Contract Oriented Development of Component Software

Chapter · January 2004

DOI: 10.1007/1-4020-8141-3_28

CITATIONS
27

READS
116

3 authors, including:

Zhiming Liu - 刘志明
Northwestern Polytechnical University
215 PUBLICATIONS **2,264** CITATIONS

SEE PROFILE

Xiaoshan Li
University of Macau
88 PUBLICATIONS **1,392** CITATIONS

SEE PROFILE

CONTRACT ORIENTED DEVELOPMENT OF COMPONENT SOFTWARE*

Zhiming Liu[1,3], He Jifeng[1,4], and Xiaoshan Li[2]

[1]*International Institute for Software Technology, The United Nations University, Macao SAR, China*
lzm@iist.unu.edu, hjf@iist.unu.edu

[2]*Faculty of Science and Technology. The University of Macau, Macau*
xsl@umac.mo

[3]*Department of Computer Science, The University of Leicester, U.K.*

[4]*East China Normal University, Shanghai, China*

Abstract We present a model for component software. We describe how components are specified at the interface level, design level and how they are composed. From its external view, a component consists a set of interfaces, *provided* to or *required* from its environment. From its internal view, a component is an executable code that can be coupled with other components via its interfaces. The developer has to ensure that the specification of a component is met by its design and implementation. We also combine component-based and object-oriented techniques in component-based software development.

Keywords: Component, Contract, Interface, Object-Orientation, Refinement

1 Introduction

Using components to build and maintain software systems is not a new idea. However, it is today's growing complexity of these systems that forces us to turn this idea into practice [Szyperski, 2002, Cheesman and Daniels, 2001, Heineman and Councill, 2001]. While component technologies such as COM, CORBA, and Enterprise JavaBeans are widely used, there is so far no agreement on standard technologies for designing and creating components, nor on methods for composing them. Finding appropriate formal approaches for specifying components, the architectures for composing them, and the methods for component-based software construction, is correspondingly challenging. In this paper, we consider a contract-oriented approach to the specification, design and composition of components. Component specification is

* This work is partly supported by the research grant 02104 MoE and the 973 project 2002CB312000 of MoST of P.R.China.

essential as it is impossible to manage change, substitution and composition of components if components have not been properly specified.

When we specify a component, it is important to separate different views about the component. From its user's (i.e. external) point of view, a component P consists a set of *provided services* [Szyperski, 2002]. The syntactic specification of the provided services is described by an interface, defining the operations that the component provides with their signatures are. This is also called the *syntactic specification* of a component, such as COM and CORBA that use IDL, and JavaBeans that uses Java Programming Language to specify component interfaces. Such a syntactic specification of a component does not provide any information about the effect, i.e. the *functionality* of invoking an operation of a component or the *behavior,* i.e. the temporal order of the interface operations, of the component.

For the functional specification of the operations in an interface, it is however necessary to know the conceptual state of the component. Consequently, the interface specification contains a so-called *information model* [Cheesman and Daniels, 2001, Filipe, 2002]. In the context of such a model, we specify an operation op by a *design* $p(x) \vdash Q(x, x')$ in Hoare and He's Unifying Theories of Programming (UTP) [Hoare and He, 1998] that is seen as a *contract* between the component and its client [Cheesman and Daniels, 2001, Heineman and Council, 2001]. This definition of a *contract* also agrees with that of [Meyer, 1992, Meyer, 1997]. To use the service op, a client has to ensure the pre-condition $p(x)$, and when this is true the component must guarantee the post-condition Q. We then define a *contract* of an interface by associating the interface with a set of *features* that we will call *fields* and assigning each a design $MSpec(op)$ to each interface operation op. The types of the fields are given in a *data/class model.*

The contract for the provided interface of a component allows the user to check whether the component provides the services required by other components in the system, without the need to know the design and implementation of the component. It also commits (or requires) the designers of the component who have to design the component's provided services. A designer of the component under consideration *(CuC)* may decide to use services provided by other components. These services are called *required services* [Szyperski, 2002] of CuC. Components that provide the required services of CuC can be built by another team or bought as a component-off-the-self (COTS). To use a component to assemble a system, one needs to know the specifications of both its provided and required services.

We will specify the design of a component by giving each operation op in the provided interface a program specification text $MImpl(op)$ in the object-oriented specification language (OOL) defined in [Liu et al., 2004c]. In $MImpl(op)$, calls to operations in a required interface are allowed. With the *refinement calculus of object-oriented designs* (RCOOD) in OOL [He et al., 2002, Liu et al., 2004b], we can verify whether $MImpl(op)$ refines the specification of op given in a contract of the provided interface. The *verifier* of a component needs to know the contracts of the provided interfaces, the contracts of the required interfaces, and the specification text for each operation op of the provided interface. We can thus understand a component as a relation between contracts of the required interfaces and contracts of the provided interface: given a con-

tract for each required interface, we can calculate a design of an ***op*** from ***MImpl(op)*** and check whether it conforms to the specification ***MSpec(op)*** defined by the contract of the provided interface. A design of a component can be further refined into an implementation by refining the data/class model and then operation specifications ***MImpl(op)***.

A component assumes an architectural context defined by its interfaces. We *connect* or *compose* two components P_1 and P_2 by linking the operations in the provided interface of one component to the matching operations of a required interface of another. For this, we have to check whether the provided interface of one component P_1 contains the operations of a required interface of another component P_2, and whether the contract of the provided interface of P_1 meets the contract of the required interface of P_2. If P_1 and P_2 match well, the composition $P_1||P_2$ forms another component. The provided interface of $P_1||P_2$ is the *merge* of the provided interfaces of P_1 and P_2. The required interfaces of $P_1||P_2$ are the union of required interfaces of P_1 and P_2, excluding *(by hiding)* the matched interfaces of P_1 and P_2. For defining composition, interfaces can be *hidden* and *renamed.*

A component is also replaceable, meaning that the developer can replace one component with another, may be *better,* as long as the new one provides and requests the same services. An component is better than another if it can provide more services, i.e. the contracts for its provided interfaces refine those of the other, with the same required services. Component replaceability is based on the notion of *component refinement.*

As a starting point, we only deal with the functional/service specification of components and deals with functional compatibility. In Section 5, we give a discussion on the alternative ways to deal with behavioral specification and compatibility. This paper is not about development processes either. However, we will give an overview on how the model can be used in a component-based development together with an object-oriented implementation.

