The United Nations University

UNU/IIST

International Institute for Software Technology

Contract-Oriented Component Software Development

He Jifeng, Zhiming Liu and Li Xiaoshan

March 2003

UNU/IIST Report No. 276

UNU/IIST and UNU/IIST Reports

UNU/IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU/IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macau authorities also supply UNU/IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU/IIST is to assist developing countries in the application and development of software technology.

UNU/IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU/IIST, and
7. Dissemination, in which UNU/IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU/IIST is on formal methods for software development. UNU/IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU/IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU/IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU/IIST at P.O. Box 3058, Macau or visit UNU/IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU/IIST and its report series.

Chris George, Acting Director

The United Nations University

UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

Contract-Oriented Component Software Development

He Jifeng, Zhiming Liu and Li Xiaoshan

Abstract

We present an approach to modelling component software. We describe how components are specified at the interface level, design level and how components can be composed. From its user's or the system assembler's (i.e. external) point of view, a component provides an interface consisting of a set of attributes, a set of provided services, and a set of required services. From its designer's (i.e. internal) point of view, a component consists of a collection of collaborating objects/classes that realize the interface. The verification of a component is to verify that its internal view meets its external view.

Keywords: *Component, Composition, Contract, Interface, Inheritance, Object-Orientation, Refinement*

HE Jifeng is a senior research fellow of UNU/IIST. He is also a professor of computer science at East China Normal University and Shanghai Jiao Tong University. His research interests include the Mathematical theory of programming and refined methods, design techniques for the mixed software and hardware systems. E-mail: hjif@iist.unu.edu.

LIU Zhiming is a research fellow at UNU/IIST, on leave from Department of Mathematics and Computer Science at the University of Liecester, Liecester, England where he is lecture in computer science. His research interests include theory of computing systems, including sound methods for specification, verification and refinement of fault-tolerant, real-time and concurrent systems, and formal techniques for OO development. His teaching interests are Communication and Concurrency, Concurrent and Distributed Programming, Internet Security, Software Engineering, Formal specification and Design of Computer Systems. E-mail: Z.Liu@iis.unu.edu.

LI Xiaoshan is an Assistant Professor at the University of Macau. His research areas are interval temporal logic, formal specification and simulation of computer systems, formal methods in system design and implementation. E-mail: xsl@umac.mo.

Contents

1 Introduction

The use of components to build and to 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 [18, 5, 10]. While component technologies such as COM+, CORBA, and Enterprise JavaBeans are becoming widely used, there is so far no agreement on standard technologies for designing and creating components, nor on methods of 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 essential as it would not be possible to manage change, substitution and composition of components successfully 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 C consists a set of *provided services*. The provided services form the *provided interface* of the component. The provided interface defines the operations that the component provides and what their signatures are. To specify the effects of the interface operations, it is necessary to partially know the conceptual state of the component. Consequently, the interface specification contains a so-called *information model* [5, 7] or a *conceptual model* [14, 15]. The effect of an operation op can then be described in terms of a *design* $p(x) \vdash R(x, x')$ in Hoare and He's Unifying Theories of Programming [12] , where $p(x)$ is the precondition and $R(x, x')$ the post-condition. This determines a *contract* between the component and its clients [5, 10] such that to use the service op, the client has to ensure the pre-condition $p(x)$, and when this is true the component must guarantee the post-condition R. We can thus define the *contract* of an interface as an assignment mapping that assigns a design to each interface operation.

The contract for the provided interface of a component is also taken as the goal for the designers of the component who has to design and implement the component's provided services. The designers may decide to use services provided by other components in the design and implementation. These services are called *required services* of the component under consideration (*CuC*). The designer does not have to build the components to provide the required services of the CuC themselves. The components can be built and implemented by another team of designers or bought as a component-off-the-self (COTS). To build a larger component with the CuC and other components and to verify the the CuC itself, we must formally describe the required services as contracts of a set of interfaces, called *required interfaces*. The required interfaces define the required operations and their signatures. There are often cases where the designers of a component are given the specification of the required services and asked to build a component to provide its provided services. A client who wants to use the services of a component is only interested in knowing the provided services. However, to use a component in assemble a system, the assembler needs to know the specification of bother the provided and required services.

We will specify the design of a component by assigning each operation op in the provided interface with a program text $def(op)$ described in a specification notation. We will use the object-oriented notation proposed in [9]. In $def(op)$, calls to operations in a required interface are allowed. With the notation in [8, 9], we can verify whether $def(op)$ refines the specification of op given in the contract of the provided

interface. The *verifier* of a component needs to know the contract of the provided interface, the contracts of the required interfaces, and the program text $def(op)$ for each operation op of the provided interface. In this sense, we can understand a component as as a relation between contracts of required interfaces and contracts of the provided interface. Given a contract for each required interface, we can calculate a contract for the provided interface from the program text $def(op)$ for each op.

When we assemble a system from components, we *connect* or *compose* two components P_1 and P_2 by linking the methods in the provided interface of one component to the methods of a required 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 is a refinement of the contract of the required interface of P_2. If P_1 and P_2 match well, the composition $P_1 \mid\mid P_2$ forms another component. The provided interface of $P_1 \mid\mid P_2$ is the *merge* of the provided interfaces of P_1 and P_2. The required interfaces of $P_1 \mid\mid P_2$ 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. Our model also allows multiple inheritance of interfaces and contracts. Refinement between contracts and between components are also defined.

For illustrating our approach, we will take the example of the ParcelCall system from [7]. ParcelCall is an European research and technology development project looking at creating a *parcel localization system*: an open distributed system which is to be integrated with the legacy systems of transport and logistic companies. It used in [7] to demonstrate the use of the logic MDTL in formalizing component specification. Our treatment of this example will show the simplicity of our model and the advantages of the separation of the concerns in our framework.

