Applications of a Generic Component Framework to a UML Case Study in Production Automation

Diplomarbeit im Fach Informatik, TU Berlin
Vorgelegt von Martti Piirainen

17. Februar 2003

Betreuer und 1. Gutachter:
Prof. Hartmut Ehrig, Technische Universität Berlin
2. Gutachter:
Prof. Fernando Orejas, Universitat Politècnica de Catalunya

Die selbständige und eigenhändige Anfertigung dieser Arbeit versichere ich an Eides statt.

Berlin, den 17. Februar 2003

Martti Piirainen

Zusammenfassung

In dieser Arbeit werden ein neuer Komponentenbegriff für UML-Klassendiagramme und die Komposition solcher Komponenten definiert. Dafür werden einfache Begriffe der Inklusion und Verfeinerung von statischen Modellen eingeführt.

Dieses Komponentenkonzept ist mit Hilfe der UML-Erweiterungsmechanismen formuliert. Ein Ergebnis ist ein UML-Profil für komponentenbasierte Modellierung. Als weiteres Ergebnis wird bewiesen, dass dieses Komponentenkonzept eine Instanziierung des Generischen Komponentenrahmenwerks von Ehrig et al. ist.

Das Komponentenkonzept wird auf eine UML-Fallstudie aus der Domäne Produktionsautomatisierung angewendet, indem das bereits entwickelte Modell in der in dieser Arbeit definierten Terminologie und Notation reformuliert wird. Außerdem wird skizziert, wie ein Komponentenkonzept, das alle UML-Techniken, insbesondere Verhaltensspezifikationen, umfasst, erreicht werden könnte. Die Ergebnisse dieser Arbeit werden mit anderen existierenden Ansätzen zur Komponentenmodellierung mit Hilfe der UML verglichen.

Abstract

In this thesis, a new notion of components for UML class diagrams and composition of such components are defined. Therefore, simple notions of inclusion and refinement of static models are introduced.

This component concept is formulated using the UML extension mechanisms. One result is a UML profile for component-based modelling. As a second result, it is proven that this component concept is an instantiation of the Generic Component Framework by Ehrig et al.

The component concept is applied to an existing UML case study in the domain of production automation, by reformulating the model developed previously in the terminology and notation defined in this thesis. Additionally, it is sketched how a component concept covering all of the UML techniques, especially behavioural specifications, could be approached. The results of the thesis are compared with other existing approaches to component modelling using the UML.

Contents

List of Figures

Chapter 1

Introduction

1.1 Precise Component-Based Modelling

As complexity and size of software systems have risen over the past years, various methods of decomposing systems into smaller parts have been developed and have proven useful. Component-based systems, i.e. systems built of independent, self-contained and replaceable components and suitable connectors, are currently implemented in large-scale projects using technologies such as COM (with its successors COM+ and DCOM), .NET, EJB or CORBA.

Components and component-based systems are currently not reflected in modelling and specification languages in such depth. For various existing techniques components are currently developed building on existing structuring and composition notions. Such components were developed for various formal techniques such as low-level and high-level Petri nets [22] or graph transformation systems [25]. For the semi-formal UML a notion of component exists, but it does not fulfil all requirements for a component (see section 6.2 for a detailed discussion).

The idea of a component, as used in this thesis, is as follows: A component specification consists of a part that specifies the services provided by the component, called *provisions*, a part specifying the details of realising the services provided, called *body*, and a part specifying services that are required for realising the provisions, but not realised in the component itself, called *requirements*. The provisions and body part are connected by a refinement relation, i.e. the body is a refinement of the provisions (or, the provisions are an abstraction of the body). The requirements are included resp. imported into the body part.

This thesis encourages using the UML *precisely* for the modelling of components. The term precise[1] is chosen because the level of formality in the use of the UML varies widely. In development practice, UML diagrams are often drawn with no idea of an semantics of the model at all. On the other hand, it is (today) not possible to use the UML as a formal technique, but it is possible to use large parts of it unambiguously. The UML specification has various inconsistencies; some are

[1]The term is used in a similar sense by the Precise UML Group, `www.pUML.org`.

analysed in this thesis, others are shown elsewhere; hopefully most of them are gone in the next major version of the UML. Nevertheless, this thesis is not meant to be a critique of the UML, but shall constructively discuss some problems and how to deal with them precisely.

1.2 A Generic Component Framework

A framework for component specifications that abstracts from the used specification technique is introduced in [9] and is shortly repeated here. This framework is generic in the sense that important results on compositionality were formulated and proven on an abstract level. In order to profit from these results in practice with a concrete modelling or specification language, it is required to formulate a suitable *instantiation* of the generic concept, i.e. to show that the properties required of the concrete language are fulfilled.

Components

As already informally introduced, a component consists of three specifications and two connecting relations. Following the tradition of algebraic module specifications [8], the provisions part is called *export* and the requirements part is called *import* of a component.

Export and body are connected by a *transformation* which shall define a refinement of the export by the body (or, equivalently, an abstraction from the body by the export). The *transformation framework* must have identical transformations for each specification, and must be transitive. Body and import are connected by an *inclusion* (or *import*) of the import into the body. On this general level, there are no further requirements.

For both kinds of relation one has to check, whether an appropriate formalism exists in the concrete language used. Later in this thesis will be shown that the existing notions of refinement and import have to be restricted in order to be usable in components.

A component $COMP = (EXP, BOD, IMP, exp, imp)$ can be denoted as in figure 1.1.

$$\begin{tikzcd}
 & EXP \arrow[d, Rightarrow, "exp"] \\
IMP \arrow[r, hook, "imp"] & BOD
\end{tikzcd}$$

Figure 1.1: A Component in the Generic Framework

Semantics

The semantics of components is defined in terms of a *transformation semantics*, also called *semantics of possible refinements*. For a single specification $SPEC$ let $Trafo(SPEC)$ be the class of all possible transformations (refinements). The transformation semantics of a component is a function $TrafoSem : Trafo(IMP) \to Trafo(EXP)$, mapping import transformations $tra : IMP \Rightarrow SPEC$ to export transformations $tra : EXP \Rightarrow SPEC$. This concept is shown in figure 1.2. Formally, the semantics of $COMP$ is defined as $trafoSem(COMP)(tra) = tra' \circ exp$.

$$\begin{tikzcd}
 & EXP \arrow[d, Rightarrow, "exp"] \\
IMP \arrow[r, hook, "imp"] \arrow[d, Rightarrow, "tra"] & BOD \arrow[d, Rightarrow, "tra'"] \\
SPEC \arrow[r, hook, "imp'"] & SPEC'
\end{tikzcd}$$

Figure 1.2: Transformation semantics

The existence of the extended transformation tra' and the inclusion imp' depends on the so-called *extension property*, see figure 1.3. The chosen modelling language has the extension property if a transformation $trafo : SPEC_1 \Rightarrow SPEC_2$ together with an inclusion i_1 of $SPEC_1$ into a "larger" specification $SPEC_1'$, given a certain *consistency* between $trafo$ and i_1, can always be extended to a transformation $trafo'$, resulting in a specification $SPEC_2'$ and an inclusion imp_2.

$$\begin{tikzcd}
SPEC_1 \arrow[r, hook, "i_1"] \arrow[d, Rightarrow, "trafo"] & SPEC_1' \arrow[d, Rightarrow, "trafo'"] \\
SPEC_2 \arrow[r, hook, "i_2"] & SPEC_2'
\end{tikzcd}$$

Figure 1.3: Extension diagram for the extension property

In chapter 3 the extension property for a defined class of UML models is shown, using such a consistency requirement (in order to avoid name conflicts). Additional properties of the used specification technique that are needed are so-called *horizontal* and *vertical composition* of extension diagrams. These properties are also shown in chapter 3.

In the context of the UML, which is not a formal language, it is usually impossible to give a precise operational semantics. Therefore,such a transformation semantics can be a valuable tool, because it is built rather on syntax of models than on its semantics. The intuition that the semantics of a component depends on how the requirements are fulfilled, is very well reflected.

Compositionality

The extension property is not only the key to define the semantics of single components, but also to define the composition of components and the semantics of composed components. Components $COMP_1$ and $COMP_2$ as given in figure 1.4 are *hierarchically composed* by a transformation $connect : IMP_1 \Rightarrow EXP_2$. The intuition is that this transformation is a refinement of the same kind as the export connections, so that the requirements specified in IMP_1 are fulfilled by EXP_2. Because of the (given) compositionality of transformations there is a transformation $xconnect = exp_2 \circ connect$, and because of the extension property (under assumption of consistency between $xconnect$ and imp_1), there is a specification BOD_3 with a transformation $xconnect'$ and an inclusion imp_1'. The resulting component is defined as $COMP_3 = (EXP_1, BOD_3, IMP_2, xconnect' \circ exp_1, imp_1' \circ imp_2)$.

$$\begin{tikzcd}
 & & EXP_1 = EXP_3 \arrow[d, Rightarrow, "exp_1"] \\
 & IMP_1 \arrow[r, hook, "imp_1"] \arrow[d, Rightarrow, "connect"] & BOD_1 \arrow[dd, Rightarrow, "xconnect'"] \\
 & EXP_2 \arrow[d, Rightarrow, "exp_2"] & \\
IMP_3 = IMP_2 \arrow[r, hook, "imp_2"] & BOD_2 \arrow[r, hook, "imp_1'"] & BOD_3
\end{tikzcd}$$

Figure 1.4: Composition of components

On the abstract level of the generic framework it is proven that the transformation semantics for $COMP_3$ is compositional (still assuming consistencies between transformations and inclusions): The semantics of the composed component is equal to the composition of the semantics of the single components and the connecting transformation. Formally, $TrafoSem(COMP_3) = TrafoSem(COMP_1) \circ Trafo(connect) \circ TrafoSem(COMP_2)$, where $Trafo(connect)(tra) =_{def} tra \circ connect$. This is the most important result with respect to the motivation for component-based modelling: Deriving properties of a system from the properties of the single components, without having to explicitly construct the composition.

A possible transfer of these concepts to UML models is sketched in figure 1.5 and detailedly examined in chapters 2 and 3. The single specifications for export, body and import map to packages; transformations and inclusions map to newly introduced dependencies with stereotypes «`provide`» and «`require`».

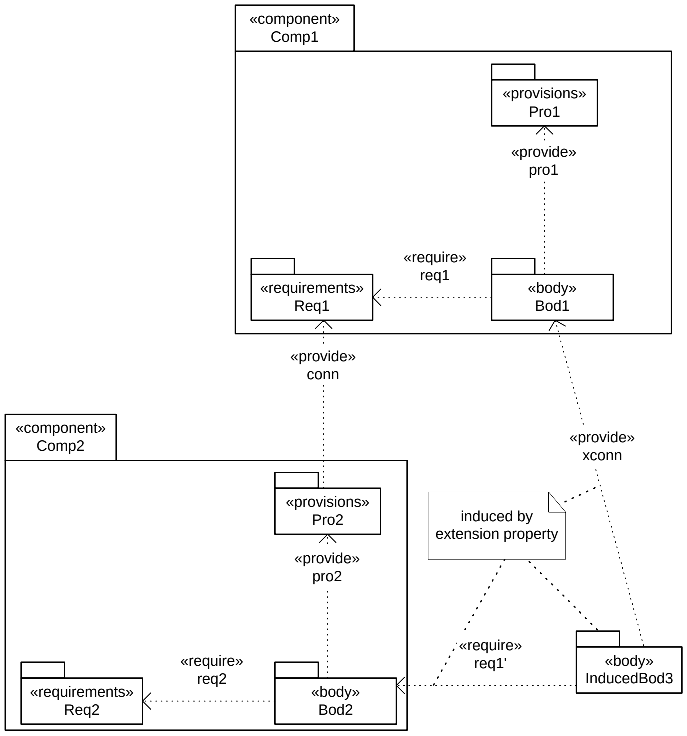


Figure 1.5: The UML component concept

1.3 The Unified Modelling Language (UML)

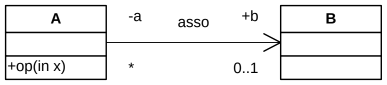
The Unified Modelling Language (UML) [28] is a modelling language that combines various modelling techniques representing software industry's "best practices" commonly used in the last years. It is developed and standardised by the Open Management Group (OMG) and widely used in the software industry, although it lacks precision in various places. This thesis is based on version 1.4 of the UML, released in 2001. Version 2.0 is expected to appear in 2003 and will cause several significant changes, probably including a new definition of components. Nevertheless, the work done in this thesis is probably not obsolete after the release of UML 2.0 since the main concepts of component design and the requirements for a component framework will stay the same.

The syntax of UML class diagrams and other diagram types is defined in terms of an abstract meta-model, well-formedness constraints formulated in the Object

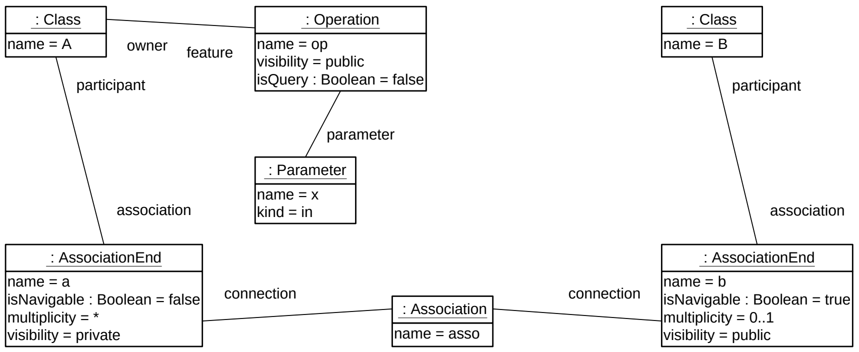
Constraint Language (OCL)[2] and additional descriptions in English language. The semantics is informally defined in English language (and in some areas intentionally left open). Additionally, a graphical notation and the mapping between concrete notation and abstract syntax are specified.

1.3.1 The UML Meta-Model

This section gives a short introduction to the UML meta-model. Using the well-known class–instance relationship, the meta-model contains *meta-classes*, instances of which are UML *model elements*.



(a) Concrete notation



(b) Abstract syntax

Figure 1.6: A simple class diagram

Figure 1.6 (a) shows a simple example of a class diagram: Two classes `A` and `B` are connected via the association `asso`. Class `A` has an operation `op` with an input parameter `x`.

Figure 1.7 shows a small excerpt from the core meta-model of the UML. The classes shown in this figure are meta-classes (and the associations are meta-associations), meaning that instances of these meta-classes are model elements such as classes or operations (and instances of the meta-associations are model element relationships such as containment, respectively) of a UML model. Figure 1.6 (b) shows the abstract syntax of the example, whereas figure 1.6 (a) shows the concrete notation.

Many more meta-model elements exist, of which the most important needed for the following chapters are attribute and parameter types, a generalisation/-

[2]OCL syntax and semantics are also specified in the UML specification [28, chapter 6]

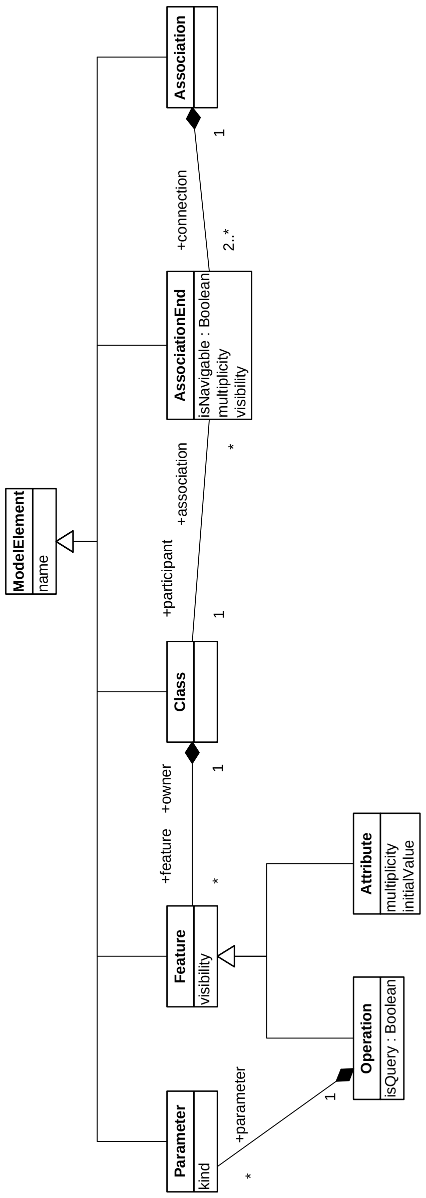


Figure 1.7: An excerpt from the UML meta-model

```
<?xml version = ’1.0’ ?>
<XMI xmi.version = ’1.2’
  xmlns:UML = ’org.omg.xmi.namespace.UML’>
  <UML:Class xmi.id = ’a1’ name = ’A’>
    <UML:Class.feature>
      <UML:Operation xmi.id = ’a2’ name = ’op’
        visibility = ’public’ isQuery = ’false’>
        <UML:Operation.parameter>
          <UML:Parameter xmi.id = ’a3’ name = ’x’
            kind = ’in’ />
        </UML:Operation.parameter>
      </UML:Operation>
    </UML:Class.feature>
  </UML:Class>

  <UML:Class xmi.id = ’a4’ name = ’B’ />

  <UML:Association xmi.id = ’a5’ name = ’asso’>
    <UML:Association.connection>
      <UML:AssociationEnd xmi.id = ’a6’ name = ’a’
        visibility = ’private’ isNavigable = ’false’>
        <UML:AssociationEnd.participant>
          <UML:Class xmi.idref = ’a1’/>
        </UML:AssociationEnd.participant>
      </UML:AssociationEnd>
      <UML:AssociationEnd xmi.id = ’a7’ name = ’b’
        visibility = ’public’ isNavigable = ’true’>
        <UML:AssociationEnd.participant>
          <UML:Class xmi.idref = ’a4’/>
        </UML:AssociationEnd.participant>
      </UML:AssociationEnd>
    </UML:Association.connection>
  </UML:Association>
</XMI>
```

Figure 1.8: XMI representation of the class diagram (simplified)

specialisation relationship, interfaces and packages.

The abstract syntax can be stored in the standardised XML Metadata Interchange (XMI) format [15] and thus be interchanged between UML modelling tools. A simplified XMI file representing the example class diagram is given in figure 1.8. The model elements are represented by XML tags, their attributes by XML attributes, and the interrelations are represented by identifiers and references (`xmi.id` and `xmi.idref` tags). XMI does not store any graphical information, though, and thus only allows for the interchange of models, not of diagrams.

The UML is defined as one instance of the OMG Meta-Object Facility (MOF) *meta-meta-model*, which is an abstraction at an even higher level. The intention of the MOF meta-meta-model is to make mappings between different modelling techniques possible. Another instance of the MOF meta-meta-model, on the same level as the UML, is the OMG Interface Definition Language (IDL) specification.

1.4 Refinement and Similar Concepts

Refinement and similar concepts appear in different dimensions of software development and modelling. One dimension is the chain of development steps from an initial requirements specification to an implementation (or an executable model), with (sub-)steps in between adding details, removing non-determinism and underspecification. Traditionally, this is a linear development (the infamous *waterfall* process model), nowadays this is usually an iterative (or cyclic) development, meaning that earlier models can be changed, enhanced or corrected as soon as problems are identified in models of a later phase. An example is given in figure 1.9, where a coarse model is refined into a more detailed one, attached with a verbal justification that this is a refinement.

The relation between refinements of this kind and the Generic Component Framework are shortly discussed in section 4.3.

Another dimension is system *evolution* over time, i.e. adapting a system to changed requirements, aimed at transforming the system into a consistent state again. This dimension has become more important in the last years as systems became more open and more distributed, and thus were subject to changes more often than in a closed, clean ”world”. The key problem is to distinguish parts that must be changed from parts that can be left as they are, aimed at a minimum effort for these changes.

The third dimension is the hiding of details of realising a service in a component body. Here, a refinement is a static relation in one model (in contrast to a development over time and between different models as in the previous two cases). This dimension has a different meaning than the first one, but is structurally similar. A component body adds detail to the provisions specification and removes underspecification from it. This kind of refinement is the essential one for this thesis. A simple example is realising an interface by a class (figure 1.10): The interface specifies operation signatures, the class defines an implementation. There

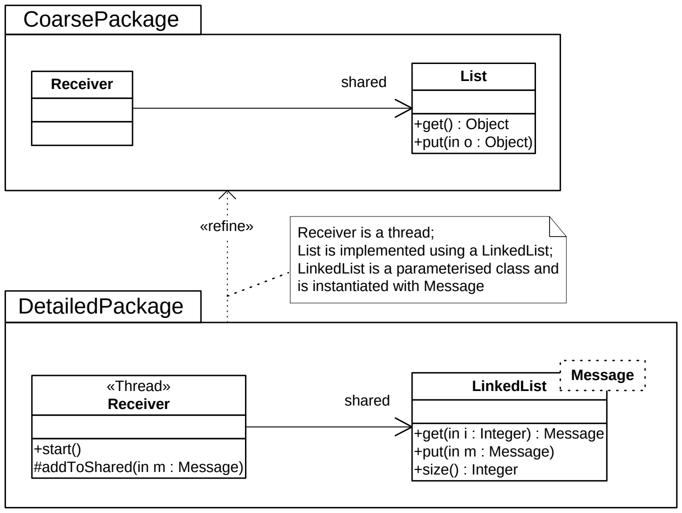
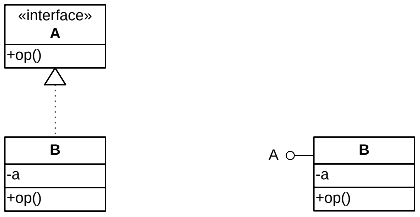


Figure 1.9: A refinement according to UML

are two different notations for interface realisation, one denoting the interface and the realisation arrow explicitly, and a shorter one hiding the interface's details.



(a) Detailed notation (b) Shorthand notation

Figure 1.10: B realises (implements) A

A quite different kind of transformation of models is *refactoring* of models. Refactoring aims at structuring and optimising design without any change in functionality. While most often done at the level of program code, it is increasingly a task performed on UML models ([2]). It will not be discussed in this thesis.

Refinement and UML

For most formal specification techniques, a refinement notion is an essential concept. In the UML, there is only a vague and imprecise notion of refinement, namely the «`refine`» stereotype for an Abstraction dependency:

> ”Refinement: Specifies refinement relationship between model elements at different semantic levels, such as analysis and design. The

> mapping specifies the relationship between the two elements or sets of elements. The mapping may or may not be computable, and it may be unidirectional or bidirectional. Refinement can be used to model transformations from analysis to design and other such changes" [28, section 2.5.2.1].

The level of formality of a refinement, though, depends on the concrete needs:

> "The degree of rigor of this traceability is variable. In a critical context, you can do these checks in mathematical detail. In more ordinary circumstances, you document the main points of correspondence to guide reviewers and maintainers and use these points as the basis for verification, design reviews, and testing." [5, section 6.1]

In the UML, any two model elements of the same kind can be connected via a dependency with a «`refine`» stereotype. Such a refinement can be used in any of the refinement dimensions mentioned above, but it lacks precision.

In chapter 2 a stricter form of refinement is defined, which, is still quite general and usable in many different situations. The most common form of refinement in the UML is class subtyping. On the syntactical level, this is an extension of features. Semantically, it has consequences on OCL constraints as well as behavioural models.

The intention that services specified in one package are realised in another package can be broken down to a refinement of classes. Addition of detail means addition of classes, attributes and operations. Adding classes to an existing class diagram implies a concept of *inclusion* of packages. A package is refined by another package by adding model elements and modifying model elements using subtyping. Thus, package refinement can be expressed as an inclusion of package contents and defining subclasses. Details are discussed in section 2.3.

1.5 Goals and Methods of the Thesis

In the following, a component concept for UML class diagrams shall be developed that is built upon one developed in the IOSIP project (see chapter 5) and that can be proven to be an instantiation of the Generic Component Framework. The intent of this thesis is to provide a precise framework for this restricted part of the UML. As the full expressiveness can be gained only by using all diagram types, it is also sketched how these can be integrated into the component framework.

In the next chapter, the instantiation is prepared by analysing the existing UML means for including and refining models, by defining new, slightly stricter notions of inclusion and refinement, and finally by defining components. Chapter 3 shows that the defined component concept is an instantiation of the generic case, by proving transitivity and the extension property of transformations, and defining composition. Chapter 4 sketches the needs for a component framework for the whole UML, i.e. ideas how to integrate the other diagram types, where statechart diagrams are examined more detailedly than the others. While chapters 2 through 4 use a ”toy size” running example (in the domain of bank accounts), chapter 5 applies the construct to parts of the IOSIP case study, allowing to prove usability of the concepts in a larger model (which is still not, of course, of realistic industry size: the full IOSIP model consists of about 100 classes with about 30 statechart diagrams). In chapter 6 the component notion introduced in this thesis is discussed in the context of different UML-based component approaches.

As a notational convention all names of UML model elements as well as all OCL specifications are laid out in `a typewriter font`. Self-defined OCL functions (used much like ”Additional operations” in the UML specification [28]) are introduced as needed and used from that point on. Some operations that are used in various places are additionally given in appendix A.

Chapter 2

A Component Framework for UML Class Diagrams

2.1 Prerequisites

Throughout this and the next chapter, the focus will be set on UML class diagrams only. First, a clarification has to made about the term *class diagram*. UML models are based on an abstract syntax, as seen in section 1.3.1, and diagrams are just the visual notation that make UML models so understandable to a human. The UML diagram types are not formally defined, in particular there is no meta-model element such as `Diagram` or `ClassDiagram`. There are only recommendations which model elements should be used in which diagram type: According to [28, section 3.19.2], a class diagram can contain ”static declarative model elements”, which in most cases means classifiers, relations and packages. Moreover, model elements can be shown in multiple diagrams, and a single diagram can leave out model elements, if they are shown somewhere else. A complete UML model is always the *sum of all diagrams* that are claimed to be part of the model.

This is a most intuitive way of modelling, as it reflects the concept of different *views* of a software system perfectly, and also reflects the idea of *decomposing* a large model into parts. For the user of a modelling tool it is natural to browse through different pages and to ”zoom into” a package or a class, or to hide details such as contents of a package or features of a class. On the other hand, it makes consistency checks on diagrams, such as type checks or logical and behavioural constraints, almost impossible, because each diagram would have to be checked with regard to each other. This is why such checks are done at the abstract syntax level, and why the component framework will not be defined for diagrams, but for the abstract syntax.

2.1.1 Motivation for Packages as Component Parts

In section 6.2 the drawbacks of UML components as defined in [28, section 2.5.2.12] and used in many practical contexts will be discussed detailedly. One of these drawbacks is the fact that UML Components are limited to provide interfaces to their environment. There is a need to provide more complex services, as will be motivated now, and thus *packages* are chosen as component parts.

The first example is not a component, but a part of a well-known class library. Among the utility class library shipped with a Java development kit, there is a service for calendar functionality, shown in figure 2.1:[1] Years, months, days are controlled using a class `Calendar` and its operations. Because the service leaves open to define other calendar types than the Gregorian calendar that is used in the western world (for example the Jewish or the ancient Julian calendars), this class is abstract (i.e. not all operations are implemented and the class cannot be instantiated) and supplemented by a concrete subclass `GregorianCalendar`. Additionally, a class `TimeZone` is needed in order to handle the impact of the time zone this calendar is defined for (which is basically the question when a day rolls over, i.e. when midnight is reached, in relation to Greenwich Mean Time).

It is imaginable that this service could be provided by a component (and used by other components), and it is obvious that this service cannot be equivalently defined using only interfaces, but needs the concept of (abstract and concrete) classes, class specialisation and associations. In particular, some of the operation signatures of `Calendar` would be syntactically invalid if the class `TimeZone` would not be accessible. This is a typical example for the situation that an operation together with a type could define a service.

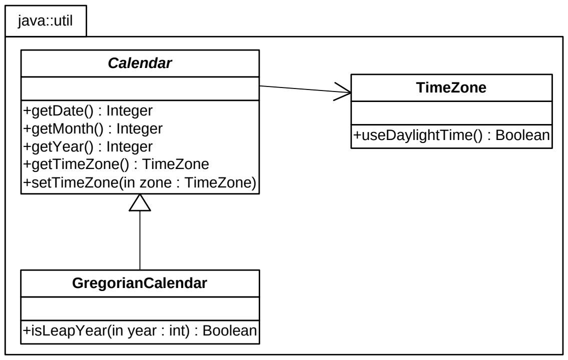

java::util
Calendar
+getDate() : Integer
+getMonth() : Integer
+getYear() : Integer
+getTimeZone() : TimeZone
+setTimeZone(in zone : TimeZone)
TimeZone
+useDaylightTime() : Boolean
GregorianCalendar
+isLeapYear(in year : int) : Boolean

Figure 2.1: The calendar service consists of several classes

As a second introducing example, the running example for the next chapters is introduced: a component that provides classes representing *accounts*, which may be part of a banking system. Figure 2.2 shows that there are two kinds of

[1]The class diagram is a simplified version of the classes in the `java.util` package; there are many more operations.

accounts, namely cheque accounts and savings accounts. A standard feature of object-oriented modelling is making use of *generalisation*: For some parts of the system the kind of an account may be irrelevant, they only need to know of accounts in general. By adding a superclass `Account`, client objects may link to account objects without knowing the concrete runtime type of the object, but still type-safely with regard to the operation signatures of `Account`, namely operations for retrieving the account number and the balance, and for depositing and withdrawing given amounts of money. This example will be extended as new concepts are introduced, but already in this excerpt it is more intuitive to provide this class diagram to the environment, than providing only three interfaces without showing their interrelations.

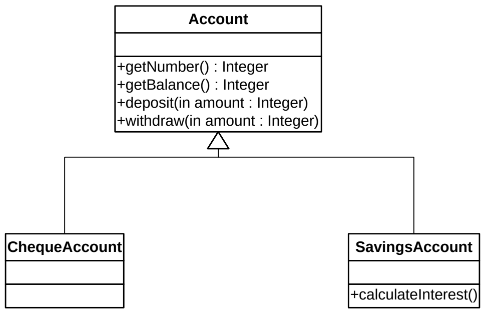


Figure 2.2: Two kinds of accounts

With packages chosen as component parts, the question rises which existing or new UML techniques to use as refinement transformations and inclusions as demanded by the Generic Framework. This is examined in the next two sections.

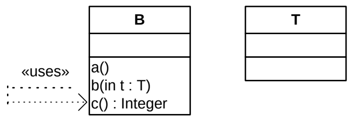
2.2 Inclusion of Packages

In [9] the relation between a component body and a component import is characterised as follows:

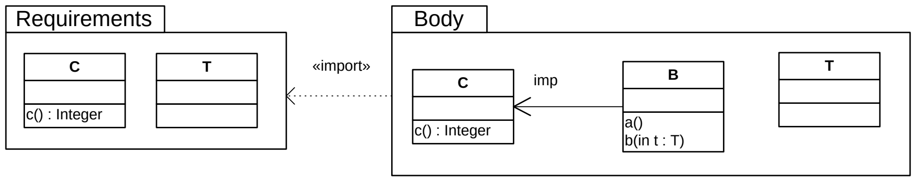
> ”We assume that the import connection is some kind of inclusion, in the sense that the functionality defined in the body is built upon the import interface.”

