Category 2: Development of formal methods

Components, Contracts, and Connectors for the Unified Modelling Language UML

Claus Pahl

School of Computer Applications
Dublin City University
Dublin 9, Ireland

Abstract. The lack of a component concept for the UML is widely acknowledged. Contracts between components can be the starting point for introducing components and component interconnections. Contracts between service providers and service users are formulated based on abstractions of action and operation behaviour using the pre- and postcondition technique. A valid contract allows to establish an interconnection - a connector - between the provider and the user. The contract concept support the re-use of components by providing means to establish and modify component interconnections. A flexible contract concept shall be based on a refinement relation for operations and classes, derived from operation abstractions. Abstract behaviour, expressed by pre- and postconditions, and refinement are the key elements in the definition of formal and flexible component and component interconnection approach.

1 Introduction

Contracts formulate an agreement between two (or more) components: a user needs additional functionality in order to fulfill his/her duties, a provider offers services which might help the user. A contract specifies obligations. The provider guarantees a certain functionality if the user guarantees a certain environment. The obligations can be expressed using the pre- and postcondition technique [Mey92, LB99]. A *connector* realises a *contract* between a service provider and a service user, i.e. it establishes an interconnection between both of them. The contract states which semantical requirements (or expectations) these services should match. The *re-usability* of components depends on the support of component abstraction in order to make components available through libraries and on the support of adaptation techniques in order to adapt library components to actual requirements, i.e. to glue service provider and user together [Wec97, Nor98, LS00].

Packages in the Unified Modelling Language UML are a grouping mechanisms which allows a designer to assemble classes (or other elements) into components. The need to improve the notion of packages in the UML has been clearly identified. Two reasons are usually given [pUG99]. Firstly, packages themselves should be developed into components in order to integrate component-based development into the UML-notation. Secondly, packages are the main element of the meta-notation used to describe the semantics of UML. Packages should help to develop a modular definition and to provide a flexible language architecture.

We propose to improve the interfaces of packages by providing import and export interfaces based on abstract semantical information. Packages shall be composed based on these interfaces. An import interface states which services from other packages shall be used and how they are expected to work. An export interface describes the services which are provided in abstract terms. The export states the properties of services that are available to prospective users. Contracts are formed based on the services required and the services provided.

We will use the UML context to motivate and present a flexible re-use oriented component composition framework. Interaction is the composition mechanism. Two components are composed by establishing an interaction infrastructure between them. The flexibility of the composition mechanism is crucial. Two issues have to be addressed: firstly, the contracts shall be formulated using a powerful constraint language, and, secondly, the connectors shall allow a flexible establishment and re-configuration of connections between components.

We believe that a refinement relation is important for the rigorous development of software artifacts, and that a powerful refinement notion can also form the glue needed to adapt services provided by some package to the requirements stated in a contract. Refinements of operation and action abstractions based on pre- and postconditions will forms the basis of a refinement relation between classes and components. The essential advantage of the pre- and postcondition technique is that it is suitable for abstracting internal object behaviour, but can also be used to constrain the interaction between objects via contracts. In [pUG99] semantics for the UML is suggested as a combination of denotational semantics and proof rules. We will follow this suggestion. We will in particular focus on a framework which allows us to establish a proof system. Modal logics [Sti92] – and its constructive variants such as TLA [Lam94, Lam99] – have motivated our formal framework for the specification and reasoning of properties of dynamic systems. The semantics of actions – and other model elements – can be given in terms of modal logics. Modal logic provides therefore the opportunity to express a more precise semantics of refinement and other forms of abstractions in terms of abstract dynamic behaviour. The way to semantical package interfaces and connectors leads via abstraction of actions.

Formulating contracts between components and formalising the infrastructure for the interaction between these components based on the contracts needs particular attention. An extension of the π-calculus [MPW92a, MPW92b, Mil99] shall be used to define contracts and establish connectors between components. The π-calculus is combined with the first-order modal reasoning, which is integrated into the calculus via a constraint language. A composition calculus for contracts and connectors is developed. Both contracts and connectors for the dynamic interaction are defined in the calculus. The extended calculus including the modal calculus can be interpreted in state-based algebraic structures (called objects). The π-calculus has been designed to deal with *mobility*, i.e. the capacity to change the connectivity of a network. We apply this idea to the space of connected (or composed) components. We use the *polyadic π-calculus* as the underlying framework to define contracts for component composition and interaction. The calculus is in particular suitable since it models the establishment of connections and also their maintenance (changing compositions due to evolving requirements). It provides the basis for a flexible re-use based concept for component composition and interaction.

Section 2 introduces behaviour abstraction, abstract interfaces and a notion of components. Their interconnection based on contracts and connectors is dealt with in Section 3. In Section 4 we present a semantical framework for behaviour abstractions, interfaces and components. Reasoning about component composition is the content of Section 5. This involves a formalisation and generalisation of the refinement and their properties. We end with related work and conclusions.

2 Abstract Behaviour, Interfaces, and Components

Among the requirements for an improved UML package concept stated by the precise UML group in their Response to the UML 2.0 Request for Information [pUG99] are *multiple imports* (a package can import several services from several packages at the same time – this means that possibly a number of contracts are formed), *renaming* (syntactical adjustment should be possible, names of service in export and import interface might not be the same, even though they might realise the same service), and *adding elements* to imported elements (it should be possible to add elements to imported elements in the importing package, thus refining the import). Formality and rigour are two general requirements which shall be added to the list. In this section, we will outline the concepts to tackle these requirements.

The case study from which excerpts shall be used to motivate and illustrate our ideas is a document authoring and management system consisting of:

– Interfaces for authors and users `UserInterface`: the operations `followLink` and `inputURL` are available to the user, whereas `sendRequest` is an internal operation which contacts the server.
– Servers for authors `AuthoringServer` and users `ContentServer`: The content server is located at a particular `address`. It requests a document (identified by a URL) from the database and returns the document. The authoring server works on a particular current document, which can be loaded, modified, and checked syntactically with respect to its internal structure.
– A shared database for documents `Document`. A document can be updated with some text at a particular document position.