2 Interfaces

An *interface I* is a set of *operation* (or *method*) *signatures,* of the form op(**in** : $\underline{U}\ \underline{x}$, **out** : $\underline{V}\ \underline{y}$, **inout** : $\underline{W}\ \underline{z}$), where ***op*** is the *name* of the operation, $\underline{x}$ are the *value* parameters of types $\underline{U}$, $\underline{y}$ the *result* parameters of types $\underline{V}$, and $\underline{z}$ the value-result parameters of types $\underline{W}$. An interface can be specified as a family of operation signatures in the following format:

Interface *I* {

Method : op_1(**in** : $\underline{U}_1\ \underline{x}_1$, **out** : $\underline{V}_1\ \underline{y}_1$, **inout** : $\underline{W}_1\ \underline{z}_1$);
...;
op_k(**in** : $\underline{U}_k\ \underline{x}_k$, **out** : $\underline{V}_k\ \underline{y}_k$, **inout** : $\underline{W}_1\ \underline{z}_k$)

}

Figure 1 shows the ParcelCall system in [Filipe, 2002] that has three main components:

- a *Mobile Logistic Server* (MLS): is an exchange point or a transport unit (container, trailer, freight wagon, etc). It always knows its current location via the GPS satellite positioning system.

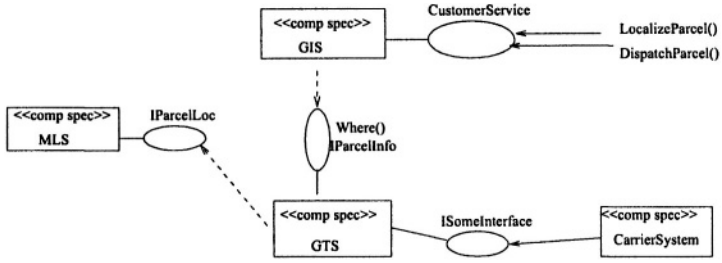

Figure 1. **Architecture of ParcelCall**

- a *Goods Tracing Server* (GTS): keeps track of all the parcels registered in the ParcelCall system. GTS is also the component which is integrated with the legacy systems of transport or logistic companies.
- a *Goods Information Server* (GIS): is the component which interacts with the customers and provides the authorized customers the current location of their parcel, keeps them informed in case of delivery delays, etc.

In Figure 1, UML notation for interfaces and components is used.The provided interface of GIS component will establish communication with the customer: for instance a customer can request for finding the current location of a parcel via *LocateParcel.* The specification of this interface can be described as follows, where we use $\mathbb{P}\mathcal{PS}$ to denote the powerset of a set $\mathcal{S}$

Interface *CustomerService* {
Method : *LocateParcel*(**in** : *PName pId, CName sId*), **out** : *Position location*);
DispatchParcel(**in** : *PName pId, CName sId*)}

Merge interfaces

It is often the case that there are a number of components, each providing a part of the operations in the required interface of another component. We thus need to *merge* these components to provide one single interface to match the interface required by the other component.

Two interfaces I_1 and I_2 are *composable* provided that every operation name in both I_1 and I_2 must be assigned the same signature. This condition is not too restrictive as to use a component designed for an application in another or specialize a generic component for a special application, renaming or adding a *connector* component [Allen and Garlan, 1997, Selic, 1998] can be used to *customize* the component.

DEFINITION 1 Let $\{I_k : \mid k \in K\}$ be a finite family of composable interfaces. Their **merge** $\uplus_{k\in K} I_k$ is defined by $\uplus_{k\in K} I_k \stackrel{def}{=} \cup_{k\in K} I_k$.

3 Contracts

Only a *syntactic specification* of its interface is not enough for the use or the design of a component. We also need to specify the effect, i.e. the *functionality,* of invoking

an interface operation. This requires one to associate the interface to a *conceptual state space,* and a specification of how the states are changed by the operation under certain pre-conditions. We view such a *functional specification* of an interface as a contract between the component *client* and the component *developer.* The contract is the specification of the component that the developer has to implement. The contract is also between a user of the component and a provider of an implementation of the interface: the component has to provide the services promised by the specification *provided* that the user uses the component according to the precondition [Szyperski, 2002].

Conceptual model

To define the conceptual state space of a contract for an interface and the types for the parameters of the interface operations, we assume that a type is either a primitive data type (such as the integers, the Booleans, etc.) or a class of objects (such as a Java class). This allows our framework to support both imperative and object-oriented programming in the design of a component.

The type definitions in fact form a *conceptual class diagram* [Liu et al., 2003, Liu et al., 2004c] that is a UML class diagram in which the classes have **no** methods and the associations have **no** direction of visibility or navigation. Figure 2 is an example of a conceptual model for a library system. A UML class diagram can be specified by a class declaration section of an object-oriented program in the object-oriented specification language (OOL) developed in [Liu et al., 2004b, Liu et al., 2004c] of the form

$$class_1; class_2; \ldots; class_n$$

where each $class_i$ is of the form

Class *N* **extends** *M* {
  public $U_1\ u_1, \ldots, U_k\ u_k$;
  }

where

- *N* and *M* are distinct names of classes, and *M* is called the direct superclass of *N*.

- The **public** declaration declares the public attributes of the class and their types. Initial values are set when an object is created.

Notice that we do not declare methods for the classes as they will be given in the implementation of the component. Also, we need to declare the public fields as functional specification of operations directly refer to them. In the design model of a component, methods are introduced to realize the specification and then data encapsulation can be applied to make the fields private or protected.