After this introduction, we introduce the ParcelCall system example in Section 2. In Section 3, we define the notations of interfaces, composition of interfaces by merging them together, and interface inheritance. We define the a notation for the specification of a contract of an interface in Section 4. A contract of an interface defines the effect of the methods of the interface in terms of their pre and post conditions. Composition, refinement of contracts, contract inheritance, method hiding are defined in this section. We devote Section 5 to the definition of components and their compositions. Section 6 briefly discusses how a client-server application is generally modelled with our approach. Section 7 concludes the paper with discussions. Throughout the paper, we use the ParcelCall System to illustrate the concepts.

2 The ParcelCall Example

For the use of the example throughout the discussion in the paper, we first give a description of the ParcelCall example.

The ParcelCall project explores the development of a low cost information infrastructure that will enable the continuous information about the exact geographic position of parcels at any time. Logistic or transportation companies (referred as carriers) will the be able to offer an additional services to customers: a customer can query the location and status of her transportation goods.

The ParcelCall system has three main components:

- a *Mobile Logistic Server* (MLS): is an exchange point or a transport unit (container, trailer, freight wagon, etc). The transport units carry the parcels. Since containers can be insider other containers, MLSs form a hierarchy. MLSs always know their current location via the GPS satellite positioning system.
- a *Goods Tracing Server* (GTS): comprises several databases which contains MLS hierarchies. Moreover, it 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.

For simplicity, we will not address the integration of ParcelCall with a carrier system nor the arising dependability issues (see Section 6 in [7]). We show how a component-based approach can be used for modelling the different views of ParcelCall system. We focus on the specification of some of the interfaces for the interactions between components within the localization system triggered by a request to localize a parcel. we will also partially specify the design of the components GIS and GTS with ideas of verification.

In [5] the stereotypes $<<$ comp spec $>>$ and $<<$ interface type $>>$ are introduced to describe component specifications and interfaces respectively. The UML notation for interfaces and components is used to describe the system architecture. Figure 1 is taken from [7] for the architecture of the ParcelCall system.

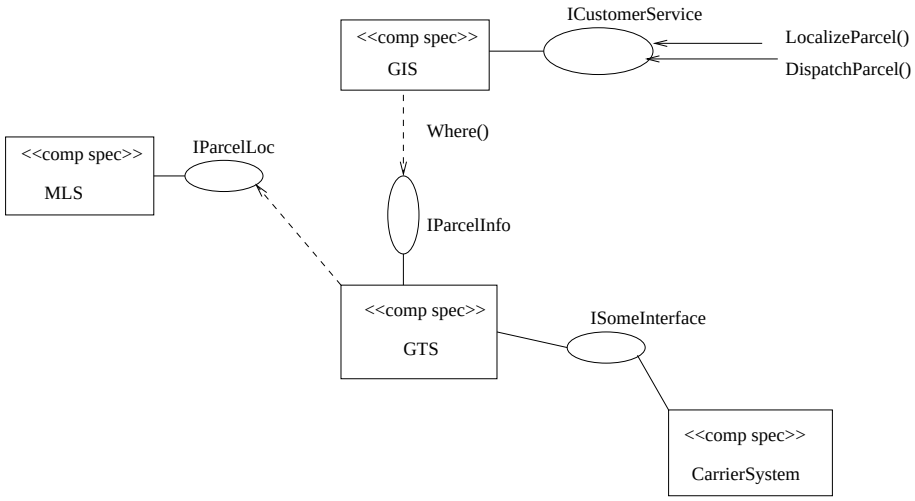

Figure 1: Architecture of ParcelCall

The figure shows the three main components described earlier, some of their interfaces and component dependencies. For example, the GIS component offers an interface $ICustomerService$ that provides the services $DispatchParcel$ and $LocalizeParcel$, and requires a service from component GTS via the interface $IParcelInfo$. The figure also shows where the legacy system from the carrier is going to be integrated with ParcelCall.

The interfaces from GIS component will establish communication with the customer: for instance a customer can enter a request to find the current location of a parcel via $LocalizeParecel$.

3 Interface

For the specification of a component, we first start in this section with the definition of interfaces. Then we show how a finite number of interfaces can be composed. This leads to the definition of interface inheritance. Interfaces are just like Java interfaces, but with an information model described as attributes.

3.1 Primitive Interface

A primitive interface I is a collection of *features* where a feature can be either an attribute $x : \mathbf{T}$ or a method $op(in : \mathbf{U},\ out : \mathbf{V})$, where in are the *value* parameters and out are the *result* parameters. Thus an interface is a named pair $I :< A,\ M >$, where A denotes the set of attributes, and M is the set of methods of the interface I. We assume that attribute names and method names are taken from two disjoint sets. We use $Atrr(I)$ and $Meth(I)$ to denote the sets of attributes and methods of I respectively. For a family $\mathcal{I}$ of interfaces, we use $Meth(\mathcal{I})$ to denote the set of methods of all the interfaces in $\mathcal{I}$. A primitive interface can be specified in the following format:

$$
\begin{array}{l}
\textbf{Interface}\ I\{ \\
\quad \textbf{Attribute}:\ a_1 : \mathbf{T}_1;\ \ldots;\ a_m : \mathbf{T}_m \\
\quad \textbf{Method}:\ op_1(in_1 : \mathbf{U}_1, out_1 : \mathbf{V}_1); \\
\qquad\qquad \ldots; \\
\qquad\quad op_k(in_k : \mathbf{U}_k, out_k : \mathbf{V}_k) \\
\}
\end{array}
$$

A method $op(in : \mathbf{U},\ out : \mathbf{V}) \in M$ takes in of type $\mathbf{U}$ as its input parameters, and out of type $\mathbf{V}$ as its output parameters. We assume that none of variables in in or out appears in the set A.

