(comp: Component |
                let r2 : Set(Usage) = comp.frameUsages in let p2 :
                    Set(InterfaceRealization) = comp.frameInterfaceRealizations in

```

```

    r1->forall(u1: Usage | r2->exists(u2: Usage |
        u1.fullName = u2.fullName and u1.sameSupplierName(u2))) and
    p1->forall(i1: InterfaceRealization | p2->exists
        (i2: InterfaceRealization | i1.fullName = i2.fullName and
        i1.sameSupplierName(i2)))

-- for each frame interface realization exists one delegation usage of
  the same name and supplier name, for each delegation interface
  realization exists a frame usage of the same name and supplier name
  and for each delegation usage exists a frame interface realization
  of the same name and supplier name
context Component
inv architecture_frame_compatibility: self.primitive or
  let fi : Set(InterfaceRealization) = self.frameInterfaceRealizations in
  let fu : Set(Usage) = self.frameUsages in let di :
    Set(InterfaceRealization) = self.delegationInterfaceRealizations in
  let du : Set(Usage) = self.delegationUsages in
    fi->forall(i: InterfaceRealization | du->one(u: Usage |
        i.fullName = u.fullName and i.sameSupplierName(u))) and
    di->forall(i: InterfaceRealization | fu->exists(u: Usage |
        i.fullName = u.fullName and i.sameSupplierName(u))) and
    du->forall(u: Usage | fi->exists(i: InterfaceRealization |
        u.fullName = i.fullName and u.sameSupplierName(i)))

-- each assembly component must not have any frame usages or interface
  realizations and all components of the same full name must be either
  assemblies, or ordinary components
context Component
inv: not self.assembly or (self.frameInterfaceRealizations->isEmpty()
  and self.frameUsages->isEmpty())
inv: Component.allInstances()->select(comp: Component | comp <> self
  and comp.fullName = self.fullName)->forall(comp: Component |
  comp.assembly = self.assembly)

```

B.2 Complete UML to preparation model transformation

In this section, the complete QVT operation transformation from a UML model to a preparation model is introduced with brief comments. For more extensive comments, see section 4.2.

```

modeltype UML uses uml('http://www.eclipse.org/uml2/3.0.0/UML');
modeltype Preparation uses preparation('http://preparation.ecore/1.0.0');
transformation UML2Preparation(in uml: UML, out Preparation);

// the main function
main() {
    uml.rootObjects()[UML::Model]->map model2Preparation();
}

// a full name of an element
query UML::NamedElement::fullName() : String {
    return if self.oclIsTypeOf(Model) then ''
    else if self.oclIsTypeOf(Package) then self.owner.oclAsType(NamedElement).
        fullName().concat(self.name).concat('.')
    else self.getNearestPackage().fullName().concat(self.name) endif endif

```

```

}

// a primitive architecture
query UML::Component::isPrimitive() : Boolean {
  return not self.allOwnedElements()->exists(e:Element |
    e.oclIsTypeOf(Component));
}

// an assembly
query UML::Component::isAssembly() : Boolean {
  return self.isActive;
}

// a diagram component
query UML::Component::isDiagram() : Boolean {
  return self.owner.oclIsTypeOf(Package);
}

// for each component/assembly type returns one component/assembly element
// it prefers an element with a composite architecture
query Set(UML::Component)::filter(assembly: Boolean) : Set(UML::Component) {
  return let rightType: Set(UML::Component) = self->select(comp: Component |
    comp.isAssembly() = assembly and not comp.isDiagram()) in
  let composite: Set(UML::Component) = rightType->select(comp: Component |
    not comp.isPrimitive()) in
  let uniqueComposite: Set(UML::Component) = composite->iterate
    (comp: UML::Component; col: Set(UML::Component) = Set{} |
    if (col.fullName()->exists(name: String | name = comp.fullName()))
    then col else col->including(comp) endif) in
  rightType->iterate(comp: UML::Component; col: Set(UML::Component) =
    uniqueComposite | if (col.fullName()->exists(name: String |
    name = comp.fullName())) then col else col->including(comp) endif);
}