Although there is no notion of inclusion of packages in the UML, there is a closely related construct called *import*. This section will shed some light on the UML import’s difficulties.

The motivation for the inclusion is that functionality that is needed in a component part is realised elsewhere. This functionality can be given, for example, by a single operation. In figure 2.3 (a) a «`uses`» dependency (informally) indicates that operation `b` needs `c` for its realisation.[2] If for some reason `c` shall not be realised in this class (because it already exists, because it is realised by someone else, or because it shall be realised later), it can be factored into another class, and each call of this operation can be replaced by a call along the association `req` that connects the two classes. See figure 2.3 (b).



(a) `b` uses `c` and class `T`



(b) `c` is factored into a separate class

Figure 2.3: Factoring requirements into a separate package

Horizontal inclusion does not necessarily imply that there is a delegation along an association to an imported class, but this is the most typical situation and will be seen in many of the following examples.

2.2.1 The UML Import Dependency

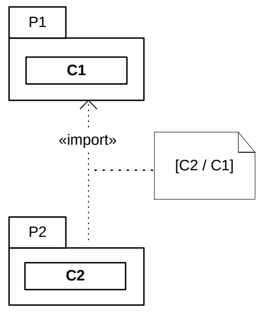
If everything that is to be realised elsewhere is put into a separate package, it can be accessed by the original package using one of two distinct dependencies:

- using a permission dependency with an «`access`» stereotype. The model elements must be referenced with their full path name, e.g. `P::C` for a class `C` that is defined in package `P`.

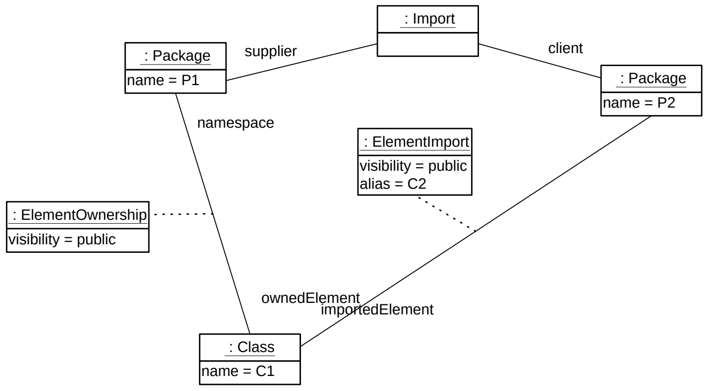
- using a permission dependency with an «`import`» stereotype. Model elements are added to the namespace of the importing package and need to be referenced only by their name. This results *almost* in a union of package contents, although on the meta-model level there is a slight difference between elements *owned* by a package and elements *imported* into a package. See figure 2.4: Model element ownership and model element import map to different elements in the abstract syntax (instances of the meta-model association classes `ElementOwnership` and `ElementImport`, respectively).

[2]This dependency could, for example, have been derived from a statechart or from code implementing this class.

In the following, the `«import»` dependency will be used. It would not be appropriate to reference every model element with the package where it was originally defined. In a model consisting of several composed components, it is not always interesting where a model element origins from, and this information is implicitly in the model in any case.



(a) Concrete notation



(b) Abstract syntax

Figure 2.4: Importing a class with an alias

In the following, the import dependency is examined more closely. Some problems will be appear that make it impossible to be used as an inclusion as it is, without putting some additional constraints into action.

Import can lead to inconsistent models

The import dependency is defined in [28, section 2.5.2.32]:

> ”Import is a stereotyped permission dependency between two namespaces, denoting that the public contents of the target package are added to the namespace of the source package.”

The fact that only *publicly visible* elements are imported leads to problems. On one hand, this property can lead to inconsistent models. On the other hand, it leads to the fact that the existing import dependency is not transitive.

An inconsistent situation raises, for example, when an imported class uses another class which is not publicly visible. In figure 2.5 package `P2` is ill-formed, because it contains a class `A` that uses a class `B` which is not accessible (not known) to this package.

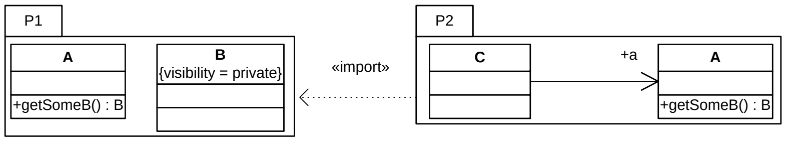


Figure 2.5: Import can lead to an ill-formed model

For this reason, we will later *constrain* the use of this kind of dependency.

Import is not transitive

The specification states:

> ”An imported element is by default private to the importing package. It may, however, be given a more permissive visibility relative to the importing package.” [28, 2.14.4.1]

This definition leads to the fact that the default way of importing model elements is not transitive. Consider an example: Assume a package `P1` defining a class `A`, package `P2` defining a class `B`, and package `P3` defining a class `C`, as shown in figure 2.6 (a). Figure (b) shows that package `P3` has *no access* to class `A` if `P3` imports `P2` and `P2` imports `P1`. The solution is an adornment to the import dependency that is mapped in the abstract syntax to a value `public` of the `visibility` meta-attribute of the meta-class `ElementImport`. There is no specified standard notation, but the chosen note ”import `A` as public” should be sufficient, as shown in figure (c).

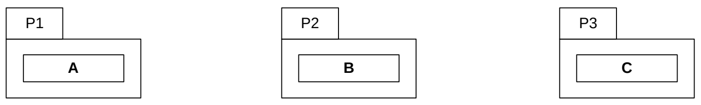
Therefore, every import as used in the following additionally sets the visibility to public for each imported element.

Alias names

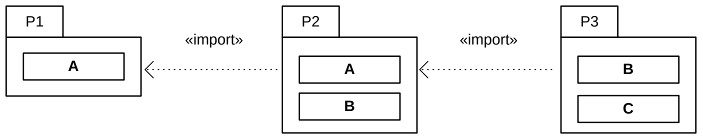
Again a quotation from the specification:

> ”It is possible to give an imported element an alias to avoid name conflicts with the names of the other elements in the namespace, including other imported elements” [28, section 2.14.4.1]

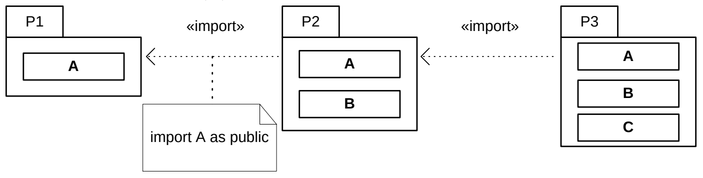
This is an interesting feature that will be used in the following. A notation is not specified, we define: `[A/X]` as a note attached to the import dependency means that element `X` is given the alias name `A` in the importing package. See figure 2.7 for an example.



(a) The basic packages

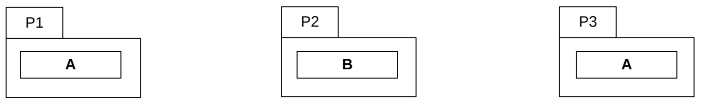


(b) P3 has no access to class A

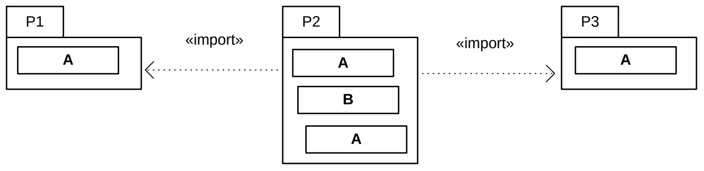


(c) An attached note defines A in package P2 as public

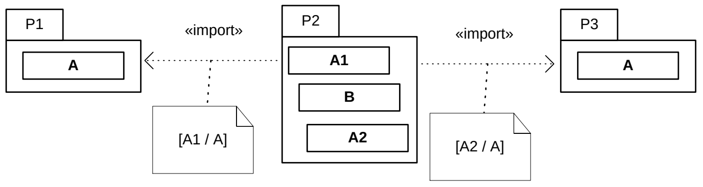
Figure 2.6: The import dependency is not transitive



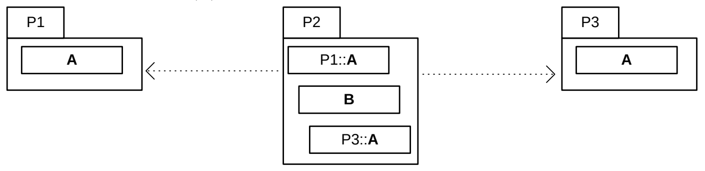
(a) The basic packages



(b) P2 is ill-formed (two elements with the same name in one namespace)



(c) Attached notes define aliases



(d) A variant: Accessing classes with full path

Figure 2.7: Import with aliases

Avoiding Name Clashes

Whenever an import is used, name conflicts can appear. The well-formedness rules of a package say that there may not be two model elements with the same name in a package, including imported elements. If such a situation exists, the model is ill-formed. One can imagine different possibilities to avoid name clashes between an importing and an imported package:

- Name clashes are detected by the modelling tool immediately when a model element or an import dependency is added. The user must choose a different name.
- The tool automatically defines a unique alias name for an imported element (like `ModelElement25`)
- The full path of a model element must be used, e.g. `P::C`

For simplicity, in the following name conflict freeness is required. Therefore, a function is defined that checks for name clashes between two packages.

Definition 2.2.1 (Name Conflict Freeness). Two packages `A` and `B` are *name conflict free*, if there are no owned elements with the same name in `A` and `B`. In OCL:

```
let isNameConflictFree(A:Package, B:Package):Boolean =
  A->ownedElement->forAll( a:ModelElement |
    not B->ownedElement->exists(
      b:ModelElement | a.name = b.name))
```

This definition is not concerned about name conflicts in imported elements. This is explained now.

Rhombic import

A typical situation is that two packages import from a common package, and another package imports these two packages. This leads to a *rhombic* (also called *diamond*) structure. In addition to name conflicts that can appear, such as in figure 2.8, such a situation raises the interesting question whether common elements are identified in the resulting package or appear twice.

This situation is similar to a *union* in other specification techniques (or, in categorical terms, to a pushout). In contrast to formal techniques such as Petri nets, there is no notion of *equivalence classes* of model elements, and therefore name clashes must be handled individually.

The clue why there is no need for name conflict freeness in the set of imported elements is that imports are, on the abstract syntax level, defined ’by reference’, meaning that elements imported into other packages are not ’copied’ there, but a meta-level link is established from any importing package to the imported element. See figure 2.9: Class `A` is imported into packages `P2` and `P3`, and again imported by `P4`. There are `ElementOwnership` links from `P2`, `P3` and `P4` to `A`.

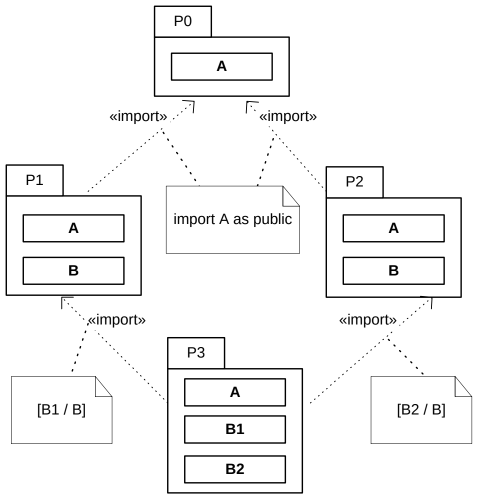


Figure 2.8: Rhombic import, using aliases to avoid name conflicts

Thus, there always results a well-formed package in a situation as in figure 2.9 with the expected property that elements imported from `P1` into both `P2` and `P3` are identified in `P4`.

This situation can be extended to imports from more than two packages under the assumption of pair-wise name conflict freeness. This situation arises in components with multiple provisions and requirements interfaces.

2.2.2 A New Import Stereotype

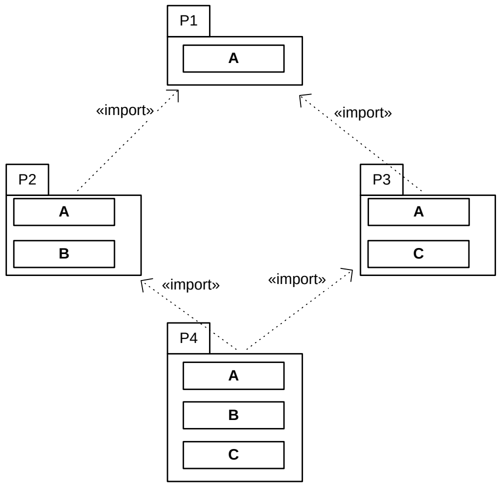
As seen, the existing import dependency raises problems that we want to eliminate for the use in a component concept. A sub-stereotype of the «`import`» dependency, called «`require`», is defined as follows:

Definition 2.2.2 (Stereotype «`require`»). An import dependency may be stereotyped as «`require`» if the following constraints hold:[3]

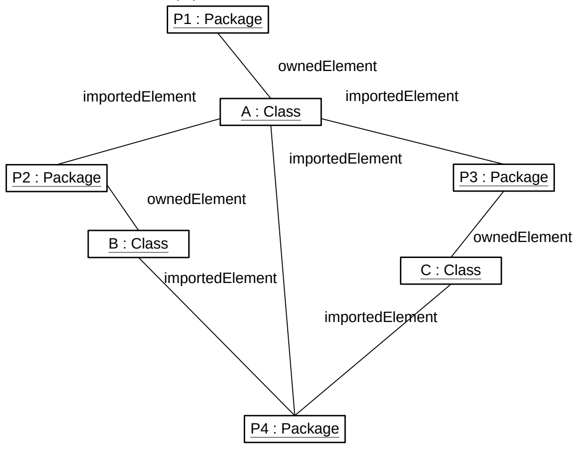
[1] «`require`» dependencies exist only between packages (not between arbitrary namespaces).

```
context Require inv:
self->client->oclIsKindOf(Package)
and
self->supplier->oclIsKindOf(Package)
```

[3]The notation is the same as for the well-formedness rules in [28]. The OCL constraint `context Require inv: X` means that property `X` is an invariant for all instances of `Require`.



(a) Concrete notation



(b) Abstract syntax

Figure 2.9: Rhombic import, elements are imported ’by reference’

[2] Every element in the imported package is public.

```
let allPublic(p:Package):Boolean =
  p.elementOwnership[ownedElement]->visibility = public
context Require inv:
allPublic(self->supplier)
```

[3] Every element is imported as public into the importing package.

```
let allImportsPublic(importing:Package,
  imported:Package):Boolean =
  importing.elementImport[importedElement]->select(
  me:ModelElement | me.namespace = imported )
  ->visibility = public
context Require inv:
allImportsPublic(self->client, self->supplier)
```

[4] No generalisations from the importing (body) to the imported (requirements) package exist.

```
let existsGeneralization(from:Package, to:Package)
  :Boolean =
  from->ownedElement->exists(
    ch:GeneralizableElement | to->ownedElement->exists(
      par:GeneralizableElement | isSpecialisation(ch,par)
  ))
context Require inv:
not existsGeneralization(self->client, self->supplier)
```

Using this stereotype, a notion of inclusion of packages in the sense of [9] can be defined, see section 3.1.

2.3 Refinement of Classes and Packages

The justification associated with a refinement can be formal or informal; it could even simply say, 'Joe said this will work'. [5, section 6.9]

Refinement relations are often missing completely in the UML practice, or used ad-hoc and implicitly, maybe even without recognising that a refinement relation exists. The authors of [5], which is a book mainly for practitioners, state that it is already an improvement if such refinement relations are recognised, made explicit, and attached with some justification.

In this section a very simple refinement concept for packages is developed, based on a simple notion of refinement of classes, which is compatible with the most typical uses of the UML. These refinement definitions could be replaced by different, especially by more formal ones if desired, where most of the component concept could stay the same. "Plugging in" other refinement notions implies, of course, an appropriate proof of the extension property for this refinement definition.

2.3.1 Refinement of Classes

As sketched in section 1.4, very different notions of refinement in the UML exist. Instead of choosing some of these and defining them formally, in the following the most basic refinement concept of object-oriented modelling, *inheritance*, is used as the foundation for refinement of classes. First, it shall be discussed whether this common usage of UML classes reflects what we expect from a refinement relation. There are very different definitions of refinement, but as a common sense, the following characterisations and requirements apply usually:

- In a refinement relationship the refining model is more concrete, more detailedly specified and more deterministic. The refined model is less concrete, less detailedly specified and more non-deterministic.
- The refinement relation is reflexive, transitive and anti-symmetric (i.e. it defines a partial order).[4]

In order to show the possible meanings of specialisation in the UML, the example introduced in section 2.1.1 is revisited with regard to refinement. Figure 2.10 shows how an account can be implemented in two different ways: Using a private attribute `balance`, or using a database interface that stores the balance in a database table. The first class is the proper one for short-term (transient) usage only, because the balance value is destroyed when the account object is destroyed, whereas the second one stores the balance persistently.

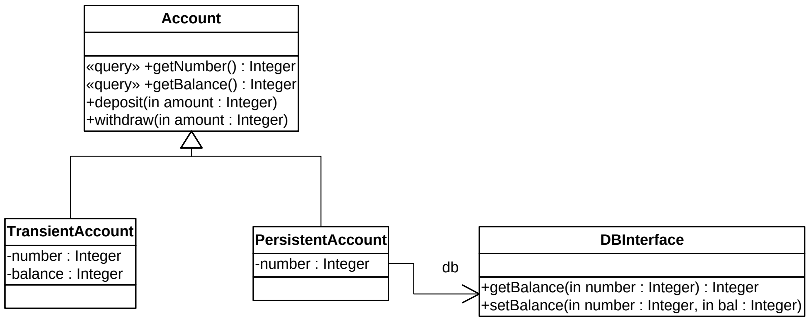

Account
«query» +getNumber() : Integer
«query» +getBalance() : Integer
+deposit(in amount : Integer)
+withdraw(in amount : Integer)
TransientAccount
-number : Integer
-balance : Integer
PersistentAccount
-number : Integer
db
DBInterface
+getBalance(in number : Integer) : Integer
+setBalance(in number : Integer, in bal : Integer)

Figure 2.10: Account and implementation classes

Intuitively, the refinement requirements are fulfilled. The subclasses `TransientAccount` and `PersistentAccount` add details to class `Account`, thus being closer to an implementation.

The classes `ChequeAccount` and `SavingsAccount`, shown in figure 2.2 on page 15, could also be viewed as two possible refinements of class `Account`. The savings account has an additional operation (the interest can be calculated), and

[4]Anti-symmetry is sometimes not required, then it defines a pre-order.

the names of the classes imply informally that one behaves like a savings account (interest is gained, but no credit is allowed) or a cheque account (no interest, but the balance may become negative). As there are no constraints defined yet, classes `Account` and `ChequeAccount` are isomorphic. This is *not* a refinement in the sense that implementation details are added in the subclasses, but rather a *subtype* relationship.

This example shows that the same mechanism, class specialisation, is used in order to express two very different *aspects* (also called *concerns*) of a system. Both aspects are orthogonal, which leads to a complex class diagram if all combinations of the dimensions are to be used, see figure 2.11. The different aspects of inheritance, the persistency and the kind of credit contract, can be denoted by so-called *discriminators* on the specialisation arrows. An implementor would probably want to put into action another use of subclasses, *implementation inheritance*, which is not interesting in the context of this thesis as implementation in a programming language is out of the scope.

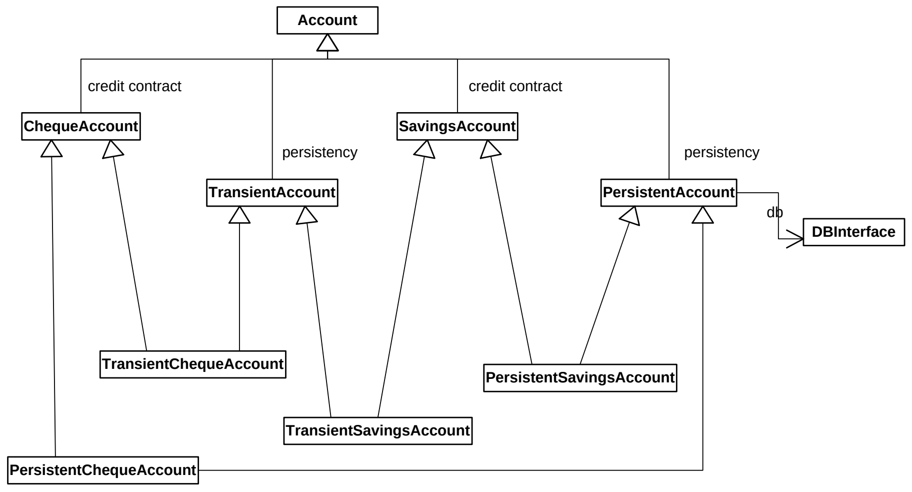


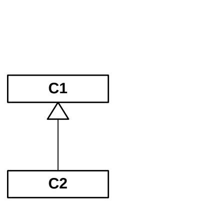
Figure 2.11: Account and all subclasses

Subclasses on the Abstract Syntax Level

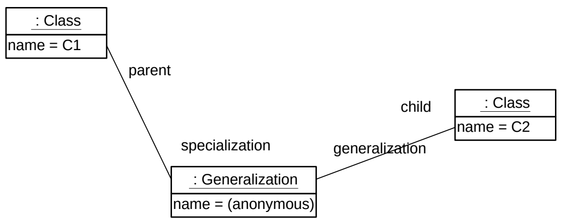
The different types of specialisation are not distinguishable on the abstract syntax level. There, a specialisation maps to a `Generalization` object connected to a child and a parent, see figure 2.12.

Various ”class-to-parent” relationships like generalisation, multiple inheritance, conformance, or realisation are discussed in detail in [27]. Some of them are proposed syntactical extensions to the UML and thus not discussed in this thesis.

Class subtyping can be expressed in terms of the UML meta-model using OCL.



(a) Concrete notation



(b) Abstract syntax

Figure 2.12: Specialisation and generalisation of classes

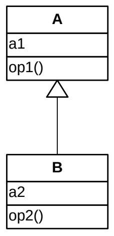
Definition 2.3.1 (Subtype and Supertype). Class `Sub` is a *subtype* of class `Super` if there is a specialisation dependency from class `Super` to class `Sub`. In OCL:

```
let isSubtype(Sub:Class, Super:Class):Boolean =
  Super.specialization->exists(
  g:Generalization | g->child->includes(Sub))
```

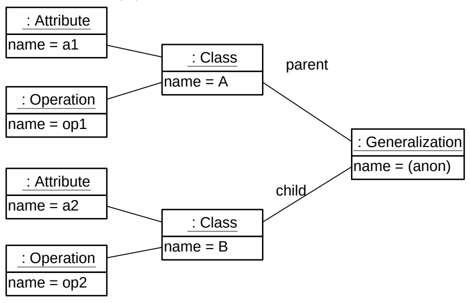
`Super` is a *supertype* of `Sub` if `Sub` is a subtype of `Super`. In OCL:

```
let isSupertype(Super:Class, Sub:Class):Boolean =
  isSubtype(Sub, Super)
```

An example is shown in figure 2.13: Class `B` adds an operation and an attribute to the ones inherited from class `A`.



(a) Concrete notation



(b) Abstract syntax

Figure 2.13: Superclass `A` and subclass `B`

But classes are not the only model element that can be refined this way: Interfaces, associations, association classes and several other model elements are subclasses of `GeneralizableElement` in the meta-model. Thus, the definition can be extended to all generalisable (and thus also specialisable) model elements. Meta-classes that are subtypes of `GeneralizableElement` include `Classifier`, `Association`, `Package`, `Stereotype`, `Class`, `Interface`, `Usecase`, and `Actor`.

One additional situation which is not a generalisation/specialisation relationship in the meta-model, has a similar meaning and is important and often used: Realisation of an interface. A class can realise one or more interfaces (see figure 1.10 on page 10), which means that the operations defined in the interface are implemented by the class (or a subclass). Interface realisation is, on the abstract syntax level, an Abstraction dependency stereotyped with `«realize»`.

Definition 2.3.2 (Specialisation and Generalisation). A generalisable element `Spec` is a *specialisation* of a generalisable element `Gen` if there is a specialisation dependency from `Gen` to `Spec`, or `Gen` is an interface realised by `Spec`. In OCL:

```
let isSpecialisation(Spec:GeneralizableElement,
  Gen:GeneralizableElement):Boolean =
    Gen.specialization->exists(g:Generalization
      | g->child->includes(Spec))
  or
    Gen.supplierDependency->exists(r:Realization
      | r->client->includes(Spec))
```

`Gen` is a *generalisation* of `Spec` if `Spec` is a specialisation of `Gen`. In OCL:

```
let isGeneralisation(Gen:GeneralizableElement,
  Spec:GeneralizableElement):Boolean =
    isSpecialisation(Spec, Gen)
```

2.3.2 Excursus: Formalisation in Set Theory

Subtyping can also be expressed in terms of set theory. Heavily simplifying, classes consist of a name and features, and subtyping is just adding features. The terms *segment descriptor* and *full descriptor* of a class are used in [28] in order to describe classes and subtyping, but they are not mathematically defined. The formal descriptor is the description of a class collecting all information from the superclasses, and is needed when instantiating a class (i.e. creating an object). A formalisation of a class, given the sets $name$ for all class names and $attrs$ and $opns$ for all attribute and operation specifications, respectively, could be:

Definition 2.3.3 (Segment Descriptor). The segment descriptor of a class C consists of a name, attributes and operations:[5]

[5] It is not distinguished between attributes and associations ends at the opposite end of an association.

$sd(C) = (name, attrs, opns)$ with
$name \in String$,
$attrs \in \mathbb{P}(Attribute)$, $opns \in \mathbb{P}(Operation)$

Definition 2.3.4 (Feature). Attributes and operations are features.
$Attribute \cup Operation = Feature$ and
$Attribute \cap Operation = \emptyset$

Definition 2.3.5 (Full Descriptor). The full descriptor of a class consists of the name of the class and the union of the classes' features with the features of all its ancestors.

Using these definitions, the syntactical basis for subtyping is established.

Definition 2.3.6 (Subtype). Let $fd(C) = (name_C, attrs_C, opns_C)$ and $fd(C') = (name_{C'}, attrs_{C'}, opns_{C'})$. C' is a ***subtype*** of C if
$attrs_C \subseteq attrs_{C'}$ and $opns_C \subseteq opns_{C'}$.

The example class diagram in figure 2.13 would map to sets as follows:
$sd(A) = ('A', \{a1\}, \{op1\})$
$sd(B) = ('B', \{a2\}, \{op2\})$
$fd(A) = sd(A)$
$fd(B) = ('B', \{a1, a2\}, \{op1, op2\})$
This formal level, though, is not used in the rest of this thesis, except for section 4.2.1, dealing with protocol statecharts.

2.3.3 Class Refinement is a Partial Order

Refining classes by subtyping is a partial order, i.e. it has reflexive, transitive and antisymmetric properties.

Reflexivity is ensured when defining that every single class is a refinement of itself (classes may not, though, inherit from themselves as in figure 2.14 (a)).

Antisymmetry is ensured by the UML specification:

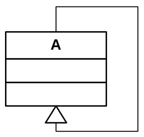
> ”Circular inheritance is not allowed.
> `not self.allParents->includes(self)`” [28, section 2.5.3.20].

Two additional operations are defined in [28, section 2.5.3.20] to express this relation in short notation:

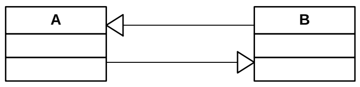
```
parent:Set(GeneralizableElement) =
  self.generalization.parent
allParents:Set(GeneralizableElement) =
  self.parent->union(self.parent.allParents)
```

Note that such errors can be found by any UML modelling tool that supports the UML 1.4 syntax and well-formedness rules. See figure 2.15 for an example screenshot of a modelling tool complaining about an ill-formed model.

Transitivity of the child-to-parent relationship is already expressed in the recursive definition of `allParents`.

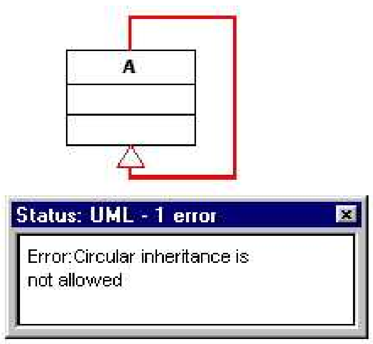


(a) Self-inheritance is not allowed



(b) Circular inheritance is not allowed

Figure 2.14: Two ill-formed models



Error:Circular inheritance is not allowed

Figure 2.15: An error is indicated by a modelling tool

2.3.4 OCL and Refinement

Class diagrams without constraints define only structure, no behaviour. In particular, operations that are not constrained can return arbitrary results and manipulate its classes' attributes arbitrarily.

Any UML model can be constrained by attaching constraints to one or more model elements. These constraints can in principle be given in any (including natural) language, but the appropriate way of constraining models precisely is to use the Object Constraint Language (OCL, [28, chapter 6]). Defining behaviour with statechart diagrams, as well as the relation between these two kinds of constraining classes, will be examined in section 4.2.

The most interesting and widely used case of OCL constraints are *class invariants* and *operation pre- and postconditions*. According to the principles of "design by contract" and "subcontracting" (as introduced by Meyer [20] and built into the Eiffel programming language), such OCL constraints on classes and operations must follow some well-known rules when specifying subtypes:

- Any invariant constraint of the subclass must be identical with or stronger than the corresponding invariant constraint of the superclass.
- For each operation, any precondition of the subclass's operation must be identical with or weaker than the corresponding precondition of the superclass's operation.
- For each operation, any postcondition of of the subclass's operation must be identical with or stronger than the corresponding postcondition of the superclass's operation.

Figure 2.16 shows a simple pair of invariants that fulfils the contract.

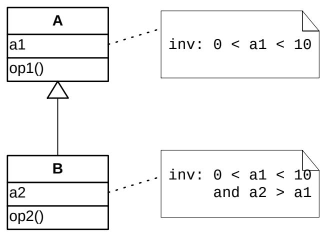
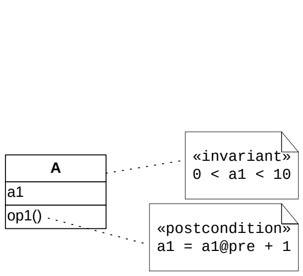
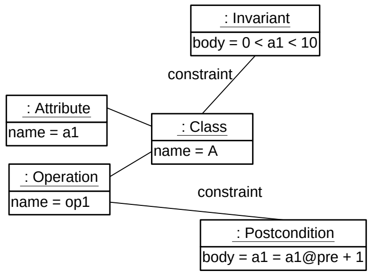


Figure 2.16: A simple example of an inheritance contract

There is no construct defined in UML or OCL to access invariants or pre-/postconditions. But after examining how these constraints are connected to classes and operations in the abstract syntax, such functions can be defined. Figure 2.17 shows how constraints stereotyped as «`invariant`», «`precondition`», or «`postcondition`» map to the abstract syntax.