Example The interfaces from GIS component will establish communication with the customer: for instance a customer can enter a request to find the current location of a parcel via $ILocalizeParecel$.

The specification of this interface can be described as

$$
\begin{array}{l}
\textbf{Interface}\ CustomerService\ \{ \\
\textbf{Attribute}:\ P:\ Set(PName);\ \ /**\text{set of parcel names} \\
\qquad\qquad S:\ Set(CName);\ \ /**\text{set of customer names} \\
\qquad\qquad owns:\ CName \times PName;\ \ /**owns(s,p)\text{: customer } s \text{ owns } p \\
\qquad\qquad loc:\ PName \longmapsto Position;\ \ /** \text{ returns the location of } p \\
\textbf{Method}:\ LocalizeParcel((PName\ pId, CName\ sId), Position\ location); \\
\qquad\qquad DispatchParcel((PName\ pId, CName\ sId)) \\
\}
\end{array}
$$

where $Position$ is the type of coordinates to represent locations.

This description can be easily shown as a UML class box or a << Interface type >> stereotype in [5].

3.2 Merge interfaces

It is often the case that onely one required interface is specified in the specification of a component, but there are a number of components, each provides a part of the operations in the required interface. We thus need to *put* these components together to provide one single interface to match the required interface. In fact, the original description of Parcelcall in [7] did have two provided interface, one for $LocalizeParcel()$ and the other for $DismatchParcel()$. We have merged these two interfaces into one in this paper.

Two interfaces $I :< A_1,\ M_1 >$ and $J :< A_2,\ M_2 >$ are *composable* if the following conditions are met.

1. Any shared attribute name must be equipped with the same type in two interfaces, i.e. if $x : \mathbf{T}_1 \in A_1$ and $x : \mathbf{T}_2 \in A_2$, then $\mathbf{T}_1 = \mathbf{T}_2$.
2. If I and J share a method name, then they must assign the same signature to that method, and furthermore they must possess the same attribute set, i.e. if $op(in_i : \mathbf{U}_i, out_i : \mathbf{V}_i) \in M_i$ for $i = 1, 2$, then $A_1 = A_2$, $in_1 \equiv in_2$, $out_1 \equiv out_2$, $\mathbf{U}_1 = \mathbf{U}_2$, and $\mathbf{V}_1 = \mathbf{V}_2$.

This condition is not too restrictive as our definition of *refinement* between contracts allows us to add more attributes into an interface. The two conditions above imply that two components in a system talk to one another about a *common conceptual model*. In a bank system for example, there is no hope of making the components reconfigurable unless all of them use the same definition of basic concepts such as customers, account, and money at their interfaces, if not internally. When one tries to use a component designed for one application in another or specialize a generic component for a special application, renaming or adding a *connector* component [1, 17] can be used.

Definition 1 (Merge of composable interfaces)

Let $\{I_k :< A_k,\ M_k > \ \mid \ k \in K\}$ be a finite family of composable interfaces. Their merge $\uplus_{k\in K} I_k$ is defined by

$$\uplus_{k\in K} I_k =_{df} < \bigcup_{k\in K} A_k,\ \bigcup_{k\in K} M_k >$$

□

3.3 Inheritance

This section deals with the issue of interface inheritance. The motivation for interface inheritance is that a component may only provide part of the services that one needs or some of the provided operations are not quite suitable for the need. We may still use such a component by rewriting some of the operations or extending it with some other operations and attributes. These newly provided services may be realized by another component, or may be programmed using the inherited operations, or by rewriting some inherited operations. If a component that realizes the inherited interface is sued in the design of the extending interface, that component must not be changed though its provided operations can be used in the design of the newly added operations or the overwritten inherited operations. Therefore, the overall purpose of inheritance is *reuse* and *extension* or *evolution*.

First consider the simple case when an interface inherits from another one. Let $J :< A_1,\ M_1 >$ be an interface. Assume that A_2 is a set of attributes and M_2 is set of methods such that no attribute of A_1 appears in A_2, and no method in M_1 is assigned with a different signature in M_2. Then the notation

$$I\ \mathbf{extends}\ J\ \mathbf{with}\ < A_2,\ M_2 >$$

represents an interface $< A,\ M >$ whose attribute and method sets are defined by

$$A =_{df} A_1 \cup A_1, \qquad M =_{df} M_1 \cup M_2$$

Multiple inheritance is denoted in the form

$$I\ \mathbf{extends}\ \{I_k \ \mid \ k \in K\}\ \mathbf{with}\ < A, M >$$

It is defined to be the interface $I\ \mathbf{extends}\ \uplus_{k\in K}\ I_k\ \mathbf{with} < A,\ M >$, where $\{I_k \ \mid \ k \in K\}$ is a set of compsable interfaces.

4 Contracts

A useful way to view interface specification is as contract between a client of an interface and a provider of an implementation of the interface: what the client needs to do to use the interface; what the provider has to implement to meet the services promised by the interface [18].

4.1 Contracts

We give a formal definition of a contract in this section.