Some of the class signatures are presented in Figure 1. We narrow our interpretation of a component to classes in this example. We can identify two instantiations of the same pattern – a 3-tiered architecture for database-supported Web-interfaced systems – with the same database part. The operations in the

```
Class UserInterface                   Class ContentServer
  attributes                            attributes
    XML_enabled:boolean                   address:URL
  operations                            operations
    followLink(url:URL)                   request(url:URL):Document
    inputURL(url:URL)
    sendRequest(url:URL)

Class Document                        Class AuthoringServer
  attributes                            attributes
    content:Text                          currentDoc:Document
    author:String                         parseDoc():boolean
  operations                            operations
    edit(pos:Position,update:Text)        loadDoc(url:URL)
                                          modifyDoc(pos:Position,update:Text)
```

Fig. 1. Classes for the Document Management System

system are rather simplistic, an in-depth modelling with substates and subactions is in general not necessary. This simplicity makes it an ideal candidate for illustration.

2.1 Actions, Operations, and their Abstraction

The *internal dynamics* of an object, i.e. which states it can have, can be described by statechart diagrams. Activities of a state can be specified. An activity consists of an event and the action which is triggered by the event (possibly guarded):

$$\textit{event-signature}\ [\ \textit{guard-condition}\ ]\ /\ \textit{action-expression} \tag{1}$$

Events cause transitions between states. Each state is described by a name and internal (state) transitions. The UML definition includes an explicit *send-clause*, a special action, which shall be subsumed here as an action for simplicity. We associate *event signatures* and *action expressions* obtaining *operation definitions* in order to simplify the notion of actions and operations for this investigation:

$$e(p_1 : t_1, \ldots , p_n : t_n) \stackrel{\text{def}}{=} \textit{action-expression} \tag{2}$$

Action expressions can be assignments to state variables $x := t$, operation calls $op(x_1, \ldots , x_n)$, send clauses $obj.op(x_1, \ldots , x_n)$ and action sequences combined using the sequence combinator ';'.

Objects interact dynamically via message exchange, realised by operation calls. *Object interaction* can be described using sequence, collaboration and activity diagrams. We have already introduced object interaction through the *send*-action. The sequence diagram allows us to describe sequences of object interactions considering several objects at the same time. It describes the interaction protocol.

An action expression or an operation definition can be *abstracted* by *pre- and postconditions* in order to express abstract dynamic behaviour. The Object Constraint Language OCL [Gro99] supports *pre- and postconditions* for the specification of operations. Abstract specifications are essential to built declarative, possibly under-determined models – an important feature for the formal development of software systems. Preconditions associate constraints with parameters and the postcondition constrains the operation result.

$$\begin{array}{l} operationName(p_1 : t_1, \ldots , p_n : t_n) \ returns \ rt \\ \quad pre : p_1 > \ldots \\ \quad post : result = \ldots \end{array} \tag{3}$$

An **abstract specification** or an **abstract interface** is a collection of abstract operation specifications using pre- and postconditions. An example shall illustrate the pre- and postconditions.

```
request ( url: URL ) returns Document
     pre  url > checkURL(url)          -- URL is well-formed
     post result = DocForURL(url)      -- Doc corresponds to URL
```

The `request` operation is provided by the `ContentServer` which can act as a service provider. The service user might the `UserInterface`. It might call `request` within its `sendRequest` operation. This functionality request would be stated in its import interface. A library of re-usable components could include a content server component. Its export interface has to satisfy the user's requirements which are formulated in a contract between both parties. A semantical adaptation might be necessary, if e.g. the library component is too general (a generic component can be instantiated).

2.2 The Specification of Interfaces and Components

We will apply the component-notion to UML packages. Packages allow us to group semantically related model elements. Packages do not provide much semantics currently [Gro99], except that packages 'own' their constituent elements, i.e. these elements can only be part of one package. There is one important relationship between packages: import (from other packages that own the desired element). Import expresses a dependency. An import section specifies what services are needed, but not where these services might come from. Packages, and classes in UML can have interfaces. An interface is described by a set of operation signatures. Other packages might relate to these interfaces. Their purpose is to support well-structured system architectures, providing contracts between participating model elements.

Let us define a **component** (or package – we shall use both terms synonymously) as a triple $C = \langle Imp, Class, Exp \rangle$ where[1]

[1] Sometimes we use projections $Imp(C)$, $Class(C)$, or $Exp(C)$ to refer to the respective elements.

```
Contract Contr
        provider   ContentServer
        user       UserInterface
    attributes
        currentURL : URL
    operations
        myRequest(url:URL) : Document
            pre   checkURL(url)
            post  resultDoc = DocForURL(url)
        mySearch(term:Text) : Document
            pre   ..
            post  ..
    syntactic match
        myRequest is matched by request
        ..
```

Fig. 2. A Contract – an Abstract Interface

- *Imp* is called the *import interface* (importing requested functionality described by pre- and postcondition-based constraints),
- *Class* is the class or package implementation (e.g. in terms of actions),
- *Exp* is called the *export interface* (abstracting the services provided by the component in terms of pre- and postconditions).

An abstract interface is decribed by a signature, pre- and postconditions for each operation, and invariants. A notion of *correctness* shall be introduced: the export *Exp* has to be an abstraction of the implementation *Class*. Each abstract operation in an interface is specified by pre- and postconditions in a form that generalises the OCL here:

$$\begin{array}{l} operationName(p_1 : t_1, \ldots , p_n : t_n) : rt \\ \quad pre : F \\ \quad post : G \end{array} \tag{4}$$

F and G can be arbitrary first-order formulas. Each class, interface or operation has a associated *signature*. A signature assembles the sorts of the constituent elements of the particular element. Figure 1 contains class and operation signatures.

Figure 2 contains an example of a (rather incomplete) abstract interface – the attribute and operation parts. This abstract interface is wrapped up by a contract between two components. The export interface of a suitable re-usable library component has to satisfy the requirements stated in the contract. *Contracts* are extensions of abstract interfaces that will specify the requirements of the prospective service user which a service provider is supposed to satisfy. Additionally, a contract includes syntactic matching information, here that the `myRequest` operation – as the operation might be called in the user interface – is matched by `request` of the service provider. A component can import from various other components, i.e. it can make separate contracts with each of these components. Contracts are formulated in the customer's (service user's) terminology. This is sensible because names in the library components might be too generic and thus not suitable for the application context.