Consider the simple conceptual class diagram in Figure 2 as an example. It is specified as

Class Lib $\{String\ name, String\ address\}$;
Class Publication $\{String\ isbn, String\ title, String\ author\}$;
Class Copy $\{String\ id, String\ location\}$;
Class Own $\{Lib\ lib, Publication\ p\}$; // * **Association**
Class Contain $\{Publication\ p, Copy\ c\}$; // * **Association**

Please see [Liu et al., 2003, Liu et al., 2004b] for details on the formalization of UML, and [Liu et al., 2004c] for the semantics of class declarations.

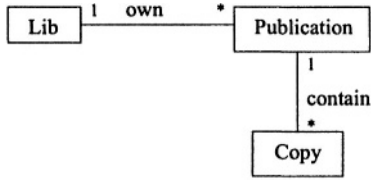

Figure 2. **A conceptual class diagram**

Contract

Given an interface I, a conceptual model M, a set A of variable declarations of the form $T\,x$ where T is either a primitive type or a class declared in M, called the type of x, we define the alphabet α as the union of sets of the variables, the input and output parameters of the operations of I.

$$in\alpha \stackrel{def}{=} A \cup \{x \in \underline{x} \cup \underline{z} \mid op(\mathbf{in} : \underline{U}\ \underline{x}, \mathbf{out} : \underline{V}\ \underline{y}, \mathbf{inout} : \underline{W}\ \underline{z}) \in I\}$$
$$out\alpha \stackrel{def}{=} A \cup \{y \in \underline{y} \cup \underline{z} \mid op(\mathbf{in} : \underline{U}\ \underline{x}, \mathbf{out} : \underline{V}\ \underline{y}, \mathbf{inout} : \underline{W}\ \underline{z}) \in I\}$$
$$out\alpha' \stackrel{def}{=} \{x' \mid x \in out\alpha\}, \quad \alpha \stackrel{def}{=} in\alpha \cup out\alpha$$

A *conceptual state* for $\langle I, M\rangle$ is a well-typed mapping from the variables α to their value spaces. It is in fact a UML object diagram of M plus values of variables in $out\alpha$ of primitive types that is a snapshot of the models consisting the current objects of the classes and links by the associations among these objects, as well as the values of variables of primitive types. A change of such a state is carried out by creating or destroying existing objects, forming or breaking links among objects, modifying values of object attributes, and changing values of some variables of primitive types. A *specification* of an operation $op(\mathbf{in} : \underline{U}\ \underline{x}, \mathbf{out} : \underline{V}\ \underline{y}, \mathbf{inout} : \underline{W}\ \underline{z})$ in an alphabet α is a *framed design* $\beta : D$ where

- β, a subset of $in\alpha$, is the *frame* of D containing the variables to be changed by D.
- $D = (b \vdash Q)$ is a design [Hoare and He, 1998] describing the behavior of the method:

$$\beta : (b \vdash Q) \stackrel{def}{=} b \wedge ok \Rightarrow Q \wedge ok' \wedge \wedge_{x \in in\alpha \setminus \beta}(x' = x)$$

Predicate b is the assumption on the variables and input parameters which the method can rely on when it is activated, while predicate Q *is* the commitment which must be true when the execution terminates. ok and ok' are used to describe the termination behavior of op. We will omit the frames in the examples by assuming that a design only changes those variables whose primed versions appear are mentioned.

The variables of $in\alpha$ are used to record the values of the variables in A and input parameters $\underline{x}$ on the activation of op, and the variables of $out\alpha$ the values of the corresponding variables and outgoing parameters $\underline{y}$ on the termination of a method. For the conceptual model in Figure 2, let

$$A \stackrel{def}{=} \{\mathbb{P}Publication\ P, \mathbb{P}Copy\ Cp, \mathbb{P}Contain\ con, Lib\ lib, \mathbb{P}Own\ own\}$$

where $\mathbb{P}S$ is the power set of S. The operation *RecordCopy*() that records a new copy of a given publication can be specified as

$$\begin{array}{lll} \{con, Cp\}: & & \exists p \in P.p.isbn = pid \wedge < lib, p > \in own \wedge \neg \exists c \in Cp.c.id = cid \vdash \\ & & \exists c' \in Cp'.c'.id = cid \wedge \exists p \in P.p.isbn = pid \\ & \wedge & con' = con \cup \{< p, c' >\} \end{array}$$

DEFINITION 2 A **contract** is a tuple $C = (I, M, A, MSpec, Init)$ where I is an interface, M is a conceptual model, A is a set of variables, called the *fields* of C, whose types are either declared in **M** or as primitive types, and *MSpec* a function that maps each operation of I to a specification, and *Init* an initial condition that defines some values to fields as their initial values.

If no field is of an object type, we will omit the conceptual model from the specification of a contract.

In modular programming, a primitive contract is a specification of a module that defines the behavior of the operations in its interface. However, later we will see that contracts can be *merged* to form another contract and this corresponds to the merge of a number of modules. In object-orient programming, a primitive contract specifies an initialized class, i.e. an object, whose public methods are operations in the interface. This class *wraps* the classes in the conceptual model M, and provides the interface operations to the environment. In the Java-like OOL [Liu et al., 2004c], such a contract can be specified as

Interface I **{Meth** : $\{m() \mid m() \in I\}$**};**
M; // * **class declarations for the conceptual model**
Class C implements I **{Attr** : $A = Init$**;**
Meth : $\{m()\{MSpec(m)\} \mid m \in I\}$**;**
main(){**C**.$New(x)$}
}

where **main** provides the condition *Init* when creating the new object of **C** attached to x with the initial values of the attributes in A (see Section 4.1 for this command and [?, Liu et al., 2004c] for its semantics).