(a) Concrete notation



(b) Abstract syntax

Figure 2.17: OCL constraints in stereotyped notes

As classes and operations may have multiple constraints, a function is defined that conjuncts them (textually). Note that OCL functions can be defined recursively. They are not meant to have some operational behaviour, but just be a logical constraint for an implementation. The rather long expression `set->asSequence() ->first()` is needed to access a singleton set's only element; there is no shorter way.[6]

```
let conjunctExpressions(set:Set(BooleanExpression))
  :BooleanExpression =
  if set->isEmpty()
    then 'true'
    else
      if set->size() = 1
        then set
        else
          let first = set->asSequence()->first()
            conjunctExpressions( set->excluding(first) )
              .concat(' and ').concat(first)
      endif
  endif
let getInvariant(C:Class):BooleanExpression =
  conjunctExpressions(C->constraint->collect(
    invar:Invariant | invar.language = 'OCL' ))
let getPrecondition(Op:Operation):BooleanExpression =
  conjunctExpressions(C->constraint->collect(
    prec:Precondition | prec.language = 'OCL' ))
let getPostcondition(Op:Operation):BooleanExpression =
  conjunctExpressions(C->constraint->collect(
    postc:Postcondition | postc.language = 'OCL' ))
```

Definition 2.3.7 (Fulfilment of an Inheritance Contract). Let class `C'` be a sub-

[6]Perdita Stevens comments on this in `www.cs.york.ac.uk/puml/puml-list-archive/0082.html`: "I agree that OCL is fundamentally broken in this respect."

class of `C`. The two classes *fulfil their inheritance contract*, if
`getInvariant(C′)`$\Longrightarrow$`getInvariant(C)`
For all operations `op` in `C`:
`getPrecondition(C::op())`$\Longrightarrow$`getPrecondition(C′::op())`
For all operations `op` in `C`:
`getPostcondition(C′::op())`$\Longrightarrow$`getPostcondition(C::op())`

A notation other than using stereotyped constraints, is giving the context of a constraint with the `context` keyword. Let us revisit the account example, given in figure 2.18. Class `Account` defines the behaviour of operation `withdraw` using a pair of `getBalance()` queries: The balance before and after the execution of the operation. The subclass `TransientAccount` has a different postcondition for `withdraw` using the attribute `balance`. Additionally, a postcondition for `getBalance` specifies that this attribute value is returned as the result of `getBalance`. Thus, the contract can be shown to be fulfilled, with the result that `TransientAccount` is a subtype of `Account`.

Proof. The operation postconditions of the superclass can be deduced from the operation postconditions of the subclass:[7]

```
    getPostcondition(TransientAccount::getBalance():Integer)
 =  getBalance() = balance
 ⇒  false
 =  getPostcondition(Account::getBalance():Integer)

    getPostcondition(TransientAccount::withdraw(
         amount:Integer ))
    and
    getPostcondition(TransientAccount::getBalance():Integer)
 =  balance = balance@pre - amount
    and getBalance() = balance
 ⇒  getBalance() = getBalance@pre() - amount
 =  getPostcondition(Account::withdraw( amount:Integer ))
```

□

The UML specification leaves open some questions regarding consistency of OCL contracts. One situation not mentioned is that there can be contradicting invariants within a class diagram, see figure 2.19. Two possible interpretations of this situation are that either the model is ill-formed (but it is not, according to [28]), or that the model is well-formed but class `A` may never be instantiated. Another question is how classes that are constrained with OCL are related to ones not constrained. In order to be compatible with inheritance contracts, one can

[7]Under the assumption that `getBalance()` does not *change* the value of `balance`, which could have been denoted by the stereotype «`query`».

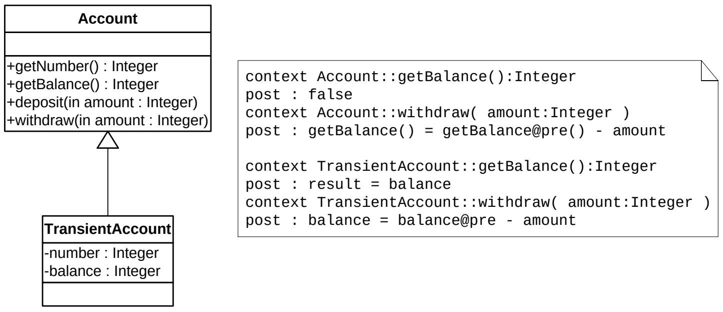

Account
+getNumber() : Integer
+getBalance() : Integer
+deposit(in amount : Integer)
+withdraw(in amount : Integer)
TransientAccount
-number : Integer
-balance : Integer
context Account::getBalance():Integer
post : false
context Account::withdraw(amount:Integer)
post : getBalance() = getBalance@pre() - amount
context TransientAccount::getBalance():Integer
post : result = balance
context TransientAccount::withdraw(amount:Integer)
post : balance = balance@pre - amount

Figure 2.18: Accounts with an OCL contract

assume that classes without OCL have no invariant, operations may not be called and return with arbitrary behaviour, i.e. `inv:true`, `pre:false` and `post:true`. But this is surely not appropriate in every situation.

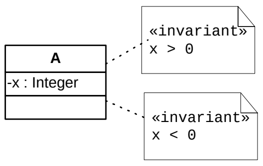


Figure 2.19: Contradicting invariants

2.3.5 Refinement of Packages

The concept of generalising and specialising packages in the UML is not suitable for this component concept, for two reasons. On the syntactical level, it lacks the possibility of aliases and changing visibility (compared to an import). More importantly, the intuition of a generalisation/specialisation relation is not matched if applied to component parts: It is hardly imaginable that a component body *is a special kind of* a component provisions part, or that a component requirements part *is a more general case of* a component provisions part.

The refinement of packages, as it will defined shortly, is mainly an extension of the refinement of single classes. At first sight, a package is just a set of classes and other model elements.

A New Refinement Stereotype

A component *provides* services to its environment. Thus, it is most intuitive to define a stereotype reflecting this. The stereotype «`provide`» is a sub-stereotype of both Import and Refinement. This is syntactically legal ([28, 2.6.4]) and reflects the intuition that we want.

Definition 2.3.8 (Stereotype «`provide`»). An abstraction dependency may be stereotyped as «`provide`» if the following constraints hold:

[1] «`provide`» dependencies exist only between packages (not between arbitrary namespaces).

```
context Provide inv:
self->client->oclIsKindOf(Package)
and
self->supplier->oclIsKindOf(Package)
```

[2] Every element in the refined (abstract) package is public.

```
context Provide inv:
allPublic(self->supplier)
```

[3] Every element is imported as public into the refining (concrete) package.

```
context Provide inv:
allImportsPublic(self->client, self->supplier)
```

[4] No generalisations from the refined (abstract) to the refining (concrete) package exist.

```
context Provide inv:
not existsGeneralization(self->supplier, self->client)
```

A refinement transformation of the same kind arises when connecting two components.

This stereotype can be used to define refinement transformations of packages in the sense of [9], this is done in section 3.1. A different, stricter definition of refinement of packages is shortly discussed in section 2.7

2.4 Components

Components will now be defined, using stereotypes for packages. The purpose of the stereotypes is twofold: On one hand, they are a notation convention that helps to oversee a large diagram 'at first sight'. On the other hand, they force the packages to fulfil constraints that are needed for the compositionality results in the next chapter.

Definition 2.4.1 (Provisions, Body, Requirements Stereotypes). A package can be stereotyped with «`provisions`», «`body`» or «`requirements`» if all owned elements are public. In OCL:

```
context Provisions inv:
allPublic(self)

context Body inv:
allPublic(self)

context Requirements inv:
allPublic(self)
```

Definition 2.4.2 (Component Stereotype). A package can be stereotyped with «`component`» if all owned elements are public, it contains exactly one package stereotyped with «`body`», and it contains only packages stereotyped with «`provisions`», «`body`» or «`requirements`». In OCL:

```
context Component inv:
allPublic(self)
and
self->ownedElement->select(p:Package
  | p.oclIsKindOf(Body))->size() = 1
and
self->ownedElement->forAll(p:Package
  | p.oclIsKindOf(Provisions) or p.oclIsKindOf(Body)
    or p.oclIsKindOf(Requirements))
```

There are some remarks to be made about the fact that everything shall be publicly visible, see section 2.7 for a discussion.

Definition 2.4.3 (Component). A *component* is a package with the stereotype «`component`» (and must obey this stereotype's constraints).

The account types seen in the examples before can be modelled as a component, see figure 2.20.

2.4.1 Notation variants

Associations, generalisations and other relations between elements from an imported package and an importing package can be denoted in two ways: The imported elements are either duplicated in the importing package, or they are drawn only once and associations etc. are drawn over package borders. In fact one could duplicate a model element in a diagram arbitrarily often, the notations are equivalent and unambiguous on the abstract syntax level. In particular, the association, dependency or generalisation objects must always be owned by the importing, not by the imported package, because the participants from the importing package are not visible in the imported package. See figure 2.21 for an example (remember that «`provide`» is a special kind of «`import`»).

Thus, there are also two equivalent notations for a component: The first includes provisions and requirements classes in the body package and is more complete; the second leaves them out and is thus more compact. In the rest of the thesis,

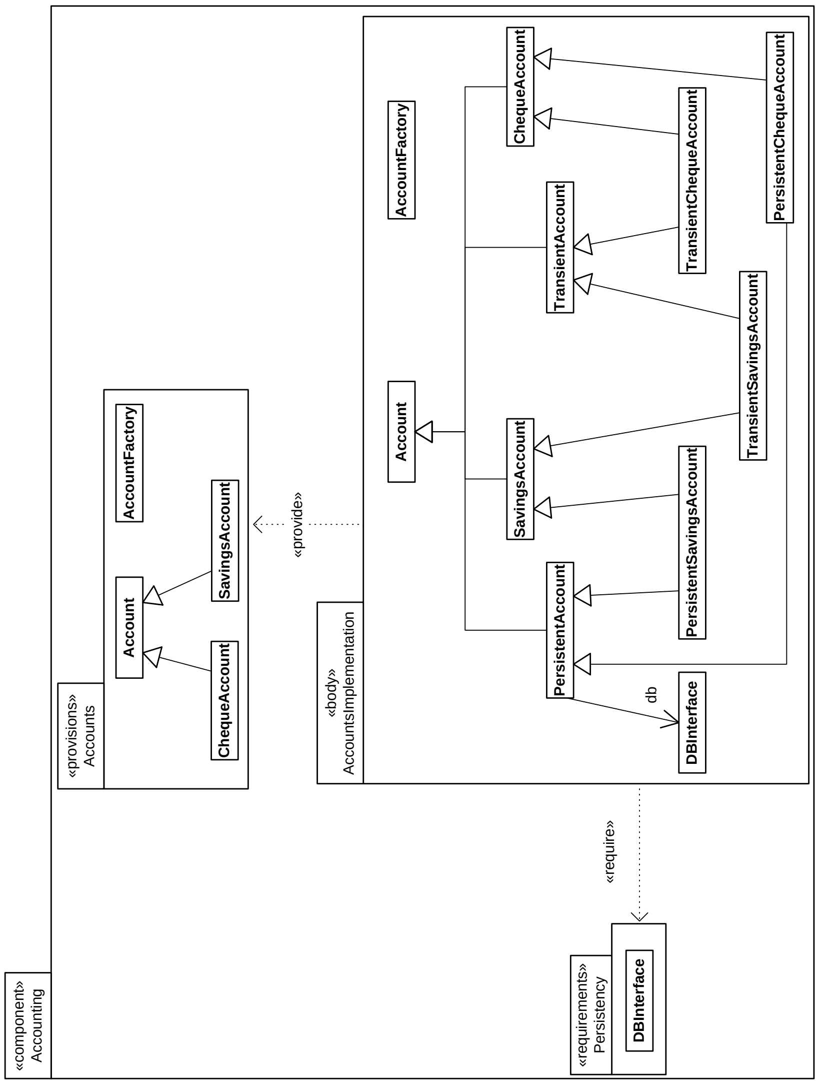
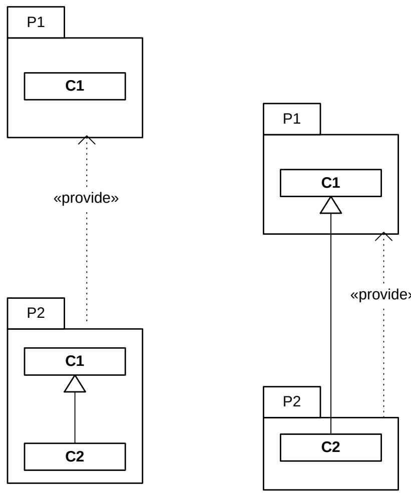
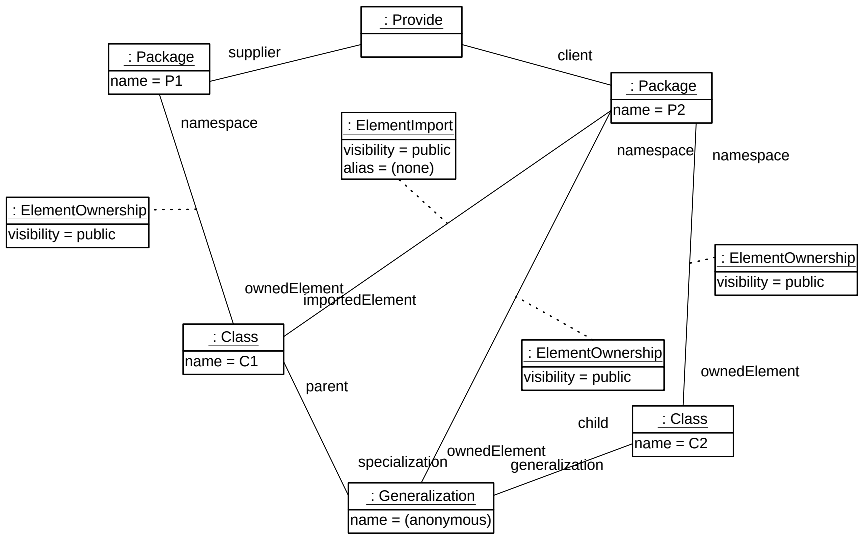


Figure 2.20: The accounting component



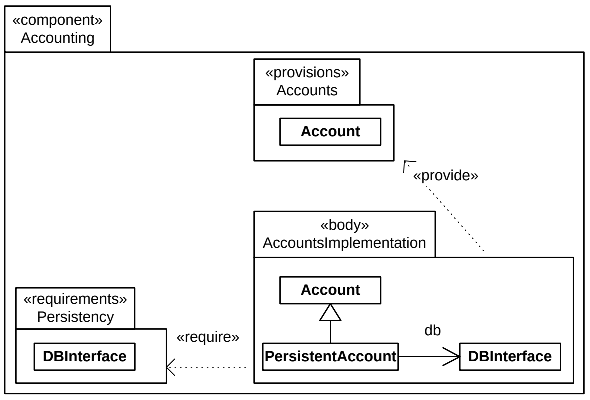
(a) Detailed notation (b) Short notation



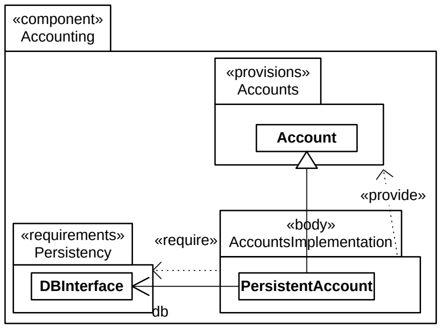
(c) Abstract syntax

Figure 2.21: A generalisation between an owned and an imported element

the short notation is preferred. An example is a shortened version of the accounting component in figure 2.22.



(a) Detailed notation



(b) Short notation

Figure 2.22: Notation variants for excerpt of the accounting component

2.5 Component Interface Specifications

A component consists of interfaces and a body. But there are situations where the details of the realisation of the interfaces, the component body, is not interesting. The need for such constructs is already recognised in [8, section 5B] (where *module* stands for *component* in current terminology):

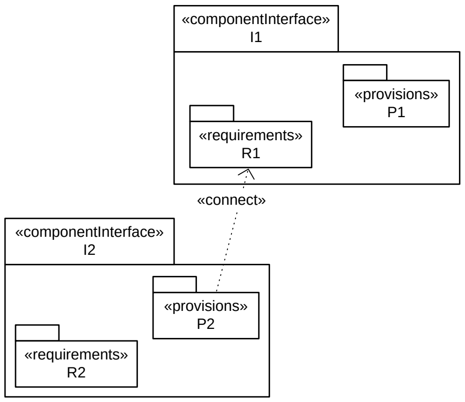
”From the software development point of view interface specifications are very useful to be considered before the specification of modules and modular systems. (...) In order to develop a modular system it is advisable to start with the interface specifications of the corresponding modules and to define the interconnection of the modular system already on the level of interface specifications.”

Definition 2.5.1 (Stereotype «`componentInterface`»). A package may be stereotyped as «`componentInterface`» if the package contains only packages that are either provisions or requirements.

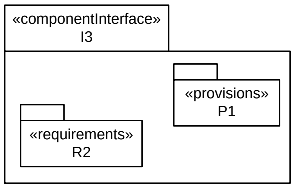
```
context ComponentInterface : inv
self.contents->forAll( p:Package |
  p.oclIsKindOf(Provisions) or p.oclIsKindOf(Requirements))
```

Definition 2.5.2 (Component Interface Specification). A *component interface specification* is a package stereotyped as «`componentInterface`».

An example is given in figure 2.23 (a): The diagram states which services are provided and required by the components, and how they are hierarchically composed. The realisation of the interfaces must be specified in a later development phase.



(a) Two component interface specifications and a connector



(b) The composed component interface specification

Figure 2.23: Component interface specifications

Component interface specifications are particularly interesting as they are conceptually similar to components in the CORBA Component Model, see section 6.6.

2.6 A UML Profile for Component Modelling

The extension mechanisms of the UML allow to define new stereotypes and datatypes in the context of a *profile*. The intention of a profile is to restrict the UML to a certain application or technology domain. A well-known example is the ”UML Profile for CORBA”, which is itself an OMG standard [14]. This profile aims at providing a means to express CORBA IDL specifications with the UML, and to be able to convert UML models to CORBA IDL. Among other elements, stereotypes for classifiers (such as «`CORBAInterface`») and for packages («`CORBAModule`») and CORBA datatypes (for example `unsigned long`) are introduced.

The component modelling profile is defined as a package, shown in figure 2.24.[8] This kind of diagram is also called ”virtual meta-model”, because stereotypes define ”virtual subclasses” in the meta-model (the meta-model is not really changed). It is equivalent to the textual representation in table 2.1. Note the difference between the name of an element in the virtual meta-model, which by convention starts with a capital letter (e.g. `Provide`, `Component`), and the notation of the stereotype, which starts with a lower letter (e.g. «`provide`», «`component`»). Stereotypes defined in a profile can also be given a new visual icon, but this is omitted in this thesis.

A profile is formally applied to a model by adding an «`appliedProfile`» dependency between the profile package and the model package, see figure 2.25. This has the effect that the new stereotypes are available, and that all constraints on the stereotyped model elements must hold. When modelling with OCL, a stereotype’s profile can be found using the additional operation `findProfile(): Set(Package)`, e.g. `Provide->findProfile() = Set {ComponentModelling}`.

The stereotypes that are defined by this profile are summarised in table 2.1.

2.7 Discussion of Design Decisions and Alternatives

Constraining Models with Stereotypes

The component concept introduced in this thesis uses the UML extension mechanisms, namely profiles and stereotypes, for its definition. There would have been other choices:

The component concept could have been described *informally* without making use of the profile mechanism. This would have saved some effort in writing, but as

[8]This is a non-standard notation, used e.g. in [14], which is more elegant than the one specified in [28].

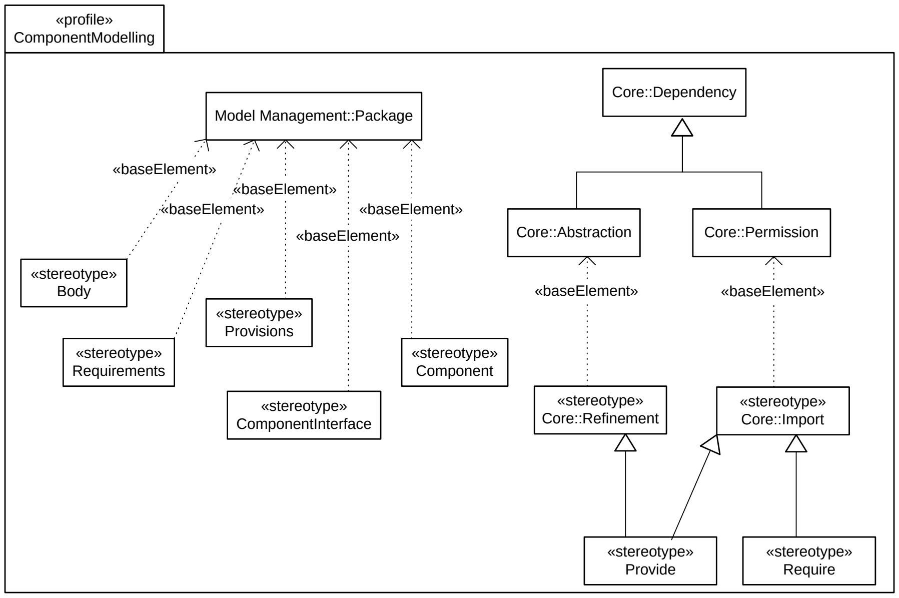


Figure 2.24: Component modelling profile

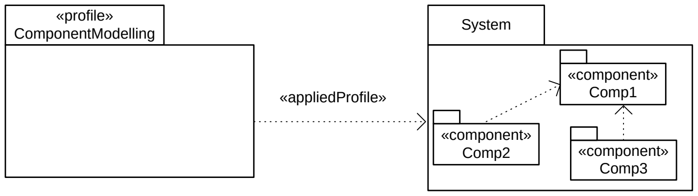


Figure 2.25: Application of a profile

Name	**Base Class**	**Parent**	**Constraints**	**Description**
`«requirements»`	Package	none	see sect. 2.4	Specifies services required by a component
`«body»`	Package	none	see sect. 2.4	Specifies the realisation of services provided by a component
`«provisions»`	Package	none	see sect. 2.4	Specifies services provided by a component
`«component»`	Package	none	see sect. 2.4	Specifies a component
`«componentInterface»`	Package	none	see sect. 2.5	Specifies a component interface
`«require»`	Dependency	`«import»`	see sect. 2.2.2	Connects a component's body part with a requirements part
`«provide»`	Dependency	`«import»`, `«refine»`	see sect. 2.3.5	Connects a component's body part with a provisions part, or a component's requirements part with another component's provisions part

Table 2.1: Stereotypes of the component modelling profile

the relation between the existing UML specification would be unclear and possibly imprecise, hardly any existing modelling tool could be extended to deal with such a description.

Similarly, the existing semantics of UML model elements could have been altered, e.g. by saying ”from now on, each import dependency sets visibility as public”. This is an absolutely inappropriate approach because it would lead to confusion among modellers, it would induce incompatibilities between models designed within and outside this framework, and would clearly contradict the UML specification (which is, not to forget, a standard used widely in software industry).

New meta-model elements, especially a meta-class `Component` could have been added to the UML specification, together with its own well-formedness rules and its own semantics specification. This was not done because substantial effort would arise to develop tools that handle the new constructs. Such extensions are referred to in [28] as ’heavy-weight’ extensions that are possible in principle owing to the MOF meta-meta-model, whereas using profiles and stereotypes, without altering the meta-model, is called ’light-weight’ extension.

Summarising, choosing the UML extension mechanisms was the right thing to do with respect to a possible tool assistance. But it is another question whether the definitions introduced are suitable from the point of view of *usability* in software design. First, one has to realise that the semantic impact of stereotypes is limited:

> ”A fundamental constraint on all extensions defined using the profile extension mechanism is that extensions must be strictly additive to the standard UML semantics. This means that such extensions must not conflict with or contradict the standard semantics. In effect, these extension mechanisms are a means for refining the standard semantics of UML and do not support arbitrary semantic extension.” [28, section 2.6.1]

In my opinion, the restrictions imposed by the stereotypes do not restrict models very strictly. Most of the constraints are concerned with visibility and consistency of naming. One detail that is probably one of the most discussable is that the need to define all package contents publicly visible *breaks encapsulation*. Granted, but this is a simplification chosen in order to have a simple definition of consistency of refinements and inclusions. One could imagine more flexibility in this area, but this would imply a stricter and probably more complicated compatibility definition, because careless use of import arrows can lead to ill-formed models (section 2.2.1). The intuition of visibility in component modelling is different from the one in programming. Component body parts are ’invisible’ to the user of a component; the interface parts of are component must always be visible completely. The UML specification also notes that visibility is a minor problem on the modelling level ([28, p. 3-45]): ”Actually all forms of nonpublic visibility are language-dependent”.

The only constraint that really restricts the expressiveness of UML is the prohibition of generalisations from a provisions part to a body part. But this restriction naturally reflects the intuition of components. This fulfils another requirement stated in [28, section 2.6.1]:

> ”When defining profiles modelers should be careful to base their extensions on the most semantically similar constructs in the UML meta-model. Failure to observe this can easily result in semantically incorrect or semantically redundant language extensions.”

Although the existing «`import`» and «`refine`» stereotypes could not be used, sub-stereotypes of these were defined that kept the intuitive meaning and added the needed syntactical strength.

Different Approaches to Components

The component notion used in the IOSIP case study leaves open the kind of relation between the component parts. Three different approaches are illustrated here. The concept defined in this thesis is a kind of compromise between the first two.

First, component parts could be connected by associations between their classes. At runtime, this would result in a chain of delegations of method calls along links that connect the objects. Figure 2.26 shows how a class diagram (without showing packages) would look like. Calls to a provisions object are delegated to the body object, possibly delegated again to the requirements object and so on. This could result in a large number of objects and a large number of method calls, which seems both confusing and inefficient.

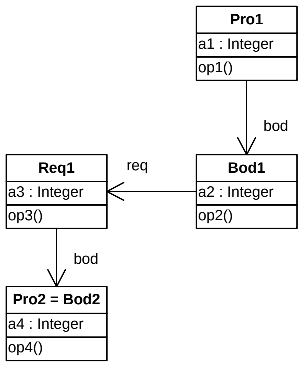
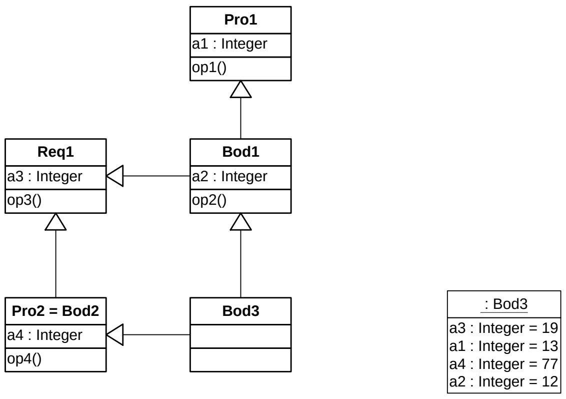


Figure 2.26: Components using delegation

Second, use of delegation could abandoned completely, by using inheritance only. Figure 2.27 (a) shows the same components, but with arrows indicating inheritance. There are several problems with this structure: At runtime, the composed system would collapse into *one* object, see figure 2.27 (b), having all attributes and methods defined in the components, which seems quite un-intuitive.

The fact that class `Bod3` inherits from class `Req1` via two different paths, a diamond or rhombic inheritance situation, is difficult because the UML is imprecise about which method implementation to choose if an operation is defined in both `Bod2` and `Bod1`. Additionally, the vertical and horizontal inheritance have slightly different meanings (the vertical ones are subtypes, the horizontal ones implementation inheritance), which is dangerously confusing.



(a) Class diagram (b) Object diagram

Figure 2.27: Components using inheritance

A third possible definition, having only interfaces and no classes in component provisions and requirements, can be ruled out as an instantiation of the Generic Framework for syntactical reasons immediately. But this approach is taken in several other works, for example in UML components (see section 6.2) and in [23]. The problem when having interfaces in the provisions and requirements parts, and classes implementing the interfaces only in the body, is that one gets different types of refinement transformations in different parts of a component. As illustrated in figure 2.28, provisions and body are connected by an *interface realisation* relationship. Requirements of the upper and provisions of the lower component are connected by *interface extension*. `Bod1` and `Bod3` are connected by an *implementation inheritance* relation between classes. This contradicts the idea of a class of uniform and composable refinement transformations. Additionally, section 2.1.1 shows why interfaces are often not sufficient for definition of services.

In object-oriented modelling and programming, inheritance and delegation can almost always be replaced by each other, but the most intuitive and appropriate, and syntactically inproblematic definition was chosen for this thesis, using inheritance in the vertical and delegation in the horizontal dimension (but still allowing interface realisation).