Contracts between components might be designed before the components itself are realised. A contract is an abstract interface describing a set of operations through pre- and postconditions. Both service provider and service user have to relate to the contract description. The provider must satisfy the contract constraints. The service user might be satisfied with less than what is described in the connector. The contract is instantiated into a *connector* for object interactions between service provider and service user.

3 Component Composition

The composition mechanism is interaction: functionality is requested by one component and provided by another via a communication channel. Two components are composed by establishing an interconnection – a connector – between them. Contracts constrain the composition. We propose a two-tiered approach for the composition of components. The *upper*, more abstract *tier* defines contracts between components, i.e. a service provider and a service user. Technically, a contract establishes a communication infrastructure on which the components can interact. A private interaction channel between provider and user is created, if the contract constraints are satisfied. The *lower*, more implementation-oriented *tier* realises a connector, an interaction channel, between provider and user. Messages can be passed along that channel, i.e. provider services can be invoked and results can be transferred back.

We assume a collection of re-usable library components (service providers) and a collection of components part of a system to be developed. The latter ones (service users) require functionality in order to be executable. These requirements are formulated in form of an import interface. A contract establishes a relationship between the import requirements and provided services.

All components shall initially be connected via a **select channel** sC of sort $selectChan$ which shall help us to formulate the interconnection of two components related by a contract. A suitable service provider has to be selected based on a component's import requirements. The most suitable should be selected among the available ones. Technically, a request from the component C

$$C \stackrel{\text{def}}{=} \textsc{Select}\ \overline{sC}\langle cC\rangle.C' \tag{5}$$

should be answered by the most suited service provider P_i from the library

$$P_i \stackrel{\text{def}}{=} \textsc{Choose}\ sC(y).\overline{y}\langle\epsilon\rangle.P_i' \tag{6}$$

using the **contract channel** $cC\!:\!contractChan$ between C and P_i (supplied by C and bound to the formal parameter y in P_i). $\overline{sC}\langle cC\rangle$ denotes the output of cC on channel sC and $sC(y)$ denotes input of parameter y via the same channel.

The provider replies to the user sending an empty data token via the contract channel cC, which is bound to its formal parameter y.

3.1 Contracts

The situation before establishing the contract shall be described as follows: The component C (the user) requires a service (an operation) m and the provider P offers a service (an operation) n. Both operations m and n are described by pre- and postconditions, e.g. $pre(m)$ and $post(m)$.

Several constraints have to be considered. The *first syntactical issue* to be considered is the proper use of names in interactions. This shall be captured in a variant of the standard React-rule which describes the state transformation triggered by an interaction realising a *sorting discipline* [Mil99].

$$\textsc{React}_S: \quad \overline{z}\langle x\rangle.C|z(y).P \longrightarrow C|P \quad \text{iff } sort(z) = sort(x) = sort(y) \tag{7}$$

This expresses that an interaction can only take place using channel z if the parameters x and y are of the same sort. The sorting *sort* applied to a channel name is a mapping which characterises the sorts of elements that can be passed along a channel. The sorting is preserved by the interaction rule [Mil99]. The *second syntactical issue* relates to the *syntactical matching* between service user and service provider, see e.g. Figure 2. The syntactical constraint can formally expressed by the existence of a signature morphism $\rho : C \to P$. The signature morphism ρ has to be applied to show that all elements required are actually provided in the correct form. The *semantical condition* is the existence of a refinement relation between m and $\rho(m)$, expressed as $m \overset{R}{\rightsquigarrow} \rho(m)$ or m is refined by $\rho(m)$ (or $\rho(m)$ satisfies m). We define the refinement using pre- and postconditions.

$$m \overset{R}{\rightsquigarrow} \rho(m) \;\text{ iff }\; pre(m) \to pre(\rho(m)) \wedge post(\rho(m)) \to post(m) \tag{8}$$

Preconditions can be weakened – the refinement is more likely to be applicable – and postconditions can be strengthened – the result is better. $\rho(m)$ describes the provided service, reachable via the provider's in-port. Here, $\rho(m)$ shall refer to n. m describes the required service. It will be accessed via the user's out-port.

We shall illustrate the weakening of a precondition. `modifyDoc` is an operation which is provided by the `AuthoringServer` class and might be requested by an `Interface` class. The contract could state the following requirements (this corresponds to the user's requirements):

```
myModifyDoc ( myDoc:Document, myUpdate:Text ) returns Document
     pre   parseDoc()
     post  if parseDoc() then update() else nothing()
```

A corresponding operation might be available in the library with the abstract export specification

```
modifyDoc ( doc:Document, update:Text ) returns Document
      pre   true
      post  ..
```

which means that there are no requirements (precondition) imposed on the syntactical structure of the document. Syntactically, $\rho(\texttt{myModifyDoc}) = \texttt{modifyDoc}$ matches. The contract requires semantically that $\texttt{myModifyDoc} \overset{R}{\rightsquigarrow} \texttt{modifyDoc}$, which means $\mathit{pre}(\texttt{myModifyDoc}) \rightarrow \mathit{pre}(\texttt{modifyDoc})$ or $\texttt{parseDoc()} \rightarrow \texttt{true}$. This shows that the library operation matches the requirements. The contract is satisfied and an interconnection between the components can be established.