Example A *contract* for interface *CustomerService* of *ParcelCall* assigns a specification to each method and can be written as follows, where *MSpec*(*op*) is given as the specification following the name *op* of each operation. We present a contract in a style such that the name of the interface is followed by the fields declarations, then the initial condition, and finally the operations with their specifications:

<< Contract >> *CS*
Interface *CustomerService*
Attr : $\mathbb{P}PName\ P$; // * set of parcel names
$\mathbb{P}CName\ S$; // * set of customer names
$CName \times PName\ owns$; // * *owns*(*s*, *p*): *s* owns *p*
$PName \longmapsto Position\ loc$; // * *loc*(p): the location of *p*
Init : $P = \varnothing \wedge S = \varnothing$;
Meth : *LocateParcel*(**in** : *PName pId*, *CName sId*, **out** : *Position location*) {
$pId \in P \wedge sId \in S \wedge owns(sId, pId) \vdash location' = loc(pId)$};
DispatchParcel(**in** : *PName pId*, *CName sId*) : {
$pId \notin P \vdash (P' = P \cup \{pId\}) \wedge (S' = S \cup \{sId\}) \wedge$
$(owns' = owns \cup \{(sId, pId)\}) \wedge (loc' = loc \cup \{pId \rightarrow (0,0)\})$
}

Merge and refinement

Contracts of interfaces can be merged only when their interfaces are composable and the specifications of the common methods are *consistent.* This merge will be used to calculate the provided and required services when components are composed.

DEFINITION 3 Contracts $(I_i, M_i, A_i, MSpec_i, Init_i), i = 1, 2,$ are **consistent** if

1 I_1 and I_2 are composable.

2 If x is declared in both A_1 and A_2, it has the same type; and $Init_1(x) = Init_2(x)$.

3 Any class name C in both M_1 and M_2 has the same class declaration in them.

4 $MSpec_1(op) \Leftrightarrow MSpec_2(op)$ for all $op \in I_1 \cap I_2$

This definition can be extended to a finite family of contracts.

DEFINITION 4 Let $\{C_k = (I_k, M_k, A_k, MSpec_k, Init_k)\}$ be a consistent finite family of contracts. Their **merge,** (denoted by $\|_{k \in K}\ C_k$), is defined by

$$I \stackrel{def}{=} \uplus_k I_k, \qquad M \stackrel{def}{=} \oplus_k M_k, \qquad A \stackrel{def}{=} \oplus_k A_k,$$
$$Init \stackrel{def}{=} \oplus_k Init_k, \quad MSpec \stackrel{def}{=} \oplus_k Mspec_k$$

where $\oplus$ denotes the overriding operator, e.g. $(MSpec_k \oplus MSpec_{k+1})(op) = MSpec_{k+1}(op)$ if $op \in I_k \cap I_{k+1}$; $MSpec_k(op)$ if $op \in I_k$ but $op \notin I_{k+1}$; $MSpec_{k+1}(op)$ otherwise.

A merge of a family of contracts corresponds the construction of a conceptual model from the partial models of the application domain in the contracts. There are three cases about the partial models:

1 The contracts do not share any fields or modelling elements in their conceptual models. In this case, the system formed by the components of these contracts are most loosely coupled. All communications are via method invocations. Such a system is easy to design and maintain. Composing these components is only plug-in composition.

2 The contracts may share fields, but their conceptual models do not share any common model elements. In this case, application domain is partitioned by the conceptual models of these contracts. And components of the system are also quite loosely coupled and easy to construct and maintain. When composing these components, some simple wiring is needed.

3 The contracts share common model elements in their conceptual models. The refinement/design of the contracts has to preserve the consistency and integrity, generally specified by state invariants, of the model. The more elements they share, the more tightly the components are coupled and the more wiring is needed when composing these components.

DEFINITION 5 We say that a contract $C_1 = (I_1, M_1, A_1, MSpec_1, Init_1)$ is (*downwards*) **refined** by $C_2 = (I_2, M_2, A_2, MSpec_2, Init_2)$, denoted by $C_1 \sqsubseteq C_2$, if there is a mapping ρ from A_1 to A_2 satisfying

1 The initial state is preserved: $(\underline{x} := Init_1(\underline{x}); \rho) \sqsubseteq (\underline{y} := Init_2(\underline{y}))$, where $\underline{x}$ is the list of variables defined in A_1, and $\underline{y}$ the list of variables in A_2. Notice that we have used a UTP design to represent a refinement mapping.

2 The behavior of the operations of C_1 are preserved: every operation op declared in I_1 is also declared in I_2 and $(MSpec_1(op); \rho) \sqsubseteq (\rho; MSpec_2(op))$.

An *upwards refinement* relation can be similar defined in terms of a refinement mapping from A_2 to A_1.

The refinement relation between contracts will be used to define component refinement. The state mapping ρ allows that a component developed in an application domain can be used in another application domain if such a mapping can be found.

THEOREM 6 *Contract refinement enjoys the properties of program refinement.*

1 $\sqsubseteq$ *is reflexive and transitive and a pre-order.*

2 (**An upper bound condition**) *The merge of a family of contracts refines any contract in the family, i.e.* $\{C_k \mid k \in K\}$ *be a family of consistent contracts,* $C_i \sqsubseteq \|_{k \in K} C_k$ *for any* $i \in K$.

3 (**An isotonicity condition**) *The refinement relation is preserved by the merge operation on contracts, i.e. let* $\{C_k^i \mid k \in K\}$, $i = 1, 2$, *be families of consistent contracts without shared fields. If* $C_k^1 \sqsubseteq C_k^2$ *for all* k, *then* $\|_{k \in K} C_k^1 \sqsubseteq \|_{k \in K} C_k^2$.

We define the *equivalence relation* by $\equiv \stackrel{def}{=} \sqsubseteq \cap \sqsupseteq$.

4 Component

A component consists of a provided interface and optionally a required interface, and an executable code which can be coupled to the codes of other components via their interfaces.