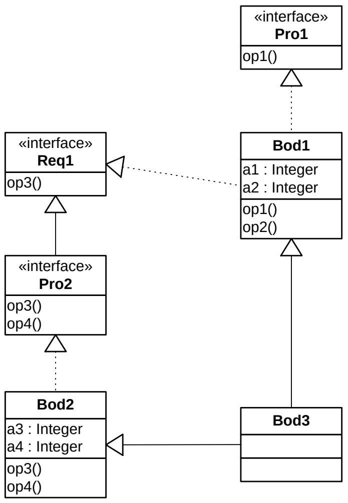


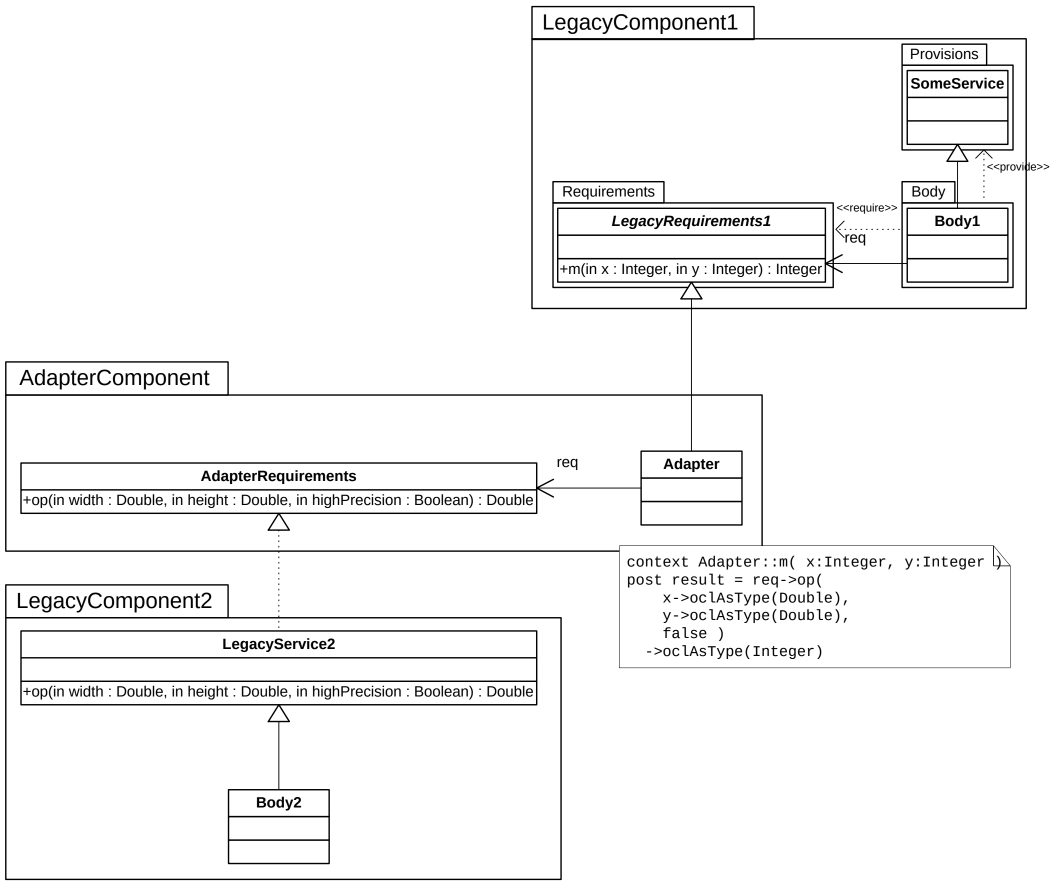
Figure 2.28: Components using interfaces only

A Stricter Definition of Package Refinement

A stricter version of refinement could *force* every class to be refined by a subclass. As a consequence, the refinement could not be a reflexive relation and thus it could not be a partial order, as many refinement calculi traditionally require. Nevertheless, it is imaginable that such a refinement could be useful for components.

Definition 2.7.1 (Strict Package Refinement). Package `P2` refines package `P1` if for *every* class in `P1` there is a subtype in `P2`.

One advantage of the loose variants chosen for the component concept allow ”trivial” components with provisions = body. These can, for example, used as adapters between components, for example in order to reuse unchangeable legacy components of a system. An example can be seen in figure 2.29: Neither the requirements part of `LegacyComponent1` nor the provisions part of `LegacyComponent2` can be changed. An adapter component implements the needed data conversions and signature adaptations. This is one of several typical component usage scenarios introduced in [19].



LegacyComponent1
Provisions
SomeService
<<provide>>
Requirements
LegacyRequirements1
+m(in x : Integer, in y : Integer) : Integer
<<require>>
req
Body
Body1
AdapterComponent
AdapterRequirements
+op(in width : Double, in height : Double, in highPrecision : Boolean) : Double
req
Adapter
LegacyComponent2
LegacyService2
+op(in width : Double, in height : Double, in highPrecision : Boolean) : Double
Body2

```
context Adapter::m( x:Integer, y:Integer )
post result = req->op(
    x->oclAsType(Double),
    y->oclAsType(Double),
    false )
  ->oclAsType(Integer)
```

Figure 2.29: An adapter component connecting legacy components

Chapter 3

Instantiation of the Generic Component Framework

3.1 Specifications, Transformations, and Inclusions

UML as a Generic Modelling Technique

The motivation for this thesis was to find out if the component concept used in the IOSIP project can be an instantiation of the Generic Component Framework introduced in [9]. There, general properties are given that a specification technique should have in order to describe systems. They are given here with a short justification why the UML class diagram components fulfil the requirements.

- *Specifications as the syntactical part:*
  The UML syntax is formalised with a meta-model and OCL well-formedness rules. One can define a package to be a *specification*.

- *Behaviour or models as the semantical part:*
  Semantics of UML class diagrams are specified in English language.

- *A constraint language in order to express properties:*
  The Object Constraint Language (OCL) fulfils the requirements of a non-formal constraint language.

- *Horizontal and vertical structuring techniques:*
  Notions of inclusion and refinement of UML packages were worked out in the previous chapter. Their properties will be further examined now.

Package Inclusion

The additional constraints for import dependencies defined in the previous chapter justify that a «`require`» dependency is an inclusion as required by the Generic Framework.

Definition 3.1.1 (Inclusion of Packages). Package `A` *includes* package `B` if there is a «`require`» dependency from `A` to `B`. In OCL:

```
let isPackageInclusion(A:Package, B:Package):Boolean =
  A->clientDependency->exists( r:Require |
    r->supplier->includes( B ))
```

This inclusion relation is reflexive: It is possible that a package has a «`require`» dependency connected to itself, contradicting with no well-formedness rule, and having no syntactical or semantical effect.

Transitivity of this kind of inclusion is ensured by the visibility rules defined in section 2.2.2, and by the 'import by reference' as sketched on page 22.

Package Refinement as a Transformation

In order to gain a component framework that reflects the desired properties, the kind of refinement transformation used is highly important. The «`provide`» dependency was defined with these properties in mind.

Definition 3.1.2 (Refinement and Abstraction of Packages). Package `Conc` is a *refinement* of package `Abstr` if there is a *provide* dependency from `Conc` to `Abstr`.

```
let isPackageRefinement(Conc:Package,Abstr:Package):Boolean =
  Conc->clientDependency->exists( p:Provide
    | p->supplier->includes( Abstr ))
```

Package `Abstr` is an *abstraction* of package `Conc` if `Conc` is a refinement of `Abstr`.

```
let isPackageAbstraction(Abstr:Package,Conc:Package):Boolean =
  isPackageRefinement(Conc, Abstr)
```

According to [9] a transformation framework consists of a class of transformations with the following properties:

- *For each object there is an identical transformation:*
  For each package `P1` an identical refinement is an empty package `P2` that imports from `P1` (remember that the «`provide`» dependency is also an import). `P1` and `P2` have identical contents.

- *The extension property is satisfied:*
  As stated in the introduction, the transformations and inclusions used must have the *extension property*. In the next section, it is proven for the instantiation defined in this thesis.

- *Transformations are closed under composition:*
  In section 2.2.1 it is shown how imports can be made transitive. The additional property of «`provide`», forbidding specialisations from the refining to the refined package, can immediately be derived from the definition of `existsGeneralisation`.

Let `B` provide `A`, and `C` provide `B`.

```
    not existsGeneralization(A, B)
and not existsGeneralization(B, C)
```

⇒ (Definition of `existsGeneralization`)

```
not A->ownedElement->exists(
    ch:GeneralizableElement | B->ownedElement->exists(
      par:GeneralizableElement | isSpecialisation(ch,par)
    ))
and
not B->ownedElement->exists(
    ch:GeneralizableElement | C->ownedElement->exists(
      par:GeneralizableElement | isSpecialisation(ch,par)
    ))
```

⇒ (`A->contents` is a subset of `B->contents`)

```
not A->ownedElement->exists(
    ch:GeneralizableElement | C->ownedElement->exists(
      par:GeneralizableElement | isSpecialisation(ch,par)
    ))
```

⇒ (Definition of `existsGeneralization`)

```
not existsGeneralization(A, C)
```

3.2 Extension Property

The extension property expresses a ”locality assumption” in the transformation framework, meaning that transformations can applied to a larger specification in a similar way than in the smaller. In the UML instantiation chosen, this property is obvious and almost trivial, as there are only additive changes to models (or, in terms of set theory, there are only injective mappings on the sets of model elements owned by a package).

With packages as component parts, package inclusion using «`require`», package refinement using «`provide`», and the consistency condition `isNameConflictFree`, it is possible to prove our models to fulfil the *extension property*. See figure 3.1 for an example: The transformation that refines `P1` by `P3` (where class `A` is refined by a subclass `B`) can equally be applied to the extended package `P2` (where `A` is associated with another class `C`).

Fact 3.2.1 (Extension Property). Given packages `P1`, `P2`, `P3`, a «`require`» dependency `r`, and a «`provide`» dependency `p` as in figure 3.2, with `isNameConflictFree(P2,P3`.

There is a package `P4`, a «`require`» dependency `r’` and a «`provide`» dependency `p’` as in figure 3.2.

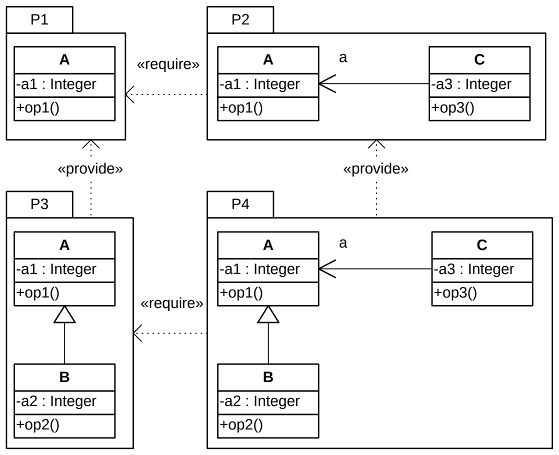


Figure 3.1: Example of extension property

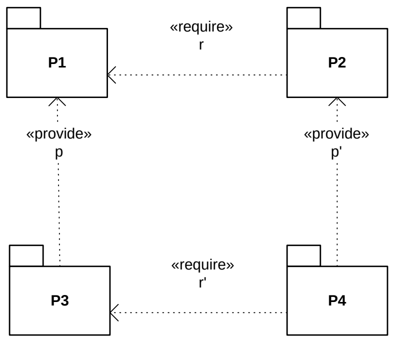


Figure 3.2: Extension diagram for packages

Proof. P4 is constructed as an empty package with P2 and P3 imported. As shown in section 2.2.1, elements that are owned by P1 and imported into P2 and P3, are identified in P4. Because of the name conflict freeness of P2 and P3, P4 is well-formed, as also shown in section 2.2.1.

As P3 and P4 contain only public elements, p′ fulfils the needed visibility constraints of a «provide» dependency. The additional property needed for p′ in order to be a «provide» dependency, is the non-existence of generalisations from P2 to P4. Informally, this is the case because no class in P2 can be a subclass of a class in P3 as the package contents are disjoint except for the common content from P1. But this can also be deducted from the assumptions using OCL:

```
not existsGeneralization(P1,P3)
and P4->contents = P3->contents->union(P2->contents)
and P2->contents->intersection(P3->contents) = P1->contents
```

⇒ (Definition of existsGeneralization)

```
not P1->ownedElement->exists(
    ch:GeneralizableElement | P3->ownedElement->exists(
      par:GeneralizableElement | isSpecialisation(ch,par)
  ))
and P4->contents = P3->contents->union(P2->contents)
and P2->contents->intersection(P3->contents) = P1->contents
```

⇒ (There can be no relationship between elements of P2 and P3 except for those from P1)

```
not P1->ownedElement->exists(
    ch:GeneralizableElement | P3->ownedElement->exists(
      par:GeneralizableElement | isSpecialisation(ch,par)
  ))
and P4->contents = P3->contents->union(P2->contents)
and not P2->ownedElement->exists(
        ch:GeneralizableElement | P3->ownedElement->exists(
          par:GeneralizableElement | isSpecialisation(ch,par)
  ))
```

⇒ (P4 does not add anything that was not in P2 or P3)

```
not P2->ownedElement->exists(
    ch:GeneralizableElement | P4->ownedElement->exists(
      par:GeneralizableElement | isSpecialisation(ch,par)
  ))
```

⇒ (Definition of existsGeneralization)

```
not existsGeneralization(P2,P4)
```

r′ fulfils a «require» dependency's visibility constraints because all elements are public in both packages P3 and P4. The additional property, not existsGeneralization(P4. P3), is shown now, in an analogue way as above. Because the contents of P4 are the union of P3 and P2, and P3 is a subset of P4, only remains to show `not existsGeneralization(P2. P3)`. `P2` and `P3` are disjoint except for the contents of `P1`. As `not existsGeneralization(P2. P1)` and `P1` is a subset of `P2`, `not existsGeneralization(P4. P3)` follows. □

3.3 Vertical and Horizontal Composition of Extension Diagrams

Extension diagrams exist only if the consistency condition holds. Analogously, composition of extension diagrams is only possible if the consistency condition can always be assured when composing extension diagrams horizontally or vertically. Figure 3.3 shows a horizontal composition of extension diagrams for packages.

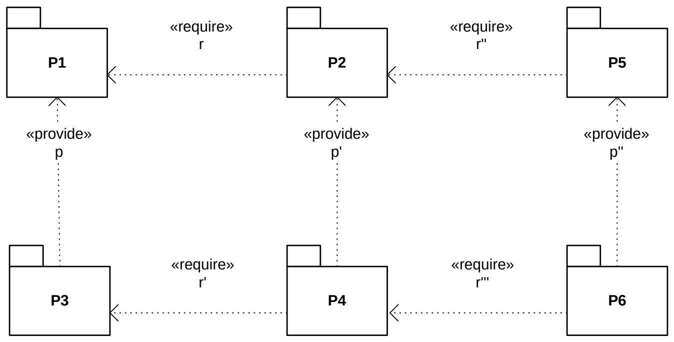


Figure 3.3: Horizontal composition of extension diagrams

Fact 3.3.1 (Horizontal Compositionality of Extension Diagrams). Given packages `P1` to `P6` and `«require»` and `provide` as in figure 3.3.
If `isNameConflictFree(P2, P3)` and `isNameConflictFree(P4, P5)`, then `isNameConflictFree(P3, P5)`.

Proof. There are no name conflicts between `P4` and `P5`. Because `P3` is a subset of `P4`, there can be no element in `P3` that would cause a name conflict between `P3` and `P5`. □

A similar diagram and proof can be constructed for vertical compositionality.

3.4 Hierarchical Composition of Components

Composition of components has already been seen by examples. In order to make the mapping of the UML component concept into the Generic Framework complete, the composition of single-interface components is now defined.

Definition 3.4.1 (Single-Interface Component). A component is called *single-interface component* if it contains exactly one provisions package and exactly one requirements package. In OCL:

```
context Component def:
let isSingleInterface():Boolean =
  self->ownedElement->select(p:Package
    | p.oclIsKindOf(Provisions))->size() = 1
  and
  self->ownedElement->select(p:Package
    | p.oclIsKindOf(Requirements))->size() = 1
```

Definition 3.4.2 (Connector). A connector between single-interface components `C1` and `C2` is a «`provide`» refinement between one component's requirements and the other's provisions package.

Definition 3.4.3 (Hierarchical Composition). The *hierarchical composition* of single-interface components `C1` and `C2` via a connector `conn` (see figure 3.4) is a single-interface component `C3` with the provisions of `C1` as its provisions part, the requirements of `C2` as its requirements part, and a package `Bod3` as its body part. Package `Bod3` as well as a «`provide`» dependency and a «`require`» are implicitly given by the extension property as shown in section 3.2, and by transitivity of refinements and inclusions.

According to the Generic Framework, hierarchical composition of single-interface components is possible if the component modelling technique has the extension property. This has been proven in section 3.2. The main practical effect of the generic compositionality results for the UML components defined here is that consistency has only to be checked *locally* when composing arbitrarily large systems hierarchically.

A different kind of composition called *structured composition* could also have been defined by requiring that both of the existing components and the connector form the new component's body, see figure 3.5. The flattened composition is more compact in notation, where the second preserves the underlying structure. Also note that the new provided and required interface packages could have been given alias names by the «`require`» and «`provide`» dependencies.

A larger and more detailed example of composition of components is shown in figures 5.16 and 5.17 (pages 92 and 93, respectively). There, multiple interfaces exist, and they are given alias names when composing.

3.5 Multi-Interface Components and Partial Composition

In many situations is is useful to define components which require services from more than one component, and which offer services to more than one component. This leads to a definition of components that have multiple provisions and requirements interfaces. The components as defined in 2.4 are such multi-interface components.

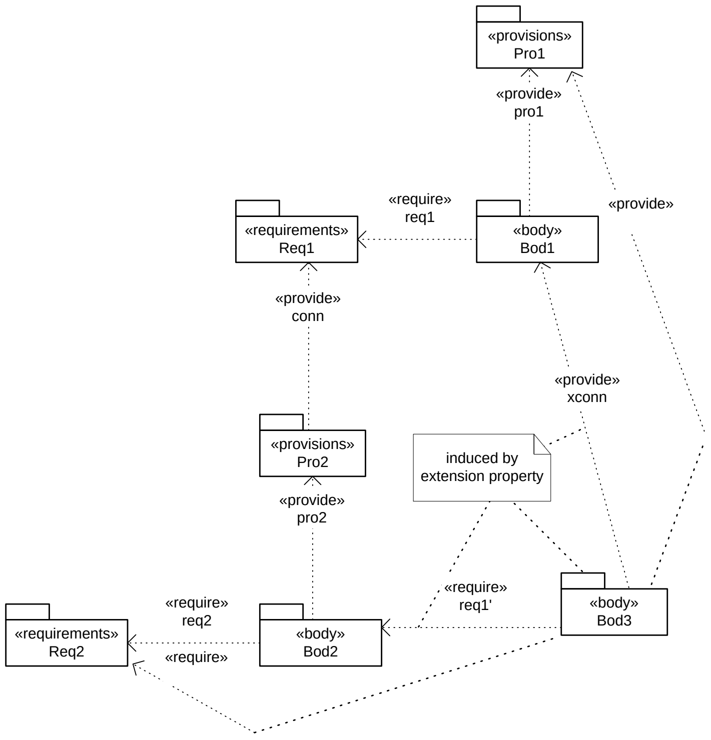


Figure 3.4: Composition of components

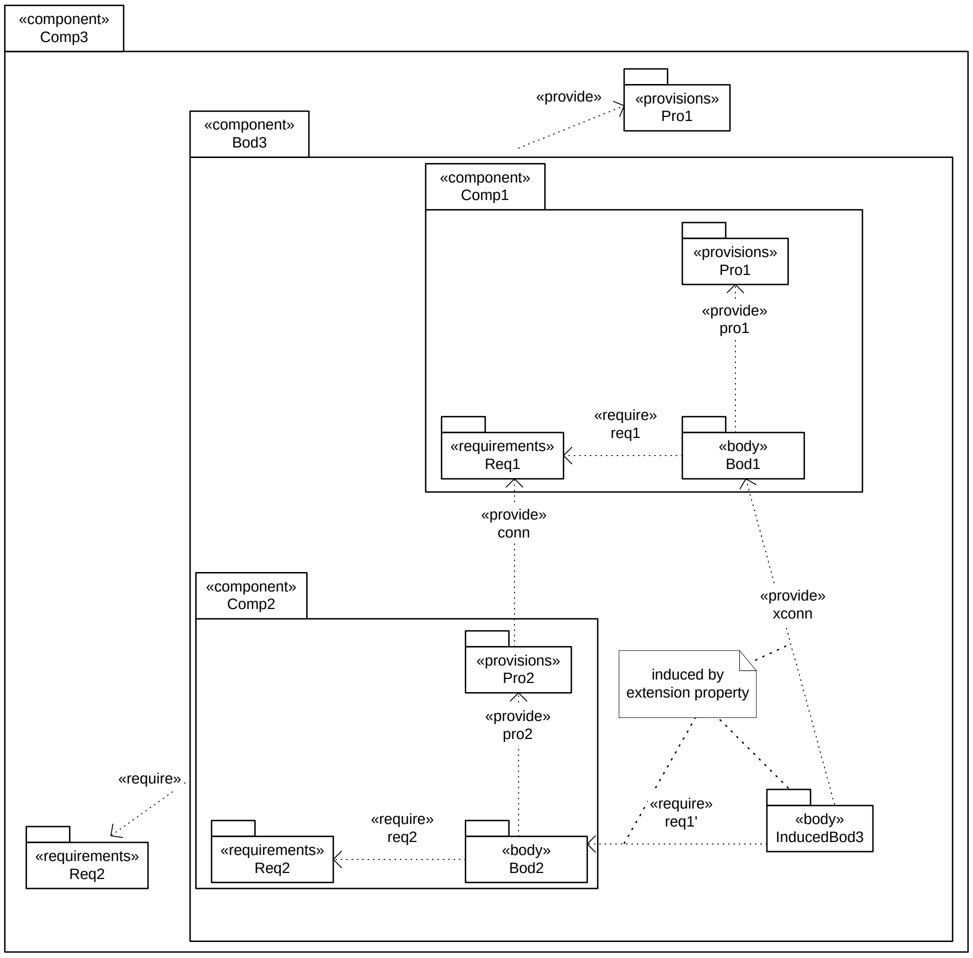


Figure 3.5: Structured composition of components

According to [10], where an instantiation of the Generic Framework with High-Level Replacement Systems is examined, the partial hierarchical composition of multi-interface components is also possible if the following properties hold additionally (see figure 3.6 for the extension diagram in the multi-interface case):

> ”The existence of a unique inclusion imp and a unique transformation $trafo$ induced by the families $(imp_i), i = 1; ...; n$ and $(trafo_i), i = 1; ...; n$ respectively. The existence of the inclusion imp is equivalent to the disjointness of the images of the import interfaces in BOD, while the existence of the transformation trafo has to be provided by the transformation framework.”

$$\begin{tikzcd}
\sum_{i=1}^{n} IMP_i \arrow[r, Rightarrow, "trafo"] \arrow[d, hook, "imp"'] & \sum_{i=1}^{n} SPEC_i \arrow[d, hook, "imp'"] \\
BOD \arrow[r, Rightarrow, "trafo'"] & SPEC'
\end{tikzcd}$$

Figure 3.6: Extension diagram for multiple interfaces

This property cannot be accordingly reformulated in the UML case, because the summed inclusion can exist very well if the images are not disjoint, and is thus not proven in this thesis. But the IOSIP case study examples, shown in chapter 5, make use of partial composition, and show that no syntactical or semantical problems seem to arise in the composed class diagrams. Some of the IOSIP example components are even composed mutually, also with no syntactical difficulties in the given situations. But in general, mutual partial composition can lead to ill-formed models, mainly because circular inheritance relations can occur.

Chapter 4

Towards a Component Framework for the UML

4.1 Components Modelled with All UML Diagram Types

A software system is usually not described only by class diagrams, but by a number of different diagram types, representing different *views* on the model. Figure 4.1 shows a rough classification of the eight digram types of the UML regarding their usage in the software development process (analysis, design, implementation) and regarding their character (static or dynamic view). The classification regarding the development process is not meant too strictly. Class diagrams, for instance, are often used in all development stages. Statechart and activity diagrams can well be used in early stages, usually replacing events and actions by some natural language.

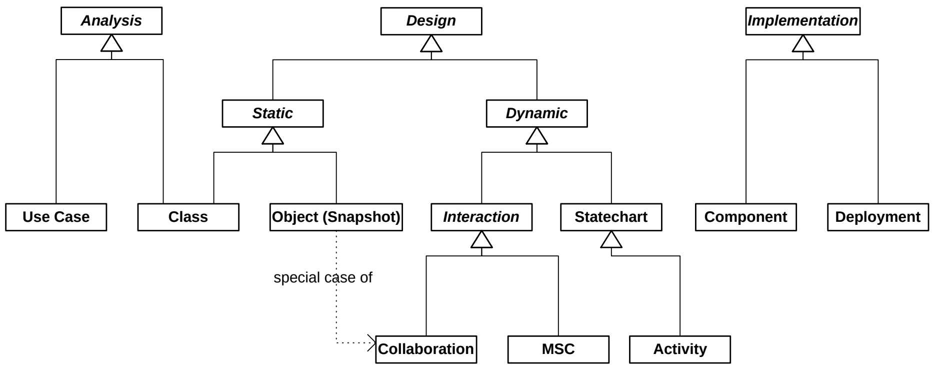


Figure 4.1: UML diagram types

Now will be very shortly discussed, how the class diagram view can be enriched by the other diagram types in component design.

- **Use Case diagrams** can be used in the very early stages of development, such as for collecting requirements, or analysing the actors and main scenarios of a system. They are informal and describe the system at a high level of abstraction. It is imaginable that a component could be adorned with a use case diagram in order to informally present the services that can be used by the environment. See figure 4.2 for an example with the well-known accounting component.

  Using use case diagrams in components raises some problems. First, required services can hardly be expressed with such diagrams. Second, the term *actor* seems inappropriate, because the environment of a component is most often another software component rather than a human actor. It is questionable if use case diagrams are helpful in a precise component model; they are mainly intended at early analysis models.

- **Statechart diagrams** are used to model behaviour. In the UML every model element can be constrained by a statechart diagram, but the most common usage is to model behaviour of *objects* with statecharts. This has interesting impacts on subtyping, which is detailedly discussed in section 4.2.

- **Activity diagrams** are semantically a special case of statechart diagrams, but with a different notation (focusing on actions rather than on states and events). This means that every activity diagram can be converted into a statechart diagram with the same operational semantics (but in general not vice versa). Therefore, they are not dealt with in the following; refer to the statecharts section. See figure 4.3 for an example of equivalent statechart and activity diagrams.

- **Message Sequence Charts** (MSCs) and **Collaboration Diagrams** have a similar relationship as statechart and activity diagrams do: They are semantically equivalent, but the notation of each emphasises different aspects. Common to both is that they show object interactions using messages. MSCs emphasise the *time* view, expressed by a vertical time line. Collaboration diagrams emphasise, as their name indicates, the collaboration roles of objects, as well as the (possibly nested) flow of control. Both can be seen as a specification that class and statechart diagrams have to conform to. There are two typical usage scenarios: On the modelling level class and statechart diagrams can be checked against the interactions (using various methodologies such as logical deduction, model checking etc.). On an implementation level, an implementation of the class and statechart diagram can be checked against test cases that are derived from the interactions (maybe using automated unit testing tools).

  Object diagrams, also called snapshots, are a syntactically restricted case of collaboration diagrams showing only static object interconnection.

  Figure 4.4 shows a sequence diagram specifying the scenario that a user withdraws exactly the amount of the current balance of an account, i.e. that he empties the account. The class and statechart diagrams of the classes involved must ensure that this scenario can occur in order to make the whole model consistent.

Components as defined in this thesis can be attached with such interaction diagrams in each part. In contrast to statechart diagrams, a refinement relation between interaction diagrams is not as important, but each component part must be consistent between class, statechart and interaction diagrams.

- **Component** and **deployment diagrams** are useful for system assembly, which nowadays is a role clearly distinct from component development. They emphasise physical and implementation aspects. Component diagrams define component types and their static interconnection, deployment diagrams show component instances, their links, and possibly their distribution on computers. The components defined in this thesis are more abstract; component diagrams may be an appropriate technique when coming close to an implementation. They are discussed further in section 6.2.

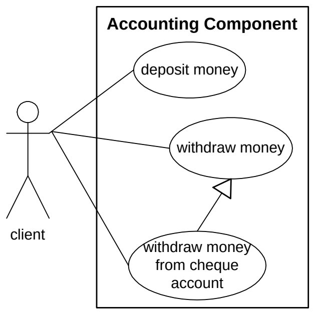
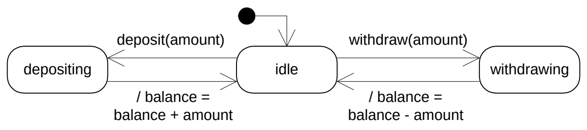


Figure 4.2: A use case diagram for the accounting component

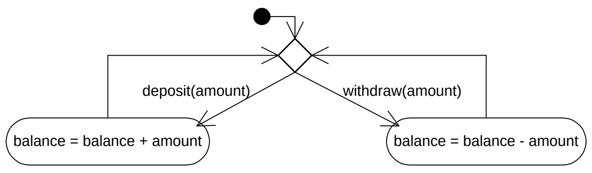
4.2 Behavioural Specification and Refinement

The most commonly used technique for modelling behaviour of objects are statechart diagrams. As per [28, sect. 2.12.1] they can be used to model the behaviour of any kind of model element. In the following, only objects – instances of classes – are modelled by statechart diagrams.

In the component framework presented in this thesis a refinement concept for classes was introduced that regards OCL specifications as semantical constraints. Another refinement concept is needed when behaviour of classes is specified with statecharts. As sketched in figure 4.5, there should be a well-defined relation between the statecharts specifying a superclass and a subclass. Constraints on this relation are developed in this section.



(a) Statechart diagram



(b) Activity diagram

Figure 4.3: Equivalent statechart and activity diagrams for `TransientAccount`

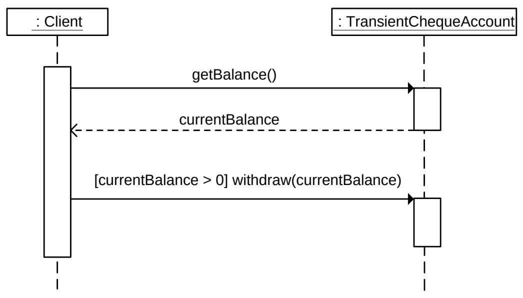


Figure 4.4: A Message Sequence Diagram representing a scenario ”emptying the account”

There is no standard way to present the relation between a class and a statechart ([28, sect. 3.74.2]). In the following, a dependency arrow will be used for this purpose. On the abstract syntax level, the model element that is specified by a statechart can be accessed using the attribute `context` of the `StateMachine` meta-class. Here, only the cases where this context is a class are dealt with.

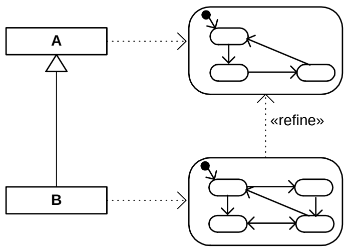


Figure 4.5: Superclass and subclass are specified with statecharts

Statecharts are specified in the UML specification with the usual techniques, i.e. a meta-model, well-formedness rules in OCL and an informal semantics description. The most important meta-model elements of statecharts are depicted in figure 4.6. Unfortunately, this description is quite imprecise regarding actions (and in some cases even contradictory, see e.g. [26]). A specification of Action Semantics is being worked out by the OMG at the time of writing and will become an addendum to the existing UML specification. Therefore, a restricted case of statecharts will be used in the following: *Protocol statecharts* are a special case of statecharts where transitions have only events and guards, but no actions.

Such statecharts and their refinements are more easy to be formalised, which is, among many other approaches, done by Engels et.al. in [11] by mapping statecharts into CSP (Communicating Sequential Processes), a formal process language, or by Ehrig et.al. in [7] by defining an operational semantics directly. Common to these and other approaches is that they deal with simplified statecharts that do not regard UML syntax, by leaving out event parameters, guards, and the four different event kinds of the UML. By having a description that is syntactically consistent with the UML meta-model, and by formulating consistency of refinements in OCL, existing UML tools can support these concepts. Existing statechart editing and simulation techniques can be used with only a fixed set of OCL constraints added.