Definition 2 (Specification of a method) Given an interface $I :< A, M >$, we define the alphabet α as the set of its attributes and input and output parameters of its methods.

$$\begin{array}{lll} \alpha & =_{df} & in\alpha I \cup out\alpha I, \textbf{ where} \\ in\alpha & =_{df} & A \cup \{in \mid op(in : \mathbf{U}, out : \mathbf{V}) \in M\} \\ out\alpha & =_{df} & \{x' \mid x \in A\} \cup \{out' \mid op(in : \mathbf{U}, out : \mathbf{V}) \in M\} \end{array}$$

The variables of $in\alpha$ are used to record the values of the attributes of A and input parameters in on the activation of op, and the variables of $out\alpha$ the values of the corresponding attributes and outgoing parameters out on the termination of a method.

A *specification* of a method $op(in : \mathbf{U}, out : \mathbf{V})$ of an interface $I :< A, M >$ is a *framed design* $\beta : D$ where

- β is a subset of α, called the *frame* of op (or D) containing the variables to be changed by op, such that $out \in v$.
- $D = (b \vdash R)$ is a design [12] describing the behaviour of the method:

$$\beta : (b \vdash R) =_{df} b \Rightarrow R \wedge \bigwedge_{x \in \alpha \setminus \beta} (x' = x)$$

The predicate b is the assumption on attributes and input parameters which the method can rely on when it is activated, while the predicate R is the commitment which must be true when the execution terminates. □

A contract is then defined to be an interface with an assignment to each of its operation with a specification.

Definition 3 (Contract) A *contract* is a triple $(I, \Phi, Init)$ where I is an interface, and the function Φ maps each method of I to a specification, and $Init$ assigns some values to attributes as their initial values.

□

Example For the ParcelCall system, a *contract* for interface $CustomerService$ assigns each method with a specification and can be written as follows, where $\Phi(op)$ is given as the specification following the name *op* of each operation:

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

In fact a (primitive) contract is a specification of a module in modular programming that defines the behaviour of the methods 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 term of object-orient programming, a (primitive) contract specifies an initialized class, i.e. an object, whose public methods are those in the interface. In a Java-like specification language [8, 9], such a contract can be specified as

Interface I {
  Meth : $Meth(I)$;
}
Class C implements I {
  Attr : $Att(I)$;
  Meth : $\{m =_{df} \Phi(m) \mid m \in Meth(I)\}$;
  main : $\mathbf{C}\ C = New\ \mathbf{C}()$
}

where **main** provides the condition $Init$.

Notice that in above description the attributes of the interface are taken to be the attributes of the class and we do not need to write them in the interface anymore. The merge of a number of contracts then becomes a set of created objects.

4.2 Merge and refinement

When interfaces are merged, their specifications should be merged too. However, they can meaningfully merged only when these interfaces are composable and their contracts are *consistent*.

Definition 4 (**Consistent contracts**) Two contracts $(I_i,\, \Phi_i,\, Init_i)$ $(i = 1,\, 2)$ are *consistent* if

1. I_1 and I_2 are composable.
2. $Init_1(x) = Init_2(x)$ for all $x : T \in A_1 \cap A_2$.
3. $\Phi_1(op) \equiv \Phi_2(op)$ for all $op \in M_1 \cap M_2$

Consistent contracts can be merged together to form a composite contract.

Definition 5 (**Merge of contracts**) Let $\{C_k = (I_k,\, \Phi_k,\, Init_k)\}$ be a finite family of consistent contracts. Their merge, (denoted by $\|_{k\in K}\ C_k$), is defined by

$$I =_{df} \uplus_k I_k, \quad Init =_{df} \oplus_k Init_k, \quad \Phi =_{df} \oplus_k \Phi_k$$

where $\oplus$ denotes the overriding operator, e.g. $(\Phi_k \uplus \Phi_{k+1})(op) = \Phi_{k+1}(op)$ if $op \in M_k \cap M_{k+1}$, $\Phi_k(op)$ if $op \in M_k$ but $op \notin M_{k+1}$, $\Phi_{k+1}(op)$ otherwise. Notice that now the framing operation should apply to the alphabet of the merged interface that is the union of those of the component interfaces. This definition ensures that none of the attributes declared outside I_k can be accessed by its methods. □

A contract can be *refined* by introducing new attributes and new methods, and by data refinement. More precisely, we have the following definition.

Definition 6 (Contract refinement) Let $I_i :< A_i,\, M_i >$ ($i = 1,\, 2$) be interfaces. A contract $C_1 = (I_1,\, \Phi_1,\, Init_1)$ is *refined* by $C_2 = (I_2,\, \Phi_2,\, Init_2)$, 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:

   $$\underline{x} := Init_1(\underline{x});\ \rho \ \Leftarrow\ \underline{y} := Init_2(\underline{y})$$

   where $\underline{x}$ is the list of variables defined in A_1, and $\underline{y}$ the list of variables in A_2.

2. The behaviour of the methods of C_1 are preserved: every method *op* declared in M_1 is also declared in M_2 and

   $$(\beta_1 : D_1);\ \rho \ \Leftarrow\ \rho;\ (\beta_2 : D_2)$$

□

It is easy to prove that the refinement relation between contracts enjoys the properties of program refinement.

Theorem 1 $\sqsubseteq$ *is reflective and transitive:*

1. $C \sqsubseteq C$.
2. *If* $C_1 \sqsubseteq C_2$ *and* $C_2 \sqsubseteq C_3$*, then* $C_1 \sqsubseteq C_3$.

□

The merge of a family of contracts refines any contract of the family.

Theorem 2 *Let* $\{C_k \mid k \in K\}$ *be a family of consistent contracts, then for any* i*,*

$$C_i \ \sqsubseteq \|_{k \in K}\ C_k$$

□

The refinement relation is preserved by the merge operation on contracts.