The external behavior of a component is specified by the contracts of its interfaces. A design of a component has to reorganize the data to realize the conceptual states, and realize the conceptual models in the contract of the component by software classes. That is the conceptual model has to be transformed into a *design model.*

Design class model

We slightly generalize the definition of a contract to allow the declarations of methods in the class model that is now called a *design class model,* which is specified as a sequence of class declarations $class_1; \ldots; class_n$, each is of the form defined in OOL:

Class N **extends** M {
private $U_1\ u_1, \ldots, U_k\ u_k$;
protected $V_1\ v_1, \ldots, V_\ell\ v_\ell$;
public $W_1\ w_1, \ldots, W_m\ w_m$;
method $m_1(parameters_1)\{c_1\}; \cdots; m_n(parameters_n)\{c_n\}$
}

where *parameter* s_i is of the form $<$ **in** : $\underline{T}_{1i}\ \underline{x}_i$, **out** : $\underline{T}_{2i}\ \underline{y}_i$, **inout** : $\underline{T}_{3i}\ \underline{z}_i$ $>$ consisting of the *value, result* and *value-result* parameters of m_i, c_i of method m_i is a command called the body of m_i. We use *Meth*(M) to denote the set of all methods declared in a design model M.

A command c is specified according to the following syntax:

$c ::=$	$\beta : p \vdash Q$	design
	$\mid skip \mid chaos$	skip and abort
	$\mid$ **var** T x=e $\mid$ **end** x	variable declaration and undeclaration
	$\mid le := e \mid c; c$	assignment and sequence
	$\mid c \lhd b \rhd c \mid b * c$	conditional choice and loop
	$\mid le.m(\underline{e}) \mid op(\underline{e})$	method call and operation call
	$\mid C.New(x)[\underline{e}]$	Creating a new object with initail values $\underline{e}$ for its attributes

where b is a Boolean expression, e is an expression, and le is an expression which may appear on the left hand side of an assignment and is of the form $le ::= x | le.a | self$ where x a simple variable and a is an attribute of an object. We use $\mathbf{if}\{(b_i \longrightarrow P_i) | 1 \leq i \leq n\}\mathbf{fi}$ to denote the multiple choice statement.

Expressions, which can appear on the right hand sides of assignments, are constructed according to the rules $e ::= x | null | self | e.a | e$ **is** $\mathbf{C} | (C)e | f(e)$, where *null* represents the special object of the special class *Null* that is a subclass of all class and has *null* as its unique object, $e.a$ is the a-**attribute** of e, $(C)e$ is the type casting, e **is** C is the type test.

Components

DEFINITION 7 A **component** P is a tuple $<$ O, I, M, A, $MImpl$, $Init$, R $>$ where

- O is an interface, called the *provided* or (*output*) *interface* of P.
- I is an interface disjoint from O, called the *internal interface* of P
- M is a design class model.
- A is a set of fields whose types are all declared in M.
- *MImpl* maps each operation declared in $O \cup I$ to a pair $(\alpha, \mathbb{Q})$, where $\mathbb{Q}$ is a command written in the above OOL, and α is the alphabet obtained from A and the input and output parameters of the operations in $O \cup I$.
- R, is the interface that is disjoint from O and I and consists of the *operations* (not methods of classes in M) which are referenced in the program text $\mathit{MImpl}(op)$ and bodies of methods in $Meth(M)$ but not in $O \cup I$, where $op \in O \cup I$. R is called the *input* or *required* interface of P.

We call $C = (O, I, M, A, MImpl, Init)$ a *generalized contract,* as it has internal operations and *MImpl* provides the specification of each operation of O in terms a general OOL command.

Hence, we will use 4-tuple $P = (C, I, O, R)$ denote a component, where C is a generalized contract for the interface $O \uplus I$.

A contract for R is called a *required service* of P, and a contract of the interface O a *provided service.* Operations in R can be seen as *holes* in the component where their specifications or implementation given in other components that are to be plugged in. Therefore, the provided services of a component depends on its required services plugged in from other components. This leads to the definition of our semantics of a component.

In the above definition, we introduced private operations so that we can hide an output operation by making it a private operation. This will keep the definition *MImpl* valid as the hidden operations may be called in $\mathit{MImpl}(op)$.

Method hiding

Hiding interface operations allows to offer different services to different clients.

DEFINITION 8 **(Hiding)** Let $C = (O, I, A, M, MImpl, Init)$ be a general contract, and $H \subseteq O$ a set of operations. The notation $C\backslash H$ represents the contract

$$(O \setminus H, I \cup H, M, A, MImpl, Init)$$

where $S\backslash S_1$ is set-subtraction.

THEOREM 9 *The hiding operator enjoys the following properties.*

1 $(C\backslash H) \sqsubseteq C$.

2 $C\backslash\emptyset \equiv C$.

3 $C\backslash H \equiv C\backslash(H \cap O)$, *where I is the interface of C.*

4 $(C\backslash H_1)\backslash H_2 \equiv C\backslash(H_1 \cup H_2) \equiv (C\backslash H_2)\backslash H_1$

5 $(\|_{k\in K} C_k)\backslash H \equiv \|_{k\in K}(C_k\backslash H)$

Semantics components

DEFINITION 10 The **semantics** of a component P is identified as a binary relation between its required services and their corresponding provided services

$$[\![P]\!](C_R, C'_O) \stackrel{def}{=} (C_R >> P) \sqsubseteq C'_O$$

where the variable C_R takes an arbitrary required service as its value, C'_O takes a provided service for O, and the notation $C_R >> P$ denotes the provided service

$$(O, F(M), A, MSpec, Init)$$

where $F(M)$ is the class model obtained from M by removing the methods of its classes, and mapping $MSpec$ is defined from the given required service

$$C_R =< R, M_R, A_R, MSpec_R, Init_R >$$

by the recursive equations $MSpec(op) = \mathcal{M}(MImpl(op))$, where $\mathcal{M}$ replaces every call of $op(inexp, outvar)$ with the actual input parameters $inexp$, output parameters $outvar$ and value-result parameters $vrexp$ of O by its corresponding specification.