4.2.1 Protocol Statecharts

Protocol state charts are mentioned in the UML specification:

> A ’protocol state machine’ for a class defines the order; that is, sequence in which the operations of that Class can be invoked. The behavior of each of these operations is defined by an associated method, rather than through action expressions on transitions. A transition in a

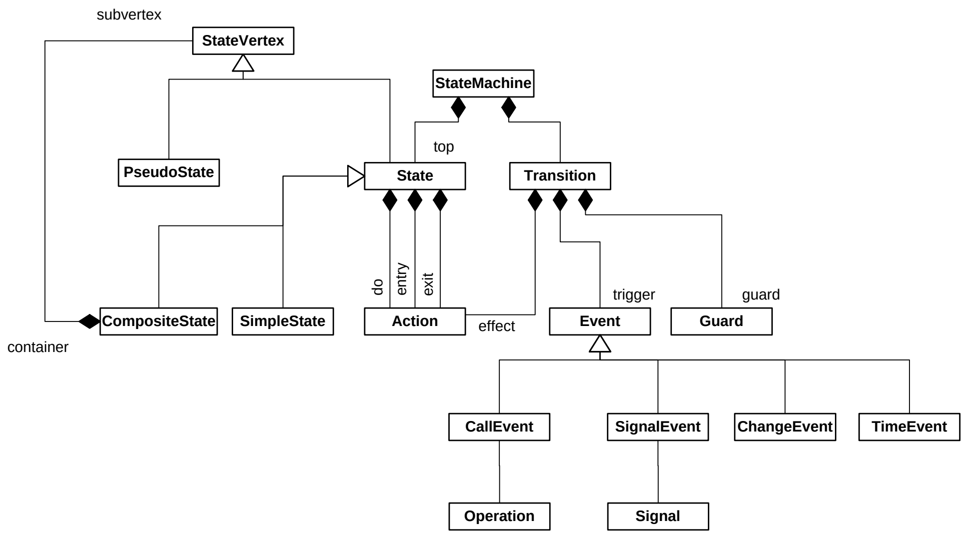


Figure 4.6: An excerpt of the statechart meta-model

> protocol state machine has as its trigger a call event that references an operation of the class, and an empty action sequence.
>
> In any practical application, a protocol state machine is made up exclusively of ’protocol’ transitions, and the entry and exit actions of its states are empty; that is, no action specifications exist other than for the methods. However, formally it is not prohibited to mix this kind of transition with transitions with explicit actions (as it does not seem worth the effort to prohibit this, and there may be some applications that might benefit from ’mixing’).[28, sect. 2.12.5.1]

In the following, we will use the definition in the stricter sense with no actions at all. From a UML meta-model viewpoint, protocol statecharts (PSCs) can be defined as a special case of statecharts, with the following restrictions:

Definition 4.2.1 (Protocol Statechart). A statechart `PSC` is called *protocol statechart*, if it specifies a class and if there are no actions in `PSC`, i.e. no entry actions, exit actions and do actions in states and no actions in transitions. In OCL:

```
context StateMachine def:
let isProtocol():Boolean =
      self.context->notEmpty()
  and self.context.oclIsKindOf(Class)
  and self.allStates.entry->isEmpty()
  and self.allStates.exit->isEmpty()
  and self.allStates.doActivity->isEmpty()
  and self.transition.effect->isEmpty()
```

There are four kinds of events defined in the UML [28, section 2.12.2] that can be used as transition labels: Events resulting from a *synchronous method call* map to a `CallEvent`, *asynchronous signal events* map to a `SignalEvent`, *timeout* events map to a `TimeEvent`, and events that indicate a *change of some boolean expression* map to a `ChangeEvent`. The first two event kinds have to be consistent with the specified classes by requiring a corresponding operation or signal reception, respectively; time events of the form `after (x)` are always well-formed as long as `x` expresses a time (such as `after(5 sec)`); change events can contain any expression resulting in a `true` or `false` value.

Any query operations are typically not interesting for a protocol: Queries do not cause any state changes, and they may be called always (except, of course, if their OCL precondition is not fulfilled. Usually queries have no preconditions). Furthermore, signal receptions cannot be queries ([28, section 2.9.3.17]).

As most of the other UML diagrams, statecharts can be used at different levels of precision. Transition labels can in practice contain event signatures that are not defined in the corresponding class, or even natural text. As this thesis aims at syntactically precise modelling, we only regard statecharts that are syntactically correct with respect to the class they specify.

Definition 4.2.2 (Syntactically Correct Protocol Statechart). A statechart is a *syntactically correct protocol statechart* if it is a protocol, there are only call events in the statechart transitions for which there is a corresponding operation defined in the class, and there are only signal events for which there is a corresponding signal reception defined in the class. In OCL:

```
context StateMachine def:
let isSyntacticallyCorrect():Boolean =
      isProtocol()
  and
      self.transitions.trigger->forAll( call:CallEvent |
        self.context.feature->exists( op:Operation |
          op = call->operation ) )
  and self.transitions.trigger->forAll( sig:SignalEvent |
        self.context.feature->exists( rec:SignalReception |
          rec->signal = sig->signal ) )
```

Note that ill-formed expressions in time events or change events are already 'ruled out' by the UML well-formedness rules.

Invokable and Observable Behaviour

Engels et al. distinguish carefully in [6] between state machines representing the *invokable* and *observable* behaviour of objects. The invokable perspective states that an object of the superclass can always be replaced by an object of the subclass without any change in behaviour. The observable one states that every trace of events possible for an object of the subclass must also be possible for an object of the superclass. The following definitions are adapted from [6]:

Definition 4.2.3 (Observable Sequences of Method Calls). Given class `A` and state machine `smA` associated with `A`. OS(smA) is the set of all sequences of method calls that might occur, though it is not guaranteed that all of these are actually executable with each instance of `A`.

Definition 4.2.4 (Invokable Sequences of Method Calls). Given class `A` and state machine `smA` associated with `A`. IS(smA) is the set of all sequences of method calls that are guaranteed to be executable.

Definition 4.2.5 (Projection of Sequences of Method Calls). Given classes `A` and `B` with `isSubClass(B,A)`, a state machine `smB` associated with `B`. For a sequence `S` of method calls in `smB`, `pr(S)` is the sequence with all calls of methods that do not exist in `A`, i.e `pr(S) = S \ {alphabet of A}` .

The following examples in figure 4.7 show some refinement situations and their relation with observability and invokability.

In case (a) the protocol is weakened when defining the subclass: A sequence `a() c() d() a() b()` is possible for class `Y`. The restriction of this sequence is `a() a() b()`, but this is forbidden in the superclass `X`. As the restriction of this sequence of events of the subclass is not included in the sequence of events of the superclass, this refinement is not consistent with respect to observable behaviour. But any sequence that can be called on the superclass, can also be called on the subclass. This refinement is consistent with respect to invokable behaviour.

In (b) the protocol of the subclass `Y` is stricter: `X` allows a sequence `a() a() b()`, but this cannot be executed in `Y`. Thus, the refinement cannot be consistent with respect to invokable behaviour. As the restriction of every sequence of events in `Y` is also a sequence of `X`, the refinement is consistent with respect to observable behaviour.

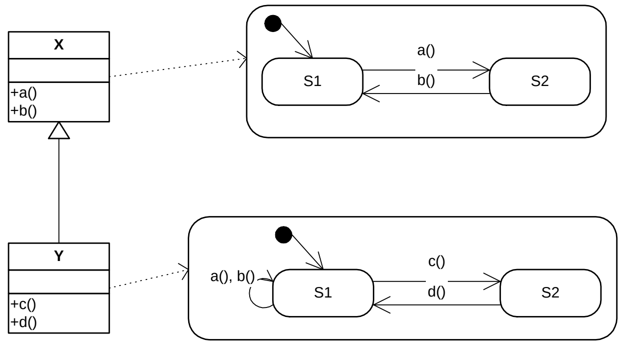
In (c) the operations new in `Y` are put between the existing operations, but the order of the operations that also exist in `X` is unchanged. Thus, each sequence in `Y`, for example `a() c() b() d() a()`, can be restricted to a sequence that also possible for `Y`, such as `a() b() a()`. This refinement is also consistent with respect to observable behaviour.

The focus in [6] is set on non-hierarchical statecharts. A statechart is non-hierarchical (or ”flat”) if the states contained by the top state do not contain sub-states themselves.

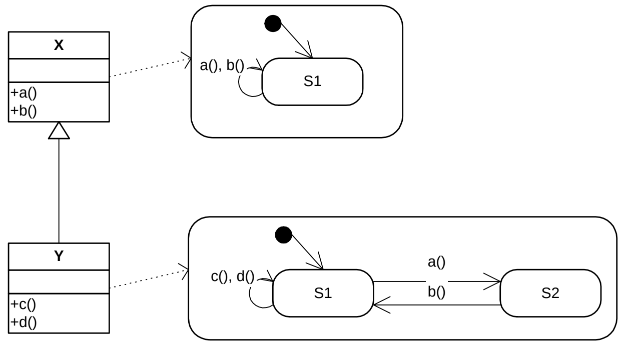
```
context StateMachine def:
let isFlat():Boolean =
  self->top->subvertex->collect( cs:CompositeState )
    ->isEmpty()
```

In such a flat state machine, an abbreviation for accessing all states is defined by the following function:

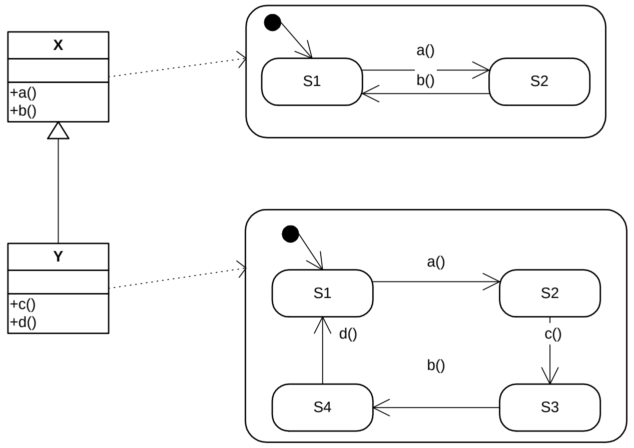
```
context StateMachine def:
let allStates:Set(StateVertex) = self->top->subvertex
```



(a) Weakening the Protocol



(b) Strengthening the Protocol



(c) Extending the Protocol

Figure 4.7: Three situations of protocol statechart refinement

As there is exactly one initial state, the following function can be used to identify the initial state:

```
context StateMachine def:
let initialState:PseudoState =
  allStates->select(ps:PseudoState |
    ps.kind = #initial)->asSequence()->first()
```

The definitions in this section are based on [6]. Here, a definition of statecharts is given that is notationally close to the one given in Z notation in [6].

Definition 4.2.6 (Protocol Statechart). Let $fd(C) = (name_C, attrs_C, opns_C)$ a class's full descriptor, $States$ be a set of object states, and $Labels = opns_C \cup \{\tau\}$ the set of transition labels, consisting of operation names and the empty transition label. A *statechart* is a labelled transition system $(States, Labels, \delta, S_0)$, where $\delta : States \times Labels \times States$ is the transition relation and $S_0 \in States$ is the initial state.

The relation to the definition above should be obvious: While this is a constructive, set-theoretical definition (from the bottom), the definition above restricts the existing UML statechart definition to a special class (from the top). It is hard to prove the equivalence because of the very different formalisms; the intuition should suffice here.

4.2.2 Protocol Statechart Refinement

In [6] refinement of statecharts is expressed in terms of statechart homomorphisms that preserve the behaviour in the desired way. The homomorphism properties are formulated as follows (again slightly modifying the original notation):

Definition 4.2.7 (Statechart Homomorphism). Let $PSCA = (States^A, Labels^A, \delta^A, S_0^A)$ and $PSCB = (States^B, Labels^B, \delta^B, S_0^B)$ be protocol statecharts. A function $h : States^A \to States^B$ is a *statechart homomorphism* if $h(S_0^A) = S_0^B$ and for each transition $s \xrightarrow{op()} t \in \delta^A$:

$h(s) = h(t)$

or in δ^B exists a transition $h(s) \xrightarrow{op()} h(t)$

or in δ^B exists a transition $h(s) \xrightarrow{\tau} h(t)$

This definition implies that the initial state is preserved, and that a label on a transition can be mapped either to a corresponding transition, to an empty transition, or to identified states (which might be interpreted as an empty transition on a single state). See figure 4.8 for the three situations.

An equivalent formulation that uses the UML notation is given now. It regards the fact that call events and signal events may be used. It restricts the statecharts to have a the same event kind in both statecharts for a corresponding transitions.

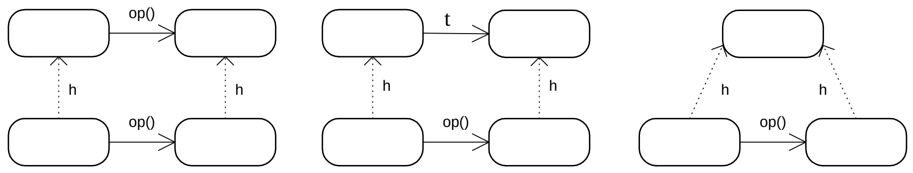


Figure 4.8: Mappings allowed in a statechart homomorphism

Relaxing this condition could lead to unexpected changes of concurrent behaviour of the specified objects. An additional function `eventsMatch` is defined for this purpose. Remember that only syntactically correct statecharts, i.e. those with operations and receptions defined in the corresponding class, are regarded.

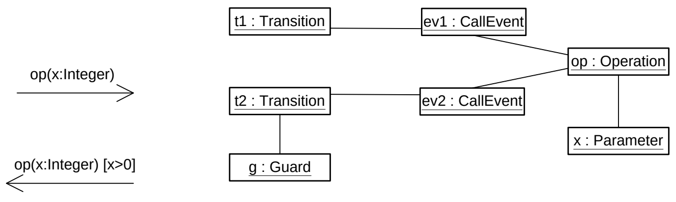
Definition 4.2.8 (Statechart Homomorphism). Let `PSCA` and `PSCB` be statecharts. A function `h(state:StateVertex):StateVertex` is a *statechart homomorphism* between `PSCA` and `PSCB` if

```
h(PSCA->initialState) = PSCB->initialState
and
PSCA->transition->forAll(tra:Transition |
  h(tra->source) = h(tra->target)
  or
  PSCB->transition->exists(trb:Transition |
    trb->source = h(tra->source)
    and trb->target = h(tra->target)
    and eventsMatch(tra->trigger, trb->trigger)
  )
  or
  PSCB->transition->exists(trb:Transition |
    trb->source = h(tra->source)
    and trb->target = h(tra->target)
    and trb->trigger->isEmpty()
  )
)
```

The definition uses on operation `eventsMatch` that checks for syntactical coherence of the events on the transitions of the two statecharts:

```
let eventsMatch(ev1:Event, ev2:Event):Boolean =
  ev1->oclIsKindOf(CallEvent) implies
    ( ev2->oclIsKindOf(CallEvent) and
      ev1.oclAsKind(CallEvent)->operation =
        ev2.oclAsKind(CallEvent)->operation
    )
  and
  ev1->oclIsKindOf(SignalEvent) implies
    ( ev2->oclIsKindOf(SignalEvent) and
      ev1.oclAsKind(SignalEvent)->signal =
        ev2.oclAsKind(SignalEvent)->signal
    )
```

If event signatures with parameters are to be regarded, rather than only the names of the operations, the definition of `eventsMatch` could have been strictened by the additional condition that the parameter list must be identical. This is redundant, though: On the abstract syntax level the operations respectively signals exist only once, there cannot be two distinct operations with the same signature in one class. See figure 4.9 for an example of two transitions, possibly occurring in two different statecharts, but referring to one operation.



(a) Concrete notation (b) Abstract syntax

Figure 4.9: Two transitions

If, additionally, transition guards are to be regarded in statechart homomorphisms, there are several possibilities for a definition. We could force the guard expressions to be identical in both transitions, leading to the following extension of the second case of the homomorphism definition:

```
PSCB->transition->exists(trb:Transition |
  trb->source = h(tra->source)
  and trb->target = h(tra->target)
  and eventsMatch(tra->trigger, trb->trigger)
  and trb->guard->expression = tra->guard->expression
)
```

We could also allow that the guard of the subclass's transition is stronger than the guard of the superclass's one. Similarly, we could allow that the second one is a syntactical restriction of the first one, because the guard of the subclass could be an expression over a larger set of attributes than the guard of the superclass.

Using the definition of a statechart homomorphism and of observable and invokable behaviour, behaviourally consistent refinement of statecharts can now be defined:

Definition 4.2.9 (Behavioural Consistency of Subclasses). Let `A`, `B` classes with `isSubclass(B,A)`, and `PSCA` and `PSCB` protocol statecharts constraining `A` and `B`, respectively. `PSCA` and `PSCB` are *behaviourally consistent with respect to observable behaviour*, if there is a homomorphism

$h : States(PSCB) \rightarrow States(PSCA)$ as defined above. PSCA and PSCB are *behaviourally consistent with respect to invokable behaviour*, if there is a homomorphism
$h : States(PSCA) \rightarrow States(PSCB)$ as defined above.

In the notation of Ebert et.al. PSCB refines PSCA if

- PSCA and PSCB represent observable behaviour
  and `pr(OS(PSCB))` $\subseteq$ `OS(PSCA)`

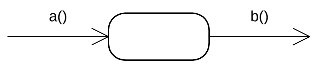
or

- PSCA and PSCB represent invokable behaviour
  and `IS(PSCA))` $\subseteq$ `IS(PSCB)`.

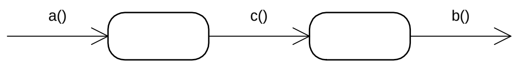
A refinement cannot be defined (and makes no sense) if PSCA represents observable and PSCB invokable behaviour or vice versa.

Introducing Hierarchy

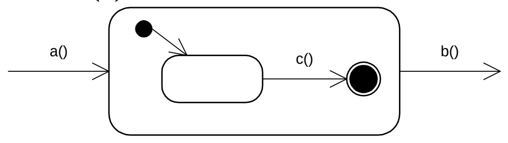
Figure 4.10 shows that the observability approach is most intuitive when refing a statechart by adding sub-statecharts to states, using the hierarchy mechanism of statecharts. A excerpt of a protocol statechart, defining that `b()` can be called after `a()` was called, can be extended by specifying that an additional operation, `c()`, has to be called in between. Using the hierarchical notation, the notational hiding exactly matches the hiding of events in the trace.



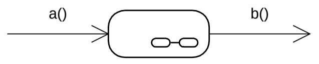
(a) The superclass's statechart



(b) The subclass's statechart



(c) Equivalent notation for (b), using hierarchy



(d) Hiding the details

Figure 4.10: Refining an excerpt of a statechart, preserving observable traces

The IOSIP case study contains several examples of this kind of relation between superclass and subclass. An example is the refinement of class `Negotiator` by class `NegotiatorBod`, see figures 5.4 to 5.7 starting at page 81.

Refinement of statecharts by introducing *parallel* substates is not handled in this thesis. There are no parallel statecharts at all in the IOSIP case study, and a parallel statechart does not reflect the intuition that a statechart specifies one (single-threaded) object's behaviour. There exists concurrency in the IOSIP examples, but it is expressed by concurrent execution of different object's statecharts. An example is the concurrent execution of `Interpreter` and `Negotiator` objects, where synchronisation is guaranteed by asynchronous signal emission and reception. See section 5.3.1 for examples. Parallel extension, though, is handled in [6] and should be easily mapped to the notations used in this thesis.

4.3 Components in the Context of Software Development Processes

An additional goal of a component concept not stated before is that components should be able to be used in different phases of a *software development process*.

The UML is also intended to be used throughout the development process, but it does not define such a process.[1] The different diagram types can be used in different phases, and refined in a phase transition. Similarly, the UML component concept introduced in this thesis shall make possible suitable refinement concepts for components. This section sketches such approaches, but does neither show how any of the proposed techniques would be defined, nor prove that they are applicable with the methods defined so far.

Refining UML models, thus getting closer to an implementation, is not trivial; many decisions are to be made. The key task is, of course, finding efficient algorithms for realising the (non-constructively) specified functionality. Programming language-specific questions include data types (is an OCL `Integer` mapped to Java type `byte`, `short`, `int`, `long`, `java.math.BigInteger`, with 8-bit, 16-bit, 32-bit, 64-bit and arbitrary precision, respectively), naming conventions, or structural mismatches with the UML model (e.g. in Java there is no multiple inheritance, but realisation of multiple interfaces is allowed. All of the following ideas do not have such depth but are rather conceptual.

The following definitions are closely related to those in [8, chapter 5] and were adapted to the terminology used in the Generic Component Framework and in this thesis.

[1]There has been some confusion about this fact in the past because early versions (before 1996) of the Unified Modelling Language (UML) were titled *Unified Method* (UM).

4.3.1 Refinement of Components

As stated above, this kind of refinement transformation can be of a more general kind than the ones used to connect the component parts, depending on the kind of properties these refinements preserve. Therefore, we assume a class $GenTrafo$ of *generic refinement transformations* between specifications, and that refinement transformations and inclusions in the sense of [9] form sub-classes thereof. As a result, compositions of inclusions and generic refinement transformations as well as compositions of refinements and generic refinement transformations are possible, resulting in a $GenTrafo$ transformation in each case.

Refining a component can be useful in two different situations: If a whole component is refined, all three parts (provisions, body, requirements) have to be refined each, such that a well-formed component results. If the interfaces shall not be changed, their realisation in the component body can be refined on its own.

Definition 4.3.1 (Refinement of Components). Given components $Comp = (Pro, Bod, Req, pro, req)$ and $Comp' = (Pro', Bod', Req', pro', req')$. A ***refinement of a component*** $compRef : Comp \to Comp'$ is given by a triple of generic refinement transformations $compRef = (refpro, refbod, refreq)$ such that $req' \circ refreq = refbod \circ req$ and $refbod \circ pro = pro' \circ refpro$. (See figure 4.11.)

$$\begin{tikzcd}
 & Pro \arrow[d, Rightarrow, "pro"] \arrow[dr, Rightarrow, "refpro"] & \\
Req \arrow[r, hook, "req"] \arrow[dr, Rightarrow, "refreq"] & Bod \arrow[dr, Rightarrow, "refbod"] & Pro' \arrow[d, Rightarrow, "pro'"] \\
 & Req' \arrow[r, hook, "req'"] & Bod'
\end{tikzcd}$$

Figure 4.11: Refinement of a component

Among the problems that have to be solved in order to apply this definition to the UML components defined before, are compatibility of component composition and refinement, and a characterisation of the properties that a refinement must fulfil so that $Comp'$ is defined and well-formed. A very simple example is shown in figure 4.12: A coarse, conceptual description of the account component, only describing that there is a provided class `Account` which is specialised in the component body to be transient or persistent, is refined to a component which renames classes, adds attributes and operations to the classes and adds a requirements package. The refinements are specified by the more general «`refine`» dependency, not by the more specific «`provide`» dependency.

If the interfaces of a component may not be changed, but the body is to be refined, a special case of the refinement construction can be defined:

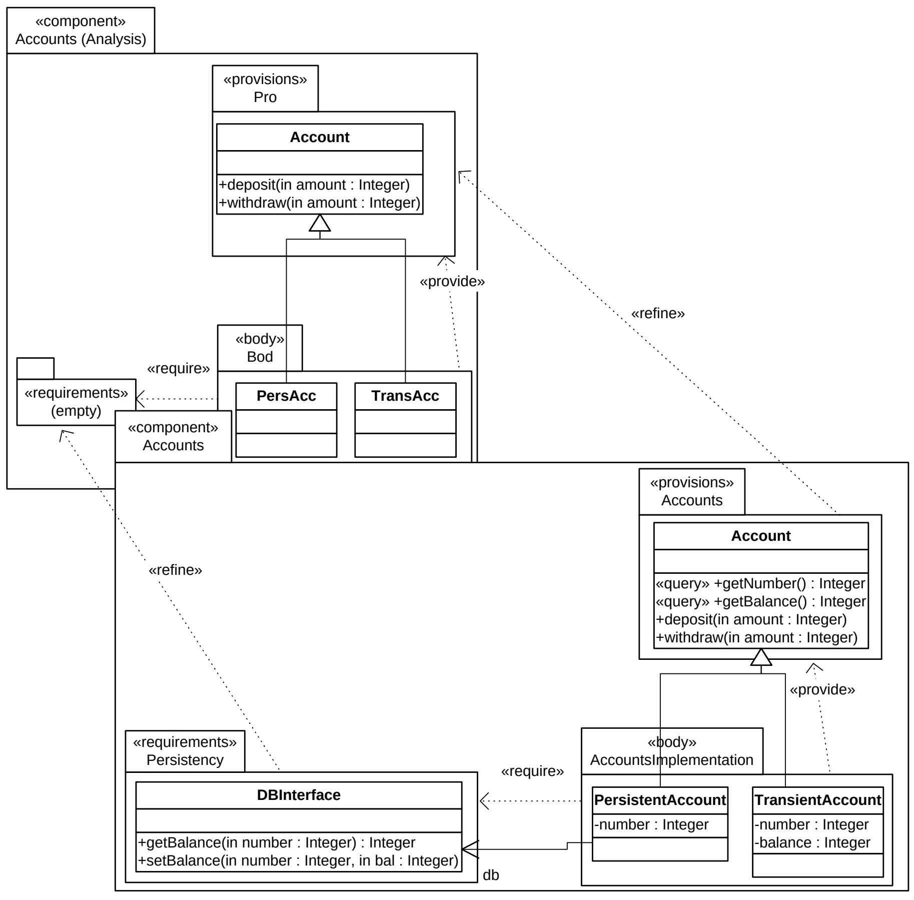


Figure 4.12: Example for refinement of a component

Definition 4.3.2 (Body Refinement of Components). Given components $Comp = (Pro, Bod, Req, pro, req)$ and $Comp' = (Pro, Bod', Req, pro', req')$. A *body refinement of a component* $compBodyRef : Comp \to Comp'$ is given by a generic refinement transformation $refbod$ such that $req' = refbod \circ req$ and $refbod \circ pro = pro'$. (See figure 4.13.)

$$\begin{tikzcd}
 & Pro \arrow[d, Rightarrow, "pro"] \arrow[dr, Rightarrow, bend left, "pro'"] & \\
Req \arrow[r, hook, "req"] \arrow[rrd, hook, bend right, "req'"'] & Bod \arrow[dr, Rightarrow, "refbod"] & \\
 & & Bod'
\end{tikzcd}$$

Figure 4.13: Body refinement

Component interface specifications, as introduced in section 2.5, can be refined to ”full” components by realisation.

Definition 4.3.3 (Component Interface Realisation). Given a component interface specification $CompInt = (Pro, Req)$ and a component $Comp = (Pro', Bod, Req', pro, req)$. A *component interface realisation* $real : CompInt \to Comp$ is given by a pair of generic refinement transformations $real = (refpro, refreq)$. (See figure 4.14.)

$$\begin{tikzcd}
 & Pro \arrow[dr, Rightarrow, "refpro"] & \\
Req \arrow[dr, Rightarrow, "refreq"] & & Pro' \arrow[d, Rightarrow, "pro"] \\
 & Req' \arrow[r, hook, "req"] & Bod
\end{tikzcd}$$

Figure 4.14: Component interface realisation

An important special case is, of course, realisation without refining provisions and requirements, i.e. $Pro' = Pro$ and $Req' = Req$ (and thus $refpro = id_{Pro}$ and $refreq = id_{Req}$).

4.3.2 Composing a System

The Generic Framework does not discuss whether a component-based software system may *only* consist of components and connectors, or if components can be mixed with non-component specifications. In principle, it is possible to add such ”basic” specifications to the system. On the ”lower end” of a composition hierarchy, a requirements specification may be refined by a single, non-component specification, such as an existing library Lib, resulting in a new component $(Pro, Bod',$ $Lib, lib' \circ pro, req')$ because of the extension property. If all requirements of a component-based system are fulfilled, it is often called *closed*, because it does not depend on the environment anymore. Such a situation is depicted in figure 4.15.

$$\begin{tikzcd}
& Pro \arrow[d, Rightarrow, "pro"] \\
Req \arrow[r, hook, "req"] \arrow[d, Rightarrow, "lib"] & Bod \arrow[d, Rightarrow, "lib'"] \\
Lib \arrow[r, hook, "req'"] & Bod'
\end{tikzcd}$$

Figure 4.15: Connecting a component with an existing library

On the ”upper end” of the composition hierarchy a single, non-component specification, such as a client Cli using the system, may include the composed component, resulting in a well-formed component $(Cli, Bod', Req, pro', cli' \circ req)$ owing to the extension property. See figure 4.16.

$$\begin{tikzcd}
& Pro \arrow[r, hook, "cli"] \arrow[d, Rightarrow, "pro"] & Cli \arrow[d, Rightarrow, "pro'"] \\
Req \arrow[r, hook, "req"] & Bod \arrow[r, hook, "cli'"] & Bod'
\end{tikzcd}$$

Figure 4.16: Client using a component

4.4 Component Lifecycle Issues

Components represented by class diagrams define static structure and constrain behaviour of objects. The component lifecycle (creation and destruction of component instances) is not yet taken into account in detail. Additional effort has to be done in order to specify how components and contained objects can be instantiated or destroyed. Depending on the level of detail of the models to be developed, this information can either be left out completely, or can be added according to usual patterns:

- Mark a provided class with stereotype «`focalClass`» ([28, section 3.14.4]). When this class is instantiated (using the programming languages' constructors), all other classes needed for the component are instantiated (or existing objects are searched and suitable ones are used). A problem arises: Subclasses hidden in the body cannot be instantiated, but because the moment of object creation determines the runtime type, this would mean that the focal class is always fully specified in the provisions.

- Add explicit creation operations to the model, for example a static (class-scope) operation `instantiateComponent(params)`. This is an example of the well-known *factory* design pattern. In this case, a runtime type from the component body can be returned, transparent to the caller.

- The underlying component technology used for implementation defines additional operations. For example, the Enterprise Java Beans [24] component concept defines how lifecycle methods such as `ejbCreate` or `ejbActivate` can be created by a tool automatically. In the CORBA Component Model [12], a so-called *component home* is declared for each component, also aimed at automatically deriving lifecycle operations.

In each of the cases mentioned, component destruction works analogously. It is imaginable that an appropriate modelling tool would either create such lifecycle functionality automatically or at least assist the user to create it.

Chapter 5

UML Case Study in Production Automation

In this chapter, some work done in the IOSIP[1] project is presented and adapted to the component syntax used in this thesis. There were only few changes to be made, because a very similar notion of component was used. First, an overview of the domain (a production automation system) is given. Then, the specified components are presented; some in full detail and others only partially. Finally, application of the partial composition operation is illustrated by explicitly showing some composed components.

Not all components are given in full detail. The components presented here were not chosen on their importance for the domain, but rather on syntactically interesting details. In particular, the components specifying the communication between H-AGVs are shown more explicitly than the components specifying the main (negotiation and transportation) functionality, although there are more interesting algorithms in the latter.