A contract between a single import m of component C and a provider P providing $\rho(m)$ should result in an interconnection between both. We assume that the contract channel cC exists with sorting $\mathit{sort}(cC) = \mathit{interactChan}$ ($\mathit{sort}(sC)$ equals $\mathit{contractChan}$). For a user

$$C' \stackrel{\mathrm{def}}{=} \textsc{Request}\ \overline{cC}\langle m\rangle.C'' \tag{9}$$

and a provider

$$P' \stackrel{\mathrm{def}}{=} \textsc{Provide}\ cC(n).P'' \tag{10}$$

we define the **contract rule**:

$$\textsc{Contract}: \qquad C|P \;\longrightarrow\; \texttt{private}\ m\!:\!iC\ (C'|P'\{^m/_n\}) \ \text{ iff } m \rightsquigarrow n \tag{11}$$

such that the sorting is guaranteed. Channel $m\!:\!\mathit{interactChan}$ is the **interaction channel**, or the **connector**, between C and P. The restriction $\texttt{private}\ m\!:\!iC$ creates a private channel m of sort iC between C and P (by introducing scopes). The sorting of m should correspond to m's signature. We illustrate this again using the `modifyDoc`-operation. The user `Interface`

$$\texttt{Interface}' \stackrel{\mathrm{def}}{=} \textsc{Request}\ \overline{cC}\langle\texttt{myModifyDoc}\rangle.\texttt{Interface}''$$

requests the service `myModifyDoc` which is provided by the `AuthoringServer`

$$\texttt{AuthoringServer}' \stackrel{\mathrm{def}}{=} \textsc{Provide}\ cC(\texttt{modifyDoc}).\texttt{AuthoringServer}''$$

Applying the Contract-rule results in a parallel composition of `Interface` and `AuthoringServer` objects where `modifyDoc` replaces `myModifyDoc` in the provider `AuthoringServer`.

We shall briefly address a contract between a component and two providers illustrating multiple imports. Allowing a component to import functionality from several providers was one of the reasons to choose the π-calculus because of its ability to express the concurrent existence of service providers. Otherwise, a variant of the λ-calculus might have been another suitable formalism (see [Pah00]). Let $C' \stackrel{\mathrm{def}}{=} \textsc{Request}\ \overline{cC_1}\langle m_1\rangle.\overline{cC_2}\langle m_2\rangle.C''$ be the user, and $P_1' \stackrel{\mathrm{def}}{=} \textsc{Provide}\ cC_1(n).P_1''$ and $P_2' \stackrel{\mathrm{def}}{=} \textsc{Provide}\ cC_2(n).P_2''$ two service providers. The channel m_1 is local to C and P_1; m_2 is local to C and P_2.

3.2 Connectors

We look at single connectors first, i.e. connectors for a single contract. A private interaction channel $m : iC$, the connector, is established between provider and user. The provider has an input port (called n) and the user has an output port m (by default the name of the connector). The interaction between the user

$$C'' \stackrel{\text{def}}{=} \textsc{Write}\ \overline{m}\langle a\rangle.C''' \tag{12}$$

and the provider

$$P'' \stackrel{\text{def}}{=} \textsc{Read}\ m(x).P''' \tag{13}$$

can happen if permitted by the sorted $\textsc{React}_S$-rule. Here a is a single parameter (we could have used a parameter list in the polyadic π-calculus). If multiple contracts – and, thus, multiple connectors m_i – exist, the behaviour of C can be abstracted by:

$$C'' \stackrel{\text{def}}{=} \textsc{Write}\ \overline{m_1}\langle a_1\rangle.C'' + \textsc{Write}\ \overline{m_2}\langle a_2\rangle.C'' + \tau.C'' \tag{14}$$

The computation is either a call of m_1 with value a_1 or a call of m_2 with value a_2 or an empty action τ (representing some internal computation). This is executed repeatedly. The interaction channels are scoped as follows in this example: `private` $m_1 : iC\ (C|P_1)$ and `private` $m_2 : iC\ (C|P_2)$. We could introduce a *reply* construct using the same interaction channel:

$$C'' \stackrel{\text{def}}{=} \textsc{Write}\ \overline{m}\langle a\rangle.\textsc{ReadReply}\ m(x).C'''$$

and

$$P'' \stackrel{\text{def}}{=} \textsc{Read}\ mx.\textsc{WriteReply}\ \overline{m}\langle b\rangle.P'''$$

for an operation m with a return value. We will not investigate this further, see e.g. [LAN00] for a suitable concept.

4 Semantics for Components

In this section, we will give semantics to the previous constructs. It shall make some of the notions introduced only intuitively in the previous sections more precise. We will interpret entities in state-based structures, called objects. Constraints are embedded into a modal state-based logic over these structures.

4.1 Semantics of Actions and Operations

The OMG *Request For Proposals on Action Semantics for the UML* [Gro98] requests semantics for actions essentially for two reasons:

- Formality: system analysis and proof of correctness are possible within an abstract and formal framework.

– Abstractness: enables interoperability and platform-independence.

Object behaviour is essentially based on state transitions expressed by actions. These actions shall be formalised in a denotational style and abstracted by pre- and postconditions. Using dynamic logic as a framework allows us to establish a development calculus centered around proof rules.

A *labelled transition system* consists of a set of states $State$, a set of transition labels $Tran$ and a relation on $State \times Tran \times State$. One state shall be distinguished as an initial state. Behaviour is modelled as a traversal of the transition system. A *state machine* executes the actions associated with the transitions. *Objects* shall be state machines with structured states and relations on states which form functions. An object state is a binding between state variables and their values. Each transition label denotes an operation definition, characterised by a signature and an implementation consisting of actions. Operations are interpreted by functions on states,possibly producing a result value:

$$State \times S_1 \times \ldots \times S_n \rightarrow State \times S_0 \tag{15}$$

The S_i are value domains. Projections onto the resulting pair select the appropriate component of an operation instance. An **object** is an algebraic structure that has the following properties:

– a carrier set S for each sort s,
– a function of type $S_1 \times \ldots \times S_n \rightarrow S$ for each attribute with signature $s_1 \times \ldots \times s_n \rightarrow s$,
– a carrier set $State$ for sort *state* containing total assignments $\mathcal{P}Id \rightarrow F$ if F is the set of functions that match the signatures of functions,
– a function of type $(State \times S_1 \times \ldots \times S_m) \rightarrow (State \times S)$ for each operation symbol with the corresponding signature.

Objects are hidden algebras for a signature with state [Pah97, Gog99, GM00]. An object consists of a **state** that maintains bindings between identifiers and functions, **attributes**, i.e. functions of the state which allow observations of the state, and **operations**, i.e. state transitions which modify the state by modifying its attributes. *Actions* can be *interpreted* by transitions on objects:

– The *assignment* modifies the state binding between state variables and values, i.e. assigns a new value to the state variable.
– The *operation call* invokes a local operation of the object, which might result in a new state.
– The *sequence* is executed by executing the second action in the resulting state of the first action execution.