$$\begin{array}{lll}
\mathcal{M}(op(inexp,\ outvar,\ vrexp)) & \stackrel{def}{=} & \left(\begin{array}{l} \textbf{var } T_1\ x = inexp,\ T_2\ y,\ T_3\ z = vrexp; \\ MSpec_R(op); outvar, vrexp := y, z; \\ \textbf{end } x,\ y,\ z \end{array}\right) \\
 & \text{if} & op(\textbf{in} : T_1\ x, \textbf{out} : T_2\ y, \textbf{inout} : T_3\ z) \in R \\
\mathcal{M}(op(inexp,\ outvar,\ vrexp)) & \stackrel{def}{=} & \left(\begin{array}{l} \textbf{var } T_1\ x = inexp,\ T_2\ y,\ T_3\ z = vrexp; \\ MImpl(op); outvar, vrexp := y, z; \\ \textbf{end } x,\ y,\ z \end{array}\right) \\
 & \text{if} & op(\textbf{in} : T_1\ x, \textbf{out} : T_2\ y, \textbf{inout} : T_3\ z) \in \\
 & & O \cup I \\
\mathcal{M}(v := e) & \stackrel{def}{=} & v := e \\
\mathcal{M}(\mathcal{F}(\mathrm{c})) & \stackrel{def}{=} & \mathcal{F}(\mathcal{M}(\mathrm{c})) \quad \text{for any comand } c \text{ and context } \mathcal{F}
\end{array}$$

Notice that when a component P has an empty set of required interface operations, P is a *closed component* and the notation $C_\emptyset >> P$ becomes a constant that is the semantics of the closed program P.

For a given contract C_R for the required interface of P, $C_R >> P$ is a closed component. Let C_O be a contract of the provided interface of P which serves as the specification of the component. We say that P *correctly realizes* or *implements* C_O with a given required service C_R if $C_O \sqsubseteq (C_R >> P)$.

In a modular programming paradigm, a component can be designed and implemented as a module in which each of the operations in the output interface is "programmed" using procedures or functions that are defined either locally in the module or externally in other modules. In this case, the external modules that the component calls methods from must be declared, as well as the types of the attribute values and parameters of its methods. Therefore, a component is in fact not a single module, but an artifact that contains all these declared types and modules. In an object-oriented

paradigm, such as Java, a component can be seen as a class that implements the interfaces in *O:*

M; //* the declaration of the design model
Class P implements O {**Attr** : A;
public : $m \stackrel{def}{=} MImpl(m)$; //* for each $m \in O$;
private : $op \stackrel{def}{=} MImpl(op)$ //* for each $op \in I$
}

Thus, after adding the notation for interfaces and contracts to OOL in [Liu et al., 2004b, Liu et al., 2004c], the extended language provides a formal model for components and the calculus of contract refinement and component refinements [He et al., 2003], and also extends RCOOD in [He et al., 2002, Liu et al., 2004b] to for component-based development.

Example Now we define a component *GIS* in the ParcelCall system to provide the services to customers. We will use some Java conventions in writing the specification, such as assignment to a variable with a method call that has an **out** parameter.

<< Component >> *GIS*
Output Interface *CustomerService*
Attr : $\mathbb{P}PName\ P$; // * set of parcel names
$\mathbb{P}CName\ S$; // * set of customer names
$CName \times PName\ owns$; // * $owns(s,p)$: s owns p
$(PName \longmapsto Position)\ loc$; // * $loc(p)$ returns the location of p
Init : $P = \emptyset \wedge S = \emptyset$;
Meth : *LocateParcel*(**in** : *PName pId, CNamesId*, **out** : *Position location*){
if $pId \in P \wedge sId \in S \wedge owns(sId, pId)$ //* call required method
then *location* := *IParcelInfo.Where*(*pId*) **else** *Abort*};
DispatchParcel(**in** : *PName pId, CName sId*) {
if $pId \notin P \wedge sId \notin S$ **then** ($P := P \cup \{pId\}$; $S := S \cup \{sId\}$;
owns := $owns \cup \{(sId, pId)\}$; *IParcelInfo.Deal*(pId)) **else** *chaos*};
Input Interface *Parcelinfo*
Attr : $\mathbb{P}PName\ P$; // * set of parcel names
$(PName \longmapsto Position)\ loc$; // * $loc(p)$ returns the location of p;
Meth : *Where*(**in** : *PName pId*, **out** : *Coordinates location*) {
Deal(**in** : *PName pId*)};
<< Contract >> *Parcelinfo*
IParcelInfo :: **Init** : $P = \varnothing$;
IParcelInfo :: *Where*(**in** : *PName pId*, **out** : *Position location*) :
$pId \in P \vdash location := loc(pId)$;
IParcelInfo :: *Deal*(**in** : *PName pId*) : $pId \notin P \vdash loc'(pId) = (0,0)$

We can calculate *ParcelInfo* >> $GIS \sqsubseteq CS$. We have kept the attribute *loc* : *PName* $\longmapsto$ *Position* to avoid from defining a state mapping in the proof of the refinement. In the following part of the example, we provide a definition of com-

ponent GTS to implement the contract *ParcelInfo,* that need the specification of a design class.

```
<< Component >> GTS
Class Parcel{ PName id; Position location = (0,0);
              Positionloc(){return := location}};
Ouput Interface IParcelInfo
   Attr : ℙParcel Parcels : ;
   Init : Parcels = ∅;
   Meth : Deal(in : PName pId){
            Parcel.New(p)[pId]; Parcels := Parcels ∪ {p}; end p};
            Where(in : PName id, out : Position location){
            var Parcel p = P.find(pId); location := p.loc(); end p}
```

Define the refinement mapping ρ from the attributes of *Parcel* to those of *ParcelInfo:*

$$\rho(P) \stackrel{def}{=} \{p.id \mid p \in Parcel\}$$
$$\rho(loc(pId)) = p.location \text{ for all } pId \in P \text{ such that } \exists p \in Parcel.p.id = pId$$

Then $ParcelInfo \sqsubseteq GTS$.