The IOSIP case study also includes an analysis model of the domain, consisting of a coarse class diagram, use case diagrams and activity diagrams with natural language descriptions. This part is not presented here, because in this thesis the focus is set on the parts of a model that is modelled precisely. Nevertheless, this analysis model is an important part of the case study.

5.1 The Production Environment

A production system as it could be used in an automobile factory is specified in [3]. Tool machines drill and wash workpieces, an input stock stores unprocessed workpieces, an output stock stores processed workpieces. A possible layout is sketched in figure 5.1. The most interesting feature are so-called *holonic automated guided*

[1]Integration of object-oriented software specification techniques and their application-specific extension for industrial production systems on the example of automobile industry

vehicles, short H-AGVs. These are robot-like vehicles that can carry workpieces between tool machines or between machines and a stock. They are called *holonic* because they plan their moves and their strategies to carry workpieces individually and collaborating with each other in an agent-like way, without a central controlling unit.

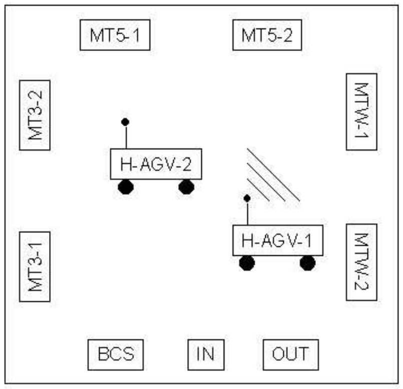


Figure 5.1: Production system overview

The focus in the IOSIP case study is to model the control software of the H-AGVs. This includes algorithms implementing the desired agent-like behaviour (called *negotiation*) and non-trivial communication issues via a wireless broadcast medium.

Miscellaneous tasks that are to be handled include an energy management for the batteries of the H-AGVs, a route reservation system for avoiding collisions between H-AGVs, and reporting the status of H-AGVs and machining tools (e.g. failures) to the rest of the system. Especially, the tools send their status including the occupancy of their input and output buffers to the H-AGVs. Thus, machining tools trigger the actions taken by the H-AGVs, that is, negotiating about the *transport order* of a workpiece among the H-AGVs and executing such an order.

5.2 The Software Components of the Case Study

The main functionality of an H-AGV is the negotiation about transport orders and executing such orders, using their ability to drive and to take workpieces. Non-trivial communication issues between the H-AGVs occur. Thus, it is natural to decompose the H-AGVs' control software into components that specify a distinguished piece of functionality. Negotiation is done in the `Negotiator` component, driving in `DrivingControl`, handover and withdrawal of workpieces in `HandoverControl`. Communication is handled by the three components `IO`, `Interpreter` and `Mediator`.

Before describing the components in detail, a shared package `TypeDefinitions` containing basic types is shown. Figure 5.2 shows an excerpt: classes that are used in various components, and that do not define functionality complex enough to be worth to be specified in detail in a component. Syntactically, one can imagine that each component has an additional requirements package containing used data types, and which is connected with this package. This is left out in all components for visual clarity.

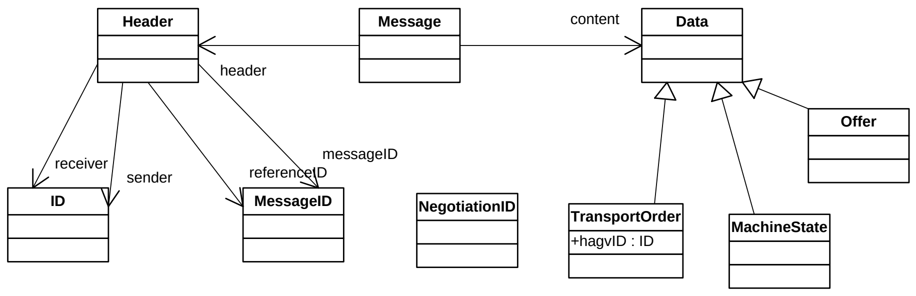


Figure 5.2: Data types used commonly

The `Negotiator` Component

The negotiation strategy of an H-AGV is realised in the `Negotiator` component. Negotiation about a transport order is triggered by a message sent by a tool machine, containing information that the input buffer of this machine is empty. This means that the machine is ready to process a new workpiece. The negotiation algorithm has to decide which of the H-AGVs (excluding those in standby or failure mode) executes the transport order. Executing a transport order means driving to the place where the workpiece is (input stock or a machine's output buffer), withdrawing it from there, carrying it to the next position (input buffer of the next machine, or output stock) and handing it over there.

The negotiation algorithm is as follows: The machine state, sent by broadcast, is received by at least one H-AGV. If the machine state implies a transport order, one H-AGV becomes the *moderator* of the negotiation. It calculates an expense factor for this transport order, which may be calculated from the distance to the machine, on the battery status or any additional criterion. This *initial offer* is sent to all other H-AGVs. H-AGVs receiving this offer calculate an offer with their own expense factor, and send it back to the moderator if it is better. The moderator chooses the best of all offers and *assigns* the transport order to the H-AGV with the best offer.

The corresponding classes are described now, see figure 5.3. In the provisions, class `Negotiator` exposes two signal receptions `processMachineState` and `processOffer`. Signal receptions are similar to regular operations, but behave asynchronously, that is, the caller does not wait until the operation is executed but continues its control flow immediately. As can be seen in the corresponding statechart diagram, an `Negotiator` instance only reacts if it is in the state `idle`. This is important when connecting the `Negotiator` with the `Interpreter` component.

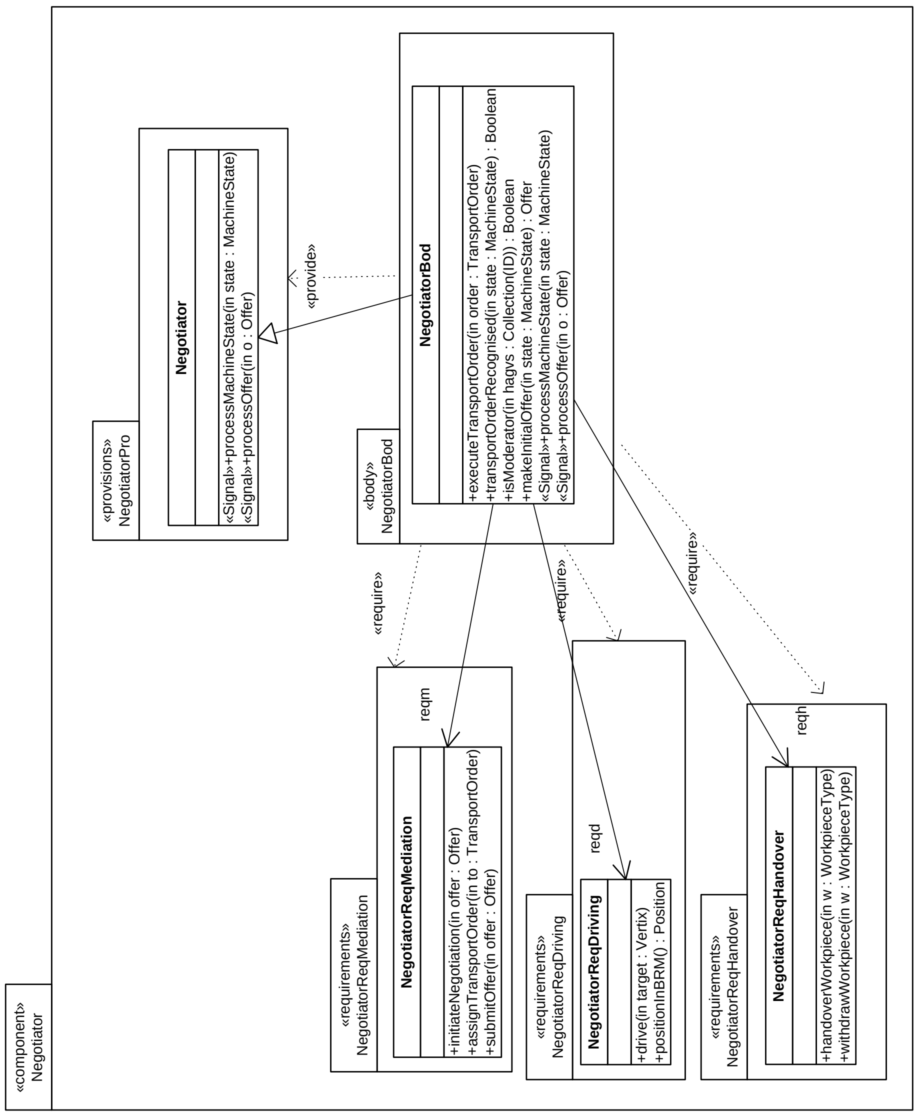

«component»
Negotiator
«provisions»
NegotiatorPro
Negotiator
«Signal»+processMachineState(in state : MachineState)
«Signal»+processOffer(in o : Offer)
«provide»
«body»
NegotiatorBod
NegotiatorBod
+executeTransportOrder(in order : TransportOrder)
+transportOrderRecognised(in state : MachineState) : Boolean
+isModerator(in hagvs : Collection(ID)) : Boolean
+makeInitialOffer(in state : MachineState) : Offer
«Signal»+processMachineState(in state : MachineState)
«Signal»+processOffer(in o : Offer)
«require»
«require»
«require»
reqm
reqd
reqh
«requirements»
NegotiatorReqMediation
NegotiatorReqMediation
+initiateNegotiation(in offer : Offer)
+assignTransportOrder(in to : TransportOrder)
+submitOffer(in offer : Offer)
«requirements»
NegotiatorReqDriving
NegotiatorReqDriving
+drive(in target : Vertix)
+positionInBRM() : Position
«requirements»
NegotiatorReqHandover
NegotiatorReqHandover
+handoverWorkpiece(in w : WorkpieceType)
+withdrawWorkpiece(in w : WorkpieceType)

Figure 5.3: `Negotiator` component

In the body, the behaviour is specified more detailedly. `NegotiatorBod`, a subclass of `Negotiator`, adds operations, has associations to several required classes, and has a more complex statechart diagram. But it can be seen immediately, comparing figures 5.4 and 5.5, that the behaviour conforms to that of the superclass with respect to the observable behaviour. Note that the body statechart is hierarchical (indicated by ⊂⊃-⊂⊃ icons) and substates are given in separate figures.

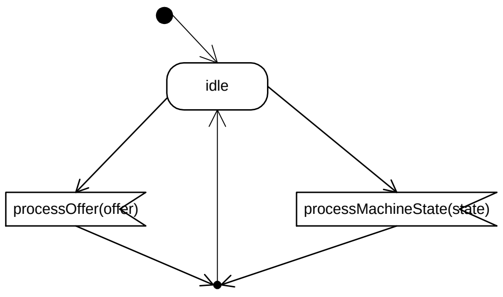


Figure 5.4: `Negotiator` statechart

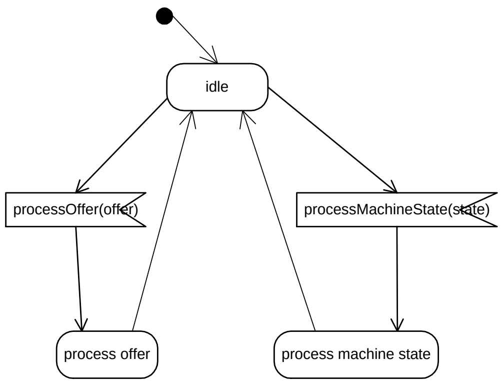


Figure 5.5: `NegotiatorBod` statechart

The `IO` component

The H-AGVs communicate via a wireless medium, a radio unit that is not specified detailedly, which can send messages. But there are some issues to be solved on the application level. First, a distinction between synchronous and asynchronous sending of messages is made, reflected by the operations `sendSynchronous` that returns answer messages, and `sendAsynchronous` which does not return anything. An instance of the `IO` component also receives messages from the radio unit using a thread that runs all the time. This is realised in the component body by the `Receiver` class, see figure 5.8.

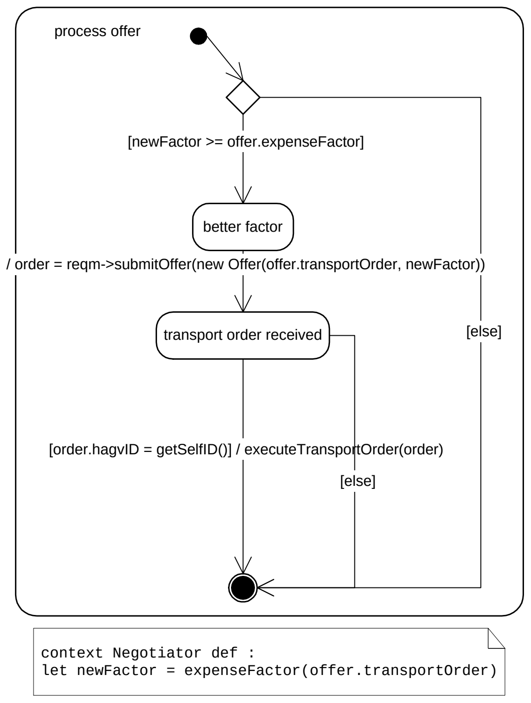
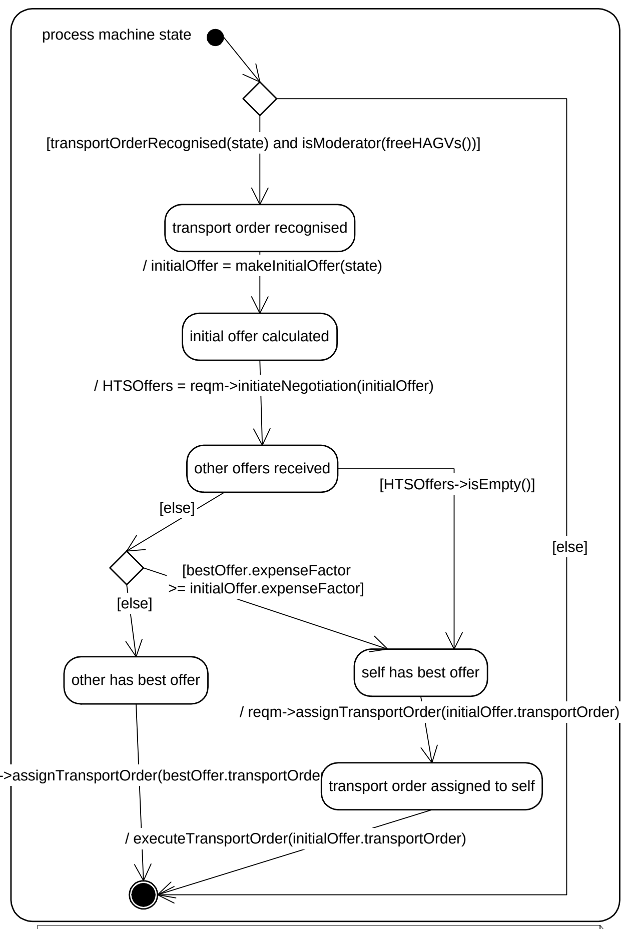

process offer
[newFactor >= offer.expenseFactor]
better factor
/ order = reqm->submitOffer(new Offer(offer.transportOrder, newFactor))
transport order received
[else]
[order.hagvID = getSelfID()] / executeTransportOrder(order)
[else]
context Negotiator def :
let newFactor = expenseFactor(offer.transportOrder)

Figure 5.6: Substate `process offer` of `NegotiatorBod` statechart


process machine state
[transportOrderRecognised(state) and isModerator(freeHAGVs())]
transport order recognised
/ initialOffer = makeInitialOffer(state)
initial offer calculated
/ HTSOffers = reqm->initiateNegotiation(initialOffer)
other offers received
[HTSOffers->isEmpty()]
[else]
[else]
[bestOffer.expenseFactor
>= initialOffer.expenseFactor]
[else]
other has best offer
self has best offer
/ reqm->assignTransportOrder(initialOffer.transportOrder)
/ reqm->assignTransportOrder(bestOffer.transportOrde
transport order assigned to self
/ executeTransportOrder(initialOffer.transportOrder)

```
context Negotiator def :
let bestOffer = HTSOffers->iterate( o:Offer; min:Offer = null |
  if (min = null) or (o.expenseFactor < min.expenseFactor)
    then o
    else minOffer
  endif
)
```

Figure 5.7: Substate `process machine state` of `NegotiatorBod` statechart

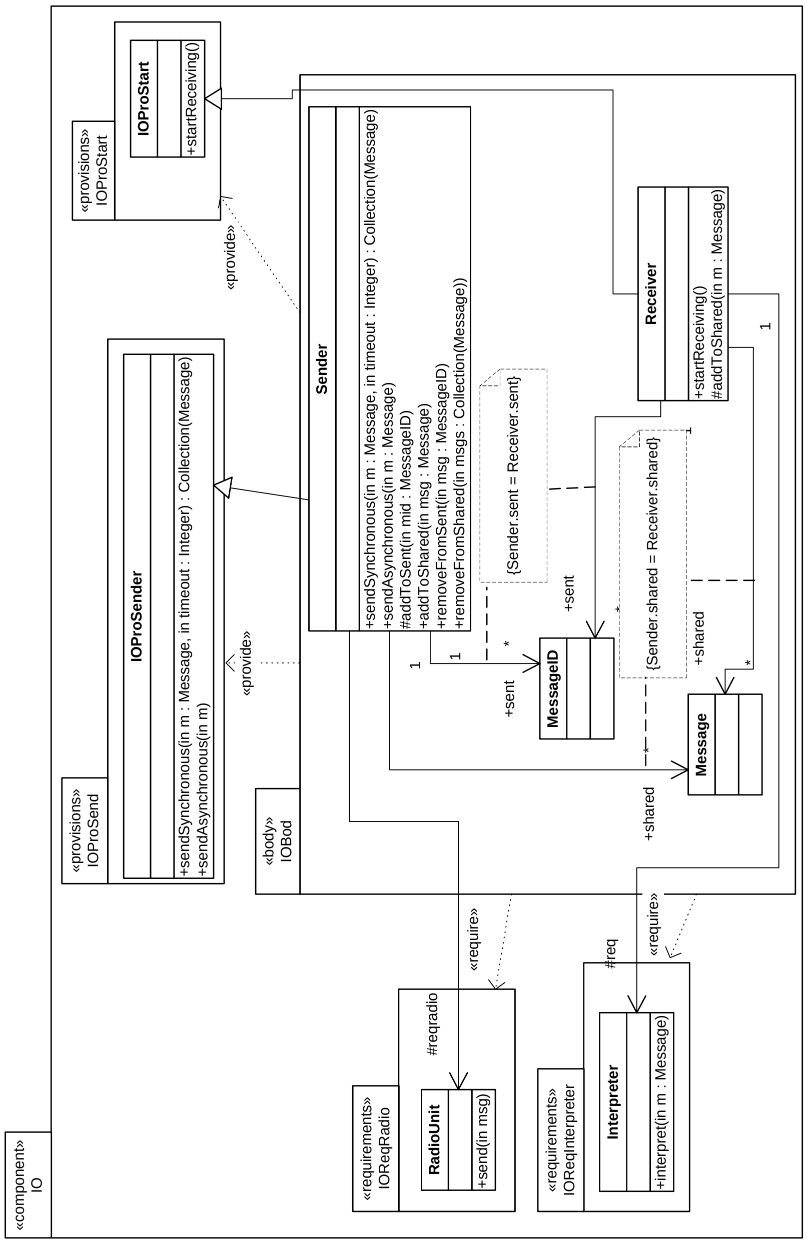

«component»
IO
«provisions»
IOProStart
IOProStart
+startReceiving()
«provisions»
IOProSend
IOProSender
+sendSynchronous(in m : Message, in timeout : Integer) : Collection(Message)
+sendAsynchronous(in m)
«provide»
«provide»
«body»
IOBod
Sender
+sendSynchronous(in m : Message, in timeout : Integer) : Collection(Message)
+sendAsynchronous(in m : Message)
#addToSent(in mid : MessageID)
+addToShared(in msg : Message)
+removeFromSent(in msg : MessageID)
+removeFromShared(in msgs : Collection(Message))
Receiver
+startReceiving()
#addToShared(in m : Message)
{Sender.sent = Receiver.sent}
{Sender.shared = Receiver.shared}
MessageID
Message
+sent
+sent
+shared
+shared
1
1
1
1
*
*
*
*
«require»
«require»
«requirements»
IOReqRadio
#reqradio
RadioUnit
+send(in msg)
«requirements»
IOReqInterpreter
#req
Interpreter
+interpret(in m : Message)

Figure 5.8: I/O component

`Sender` and `Receiver` share data: The IDs of outgoing messages that are sent synchronously are stored in a collection `sent`, which is also used by the receiver in order to recognise incoming messages that are a reply to a sent answer. An OCL constraint `Sender.sent = Receiver.sent` ensures that the same objects are referenced by a `Sender` and a `Receiver` instance. A similar constraint applies to messages in `shared`, which is used to collect the incoming replies. The behaviour is specified in figures 5.9 and 5.10.

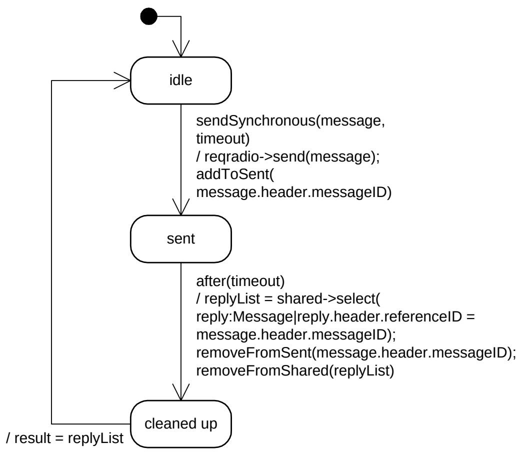

idle
sendSynchronous(message,
timeout)
/ reqradio->send(message);
addToSent(
message.header.messageID)
sent
after(timeout)
/ replyList = shared->select(
reply:Message|reply.header.referenceID =
message.header.messageID);
removeFromSent(message.header.messageID);
removeFromShared(replyList)
cleaned up
/ result = replyList

Figure 5.9: `Sender` statechart

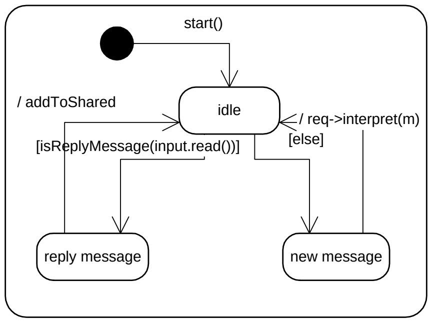


Figure 5.10: `Receiver` statechart

`Interpreter` and `Mediator` Components

The provisions interface of the `IO` component takes messages of type `Message` as one parameter. As defined in the `TypeDefinitions` package, such a message can contain complex data structures. In order to avoid that many components that communicate something to the production system have to know the details of these types, a design pattern was used that helps to decouple this dependency. A `Mediator` component is put in between that provides functionality for various components that always results in sending a message. This pattern is called mediator (or sometimes expert pattern), because it hides the (expert) knowledge of communication details from the components that use the `Mediator`. As an example, only the part of the statechart specifying one of the operations of the `Mediator`, `requestAction`, is given in figure 5.12; the other ones are very similar.

The statechart only consists of steps in composing a message of an appropriate type and with appropriate values. Note that this notation relies heavily on the chosen action language.

A similar approach is chosen for incoming messages: The IO component is not responsible for knowing which message is important for which component. An Interpreter component is placed in between which has only the functionality of analysing incoming messages and triggering appropriate actions in other components.

The provisions of the Interpreter consist of only one operation, `interpret( m:Message)` in class `Interpreter`. The details are hidden in the body: A statechart for `InterpreterBod` reveals how the type and the contents of the message are checked, and one or more actions are triggered. Interestingly, the actions triggered for incoming machine states and offers, which are executed in the negotiator, are asynchronous. This has the effect that the receiving thread is not blocked until the negotiation has finished, but continues to work as soon as the message was interpreted.

Driving Control and Handover Control

When a transport order has been negotiated about and has been assigned, the H-AGV starts its physically observable actions: It drives to and from stocks or machining tools, and withdraws and hands over workpieces. These components are not given here in detail, just the dependencies to other components are described shortly: Both of them use messages to communicate with the route reservation system, the tool machines and the stocks, and therefore make use of the Mediator. The interface to the physical part of the H-AGV, i.e. the sensors and actuators needed for moving and for the crank shaft, is given by a component Sensors/Actuators which is outside the scope of the case study.

Storage

Some data relevant for the production are stored in the Storage component. This includes the amount and identities of H-AGVs currently available, the current production layout (the order of machines that a workpiece must be processed in), and the block route map defining legal routes and parking positions for the H-AGVs.

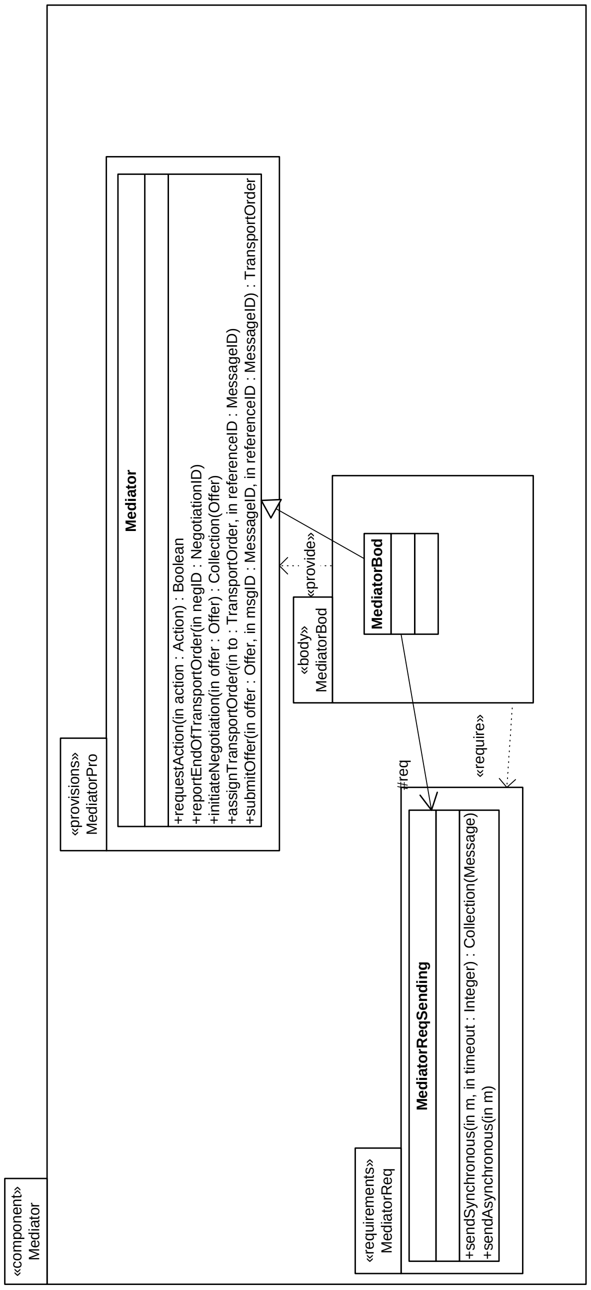

«component»
Mediator
«provisions»
MediatorPro
Mediator
+requestAction(in action : Action) : Boolean
+reportEndOfTransportOrder(in negID : NegotiationID)
+initiateNegotiation(in offer : Offer) : Collection(Offer)
+assignTransportOrder(in to : TransportOrder, in referenceID : MessageID)
+submitOffer(in offer : Offer, in msgID : MessageID, in referenceID : MessageID) : TransportOrder
«body»
MediatorBod
MediatorBod
«provide»
«require»
#req
«requirements»
MediatorReq
MediatorReqSending
+sendSynchronous(in m, in timeout : Integer) : Collection(Message)
+sendAsynchronous(in m)

Figure 5.11: `Mediator` component

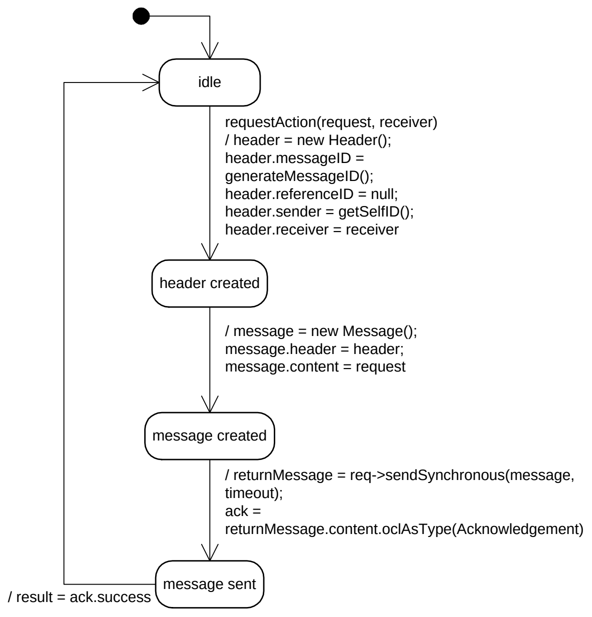

idle
requestAction(request, receiver)
/ header = new Header();
header.messageID =
generateMessageID();
header.referenceID = null;
header.sender = getSelfID();
header.receiver = receiver
header created
/ message = new Message();
message.header = header;
message.content = request
message created
/ returnMessage = req->sendSynchronous(message,
timeout);
ack =
returnMessage.content.oclAsType(Acknowledgement)
message sent
/ result = ack.success