// maps a UML model to a preparation model
mapping UML::Model::model2Preparation() : Preparation::Preparation {
  interfaces := self.allOwnedElements() [UML::Interface]->
    map interface2InterfaceType();
  components := self.allOwnedElements() [UML::Component]->filter(false)->
    map component2ComponentType();
  assemblies := self.allOwnedElements() [UML::Component]->filter(true)->
    map component2ComponentType();
  self.allOwnedElements() [UML::Component]->select
    (comp | not comp.isDiagram())->map component2ComponentType();
}

// stores mapping from a UML Interface to a preparation InterfaceType
property interfaceMap : Dict(String, Preparation::InterfaceType) = Dict{};

// maps a UML Interface to a preparation InterfaceType
mapping UML::Interface::interface2InterfaceType():Preparation::InterfaceType{
  init { result := interfaceMap->get(self.fullName()); }
  name := self.fullName();
  end { if (not interfaceMap->hasKey(self.fullName())) then
    interfaceMap->put(self.fullName(), result) endif; }
}

// stores mapping from a UML Component to a preparation Component

```

```

property componentMap : Dict(String, Preparation::Component) = Dict{};

// maps a UML Component to a preparation Component
mapping UML::Component::component2ComponentType() : Preparation::Component {
  init { result := componentMap->get(self.fullName()); }
  if (not componentMap->hasKey(self.fullName())) then
  {
    name := self.fullName();
    subcomponents := self.ownedElement[UML::Component]->
      map component2Subcomponent();
    providedInterfaces := self.getSourceDirectedRelationships()
      [UML::InterfaceRealization]->select(ir | ir.supplier->
        exists(owner = self.owner))->map dependency2Interface();
    requiredInterfaces := self.getSourceDirectedRelationships()
      [UML::Usage]->select(us | us.supplier->
        exists(owner = self.owner))->map dependency2Interface();
    connections := self.getSourceDirectedRelationships() [UML::Usage]->select
      (us | us.supplier->exists(owner = self))->map usage2Connection(self)->
      union(self->ownedElement[UML::Component].getSourceDirectedRelationships()
        [UML::Usage]->select(us | us.supplier->exists(owner = self))->
        map usage2Connection(self));
  } endif;
end { if (not componentMap->hasKey(self.fullName())) then
  componentMap->put(self.fullName(), result) endif; }
}

// maps a UML component to a preparation Subcomponent
mapping UML::Component::component2Subcomponent() : Preparation::Subcomponent {
  name := self.fullName();
  component := self.late resolveone(Preparation::Component);
}

// maps a UML InterfaceType or Usage to a preparation Interface
mapping UML::Dependency::dependency2Interface() : Preparation::Interface {
  name := self.name;
  interface := self.supplier->asSequence()->first().oclAsType(Interface).
  map(Preparation::InterfaceType);
}

// maps a UML Usage to a preparation Connection
mapping UML::Usage::usage2Connection(parent : UML::Component)
  : Preparation::Connection {
  let comp : UML::Component = self.client->asSequence()->first() in
  let interface : UML::Interface = self.supplier->asSequence()->first() in
  let otherRel : UML::InterfaceRealization =
    interface.getTargetDirectedRelationships() [UML::InterfaceRealization]->
    asSequence()->first() in
  let otherComp : UML::Component = otherRel.client->asSequence()->first() in
  endpoints := Set{
    object Endpoint {
      interfaceName := self.name;
      subcomponentName := if (comp != parent) then comp.fullName()
        else null endif; },
    object Endpoint {
      interfaceName := otherRel.name;
      subcomponentName := if (otherComp != parent) then otherComp.fullName()
        else null endif; } }
}

```