Refinement and composition of components

For a component P with provided and required interfaces O and R, the semantics $[\![P]\!]$ is a binary relation between the input services and output services.

THEOREM 11 (***Monotonitity and Upwards Closure*** **[Smyth, 1978])** *Let P =< C, I, O, R > and* $\sqsubseteq_R$ *and* $\sqsubseteq_O$ *are the refinement relations among contracts of R and among contracts of O respectively. Then* $\sqsubseteq_R \circ [\![P]\!] \circ \sqsubseteq_O = [\![P]\!]$, *where* $\circ$ *denotes relational composition.*

Thus, for any required services $C_R \sqsubseteq C'_R$, and provided services $C_O \sqsubseteq C'_O$, then

$$[\![P]\!](C_R, C'_O) \Rightarrow [\![P]\!](C'_R, C_O)$$

A component P_1 is a *refinement* of a component P_2, denoted by $P_2 \sqsubseteq P_1$, if P_1 is a sub-relation of P_2.

DEFINITION 12 Component P_1 is a **refinement** of P_2 if

$$R_1 = R_2 \wedge O_1 = O_2 \wedge [\![P_1]\!] \Rightarrow [\![P_2]\!]$$

P_1 refines P_2 iff for any required service C_R, $(C_R >> P_2) \sqsubseteq (C_R >> P_1)$.

We therefore have when P_1 refines P_2, then for any given required service C_R and a contract a provided service C_O as the specification, P_1 realizes C_O with C_R if P_2 realizes C_O with C_R.

DEFINITION 13 Let $P_i = (C_i, I_i, O_i, R_i)$ be two components with contracts $C_i = (O_i \cup I_i, M_i, A_i)$, for $i = 1, 2$. Assume that $I_1 \cap I_2 = \emptyset$, $O_1 \cap O_2 = \emptyset$ and $R_1 \cap R_2 = \emptyset$. The **composition** $P_1 || P_2$ is defined to merge their contracts, output

interfaces and input interfaces, and to remove those input interfaces of each component that are matched by the output interfaces in another:

$$P_1 \| P_2 \stackrel{def}{=} < C_1 \| C_2, I_1 \cup I_2, O_1 \uplus O_2, R_1 \backslash O_2 \cup R_2 \backslash O_1 >$$

Let $I \stackrel{def}{=} I_1 \cup I_2$, $R \stackrel{def}{=} R_1 \backslash O_2 \cup R_2 \backslash O_1$ and $O \stackrel{def}{=} O_1 \cup O_2$. The composition of P_1 and P_2 is defined by

$$\begin{aligned} [P_1 \| P_2](C_R, C'_O) \stackrel{def}{=} \; & \exists C_{R_1}, C'_{O_1}, C_{R_2}, C'_{O_2} \bullet \\ & [P_1](C_{R_1}, C'_{O_1}) \wedge [P_2](C_{R_2}, C'_{O_2}) \wedge \\ & C_{R_1} \backslash (R_1 \setminus O_2) = C'_{O_2} \backslash (O_2 \setminus R_1) \wedge \\ & C_{R_2} \backslash (R_2 \setminus O_1) = C'_{O_1} \backslash (O_1 \setminus R_2) \wedge \\ & C_R = C_{R_1} \backslash (R_1 \backslash O_2) \| C_{R_2} \backslash (R_2 \backslash O_1) \wedge \\ & C'_O = C'_{O_1} \backslash (R_2 \backslash O_1) \| C'_{O_2} \backslash (R_1 \backslash O_2) \end{aligned}$$

This definition allows an output interface and thus part of provided service of one component to be shared among a number other components. Hiding can be used to *internalize* the part of a provided service of one component that is used in another component: $(P_1 \| P_2) \backslash (R_1 \cap O_2) \backslash (R_2 \cap O_1)$.

Example We can now compose *GIS* and *GTS*. $(GIS \| GTS) \backslash IParcelInfo$. If we do not consider the relation between *GTS* with other components of the Parcel-Call system, this composite component is a closed system that only provides services according to the contract of *CS,* but it does not have any required interface. However, to complete the ParcelCall system, we can add a required service interface to get the new location of a parcel from the Mobile Logistic Server component *MLS.* Alternatively, we add another provided interface *ChangLoc*() that will be needed as a required interface of Mobile Logistic Server component *MLS* to update the location of a parcel.

Client-server systems are often seen as applications in component software. The architecture of such a system is organized as a layered structure and can be model with in our model as shown in the full paper [Liu et al., 2004a].

5 Conclusion

We have proposed a model for software components and defined composition and refinement of components. This allows us to use the existing calculus in [Hoare and He, 1998, Liu et al., 2004b, Liu et al., 2004c] to reason about and refine components. We have separated the different views about a component. The different views are specified at different levels of abstraction. A component is constructed to provide certain services and these services are specified in terms of the component's interface and contract. This specification is taken as the requirement specification of the component. The designer of the component has to design and implement the component to satisfy this requirement specification. A design can be specified in the object-oriented specification notation developed in, that supports incremental and step-wise construction of a component [Liu et al., 2004b, Liu et al., 2004c]. Merge and hiding of interfaces for components add more support to incremental construction of component software as well as to restrict the use of some services by certain users.

When composing components, one has to check the matchability of the provided services of one component with the specification of the required services of another, both syntactically and semantically. The syntactic check is only to check the signature of the interface methods. The semantic check is to ensure that the provided service of one component does ensure the service required by another component. This is to check the pre and post conditions of in the specification of the services.