Figure 5.12: Statechart for the operation `requestAction`

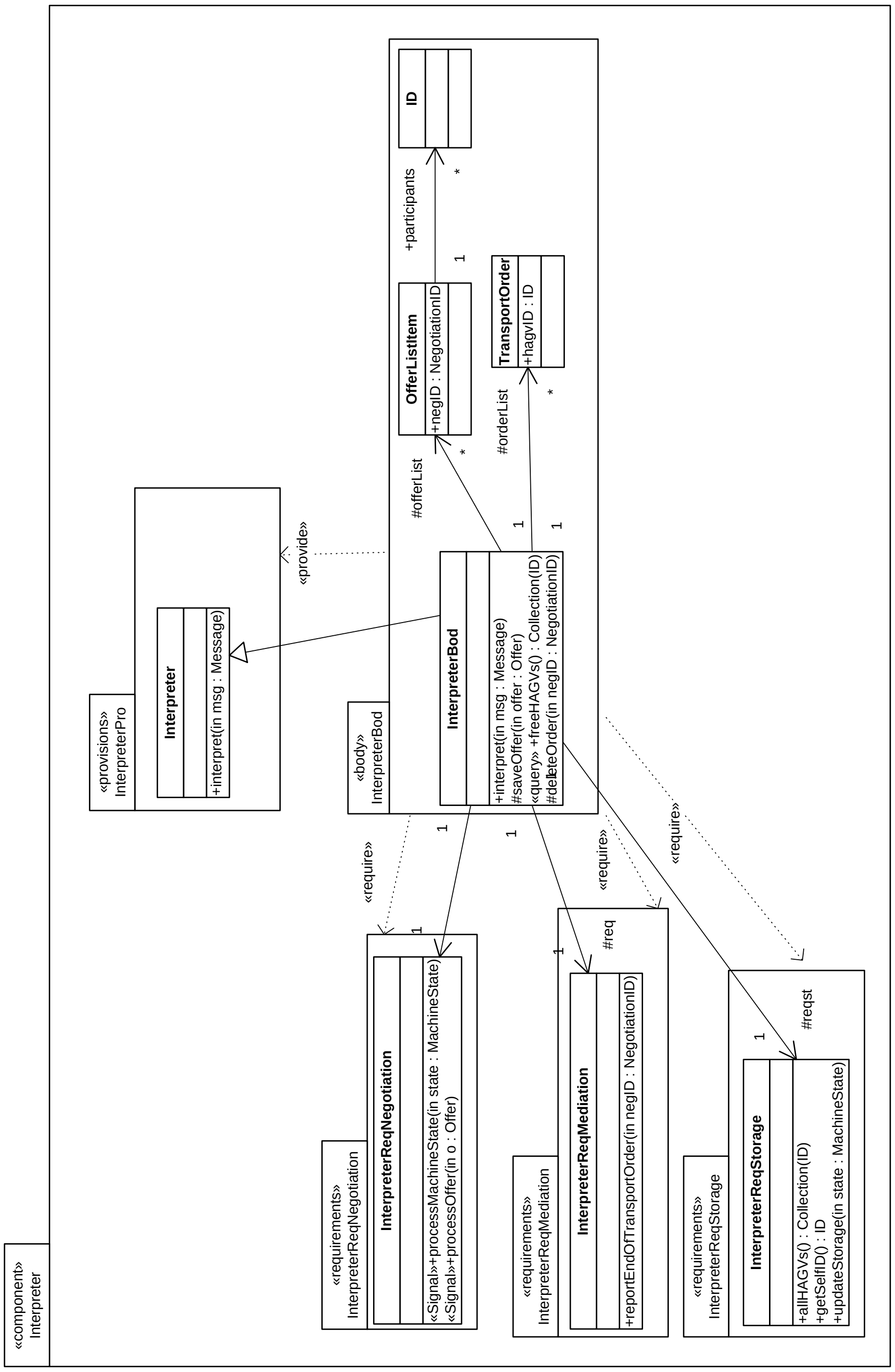

«component»
Interpreter
«provisions»
InterpreterPro
Interpreter
+interpret(in msg : Message)
«provide»
«body»
InterpreterBod
InterpreterBod
+interpret(in msg : Message)
#saveOffer(in offer : Offer)
«query» +freeHAGVs() : Collection(ID)
#deleteOrder(in negID : NegotiationID)
OfferListItem
+negID : NegotiationID
ID
+participants
TransportOrder
+hagvID : ID
#offerList
#orderList
1
*
«require»
«requirements»
InterpreterReqNegotiation
InterpreterReqNegotiation
«Signal»+processMachineState(in state : MachineState)
«Signal»+processOffer(in o : Offer)
«requirements»
InterpreterReqMediation
InterpreterReqMediation
+reportEndOfTransportOrder(in negID : NegotiationID)
#req
«requirements»
InterpreterReqStorage
InterpreterReqStorage
+allHAGVs() : Collection(ID)
+getSelfID() : ID
+updateStorage(in state : MachineState)
#reqst

Figure 5.13: `Interpreter` component

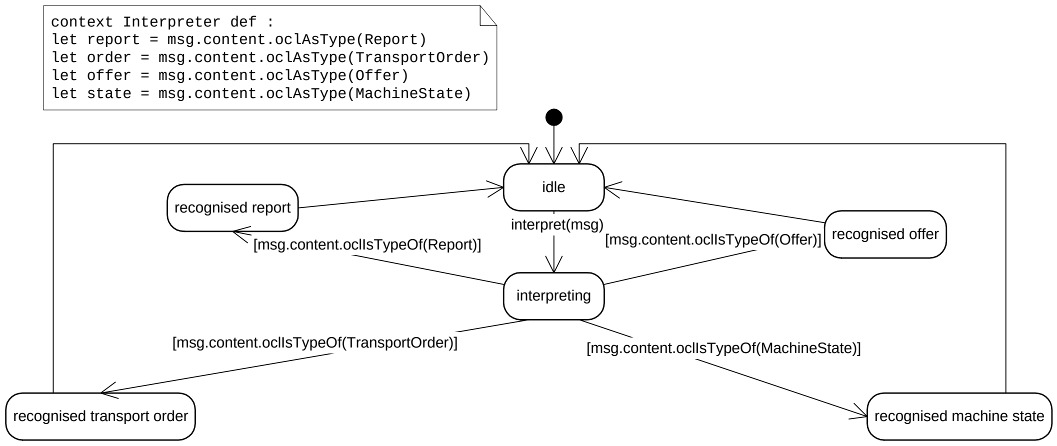

context Interpreter def :
let report = msg.content.oclAsType(Report)
let order = msg.content.oclAsType(TransportOrder)
let offer = msg.content.oclAsType(Offer)
let state = msg.content.oclAsType(MachineState)
idle
recognised report
interpret(msg)
[msg.content.oclIsTypeOf(Report)]
[msg.content.oclIsTypeOf(Offer)]
recognised offer
interpreting
[msg.content.oclIsTypeOf(TransportOrder)]
[msg.content.oclIsTypeOf(MachineState)]
recognised transport order
recognised machine state

Figure 5.14: `InterpreterBod` statechart

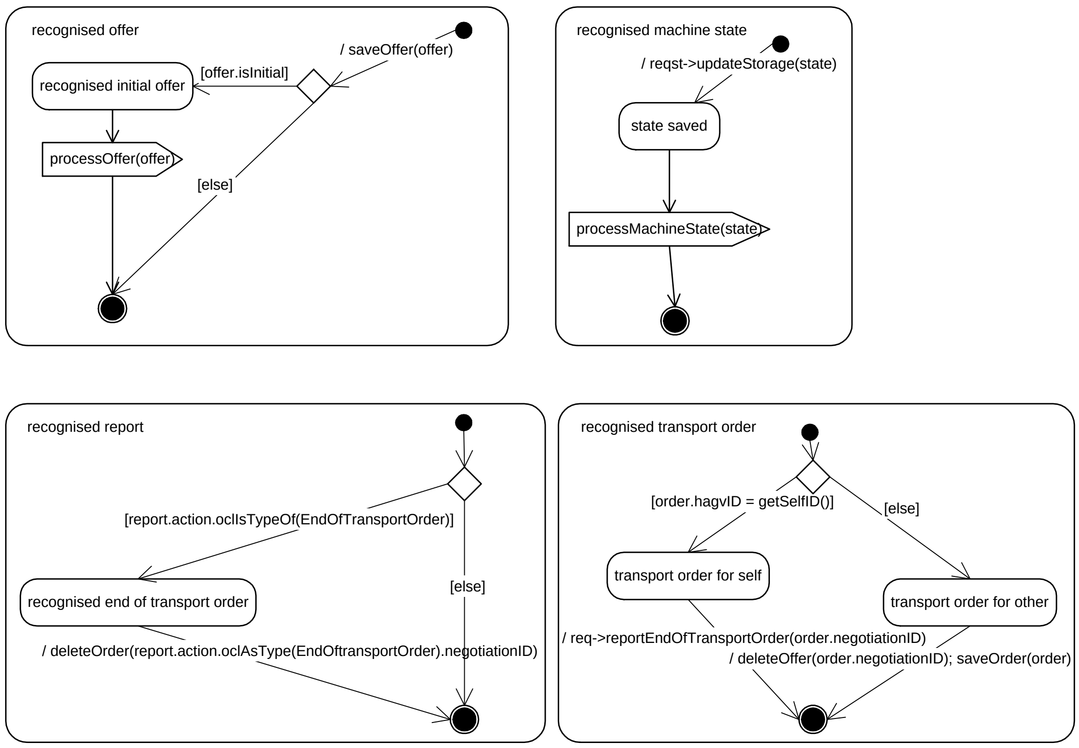

recognised offer
/ saveOffer(offer)
[offer.isInitial]
recognised initial offer
processOffer(offer)
[else]
recognised machine state
/ reqst->updateStorage(state)
state saved
processMachineState(state)
recognised report
[report.action.oclIsTypeOf(EndOfTransportOrder)]
recognised end of transport order
[else]
/ deleteOrder(report.action.oclAsType(EndOftransportOrder).negotiationID)
recognised transport order
[order.hagvID = getSelfID()]
[else]
transport order for self
transport order for other
/ req->reportEndOfTransportOrder(order.negotiationID)
/ deleteOffer(order.negotiationID); saveOrder(order)

Figure 5.15: Substates of `InterpreterBod` statechart

5.3 Composing the Software System

As soon as the components are specified, their interconnection can be defined. In most cases, we have trivial connectors only, because requirements classes and corresponding provisions classes have equal signatures. In these cases, just a renaming (alias) of classes has to be defined.

In a development environment with appropriate tools, no further work would have to be done because the resulting component could be calculated automatically. In order to get an impression of what partial hierarchical composition is all about, some compositions will be shown in detail.

The three components that deal with communication issues, IO, Interpreter and Mediator, can be composed to a single larger component which will be called Communication. The following connectors are needed: `IOReqInterpreter` is fulfilled with `InterpreterPro`, where class `IOReqInterpreter::Interpreter` can be identified with (or renamed to) `InterpreterPro::Interpreter`.

`InterpreterReqMediation` is fulfilled by `MediatorPro`, because class `MediatorPro::Mediator` can be a subclass of `InterpreterReqMediation::InterpreterReqMediation` as it has the desired operation `reportEndOfTransportOrder(negID:NegotiationID)`.

`MediatorReq` is fulfilled by `IOProSending` because of identical signatures of classes `MediatorReqSending` and `IOProSender`.

`IOReqRadio` is not fulfilled and thus becomes an open requirements part of the communication component, but is renamed to `CommunicationReqRadio`. This similarly applies to `InterpreterReqNegotiation` and `InterpreterReqStorage`.

The composed component is given in figure 5.16, its body is given in detail in figure 5.17.

A similar structure is gained when composing the components Negotiator, HandoverControl and DrivingControl to a component Production, see figure 5.18. In this case, five requirements packages are left open, and one provisions package is exposed to the environment.

The Communication and Production components can be composed mutually and with Storage, RadioUnit, and SensorsActuators accordingly, thus gaining a complete model of the system. In figure 5.19, the resulting system is given with insight into the components. If we abstract even one level higher, hiding all component contents, a very compact overview of the system results (figure 5.20). The notation for the latter, though, is not strictly correct as the `«provide»` dependency should be used to connect provisions and requirements parts rather than components.

5.3.1 Behavioural Composition

As stated before, there is no action language defined in the UML. But it should be obvious that an action that represents a method call on another object synchronises

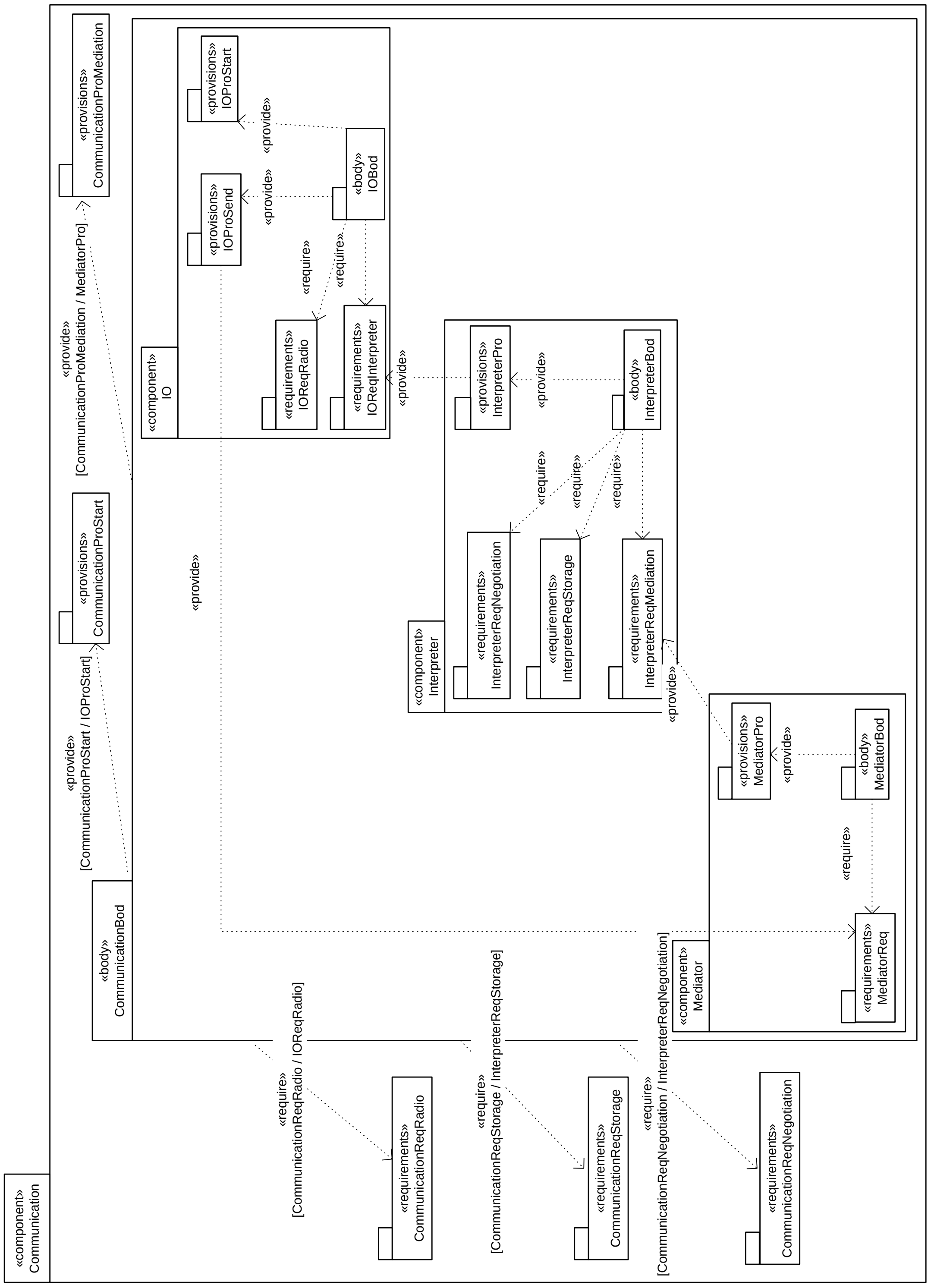

«component» Communication
«provisions» CommunicationProMediation
«provisions» CommunicationProStart
«provide» [CommunicationProMediation / MediatorPro]
«provide» [CommunicationProStart / IOProStart]
«body» CommunicationBod
«provide»
«component» IO
«provisions» IOProStart
«provisions» IOProSend
«body» IOBod
«provide»
«provide»
«require»
«require»
«requirements» IOReqRadio
«requirements» IOReqInterpreter
«provide»
«component» Interpreter
«provisions» InterpreterPro
«body» InterpreterBod
«provide»
«require»
«require»
«require»
«requirements» InterpreterReqNegotiation
«requirements» InterpreterReqStorage
«requirements» InterpreterReqMediation
«provide»
«component» Mediator
«provisions» MediatorPro
«body» MediatorBod
«provide»
«require»
«requirements» MediatorReq
«require» [CommunicationReqRadio / IOReqRadio]
«require» [CommunicationReqStorage / InterpreterReqStorage]
«require» [CommunicationReqNegotiation / InterpreterReqNegotiation]
«requirements» CommunicationReqRadio
«requirements» CommunicationReqStorage
«requirements» CommunicationReqNegotiation

Figure 5.16: Communication component

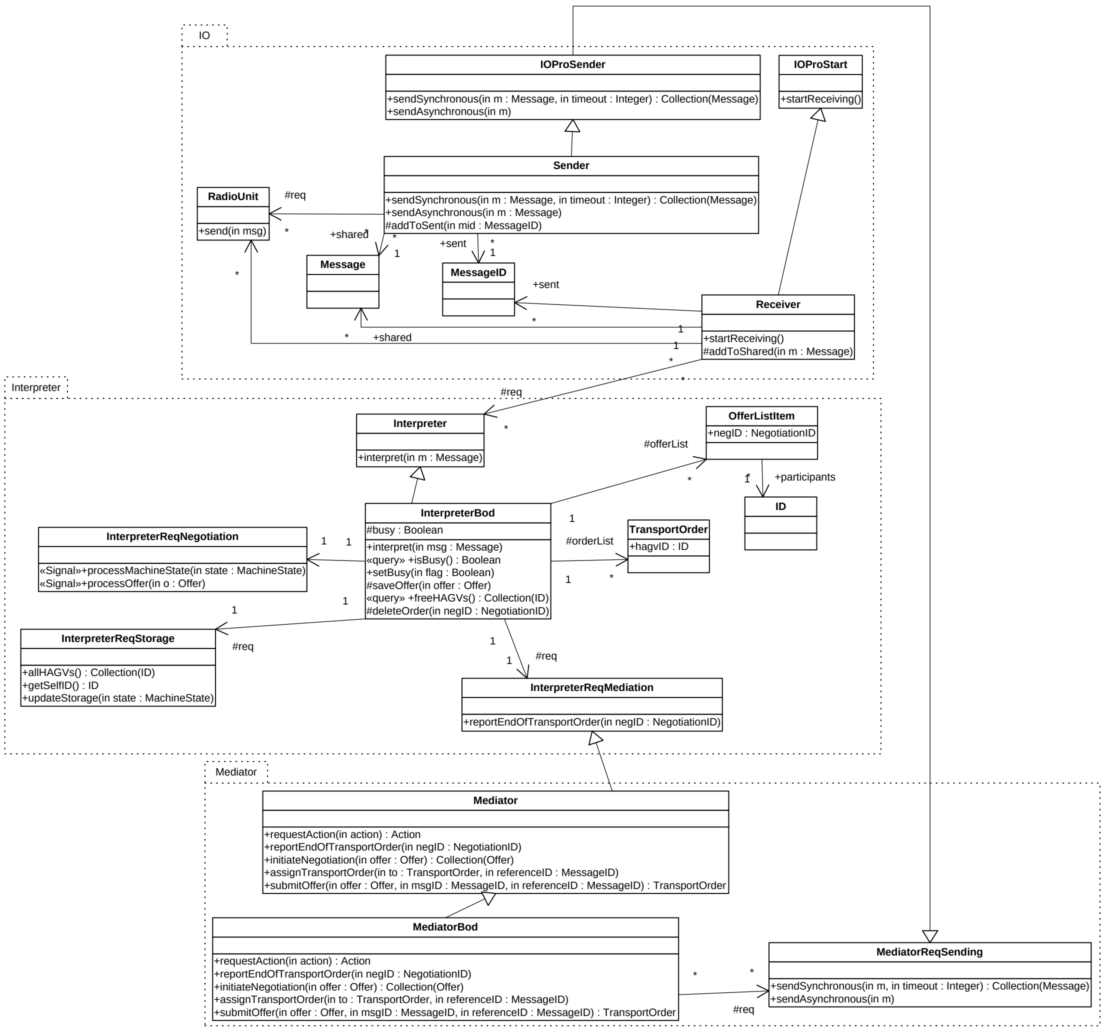

IO
IOProSender
+sendSynchronous(in m : Message, in timeout : Integer) : Collection(Message)
+sendAsynchronous(in m)
IOProStart
+startReceiving()
Sender
+sendSynchronous(in m : Message, in timeout : Integer) : Collection(Message)
+sendAsynchronous(in m : Message)
#addToSent(in mid : MessageID)
RadioUnit
+send(in msg)
#req
+shared
+sent
Message
MessageID
Receiver
+startReceiving()
#addToShared(in m : Message)
Interpreter
+interpret(in m : Message)
OfferListItem
+negID : NegotiationID
#offerList
+participants
ID
InterpreterBod
#busy : Boolean
+interpret(in msg : Message)
«query» +isBusy() : Boolean
+setBusy(in flag : Boolean)
#saveOffer(in offer : Offer)
«query» +freeHAGVs() : Collection(ID)
#deleteOrder(in negID : NegotiationID)
TransportOrder
+hagvID : ID
#orderList
InterpreterReqNegotiation
«Signal»+processMachineState(in state : MachineState)
«Signal»+processOffer(in o : Offer)
InterpreterReqStorage
+allHAGVs() : Collection(ID)
+getSelfID() : ID
+updateStorage(in state : MachineState)
InterpreterReqMediation
+reportEndOfTransportOrder(in negID : NegotiationID)
Mediator
+requestAction(in action) : Action
+reportEndOfTransportOrder(in negID : NegotiationID)
+initiateNegotiation(in offer : Offer) : Collection(Offer)
+assignTransportOrder(in to : TransportOrder, in referenceID : MessageID)
+submitOffer(in offer : Offer, in msgID : MessageID, in referenceID : MessageID) : TransportOrder
MediatorBod
+requestAction(in action) : Action
+reportEndOfTransportOrder(in negID : NegotiationID)
+initiateNegotiation(in offer : Offer) : Collection(Offer)
+assignTransportOrder(in to : TransportOrder, in referenceID : MessageID)
+submitOffer(in offer : Offer, in msgID : MessageID, in referenceID : MessageID) : TransportOrder
MediatorReqSending
+sendSynchronous(in m, in timeout : Integer) : Collection(Message)
+sendAsynchronous(in m)

Figure 5.17: Communication component body

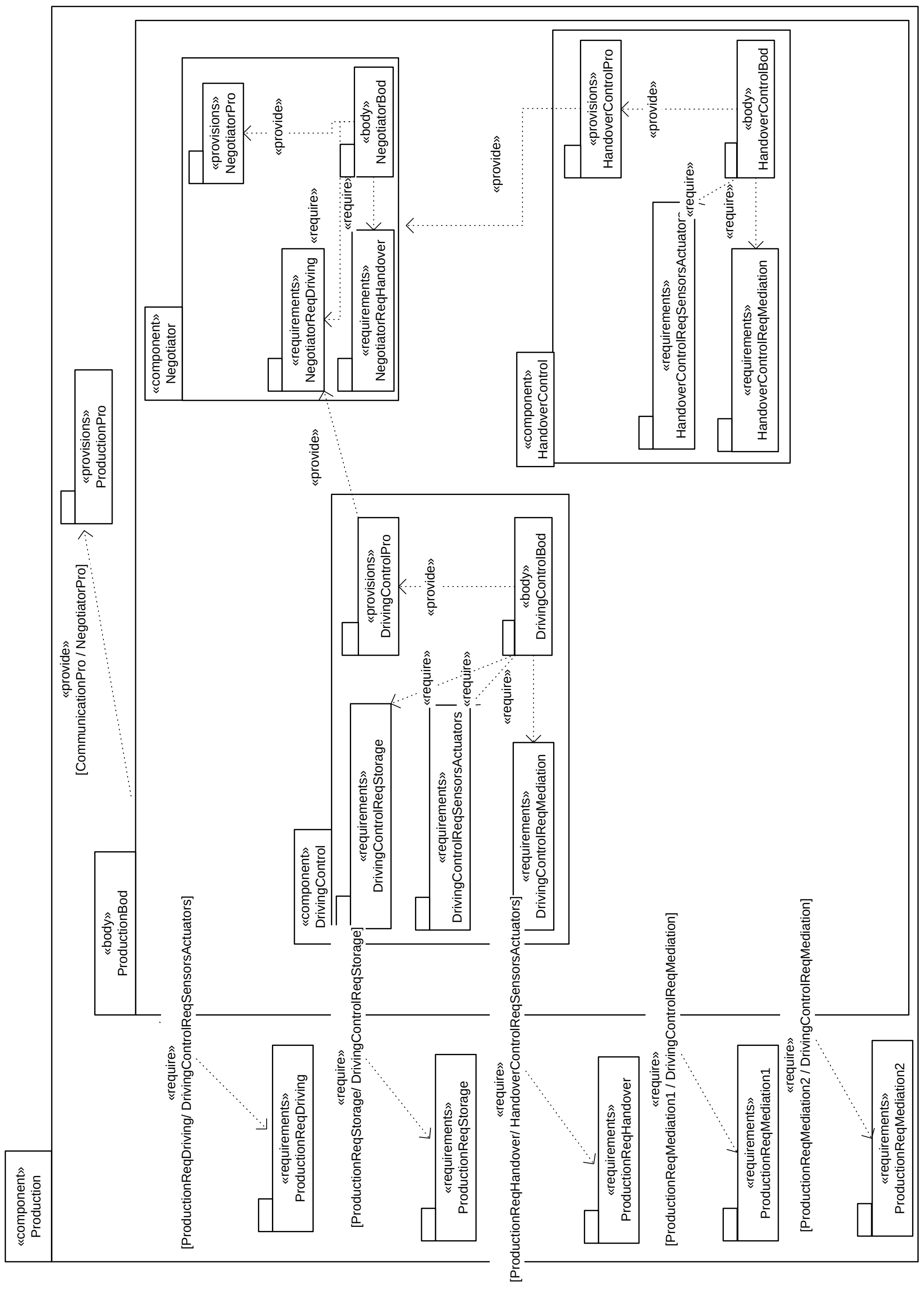

«component» Production
«body» ProductionBod
«provisions» ProductionPro
«provide» [CommunicationPro / NegotiatorPro]
«component» Negotiator
«provisions» NegotiatorPro
«body» NegotiatorBod
«provide»
«require»
«require»
«requirements» NegotiatorReqDriving
«requirements» NegotiatorReqHandover
«component» HandoverControl
«provisions» HandoverControlPro
«body» HandoverControlBod
«provide»
«provide»
«require»
«require»
«requirements» HandoverControlReqSensorsActuator
«requirements» HandoverControlReqMediation
«component» DrivingControl
«provisions» DrivingControlPro
«body» DrivingControlBod
«provide»
«provide»
«require»
«require»
«require»
«requirements» DrivingControlReqStorage
«requirements» DrivingControlReqSensorsActuators
«requirements» DrivingControlReqMediation
«require» [ProductionReqDriving/ DrivingControlReqSensorsActuators]
«require» [ProductionReqStorage/ DrivingControlReqStorage]
«require» [ProductionReqHandover/ HandoverControlReqSensorsActuators]
«require» [ProductionReqMediation1 / DrivingControlReqMediation]
«require» [ProductionReqMediation2 / DrivingControlReqMediation]
«requirements» ProductionReqDriving
«requirements» ProductionReqStorage
«requirements» ProductionReqHandover
«requirements» ProductionReqMediation1
«requirements» ProductionReqMediation2

Figure 5.18: Production component

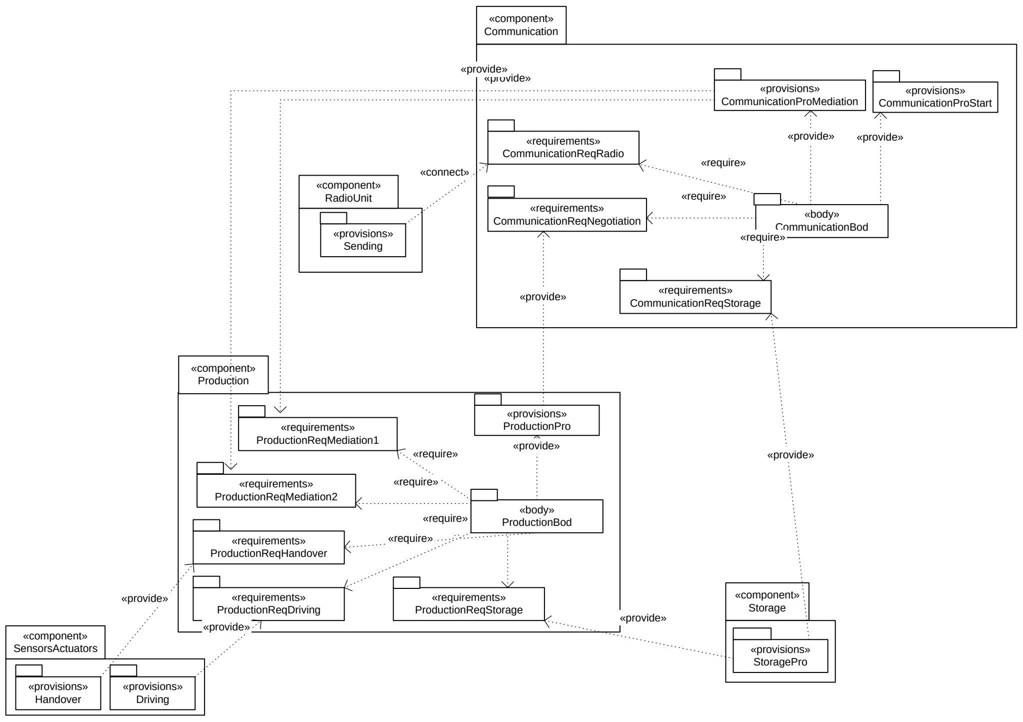

«component» Communication
«provisions» CommunicationProMediation
«provisions» CommunicationProStart
«requirements» CommunicationReqRadio
«requirements» CommunicationReqNegotiation
«body» CommunicationBod
«requirements» CommunicationReqStorage
«component» RadioUnit
«provisions» Sending
«component» Production
«requirements» ProductionReqMediation1
«requirements» ProductionReqMediation2
«requirements» ProductionReqHandover
«requirements» ProductionReqDriving
«provisions» ProductionPro
«body» ProductionBod
«requirements» ProductionReqStorage
«component» Storage
«provisions» StoragePro
«component» SensorsActuators
«provisions» Handover
«provisions» Driving
«provide»
«require»
«connect»

Figure 5.19: System component

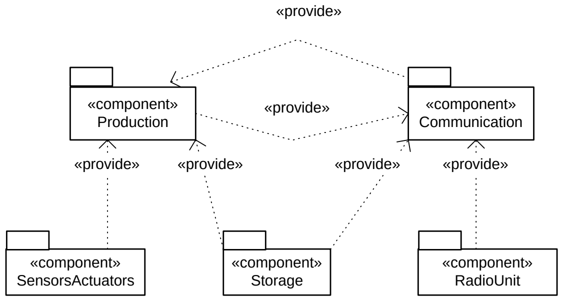


Figure 5.20: System component – Overview

the two objects' statecharts: The one calling the action waits for the execution of method to be completed. The called object triggers a transition because it receives a `CallEvent`.

The `Receiver` statechart on page 85 has an action `req->interpret(m)`. This means navigating along the association to the association end `req`, and calling the method `interpret` there with the parameter `m`. The `Interpreter` class at the association end `req` has only one operation, and thus a trivial statechart, depicted in figure 5.21. When the `IO` and `Interpreter` components are connected properly, the interpreting functionality is specified in class `InterpreterBod`, which has the statechart in figures 5.14 and 5.15 on page 90. The class diagram 5.22 shows the three involved classes.