Theorem 3 *Let* $\{C_k^i \mid k \in K\}$ *(*$i = 1, 2$*) be families of consistent contracts. If* $C_k^1 \sqsubseteq C_k^2$ *for all* k*, then*

$$\|_{k \in K}\ C_k^1 \ \sqsubseteq \|_{k \in K}\ C_k^2$$

□

4.3 Contract inheritance and method hiding

When an interface inherited by another, its contract can also be inherited.

Definition 7 (Contract inheritance) Let $C_i = (I_i, \Phi_i, Init_i)$ $(i = 1, 2)$ be contracts. Assume that no attribute defined by I_1 is redefined by I_2. Then the notation

$$C_2 \,\mathbf{extends}\, C_1$$

denotes the contract

$$(I_2 \,\mathbf{extends}\, I_1 \,\mathbf{with}\, < A_2, M_2 >, \Phi, Init)$$

where

$$Init =_{df} Init_1 \cup Init_2, \quad \Phi =_{df} \Phi_1 \oplus \Phi_2$$

which indicates that Φ_2 can override the definition of a method given by Φ_1. □

The refinement relation is preserved by inheritance.

Theorem 4 *If $C_1 \sqsubseteq C_2$, then*

$$(1) \quad (C \,\mathbf{extends}\, C_1) \sqsubseteq (C \,\mathbf{extends}\, C_2)$$
$$(2) \quad (C_1 \,\mathbf{extends}\, C) \sqsubseteq (C_2 \,\mathbf{extends}\, C)$$

□

We allow to hide some interface methods in a contract in order to offer different services to different customers.

Definition 8 (Hiding) Let $C = (I :< A, M >, \Psi, Init)$ be a contract, and HM a set of methods. The notation $C \backslash HM$ represents the contract

$$(I :< A, M \setminus HM >, (M \setminus HM) \lhd \Psi, Init)$$

where $X \lhd f$ represents the mapping f restricted to the domain X. □

Theorem 5 *The hiding operator enjoys the following properties.*

1. $(C \backslash HM) \sqsubseteq C$
2. $C \backslash \emptyset \equiv C$
3. $C \backslash HM \equiv C \backslash (HM \cap M)$
4. $(C \backslash HM_1) \backslash HM_2 \equiv C \backslash (HM_1 \cup HM_2) \equiv (C \backslash HM_2) \backslash HM_1$
5. $(\|_{k \in K} C_k) \backslash HM \equiv \|_{k \in K} (C_k \backslash HM)$
6. $C_1 \, \mathbf{extends} \, C_2) \backslash HM \equiv (C_1 \backslash HM) \, \mathbf{extends} \, (C_2 \backslash HM)$ □

As for interfaces, we allow multiple inheritance for contracts.

Definition 9 (**Multiple interface inheritance**) Let $\{I_k \mid k \in K\}$ be a family of composable interfaces. Assume that none of attributes defined by I_k is used in A.

The notation $I \, \mathbf{extends} \, \{I_k \mid k \in K\} \, \mathbf{with} < A, \, M >$ represents the interface

$$I \, \mathbf{extends} \uplus_{k \in K} I_k \, \mathbf{with} < A, \, M >$$

Let $C = (I, \, \Phi, \, Init)$ be a contract for I, and $\{C_k = (I_k, \, \Phi_k, \, Init_k >) \mid k \in K\}$ a family of consistent contracts for $\{I_k \mid k \in K\}$. Assume that no attribute defined in I_k appears in the attribute set of I. We define

$$C \, \mathbf{extends} \, \{C_k \mid k \in K\} =_{df} C \, \mathbf{extends} \, \|_{k \in K} \, C_k$$

□

5 Component

This section show how to define components and how to compose components.

5.1 Defining components

Now have defined the basic elements of a component. We slightly generalize the definition of a contract to allow it to have internal methods as well as public methods. However, the internal methods can only be used in the specification of the methods of their own contracts, but they cannot be used in the specification of methods of an external contract.

Definition 10 (Component) A *component* P is a triple $< C, \mathcal{O}, \mathcal{I} >$ where

- C is a generalised contract

  $$(I :< A, PubM, PriM >, \Theta, Init)$$

  where

  (1) A is a set of attributes.

  (2) $PubM$ is a set of public methods.

  (3) $PriM$ is a set of private methods.

  (4) Θ maps each method declared in the set $PubM \cup PriM$ into a pair $(\alpha, \mathtt{P})$, where $\mathtt{P}$ is a program text written in the specification language of [8, 9], and α is the alphabet obtained from A and the input and output parameters of the method.

  We use $ReqM$ to denote the set of all methods which are referenced in the program text $\Theta(op)$ but not contained in $PubM \cup PrivM$, where $op \in PubM \cup PrivM$.

- a set $\mathcal{O}$ of composable *output interfaces* whose merge has $PubM$ as its methods, i.e. $Meth(\uplus\mathcal{O}) = PubM$.

- a set $\mathcal{I}$ of composable *input interfaces*, whose merge has $ReqM$ as its method set, i.e. $Meth(\uplus\mathcal{I}) = ReqM$. □

A contract on $\uplus\mathcal{I}$ is called a *required service* of P, and a contract of the interface $\uplus\mathcal{O}$ a *provided service*. Methods in $\mathcal{I}$ can be seen as *holes* in the component where their specifications or implementation given in another component, can be plugged in. Therefore, the provided service of a component depends on its required service plugged in from other components. This leads to the definition of our semantics of a component.

In the above definition, we introduced private methods so that when we hide a public method, the hidden public method is changed to a private method. This will keep the definition Θ valid as a public method may be called in $\Theta(op)$. We do not introduce private attributes as local variables can be declared in the program text $\Theta(op)$.