Entities that we have used in the definition of contract and connector channels also have denotations in this semantics structure. The semantics of a component C is an object. That of a channel z is an operation of an object: an output $\overline{x}\langle y \rangle$ is an interaction (or send-activity) which invokes an operation at the other object, an input $x(y)$ is an operation invocation at the current object itself.

4.2 Abstraction of Actions and Operations

The *Action Semantics RFP* [Gro98] requests a framework to carry out formal analysis and proofs of correctness. *Modal logic* – a logic with a notion of state or time – is a suitable framework for reasoning about concurrent and reactive systems. In order to express abstract constraints on states and transitions (operations), we propose an extension of the OCL-notion of pre- and postconditions based on a simplified *dynamic logic*:

$$\begin{aligned}&opName(p_1 : t_1, \ldots , p_n : t_n) : rt\\ &\quad pre : F\\ &\quad post : G\end{aligned} \tag{16}$$

where F and G are arbitrary first-order formulas. We have simplified the modal calculus in order to avoid reasoning about nested modal combinators in the context of UML. As usual, the name *self* can be used, and values of state variables in the previous state can be accessed by the @*pre*-postfix. Pre- and postconditions are *observations on states*, they describe properties of states. Additionally, using the reserved name *result* we can specify the return value of the operation in a postcondition. The precondition F corresponds to the guard from transition descriptions in statechart diagrams.

The *semantics of constraints* shall be given in form of a *satisfaction* relation. *State properties* can be specified using **equations** based on expressions involving state variables and attributes. An equation $x = y$ is *satisfied* in a state if the interpretations of both sides are the same (x and y are expressions consisting of values, operators, and operation applications). The **implication** $f \to g$ holds iff f and not g holds. The **formula** $\Box(F \to [P]G)$ shall abbreviate the pre- and postcondition specification in (16) with $P \equiv opName(x_1, \ldots , x_n)$. The formula $\Box([P]G)$ holds iff the execution of P terminates in a state satisfying G. The UML definition assumes a non-partial (terminating) behaviour[2].

An **abstract specification** or **abstract interface** consists of a signature and well-formed axioms in dynamic logic $S = \langle \Sigma, E\rangle$ describing operations on objects in abstract terms. An axiomatic specification and the interpretation of elements in semantic structures gives rise to a *notion of model classes*, here the class of objects which satisfy some specification. The semantics of a specification S is a model class $Mod(S)$.

Inference systems for modal logics are documented in the literature [Sti92, Lam94]. Particularly important are *rules* which allow us to *compose and decompose action or operation specifications*. The following shall illustrate how to use proof rules and how to derive development support from it. An operation specification $\Box([P]F)$ can be described as a decomposition $\Box([P1; P2]F)$ where $\Box([P1; P2]F)$ is defined as $\Box([P2][P1]F)$.

$$\Box([P; Q]F) \Leftrightarrow \Box([P]\Box([Q]F)) \tag{17}$$

[2] If we would extend our approach to partial correctness (the above is a total correctness assertion), we would add a *liberal* variant of the formula involving an undefinedness predicate, see [LB99].

The previous rule is an example which shows us how to compose to operation specifications and how to introduce the sequence operator. Another rule related to the introduction of the sequence operator is the transitivity rule:

$$\Box(F_1 \to [P]F_2) \land \Box(F_2 \to [Q]F_3) \Rightarrow \Box(F_1 \to [P;Q]F_3) \tag{18}$$

5 Reasoning about Composition and Contracts

Elements of an inference system have already been introduced. Now, we will extend our formal framework in order to allow more reasoning about component composition and contracts. Two main issues will be addressed. Firstly, we generalise the refinement relation into a general abstraction/implementation relation. The relations play an important role in the definition of a flexible composition mechanism. Secondly, we address the so-called horizontal composition property.

UML offers several ways of relating classes statically. The main relationships are association, generalisation, dependency and refinement. We will concentrate on the *abstraction* relations, in particular *refinement* and *implementation*. A refinement relates two elements describing the same on different levels of abstraction. The name indicates that structure, knowledge, or properties are added in a refinement. It also suggests that properties of the more abstract description should be preserved. The refinement is particulary important since it can form the basis of a formally supported stepwise development method. Refinement is defined for the UML [Gro99] as the description of something on a lower level of abstraction. Lowering the level of abstraction means to make a description more concrete by adding details. These can be details about the underlying structure or can be details about the behaviour of operations. Certainly, we expect that *properties* specified on the abstract level are *preserved* in a refinement.

5.1 Implementing and Refining Abstract Specifications

We can distinguish two dimensions of development: horizontal and vertical development. Horizontal development refers to the composition of packages, vertical development means using refinement, realisation, implementation or any other construct which lowers the level of abstraction.

Implementation captures the idea of making design decisions, i.e. lowering the level of abstraction. Formally, this can be expressed by model class inclusion

$$S \overset{I}{\rightsquigarrow} S' \quad \text{iff} \quad Mod(S') \subseteq Mod(S) \tag{19}$$

for two specifications S and S'. The inverse of an implementation is an **abstraction**. We have introduced a **correctness** condition on components $C = \langle Imp, Class, Exp \rangle$ earlier on: Exp is an abstraction of $Class$. We can now formalise this notion.

$$sig(Exp) \subseteq sig(Class) \ \land \ Exp \overset{I}{\rightsquigarrow} Class \tag{20}$$

We require that only a subset of attributes and operations is exported (or visible) and that *Exp* abstracts *Class*, or *Class* implements *Exp*.

The **refinement** is a constructive support for the implementation only based on pre- and postconditions: preconditions are weakened and/or postconditions are strengthened. *Implication* is the formal basis of the refinement [Lam96], relating behavioural abstractions of operations in terms of pre- and postconditions. We reformulate the refinement in terms of the box-operator notation:

$$\frac{F \to F',\ \Box(F \to [P]G),\ G' \to G}{\Box(F' \to [P]G')} \tag{21}$$

Proving an implication between pre- or postconditions is usually less complex compared to proper modal formulas. In modal logics, the rule above is known as the *consequence rule*. In refinement calculi, it is known as a combination of the *weaken precondition*- and *strengthen postcondition*-rule [Bac88, Mor90, Mor94]. Ideas from refinement calculi can be used to provide a *constructive calculus of derivations*.The refinement here is only basic definition and needs to be accompanied by an appropriate calculus to *support the modelling process*.