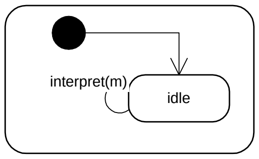


Figure 5.21: `Interpreter` statechart

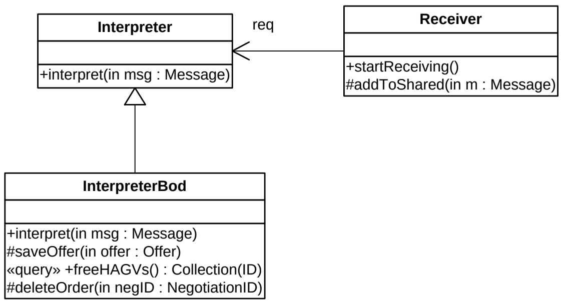

Interpreter
+interpret(in msg : Message)
req
Receiver
+startReceiving()
#addToShared(in m : Message)
InterpreterBod
+interpret(in msg : Message)
#saveOffer(in offer : Offer)
«query» +freeHAGVs() : Collection(ID)
#deleteOrder(in negID : NegotiationID)

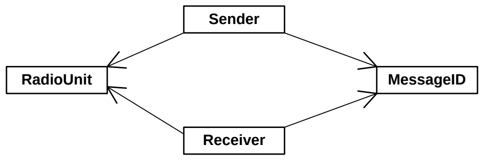
Figure 5.22: Composition of classes that are specified by statecharts

This is an example for a (horizontal) composition of statecharts in conjunction with a (vertical) statechart refinement that preserves the observable behaviour. Another similar example, but with a synchronisation via *asynchronous signals* rather than via synchronous method calls, is the composition of the `InterpreterBod` statechart (figure 5.15) and the `Negotiator` resp. `NegotiatorBod` statecharts (figures 5.4 to 5.7). When a `InterpreterBod` has sent the signal `processOffer`, the interpreter object does not wait for the negotiator to execute the method.

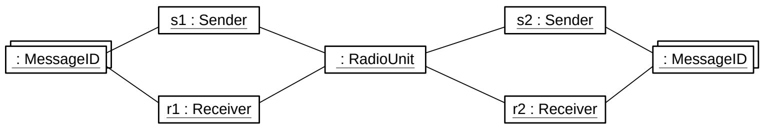
5.3.2 Component Instantiation

As discussed in section 4.4, component instantiation is an interesting topic that is not detailedly handled in this thesis. For illustration purposes an example will be shown.

Several H-AGVs communicate via the same radio unit instance. A static 'glueing' of models over `RadioUnit` is *not* the suitable technique to achieve this, but it is rather an instantiation issue. Figure 5.23 shows how objects (instances of the classes) can be connected via links (instances of the associations) in a way that connects two H-AGVs via one radio unit.



(a) Class diagram



(b) Snapshot (object diagram)

Figure 5.23: An excerpt of the composed system

Note that this example is simplified, especially when saying that there is a shared instance of `RadioUnit` in this distributed system; there would rather be an instance of `RadioUnit` for each H-AGV communicating with some wireless protocol that is specified elsewhere.

Chapter 6

Related Work and Conclusion

Component-based development is one of the most often heard *buzz-words* in software development industry today. Unfortunately, the term is often used with a rather weak meaning, not reflecting the needs of a component stated in the Generic Component Framework and in this thesis. Additionally, component-based development is often *implementation-oriented*, relying on technology platforms called *component infrastructures*. In this chapter, some component modelling approaches related with the UML are shown and related with the one developed in this thesis. Additionally, one implementation platform, the CORBA Component Model, is sketched and the similarities with the component concept of this thesis are shown.

In order to get a feeling for the concepts and the notation of each approach, the running example, the accounting component, is reformulated according to each approach's notation (as far as possible). For the ”original” example, see figure 2.20 on page 36.

6.1 UML Component Concept Developed in the IOSIP Case Study

The component concept introduced in this thesis has its origin in the work on the IOSIP case study. The idea that stereotyped packages can define components and component parts, was presented in [18].

The main issue that was left open is the kind of dependencies between provisions and body and between requirements and body of a component, and between components. As depicted in figure 6.1, there are arrows indicating that the provisions depend on the body, and the body depends on the requirements. Details of these dependency relationships were not given; this was made more precise in this thesis by defining refinement and inclusion of packages.

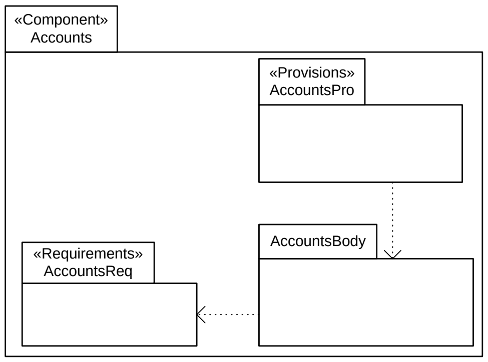


Figure 6.1: Accounting component in the IOSIP notation

6.2 UML Components

The UML has an own component notion, introduced in [28, section 2.5.2.12]:

> ”A component represents a modular, deployable, and replaceable part of a system that encapsulates implementation and exposes a set of interfaces. A component is typically specified by one or more classifiers that reside on the component. A subset of these classifiers explicitly defines the component's external interfaces. A component conforms to the interfaces that it exposes, where the interfaces represent services provided by elements that reside on the component. A component may be implemented by one or more artifacts, such as binary, executable, or script files.”

The accounting example could be denoted as in figure 6.2. All defined classes reside inside the components and are typically not shown in a component diagram. The «uses» dependency indicates that the database component is required by the accounting component.

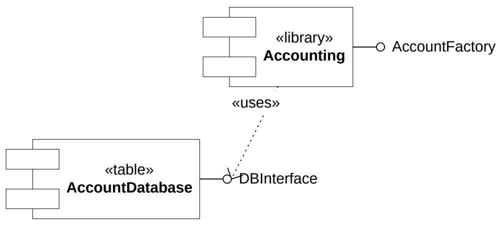


Figure 6.2: Accounting, using UML components

The quotation from the specification and the example expose the main differences between UML components and Generic Framework components:

- There is no way to define abstract import interfaces. This means that a component directly references all used classes, and thus cannot be called self-contained and independent of the environment.
- In their most typical use, UML components are rather implementation-oriented. Their intent is to define code or compilation units rather than model units.
- As discussed in section 2.1.1, there is sometimes a need to have more complex interface specifications than UML interfaces.

Using Components According to Cheesman/Daniels

Cheesman and Daniels propose in their method [4] a use of components that comes somewhat closer to the Generic Framework by demanding that a component consists of a specification part and and implementation part, see figure 6.3. This is somewhat comparable to the idea of an (arbitrarily complex) provisions part and a body part realising the provisions. Still, it is missing a requirements part, so that every import is concrete and not abstract.

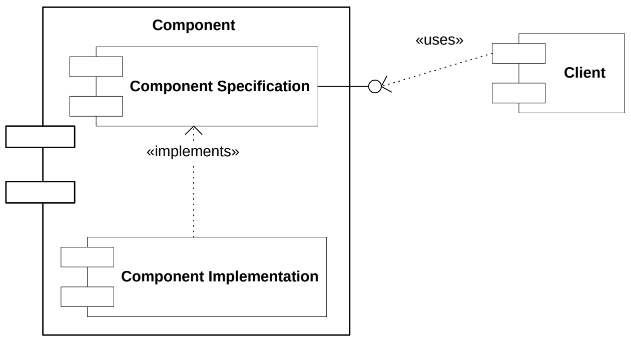


Figure 6.3: Example of a Component according to Cheesman/Daniels

6.3 Component Interface Diagrams

Huber, Rausch, Rumpe introduce a new diagram type in [17, 16] called *component interface diagram*. Components are defined as a dynamically changing set of objects. Each component has exactly one *principal object*, the lifecycle of which is exactly the lifecycle of the component. Clients can ask this object to receive instances of other classes. A new arrow type between operations and classes is labelled with information about the multiplicity of the returned objects, whether a new instance shall be created or an existing shall be reused (as in the Singleton design pattern), and which object shall get the reference to the returned objects (usually the `caller` of the method). Each class is given a multiplicity (in the upper right corner) stating how many instances may exist (for the principal object, typically one).

By being exposed to the environment, an object becomes part of the interface of the component. This way, the interface of a component is dynamically changing.

A component interface diagram does not exist for its own, but is only a higher-level view of a class diagram (similarly as in the UML components case).

The account example could be denoted as in figure 6.4. The labelling `$1->caller` means that a reference to one `ChequeAccount` object, which may have existed before (denoted by `$`), is assigned to the object that called the operation `getAccount(number:Integer)`.

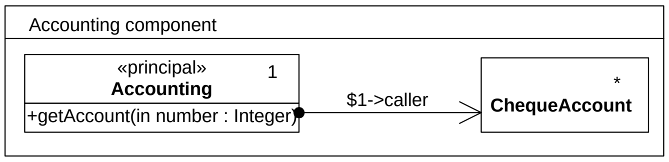


Figure 6.4: Example of a Component Interface Diagram

One main difference is that component interface diagrams emphasise the role of *objects* (instances), whereas the concepts of this thesis mainly deal with *classes* (types). Note especially that the term *interface* is used in a different way; the methods of the principal class are comparable to what was called interface specification in this thesis. An explicit notion of composition does not exist, neither an import or requirements part. An obvious problem is that a new diagram type with new syntactic elements is introduced, which leads to acceptance problems in the UML user community.

6.4 Contract-Aware Components

Weis, Becker, Geihs, Plouzeau introduce a new meta-model element *contract* in [29]. They enhance UML component diagrams by stating that components are specified, in the perspective of the environment, by offered and required *contracts* rather than by pure interfaces. The idea is that such contracts can contain a broader set of properties, including non-functional (e.g. quality of service) specifications. Some of these may be expressed in OCL, others in different formal or informal languages.

The account example could be modified by defining a contract stating that the `Accounting` component requires a possibility to make account data persistent, and a contract stating that the database component offers SQL functionality. The components can be connected if the SQL contract fulfils the persistency requirements, see figure 6.5.

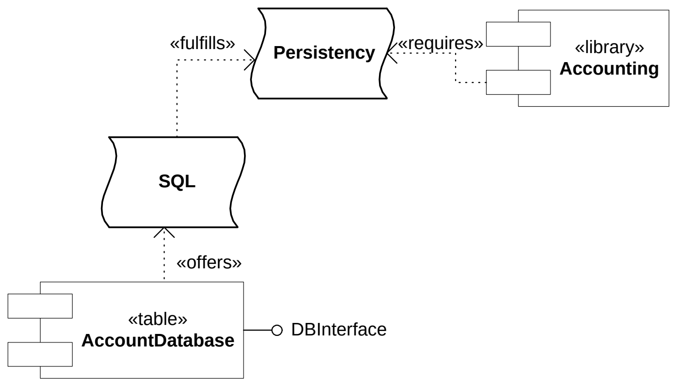


Figure 6.5: Example of contract-aware components

This concept is quite vague as long as more concrete syntax and semantics for contracts are not defined. But the idea seems to reflect the requirements of the Generic Component Framework properly, and goes beyond it by explicitly including non-functional specifications.

6.5 Architectural Connectors and Coordination Contracts

A similar notion of *contract* is used by Fiadeiro et al. in [1], but has a different background, namely *architectural connectors*. This is a concept of *coordinating* components (or other modelling or programming artifacts) by a connector that is set on top of two or more components. This coordination can, for example, be achieved by intercepting method calls at runtime in order to achieve the behaviour defined in the contract.

In the account example, a coordination contract could specify that a balance of a savings account that is below zero may not be written into the database. A simplified notation is given in figure 6.6.

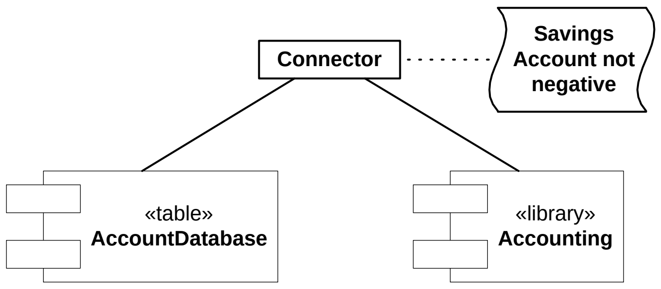


Figure 6.6: Example of a coordination contract

The coordination concept is worked out more detailedly, describing how a component and contract architecture can be set up using well-known design patterns. Coordination contracts are a flexible tool and differ from the Generic Framework especially in the sense that coordination between components on one hierarchy level is the most basic composition operation, rather than a hierarchical composition.

6.6 CORBA Component Model

Version 3 of the Common Object Request Broker Architecture (CORBA, [13]) introduces a new component concept: The CORBA Component Model (CCM, [12]). CORBA is a standard for implementing distributed software system across operating systems and computer architectures. It is widely and increasingly used in the industry. It comes with an object-oriented Interface Definition Language (IDL) that abstracts from the concrete programming language. Mappings from CORBA IDL into most of currently important programming languages exist. Implementations of the CORBA architecture exist for many operating systems running on different platforms. For example, a CORBA-based system can consist of a piece of software programmed in Visual Basic and running on a PC under the Windows operating system, a second piece programmed in Java and running on an Apple Macintosh, and a third piece programmed in C++ and running on a Sparc architecture running Unix, with the pieces communicating seamlessly, and without much additional effort for the developers.

CCM enhances the existing CORBA IDL by *components* with the following keywords:

- `component`, defines a component consisting of an arbitrary number of the next four elements. Reflects the same intuition as `«component»`.
- `provides`, defines provided interfaces. Reflects the same intuition as `«provisions»`.
- `uses`, defines required interfaces. Reflects the same intuition as `«requirements»`
- `publishes`, specifies asynchronous events (signals) sent by this component.
- `consumes`, specifies asynchronous events (signals) received by this component.

Additionally, a *component home* is defined for each component which manages a component's lifecycle, i.e. which can be asked to instantiate a component and return a reference, and to destroy such a component instance.

The intention of an *interface* definition language is to hide implementation details, whereas a component body intends to show such details. As a result, CCM software can most intuitively be modelled by component interface specifications as defined in section 2.5. Figure 6.7 shows an excerpt of the IOSIP model, figure 6.8 shows a possible mapping to CORBA IDL (without making use, for simplicity, of the `module` keyword that defines namespaces).

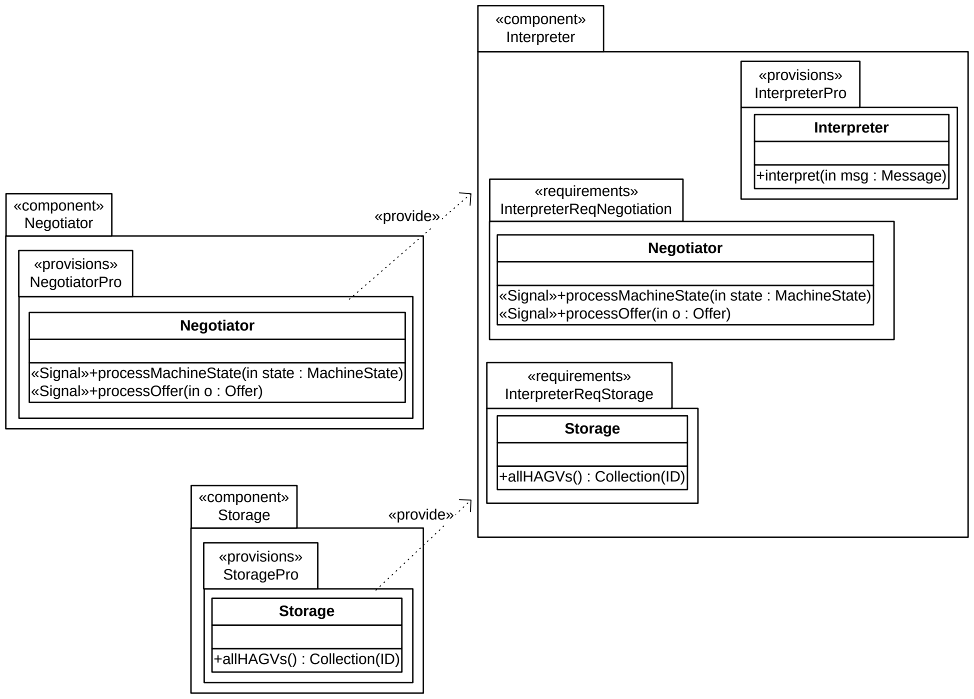

«component»
Interpreter
«provisions»
InterpreterPro
Interpreter
+interpret(in msg : Message)
«requirements»
InterpreterReqNegotiation
Negotiator
«Signal»+processMachineState(in state : MachineState)
«Signal»+processOffer(in o : Offer)
«requirements»
InterpreterReqStorage
Storage
+allHAGVs() : Collection(ID)
«component»
Negotiator
«provide»
«provisions»
NegotiatorPro
Negotiator
«Signal»+processMachineState(in state : MachineState)
«Signal»+processOffer(in o : Offer)
«component»
Storage
«provide»
«provisions»
StoragePro
Storage
+allHAGVs() : Collection(ID)

Figure 6.7: An excerpt of the IOSIP model

The CCM specification goes beyond the pure IDL: it also defines how components are implemented, how they are deployed into a *component container*, and how they are configured by *deployment descriptors*. A component container is a software running on an application server that handles aspects the component developer does not explicitly have to program but which are important in enterprise and other high-quality software, such as security issues, persistency, transaction safety, and distribution over multiple computers. A deployment descriptor is a text file that stores attributes and parameters of a component that can be changed without recompiling the component.

The CCM contains the same basic assumption about components as the Generic Framework, namely that they consist of provided and required interfaces and a hidden part realising the provisions using the requirements. Therefore, a mapping of the UML component concept defined in this thesis to the CCM is possible. But the comparison also shows that modern component technologies contain many features that are often not appropriately modelled in UML. It depends on project size and on the tools and methods used, in which phases of a development process it is

```
// basic types
interface Message{
  // ...
};
interface ID{
  // ...
};
typedef sequence<ID> IDCollection;

// component interfaces
interface Interpreter{
  void interpret( in Message m );
};
interface Storage{
  IDCollection allHAGVs();
};

// event types
eventtype MachineState{
  // ...
};
eventtype Offer{
  // ...
};

// component types
component StorageComp{
  provides Storage storage;
};
component InterpreterComp{
  provides Interpreter interpreter;
  uses Storage storage;
  publishes MachineState state;
  publishes Offer offer;
};
component NegotiatorComp{
  consumes MachineState state;
  consumes Offer offer;
};

// component homes
home StorageHome manages StorageComp {};
home InterpreterHome manages InterpreterComp {};
home NegotiatorHome manages NegotiatorComp {};
```

Figure 6.8: CORBA 3 IDL for the IOSIP model

correct to ”abstract away” such implementation issues, and in which they have to be taken into account. It is not useful to start modelling a system with all details of the target programming language in mind. But it is also dangerous to forget all technological issues, because an un-implementable system could be modelled then.

6.7 Conclusion

In this thesis it was analysed in how far the Generic Component Framework can be instantiated by the UML. Existing UML model elements were discussed and enhanced by the UML extension mechanisms. The results are a UML profile for component-based modelling, and a justification that models conforming to this profile can be viewed as instances of the Generic Framework.

A developer using the UML with this new constructs gains a concept for structuring large models that is at a higher level than the existing techniques. The most important feature of components is the abstract import of services, resulting in a true decoupling of components. This is a small step towards the ”Holy Grail” of *effective reuse* of software (which should include models, code, and other documentation).

In some parts of the thesis the UML was used as if it were a formally defined specification language, by using terms such as refinement transformation and inclusion. But it has to be kept clearly in mind that the UML 1.4 lacks formality and precision in many places. Not every aspect of the Generic Framework can be reflected easily in the UML. Even for the simpler case of class diagrams, problems had to be solved that do not occur in other specification techniques, such as the non-transitive import dependency, see section 2.2.1.

The UML is a complex language; diagram elements of very different nature can be combined almost arbitrarily in order to specify different views on a software system. This causes consistency problems in different dimensions: within one diagram, between diagrams that represent different views of a model, between diagrams at different levels of abstraction, and between diagrams of a system evolution over time. At first sight, modelling a system with components solves none of these problems, but rather introduces a new dimension of complexity. But when a complete and sufficiently formal concept of components and component composition exists, some of the consistency checks have to be performed only on small parts of the system, because the component framework ensures compositionality. Similar arguments hold for other kinds of property preservation. Possible further work could include a concretisation of the refinement notions for class and statechart diagrams, which are left very simple in this thesis.

While working on the UML component concept, it also became obvious that the Generic Component Framework is a quite weak construct as long as there are no restrictions on the transformations and inclusions other than the intuitive descriptions. The UML component concept was designed to fulfil these intuitions, but lacks more precise properties as well as the abstract framework.

If a formal semantics for the UML were defined, the results of this thesis could be reformulated within that formal framework. This could result in a fully formal component framework. But even on the non-formal level this thesis could be used as a starting point for more concrete component concepts, especially by ”plugging in” different notions of refinement that preserve important properties. The component concept as presented in this thesis ensures some basic syntactical consistency. One kind of formalisation of the UML useful in the context of components could be a categorical formalisation, maybe even a description of (a subset of) the UML in terms of High-Level Replacement categories.

An interesting formalisation of refinement of classes by predicate transformer semantics, and a corresponding refinement calculus are introduced in [21].

Besides reuse, another aim of decomposing a system into self-contained parts is the ability to perform various checks and proofs of properties on the model. The ability to formally prove properties of a model is the key advantage of formal modelling techniques. But also with the UML precise modelling is possible. This thesis uses some features of the UML that are not often used, such as a profile definition using OCL, but deliver very precise models. In software development practice, the UML is most often not used with the maximum precision it offers. This becomes obvious, for example, when looking at the UML modelling tools on the market which usually can generate program code from class diagrams without regarding OCL or behavioural specifications (that is, classes and empty method ’stubs’). The developers use only informal specifications and class diagrams as a design model, and implement all behaviour directly in program code. This is appropriate in many practical situations, but does not allow for modern software engineering analysis techniques such as model checking or theorem proving, which are essential in the development of high quality (e.g. safety-critical) software.

The concepts defined in this thesis should, of course, be proven to be useful in practice. The techniques used to define the component concept conform to the UML 1.4 specification, and thus allow to use existing modelling tools. The new stereotypes, constrained by OCL, could be added to a tool’s set of modelling elements. Unfortunately, most UML modelling tools do either lack the possibility to add profiles and stereotypes, or are not able to evaluate OCL, or have a different (proprietary) means to add profiles. In order to make the new stereotypes more usable, new visual notations could additionally be defined.

Acknowledgements

This work is partially supported by the DFG within the research project IOSIP, which is part of the DFG priority program ”Integration of Software Specification Techniques for Applications in Engineering”.

I would like to thank Hartmut Ehrig, Benjamin Braatz, Markus Klein and Mesut Özhan for valuable comments and constructive discussions on the topics of this thesis.

Appendix A

Self-Defined OCL Functions

`allImportsPublic`: Is every element imported into `importing` as public?

```
let allImportsPublic(importing:Package,
  imported:Package):Boolean =
  importing.elementImport[importedElement]->select(
  me:ModelElement | me.namespace = imported )
  ->visibility = public
```

`allPublic`: Do all elements in package `p` have public visibility?

```
let allPublic(p:Package):Boolean =
  p.elementOwnership[ownedElement]->visibility = public
```

`existsGeneralization`: Is there any generalisation from an element in package `from` to an element in package `to`?

```
let existsGeneralization(from:Package, to:Package):Boolean =
  from->ownedElement->exists(
    ch:GeneralizableElement | to->ownedElement->exists(
      par:GeneralizableElement | par->specialization
       = ch->generalization
  ))
```

`isNameConflictFree`: Is there a name conflict between packages `A` and `B`?

```
let isNameConflictFree(A:Package, B:Package):Boolean =
  A->contents->forAll( a:ModelElement |
    not B->contents->exists(
      b:ModelElement | a.name = b.name))
```

`isSubtype`: Is `Sub` a subtype of `Super`?

```
let isSubtype(Sub:Class, Super:Class):Boolean =
  Super.specialization->exists(
  g:Generalization | g->child->includes(Sub))
```

`isSupertype`: Is `Super` a supertype of `Sub`?

```
let isSupertype(Super:Class, Sub:Class):Boolean =
  isSubtype(Sub, Super)
```

`isSpecialisation`: Is `Spec` a specialisation of `Gen`?

```
let isSpecialisation(Spec:GeneralizableElement,
  Gen:GeneralizableElement):Boolean =
    Gen.specialization->exists(g:Generalization
      | g->child->includes(Spec))
  or
    Gen.supplierDependency->exists(r:Realization
      | r->client->includes(Spec))
```

`isGeneralisation`: Is `Gen` a generalisation of `Spec`?

```
let isGeneralisation(Gen:GeneralizableElement,
  Spec:GeneralizableElement):Boolean =
    isSpecialisation(Spec, Gen)
```

Bibliography

[1] Luís Filipe Andrade and José Luiz Fiadeiro. Contracts: Supporting architecture-based evolution. submitted. `www.fiadeiro.org/jose/papers/`.

[2] Dave Astels. Refactoring with UML. In *Proc. 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (XP2002)*, 2002. `www.xp2002.org/prog_full.html`.

[3] Arnulf Braatz and Arno Ritter. Referenzfallstudie Produktionstechnik Version 2.0, 2001. `tfs.cs.tu-berlin.de/projekte/indspec/SPP/themenbereiche.html`.

[4] John Cheesman and John Daniels. *UML Components*. Addison-Wesley, 2001.

[5] Desmond D'Souza and Alan Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.

[6] Jürgen Ebert and Gregor Engels. Specialization of object life cycle definitions. Fachbericht Informatik 19/95, Universität Koblenz-Landau, Fachbereich Informatik, 1995.

[7] Hartmut Ehrig, Robert Geisler, Marcus Klar, and Julia Padberg. Horizontal and Vertical Structuring Techniques for Statecharts. In A. Mazurkiewicz and J. Winkowski, editors, *LNCS Vol. 1243, CONCUR'97: Concurrency Theory, 8th International Conference, Warsaw, Poland*, pages 181–195. Springer Verlag, 1997.

[8] Hartmut Ehrig and Bernd Mahr. *Fundamentals of Algebraic Specification 2: Module Specifications and Constraints*. Springer, Berlin, 1990.

[9] Hartmut Ehrig, Fernando Orejas, Benjamin Braatz, Markus Klein, and Martti Piirainen. A Generic Component Concept for System Modeling. In *Proc. FASE 2002: Formal Aspects of Software Engineering*, 2002.

[10] Hartmut Ehrig, Fernando Orejas, Benjamin Braatz, Markus Klein, and Martti Piirainen. A Component Framework Based on High-Level Replacement Sys-

tems. In *Proc. International Workshop on Graph Transformation and Visual Modeling Techniques*, 2002.

[11] Gregor Engels, Reiko Heckel, and Jochen Malte Küster. Rule-Based Specification of Behavioral Consistency Based on the UML Meta-model. In *Proc. UML 2001: Formal Aspects of Software Engineering*, volume 2185 of *Lecture Notes in Computer Science*, pages 272–286. Springer-Verlag, 2001.

[12] Object Management Group. Common object request broker architecture: Component model specification (version 3.0). Technical Report formal/2002-06-65, Object Management Group, 2002. `www.omg.org/cgi-bin/doc?formal/02-06-65`.

[13] Object Management Group. Common object request broker architecture: Core specification (version 3.0). Technical Report formal/2002-12-06, Object Management Group, 2002. `www.omg.org/cgi-bin/doc?formal/02-12-06`.

[14] Object Management Group. UML profile for CORBA specification (version 1.0). Technical Report formal/02-04-01, Object Management Group, 2002. `www.omg.org/cgi-bin/doc?formal/02-04-01`.

[15] Object Management Group. XML metadata interchange (XMI) specification (version 1.2). Technical Report formal/02-01-01, Object Management Group, 2002. `www.omg.org/cgi-bin/doc?formal/02-01-01`.

[16] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Component interface diagrams: Putting components to work. Technical Report TUM-I9831, Technische Univerität München, 1998.

[17] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling dynamic component interfaces. In Madhu Singh, Bertrand Meyer, Joseph Gil, and Richard Mitchell, editors, *Technology of Object-Oriented Languages and Systems (TOOLS 26)*. IEEE Computer Society, 1998.

[18] Markus Klein, Mesut Özhan, and Martti Piirainen. Agent-based material flow. In Reiko Heckel, Tom Mens, and Michel Wermelinger, editors, *Proc. Workshop on Software Evolution Through Transformations*, volume 72. Elsevier Science Publishers, 2002.

[19] Stefan Mann, Alexander Borusan, Hartmut Ehrig, Martin Große-Rhode, Rainer Mackenthun, Asuman Sünbül, and Herbert Weber. Towards a component concept for continuous software engineering. Technical Report 55/00, Fraunhofer Institute for Software and Systems Engineering, 2000.

[20] Bertrand Meyer. Applying ”Design by Contract”. *IEEE Computer*, 25(10):40–51, October 1992.

[21] Anna Mikhajlova and Emil Sekerinski. Class refinement and interface refinement in object-oriented programs. In John Fitzgerald, Cliff B. Jones, and Peter Lucas, editors, *Proc. FME'97*, volume 1313 of *Lecture Notes in Computer Science*, pages 82–101. Springer-Verlag, 1997.

[22] Julia Padberg, Herbert Weber, and Asuman Sünbül. Petri net based components for evolvable architectures. In *Transactions of the SDPS*, volume 6(1). Society for Design and Process Science, 2002.

[23] Bernhard Rumpe, M. Schoenmakers, A. Radermacher, and Andy Schürr. UML + ROOM as a standard ADL? In *Proc. ICECCS'99 Fifth IEEE International Conference on Engineering of Complex Computer Systems*, 1999.

[24] Enterprise JavaBeans specification (version 2.0). Technical report, Sun Microsystems, 2001. `java.sun.com/products/ejb/docs.html`.

[25] Gabriele Taentzer. A visual modeling framework for distributed object computing. In Bart Jacobs and Arend Rensink, editors, *Formal methods for open object-based distributed systems*, volume V. Kluwer Academic Publishers, 2002.

[26] Jennifer Tenzer and Perdita Stevens. Modelling recursive calls with UML state diagrams. To appear in Proc. FASE 2003.

[27] Laurence Tratt, Tony Clark, and Andy Evans. Modelling generalization and other class-to-parent relationships. Technical Report TR-02-05, Department of Computer Science, King's College London, July 2002.

[28] UML Revision Taskforce. OMG Unified Modeling Language Specification (version 1.4). Technical Report formal/2001-09-67, Object Management Group, 2001. `www.omg.org/cgi-bin/doc?formal/01-09-67`.

[29] Torben Weis, Christian Becker, Kurt Geihs, and Noël Plouzeau. A UML meta-model for contract aware components. *Lecture Notes in Computer Science*, 2185:442–456, 2001.