Points of discussion The model of components is simplified in the sense that behavior or protocols of the interfaces are not described. There are several possible ways to address the problem of protocols. First, we can introduce control state variables in contracts and thus in components. This will allow us to define a contract as a state machine or statechart, e.g. [Selic, 1998, Wirsing and Broy, 2000]. Then when two components are composed, deadlock freedom needs to be verified and this is not an easy task. Second, in addition to the state information, we can add a CSP-like specification of the order of the methods in a component, e.g. [Allen and Garlan, 1997]. Again, matching between protocols in different components has to be checked and deadlock needs to be avoided. As we know from the model of CSP, this is not a trivial task either. We would like to propose a weak approach in which protocols of the provided interface and required interface of a component are described independently in terms of *regular languages* on the method names of the interfaces. To check the matchability between a provided interface with a required interface is then to check the provided interface protocol is a subset of the required interface protocol in terms of the regular languages that are defined for the protocols, and this can be automated.

Related work There is much work on the definitions of software components. We take the informal views of [Cheesman and Daniels, 2001, Szyperski, 2002] that a component both provides to and requires services from other components. We used the notion of contract for formal specification of provided and required services, A contract here is similar to that of Meyer [Meyer, 1992]. However, we have provided the notion of composition and there is a standard calculus for reason about and refine components at different levels of abstracts. A distinctive nature of our framework is the natural link of the component contract specification and its object-oriented implementation.

A contract in [Helm et al., 1990] models the collaboration and behavioral relationships between objects. In our approach, we provide the separation between the specification of a contract for an interface from the specification of the behavior of the component that realizes the contract. A contract in [Andrade and J.L.Fiadeiro, 1999] describes the coordinations among a number of partners (i.e. components or objects). Its main purpose is to support system architectural evolution and to deal with changes in business rules of the system application. Our contracts here specify the services of components while we treat interaction and coordinations as part of the implementation of the components. Our aim is to support construction of software components and component software systems.

Acknowledgement We thank the referees for their careful review and constructive and helpful comments. We also thank our colleague Dang Van Hung for his comments on the earlier version of the paper.

References

[Allen and Garlan, 1997] Allen, R. and Garlan, D. (1997). A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology,* 6(3).

[Andrade and J.L.Fiadeiro, 1999] Andrade, L. F. and J.L.Fiadeiro (1999). Interconnecting objects via contracts. In France, R. and Rumpe, B., editors, *UML'99 - Beyond the Standard, LNCS1723.* Springer-Verlag.

[Cheesman and Daniels, 2001] Cheesman, J. and Daniels, J. (2001). *UML Components. Component Software Series.* Addison-Wesley.

[Filipe, 2002] Filipe, J. (2002). A logic-based formalization for component specification. *Journal of Object Technology,* 1(3):231–248.

[He et al., 2002] He, J., Liu, Z., and Li, X. (2002). Towards a refinement calculus for object-oriented systems (keynote talk). In *Proc. ICCI02, August 19-20, 2002, Alberta, Canada.*

[He et al., 2003] He, J., Liu, Z., and Li, X. (2003). Component calculus. In Dang, V. and Liu, Z., editors, *Proc. Proc. Workshop on Formal Aspects of Component Software (FACS'03), Satellite Workshop of FME2003, Pisa, Italy, 8-9 September, 2003.* UNU/IIST Report No 284, UNU/IIST, P.O. Box 3058, Macao.

[Heineman and Councill, 2001] Heineman, G. and Councill, W. (2001). *Component-Based Software Engineering, Putting the Pieces Together.* Addison-Wesley.

[Helm et al., 1990] Helm, R., Holland, I., and Gangopadhyay, D. (1990). Contracts: Specifying behavioral compositions in object-oriented systems. In *Proc. OOPSLA'90/ECOOP'90,* pages 169–180. ACM.

[Hoare and He, 1998] Hoare, C. and He, J. (1998). *Unifying theories of programming.* Prentice-Hall International.

[Liu et al., 2004a] Liu, Z., He, J., and Li, X. (2004a). Contract-oriented component software development. Technical Report UNU/IIST, Report No 298. http://www.iist.unu.edu/newrh/III/1/page.html.

[Liu et al., 2004b] Liu, Z., He, J., and Li, X. (2004b). Integrating and refining UML models. Technical Report UNU/IIST Report No 295, http://www.iist.unu.edu/newrh/III/1/page.html, UNU/IIST, P.O. Box 3058, Macao. Submitted for publication.

[Liu et al., 2003] Liu, Z., He, J., Li, X., and Chen, Y. (2003). A relational model for object-oriented requirement analysis in UML. Technical Report UNU/IIST, Report No 287. *Proc. ICFEM03,* 5-7 November, 2003, Singapore. Lecture Notes in Computer Science.

[Liu et al., 2004c] Liu, Z., He, J., Li, X., and Liu, J. (2004c). Unifying views of UML. Technical Report UNU/IIST Report No 288, http://www.iist.unu.edu/newrh/III/1/page.html, UNU/IIST, P.O. Box 3058, Macao. Presented at <<UML>> 2003 Workshop on Compositional Verification of UML, 21 October 2003, SF, USA. To appear in ENTCS.

[Meyer, 1992] Meyer, B. (1992). Applying design by contract. *IEEE Computer.*

[Meyer, 1997] Meyer, B. (1997). *Object-oriented Software Construction (2nd Edition).* Prentice Hall PTR.

[Selic, 1998] Selic, B. (1998). Using UML for modelling complex real-time systems. In Mueller, F. and Bestavros, A., editors, *Language Compilers, and Tools for Embedded Systems, LNCS 1474,* pages 250–262. Springer.

[Smyth, 1978] Smyth, M. (1978). Powerdomain. *Journal of Computer Science and System Sciences,* 16:23–36.

[Szyperski, 2002] Szyperski, C. (2002). *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley.

[Wirsing and Broy, 2000] Wirsing, M. and Broy, M. (2000). Algebraic state machines. In Rus, T., editor, *Proc. 8th Internat. Conf. Algebraic Methodology and Software Technology, AMAST2000. LNCS 1816,* pages 89–118. Springer.