We need to distinguish two forms of implementation and refinement: $\overset{I}{\underset{op}{\leadsto}}$ and $\overset{R}{\underset{op}{\leadsto}}$ are relations between operations, and $\overset{I}{\leadsto}$ and $\overset{R}{\leadsto}$ are relations between abstract specifications. The derivation rule above defines a notion of *refinement* for *operations*. *Refinement between interfaces* shall now be addressed. Ideally, we would find a notion which is *compositional*; a notion which defines the refinement of interfaces based on the refinement of its constituent operation specifications. That requires that invariants and other constraints can be dealt with as part of operation specifications. Implementation and refinement are *compositional*:

$$S \leadsto S' \ \text{ iff } \ P \leadsto P' \ \text{ for all constituent procedures } P \tag{22}$$

A corresponding definition can be found in [Pah00].

We can show that our refinement is a close approximation to an inclusion of model classes for a concrete and an abstract specification, see [Pah00] for details. The following theorem formalises this important property. It shows that the implementation generalises the refinement.

Proposition 1. *Refinement implies implementation*

$$S \overset{R}{\leadsto} S' \ \textit{implies} \ S \overset{I}{\leadsto} S' \tag{23}$$

for two specifications S and S'.

This proposition is based on some assumptions. Invariants *inv* are added to pre- and postconditions $\Box(F \wedge inv \to [P]\ G \wedge inv)$. Attribute definitions do not change and can, thus, be specified as invariants. These assumptions do not restrict the approach, however, they simplify proofs. The implication $S_1 \overset{I}{\leadsto} S_2$ *implies* $S_1 \overset{R}{\leadsto} S_2$ can *not* be established – which means that the notions of refinement and model class inclusion are different – if states are involved which are not reachable from any initial state or formulas specify states which are not satisfiable.

We see implementation as a fundamental relation since it captures property-preservation. Property-preservation can also be the foundation of the various UML abstraction relations. These relations are important for the *development of software components*. This can include the *vertical implementation*, but also the *horizontal composition of components* where the implementation (or the refinement) can play the role of a *correctness* criterion – e.g. the glue between a service provider and a service user. The refinement has already been used to define the notion of *contracts* between provider and user. A service provided needs to satisfy requirements formulated by a service user, i.e. the provided service refines or implements – we can generalise the definition – the user's import requirements, see [Pah00]. These results can be used in the definition of an extension and improvement of the UML package concept.

An example shall illustrate a contract between two components matched by strengthening postconditions. The refinement is the tool to prove the correctness constraint for semantical matching. The contract might specify a postcondition

$$\phi = \ \texttt{update()}$$

for an operation `myModifyDoc` and a library version `modifyDoc` might provide

$$\phi' = \ \texttt{if parseDoc() then update() else nothing()}$$

as the postcondition. We get for the corresponding interfaces

$$Imp(\texttt{myModifyDoc}) \overset{I}{\rightsquigarrow} Exp(\texttt{modifyDoc})$$

since

$$post(\texttt{myModifyDoc}) \rightarrow post(\texttt{modifyDoc})$$

This can be proven as follows: ϕ' implies `parseDoc()` $\rightarrow$ `update()` using modus ponens: $(A \rightarrow B) \wedge A$ implies B – A is here the precondition. ϕ follows since $A \wedge B$ implies A. With Proposition 1 we can deduce the more general implementation from the refinement. The refinement is used here as a proof tool to prove the correctness of a component composition with respect to a contract.

5.2 Composition of Components

A **composition** between two components can be formulated **syntactically** by

$$compose\ U = \langle Imp, Class, Exp \rangle\ with\ P = \langle Imp', Class', Exp' \rangle\ via\ \rho \tag{24}$$

expressing that a component U uses services provided by P. A contract C can be derived from the abstract interface Imp which is the requirements specification of the service user. The **correctness constraint** for composition based on a contract C with provider P and user U is the following:

$$\rho(sig(Imp(U))) = sig(Exp'(P)) \text{ and } Imp(U) \overset{I}{\rightsquigarrow} Exp'(P)_{|\rho} \tag{25}$$

This criterion is based on syntactical and semantical properties of the abstract interfaces of the components involved. With $Exp'(P)_{|\rho}$ we denote the restriction of P to elements in the range of $\rho(sig(Imp))$. Technically, the composition results in the establishment of an interaction infrastructure between components so that services requested by a user can actually be accessed. The contract rule $C|P \longrightarrow \texttt{private}\ m\!:\!iC\ (C'|P'\{^m/_n\})$ iff $m \leadsto n$ formalises the execution of the composition and creation of the connector. A component can import from several library components. Each import is defined in a separate contract and results in a separate connector. The **composition** of components can be defined **semantically** by

$$\begin{aligned} compose\ \langle Imp, Class, Exp\rangle\ with\ \langle Imp', Class', Exp'\rangle\ via\ \rho\ :=& \\ \langle Imp', Class|Class', Exp\rangle& \end{aligned} \tag{26}$$

where ρ shall be a signature morphism $\rho : sig(Imp) \rightarrow sig(Exp')$. The correctness constraint for composition needs to be applied. The composed component forms again a component with the parallel composition $Class|Class'$ of the component implementations at its core. The new import is that of the provider and the export is that of the user.

In the π-calculus, the interaction between two processes in a parallel composition is considered as not being observable from the outside. We have followed this idea, and defined the composition of two components as a new component, which encapsulates and hides the parallel composition of its interacting objects inside.

The notion of implementation shall now be extended to components. The component $I = \langle Imp', Class', Exp'\rangle$ is called an **implementation** of the component $S = \langle Imp, Class, Exp\rangle$

$$\langle Imp, Class, Exp\rangle \leadsto_C \langle Imp', Class', Exp'\rangle \tag{27}$$

if both $\langle Imp, Class, Exp\rangle$ and $\langle Imp', Class', Exp'\rangle$ are correct components, and $Exp \leadsto Exp'$ and $Imp' \leadsto Imp$ are implementations of abstract interfaces.