Definition 11 (Semantics of a Component) A component P is identified as a binary relation between its required services and their corresponding provided services

$$[\![P]\!](C_{\mathcal{I}},\, C'_{\mathcal{O}}) \;=_{df}\; (C_{\mathcal{I}} >> P) \sqsubseteq C'_{\mathcal{O}}$$

where the variable $C_{\mathcal{I}}$ takes an arbitrary required service as its value, $C'_{\mathcal{O}}$ takes a provided service for $\mathcal{O}$, and the notation $C_{\mathcal{I}} >> P$ denotes the provided service

$$(I :< A,\, PubM >,\, \Psi,\, Init)$$

where the mapping Ψ is defined from the given required service $C =< \uplus\mathcal{I}, \Phi_{\mathcal{I}}, Init_{\mathcal{I}} >$ by the recursive equations

$$\Psi(op) \;=\; \mathcal{M}(\Theta(op))$$

where $\mathcal{M}$ replaces every call of op of $ReqM$ by its corresponding specification.

$$\mathcal{M}(op(inexp,\, outvar)) \;=_{df}\; \left(\begin{array}{l} \textbf{var}\ in,\, out; \\ in := inexp; \\ \Phi_{\mathcal{I}}(op); \\ outvar := out; \\ \textbf{end}\ in,\, out \end{array}\right) \quad \text{if } op(in : U,\, out : V) \in ReqM$$

$$\mathcal{M}(op(inexp,\, outvar)) \;=_{df}\; \left(\begin{array}{l} \textbf{var}\ in,\, out; \\ in := inexp; \\ \Psi(op); \\ outvar := out; \\ \textbf{end}\ in,\, out \end{array}\right) \quad \text{if } op(in : U,\, out : V) \in PubM \cup PrivM$$

$$\mathcal{M}(v := e) \;=_{df}\; v := e$$

$$\mathcal{M}(\mathcal{F}(\mathtt{c})) \;=_{df}\; \mathcal{F}(\mathcal{M}(\mathtt{c})) \text{ for any comand } c \text{ and context } \mathcal{F}\Box$$

Notice that when a component P has an empty set $\varnothing$ of input interfaces, the notation $C_{\varnothing} >> P$ becomes a constant that is the semantics of the closed program P. In the above definition, we have used part of the specification language proposed in [8, 9]. However, here we do not have to require the object-oriented features of that language if the design and implementation of a component are not object-oriented.

In a modular programming paradigm, a component can be designed or implemented as a module in which each of the methods in the output interfaces 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 $\mathcal{O}$:

Class P implements $\mathcal{O}$ $\{$
Attr : A;
public : $m =_{df} \Phi(m)$; for each $m \in PubM$;
private : $op =_{df} \Phi(op)$; for each $op \in PrivM$
$\}$

According to the definition of the language in [8, 9], this class is well-formed only if all the types in A, all the types of parameters of the methods, and the methods called in the definition of each $\Phi(m)$ and $\Phi(op)$ must be declared. Therefore, the component P in fact is a package that contains all these definitions. In [8, 9], we define a program specification in the form

$$ClassDeclarations \bullet P$$

where $ClassDeclarations$ is a sequence of class declarations

$$ClassDecl;\ \ldots;\ ClassDecl$$

Each class declaration $Classdecl$ is of the form

Class N **extends**M $\{$
public : x : $\mathbf{T}_1$
protect : y : $\mathbf{T}_2$
private : z : $\mathbf{T}_3$
public : $m_1(in : \mathbf{U}, out : \mathbf{V}, inou : \mathbf{W})\{Command\}, \ldots,$
protected : .....
private : .....
$\}$

A command in a method definition is composed from a number of primitive commands that are either framed designs for a high level specification, or commands in a Java-like language including calls to methods of a class. The semantics deals with *visibility*, *dynamic binding* and *recursive method calls*. Therefore, that language is expressive enough to specify the *design* of a component. Adding the notation for interfaces and contracts, the extended language provides a formal model for components.

Example Now we define a component GIS in the ParcelCall system to provide the services to customers.

$<<$ **Component** $>> GIS$
Output Interface $CustomerService$
Attr : P : $Set(PName)$; $/**$set of parcel names
S : $Set(CName)$; $/**$set of customer names
$owns$: $CName \times PName$; $/**owns(s,p)$: s owns p
loc : $PName \longmapsto Position$; $/**loc(p)$ returns the location of p
Init : $P = \varnothing \wedge S = \varnothing$;
Meth : $LocalizeParcel((PName\ pId, CNamesId), Position\ location)$:
if $pId \in P \wedge sId \in S \wedge owns(sId, pId)$
$/**$**call required method**
then $location := IParcelInfo.Where(pId)$
else $\perp$
$DispatchParcel(in : (PName\ pId, CName\ sId))$:
if$pId \notin P \wedge sId \notin S$
then $P := P \cup \{pId\} \wedge S := S \cup \{sId\}$;
$owns := owns \cup \{(sId, pId)\}$;
$IParcelInfo.Deal(pId)$
else $\perp$

Input Interface $Parcelinfo$
Attr : P : $Set(PName)$; $/**$set of parcel names
loc : $PName \longmapsto Position$ $/**loc(p)$ 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 component GTS to implement the contract $ParcelInfo$. We first declare

$$
\begin{array}{l}
\textbf{Class}\ Parcel\{ \\
\quad id:\ PName; \\
\quad location = (0,0):\ Position \\
\quad Positionloc()\{ \\
\quad return := location \\
\quad \} \\
\}
\end{array}
$$

Then we specify the component GTS as follows.