Finally, we formulate the central result of this subsection, the **horizontal composition property**. Let us assume that a given system is decomposed into components. Each of these components can be realised by hand or by re-using appropriate components for implementation. If, on the level of more abstract specifications, a system of components is correctly composed, then this correctness shall be preserved on the level of implementations of these abstract specifications. The correctness of compositions on the horizontal level shall be preserved. Thus, this property is called the *horizontal composition property*.

Let C_1 and C_2 be components. $C_1(C_2)$ – or *compose* C_1 *with* C_2 – expresses, that C_2 is the actual parameter of – or the provider for – C_1. The composition $C_1(C_2)$ is correct, if C_2 satisfies the requirements of C_1. Let us now assume two more abstract components S_1 and S_2, and two less abstract (implemented) components I_1 and I_2. If the composition of abstract components $S_1(S_2)$ is correct, then the horizontal composition property requires that, if $S_1 \leadsto I_1$ and $S_2 \leadsto I_2$ are implementations, then $I_1(I_2)$ is a correct composition.

The following proposition formalises the horizontal composition property.

Proposition 2. *Assume the following conditions:*

- *Correct components* $S_1 = \langle Imp_S^1, Exp_S^1 \rangle$, $S_2 = \langle Imp_S^2, Exp_S^2 \rangle$ *(both abstract), and* $I_1 = \langle Imp_I^1, Exp_I^1 \rangle$, $I_2 = \langle Imp_I^2, Exp_I^2 \rangle$ *(both implementations).*
- *Implementations between* S_1 *and* I_1, *i.e.* $\langle Imp_S^1, Exp_S^1 \rangle \rightsquigarrow_C \langle Imp_I^1, Exp_I^1 \rangle$, *and between* S_2 *and* I_2, *i.e.* $\langle Imp_S^2, Exp_S^2 \rangle \rightsquigarrow_C \langle Imp_I^2, Exp_I^2 \rangle$.
- *A signature morphism* $\rho : sig(Imp_S^1) \rightarrow sig(Exp_S^2)$.
- *compose* $\langle Imp_S^1, Exp_S^1 \rangle$ *with* $\langle Imp_S^2, Exp_S^2 \rangle$ *via* ρ *is correct.*

Then

- $\rho' : sig(Imp_I^1) \rightarrow sig(Exp_I^2)$ *with* $\rho' := \rho$ *is a signature morphism.*
- *compose* $\langle Imp_I^1, Exp_I^1 \rangle$ *with* $\langle Imp_I^2, Exp_I^2 \rangle$ *via* ρ' *is correct.*
- *The component implementation relation holds between the compositions:*

$$\begin{array}{ll} & compose\ \langle Imp_S^1, Exp_S^1 \rangle\ with\ \langle Imp_S^2, Exp_S^2 \rangle\ via\ \rho \\ \rightsquigarrow_C & \\ & compose\ \langle Imp_I^1, Exp_I^1 \rangle\ with\ \langle Imp_I^2, Exp_I^2 \rangle\ via\ \rho' \end{array}$$

A proof for Proposition 2 can be found in [Pah00].

6 Related Work

Catalysis is a development approach building up on the UML incorporating formal aspects such as the pre- and postcondition technique [DW98]. Catalysis uses ideas from formal languages such as OBJ, CLEAR or EML. The concept of the connector that we have used here is motivated by the Catalysis approach. There, connectors allow the communication between ports of two objects. A connector defines a protocol between the ports. Several other authors also address contracts based on pre- and postconditions for the UML, including [AF99] and [BPP99]. The combination of the pre- and postcondition technique and refinement calculi is explored in e.g. [BS97] or [BPP99].

KobrA is another approach which combines the UML with the component paradigm. The basic structuring mechanism is the *is-component-of* hierarchy, forming a tree-structured hierarchy of components, i.e. sub-components. Each component is described by a suite of UML diagrams. A component consists of a specification (an abstract export interface) and a realisation.

In earlier work [Pah00], we have used a variant of the λ-calculus to define a single import using reduction as the mechanism for import actualisation. The variant is called $\lambda\pi$-calculus, and has been developed by L. Feijs [Fei89]. The calculus has been used to define module parameterisation for the state-based specification language COLD [FJ92]. This λ-calculus can be interpreted in semantic structures, as we have done it here for the constrained interaction calculus. We have used a π-calculus variant here, because it offers multiple (concurrent) connections and it allows to model two layers: contracts and connectors.

A composition language for components which is also based on the π-calculus is presented in [LAN00]. A variation of the π-calculus is used to realise a composition language which supports various forms of components, and, thus, various composition mechanisms.

Walker [Wal95] introduces object intercommunication into the π-calculus. The difference between our approach and Walker's approach is that in our approach the user is the active entity which initiates the estabishment of the connections. In Walker's formalisation, the service provider also provides the communication channels. The service user acquires the contract channel, then aquires the interaction channels via the appropriate contract channels and finally uses the interaction channels to invoke methods of the service providers.

7 Conclusions

A composition mechanism for component-based software development has been developed and illustrated in the UML-context. Components are specifications with abstract import and export interfaces which encapsulate and abstract (possibly complex) objects. We have addressed the *abstraction of behaviour* and formalised *notions of refinement and implementation*. The internal behaviour of operations is specified by actions. Pre- and postconditions specify the abstract behaviour of operations. Their specifications can be related through pre- and postcondition-based implementation and refinement relations, whereby the refinement can be used to prove implementations. These relations capture the idea of property preservation.

The basis of component composition is interaction. One component interacts with another component if it requires services of the latter. The user's requirements – or expectations how the required services will work – are the basis on which a contract between both parties is formulated. These constraints formulated by the contract are based on the refinement relation.

The essential result here is that the pre- and postcondition technique extended to a refinement approach solves two problems. Firstly, the internal behaviour of operations on objects can be abstracted by pre- and postconditions, and the refinement relation based on this can form the foundation of a stepwise development calculus. Secondly, pre- and postconditions formalise conditions necessary to constrain interactions between objects. The refinement is the tool to prove these constraints. We have addressed both the interaction infrastructure and the constraint language necessary to control the composition. The result is a composition approach which allows re-use of existing components and reasoning about composition contracts.