$$
\begin{array}{l}
<< \textbf{Component} >> GTS \\
\textbf{Ouput Interface}\ IParcelInfo \\
\quad \textbf{Attr}:\ Parcels:\ Set(Parcel); \\
\quad \textbf{Init}:\ Parcels = \varnothing \\
\quad \textbf{Meth}:\ Deal(in: PName\ pId)\{ \\
\qquad\qquad \textbf{var}\ Parcelp;\ p := \textbf{New}Parcel(pId); \\
\qquad\qquad Parcels := Parcels \cup \{p\};\ \textbf{end}\ p; \\
\qquad\qquad \} \\
\qquad\qquad Where(in:\ PName\ out:\ Position\ location)\{ \\
\qquad\qquad \textbf{var}\ Parcel\ p;\ p := P.find(pId);\ location := p.loc();\ \textbf{end}\ p \\
\qquad\qquad \} \\
\quad \}
\end{array}
$$

Define the refinement mapping ρ from the attributes of $Parcel$ to those of $ParcelInfo$ such that

$$
\begin{array}{l}
\rho(P) =_{df} \{p.id \mid p \in Parcel\} \\
\rho(loc(pId)) = p.location \text{ for all } pId \in P \text{ such that } \exists p \in Parcel.p.id = pId
\end{array}
$$

Then $ParcelInfo \sqsubseteq GTS$.

5.2 Refinement, interface hiding and composition of components

For a component P with output and input interfaces $\mathcal{O}$ and $\mathcal{I}$, we defined earlier the semantics $[\![P]\!]$ as a binary relation between the input services and output services such that with a given required service $C_{\mathcal{I}}$, we should be able to check whether the component realizes a provided service. Obviously, if with a given required service $C_{\mathcal{I}}$ the component P realizes a provided service $C_{\mathcal{O}}$, then with any required service $C'_{\mathcal{I}}$ that refines $C_{\mathcal{I}}$, P will realize $C_{\mathcal{O}}$ too. On the other way around, if P realizes $C_{\mathcal{O}}$ with a given $C_{\mathcal{I}}$, then P realizes any provided service that $C_{\mathcal{O}}$ refines. This is formalized as the following theorem of monotonicity of a component.

Theorem 6 ***(Monotonicity)*** *Let $P =< I, \mathcal{O}, \mathcal{I} >$ and $\sqsubseteq_{\mathcal{I}}$ and $\sqsubseteq_{\mathcal{O}}$ are the refinement relations among contracts of $\mathcal{I}$ and among contracts of $\mathcal{O}$ respectively. Then*

$$\sqsubseteq_{\mathcal{I}} \circ [\![P]\!] \circ \sqsubseteq_{\mathcal{O}} \ = \ [\![P]\!]$$

where $\circ$ denotes relational composition. □

Thus, for any required services $C_{\mathcal{I}} \sqsubseteq C'_{\mathcal{I}}$, and provided services $C_{\mathcal{O}} \sqsubseteq C'_{\mathcal{O}}$, then

$$[\![P]\!](C_{\mathcal{I}}, C'_{O}) \Rightarrow [\![P]\!](C'_{\mathcal{I}}, C_{\mathcal{O}})$$

A component P_1 is a *refinement* of a component P if for any given required service P_1 provides more *refined* services than P. Thus, component can be refined according to the following definition.

Definition 12 (**Component refinement**) Component P_1 is refined by P_2 (denoted by $P_1 \sqsubseteq P_2$), if

1. $\mathcal{I}_1 \ = \ \mathcal{I}_2$,
2. $\mathcal{O}_1 \ = \ \mathcal{O}_2$, and
3. $[\![P_1]\!] \Leftarrow [\![P_2]\!]$ □

Notice that the upwards closure property in the monotonicity theorem also ensures that if a component provides a service with a required service, it will also provide the service with a refined required service.

Definition 13 (**Interface hiding**) Let $P =< C, \mathcal{O}, \mathcal{I} >$ be a component with $C = (I :< A, PubM, PrivM >, \Theta, Init)$. For a set HI of interfaces, let $Meth(HI)$ be the set of the methods in HI. The notation $P \backslash HI$ denotes the component that removes the methods defined by the interfaces of HI from its provided service.

$$P \backslash HI =_{df} < C \backslash HI, \mathcal{O} \backslash HI, \mathcal{I} >$$

where

$$C \backslash HI =_{df} (< I :< A, PubM \backslash Meth(HI), PrivM \cup (HI \triangleleft PubM) >, \Theta, Init)$$

The semantics of $P\backslash HI$ is defined as

$$[\![P\backslash HI]\!] =_{df} [\![P]\!] \circ (C'_{\mathcal{O}} = C_{\mathcal{O}}\backslash Meth(HI))$$

□

We have the following equations for interface hiding.

Theorem 7

1. $P\backslash\emptyset \equiv P$
2. $P\backslash HI \equiv P\backslash(HI \cap \mathcal{O})$
3. $(P\backslash HI_1)\backslash HI_2 \equiv P\backslash(HI_1 \cup HI_2) \equiv (P\backslash HI_2)\backslash HI_1$ □

Now we are ready to define how components can be connected together to make networks of components.

Definition 14 (Network of component)

Let $C_i = (< I_i :< A_i, PubM_i, PrivM_i >, \Theta_i, Init_i)$ be general contracts, $P_i =< C_i, \mathcal{O}_i, \mathcal{I}_i >$, for $i = 1, 2$ be two components. Assume that $\mathcal{I}_1 \cap \mathcal{I}_2 = \emptyset$ and $\mathcal{O}_1 \cap \mathcal{O}_2 = \emptyset$.