One future research focus concerns the maintainability of systems and the evolution of contracts in these systems. Changing requirements make it necessary to re-negotiate contracts, i.e. to either adapt the existing partners to the new requirements or to involve other components. Assessing the suitablity of contracts in an evolving environment might be supported by the bisimilarity concept of the π-calculus. Another direction in which the adaptability of re-usable components could be investigated is the deployment of matching approaches, as presented in [ZW95] for the Larch language family.

A further direction concerns the extension of the constraint language. In *reactive systems*, *liveness* is the second important property besides *safety* (which has been addressed only so far). Liveness can be expressed using the eventually-operator: $\diamond([P]F)$ expresses that by executing P a state described by F will eventually be reached. Formally, the eventually-operator can be defined via the always operator: $\diamond([P]F) := \neg\Box([P]\neg F)$. A modal logic framework was chosen in order to be able to extend the approach to reactive systems modelling.

References

[AF99] L.F. Andrade and J.L. Fiadero. Interconnecting Objects via Contracts. In R. France and B. Rumpe, editors, *Proceedings 2nd Int. Conference UML'99 - The Unified Modeling Language*. Springer Verlag, LNCS 1723, 1999.

[Bac88] R.J.R. Back. A Calculus of Refinements for Program Derivations. *Acta Informatica*, 25:593–624, 1988.

[BPP99] R.-J. Back, L. Petre, and I.P. Paltor. Analysing UML Use Cases as Contracts. In R. France and B. Rumpe, editors, *Proceedings 2nd Int. Conference UML'99 - The Unified Modeling Language*. Springer Verlag, LNCS 1723, 1999.

[BS97] M. Büchi and E. Sekerinski. Formal Methods for Component Software: The Refinement Calculus Perspective. In *Proceedings 2nd International Workshop on Component-Oriented Programming WCOP '97*. Turku Center for Computer Science, General Publication No.5-97, Turku University, Finland, 1997.

[DW98] D. D'Souza and A.C. Wills. *Objects, Components and Frameworks in UML: the Catalysis approach*. Addison-Wesley, 1998.

[Fei89] L.M.G. Feijs. The calculus $\lambda\pi$. In *Algebraic Methods: Theory, Tools and Applications*, pages 307–328. Springer-Verlag, 1989.

[FJ92] L.M.G Feijs and H.B.M Jonkers. *Formal Specification and Design*. Cambridge University Press, 1992.

[GM00] J. Goguen and G. Malcolm. A Hidden Agenda. *Theoretical Computer Science*, 2000. Special Issue on Algebraic Engineering – to appear.

[Gog99] J. Goguen. Hidden Algebra for Software Engineering. In *Proceedings Conference on Discrete Mathematics and Theoretical Computer Science, Auckland, New Zealand*, pages 35–59. Australian Computer Science Communications, Volume 21, Number 3, 1999.

[Gro98] Object Management Group. Action Semantics for the UML – RFP, 1998. http://www.omg.org/technology/uml.

[Gro99] Object Management Group. UML 1.3 Specification, 1999. http://www.omg.org/technology/uml.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[Lam96] L. Lamport. Refinement in State-based Formalisms. SRC Technical Note 1996-001, Digital Equipment Corporation, Systems Research Center, 1996.

[Lam99] L. Lamport. Specifying Concurrent Systems with TLA^+. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. IOS Press, Amsterdam, 1999.

[LAN00] M. Lumpe, F. Achermann, and O. Nierstrasz. A Formal Language for Composition. In G.T. Leavens and M. Sitamaran, editors, *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[LB99] G.T. Leavens and A.L. Baker. Enhancing the Pre- and Postcondition Technique for More Expressive Specifications. In R. France and B. Rumpe, editors, *Proceedings 2nd Int. Conference UML'99 - The Unified Modeling Language*. Springer Verlag, LNCS 1723, 1999.

[LS00] G.T. Leavens and M. Sitamaran. *Foundations of Component-Based Systems*. Cambridge University Press, 2000.

[Mey92] Bertrand Meyer. Applying Design by Contract. *Computer*, pages 40–51, October 1992.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[Mor90] J.M. Morris. Programs from Specifications. In E.D. Dijkstra, editor, *Formal Development of Programs and Proofs*. Addison-Wesley, 1990.

[Mor94] C. Morgan. *Programming from Specification 2e*. Addison-Wesley, 1994.

[MPW92a] R. Milner, J. Parrow, and D. Walker. A calculus of Mobile Processes, part I. *Information and Computation*, 100(1):1–40, 1992.

[MPW92b] R. Milner, J. Parrow, and D. Walker. A calculus of Mobile Processes, part II. *Information and Computation*, 100(1):41–77, 1992.

[Nor98] E.K. Nordhagen. *A Computational Framework for Verifying Object Component Substitutability*. PhD thesis, University of Oslo, November 1998.

[Pah97] C. Pahl. A Model for Dynamic State-based Systems. In A.S. Evans and D.J. Duke, editors, *Proc. Northern Formal Methods Workshop, Sept.'96, Bradford, UK*. Springer-Verlag, 1997.

[Pah00] C. Pahl. Modal Logics for Reasoning about Object-based Component Composition. In *Proc. 4rd Irish Workshop on Formal Methods, July 2000, Maynooth, Ireland*. BCS, eWiC series, 2000. (to appear).

[pUG99] precise UML Group. Response to UML 2.0 Request for Information, 1999. http://www.cs.york.ac.uk/puml.

[Sti92] C. Stirling. Modal and Temporal Logics. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. II*, pages 477–563. Oxford University Press, 1992.

[Wal95] D. Walker. Objects in the π-Calculus. *Information and Computation*, 115:253–271, 1995.

[Wec97] W. Weck. Inheritance Using Contracts & Object Composition. In *Proceedings 2nd International Workshop on Component-Oriented Programming WCOP '97*. Turku Center for Computer Science, General Publication No.5-97, Turku University, Finland, 1997.

[ZW95] A. Moormann Zaremski and J.M. Wing. Specification Matching of Software Components. In Gail E. Kaiser, editor, *Proc. ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 6–17. ACM Software Engineering Notes 20(4), October 1995.