The *composition* $P_1 \mid\mid P_2$ 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:

$$P_1 \mid\mid P_2 =_{def} < C_1 \| C_2, \mathcal{O}_1 \uplus \mathcal{O}_2, \mathcal{I}_1\backslash\mathcal{O}_2 \uplus \mathcal{I}_2\backslash\mathcal{O}_1 >$$

Let $\mathcal{I} =_{df} \mathcal{I}_1\backslash\mathcal{O}_2 \uplus \mathcal{I}_2\backslash\mathcal{O}_1$ and $\mathcal{O} =_{df} \mathcal{O}_1 \uplus \mathcal{O}_2$. The behaviour of the network of C_1 and C_2 is defined by

$$\begin{array}{lll} [\![P_1 \| P_2]\!](C_{\mathcal{I}}, C'_{\mathcal{O}}) & =_{df} & \exists C_{\mathcal{I}_1}, C'_{\mathcal{O}_1}, C_{\mathcal{I}_2}, C'_{\mathcal{O}_2} \bullet \\ & & [\![P_1]\!](C_{\mathcal{I}_1}, C'_{\mathcal{O}_1}) \wedge [\![P_2]\!](C_{\mathcal{I}_2}, C'_{\mathcal{O}_2}) \wedge \\ & & C_{\mathcal{I}_1}\backslash Meth(\mathcal{I}_1 \setminus \mathcal{O}_2) = C'_{\mathcal{O}_2}\backslash Meth(\mathcal{O}_2 \setminus \mathcal{I}_1) \wedge \\ & & C_{\mathcal{I}_2}\backslash Meth(\mathcal{I}_2 \setminus \mathcal{O}_1) = C'_{\mathcal{O}_1}\backslash Meth(\mathcal{O}_1 \setminus \mathcal{I}_2) \wedge \\ & & C_{\mathcal{I}} = C_{\mathcal{I}_1}\backslash Meth(\mathcal{I}_1\backslash\mathcal{O}_2) \| C_{\mathcal{I}_2}\backslash Meth(_2\backslash\mathcal{O}_1) \wedge \\ & & C'_{\mathcal{O}} = C'_{\mathcal{O}_1}\backslash Meth(\mathcal{I}_2\backslash\mathcal{O}_1) \| C'_{\mathcal{O}_2}\backslash Meth(\mathcal{I}_1\backslash\mathcal{O}_2) \end{array}$$

□

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: $(P_1 \| P_2)\backslash(\mathcal{I}_1 \cap \mathcal{O}_2)\backslash(\mathcal{I}_2 \cap \mathcal{O}_2)$.

Example With this definition, we can now compose GIS and GTS to make a composite component. $(GIS \mid\mid GTS)\backslash IParcelInfo$. $(GIS \mid\mid GTS)\backslash IParcelInfo$ does not have any required interface if we do not consider the relation between GTS with other components of the ParcelCall system, It then becomes a closed system that only provides services defined by the contract CS for customer service 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.

6 Client-Server Systems

Client-server systems are the most often seen applications in component software. The architecture of such a system is organized as a layered structure as illustrated in Figure 2. On the top of the layers are the clients that only require services from the components in the second layer. Components in a middle layer provide services to components in the layer above, but requires services from the layer below. The components at the bottom are the basic server components that only provide services to the components above the bottom layer. Components of the same layer have disjoint provided interfaces. The layers are are organized according to interface dependencies. The whole system is the composition of the components with the interfaces hidden:

$$Q =_{df} ((P_{11} \mid\mid \cdots \mid\mid P_{1k_1}) \mid\mid \cdots \mid\mid (P_{n1} \mid\mid \cdots \mid\mid P_{nk_n}))\backslash All$$

where All denotes all the linked interfaces. The construction of such a system can be top-down:

$$\begin{aligned} Q_1 &=_{def} (P_{11} \mid\mid \cdots \mid\mid P_{1k} \mid\mid P_{21} \mid\mid \cdots \mid\mid P_{2k_2})\backslash\mathcal{I}_1 \\ Q_2 &=_{df} (Q_1 \mid\mid P_{31} \mid\mid \cdots \mid\mid P_{3k_3})\backslash\mathcal{I}_2 \\ &\cdots\cdots \\ Q_{n-1} &=_{df} (Q_{n-2} \mid\mid P_{n1} \mid\mid \cdots \mid\mid P_{nk_n})\backslash\mathcal{I}_{n-2} \end{aligned}$$

where Q_{n-1} is the resulting component, and $\mathcal{I}ij$ are the linked interfaces.

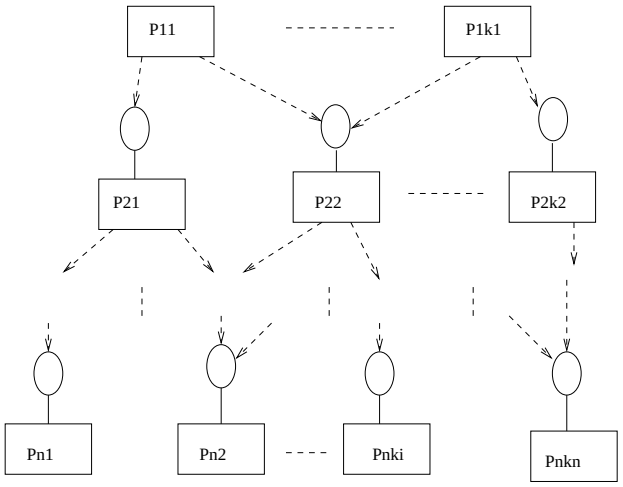

Figure 2: Client-Server Systems

The system construction can also be bottom-up:

$$
\begin{aligned}
S_1 &=_{def} (P_{n1} \,||\, \cdots \,||\, P_{nkn} \,||\, P_{n_1 1} \,||\, \cdots \,||\, P_{n-1k_{n-1}})\backslash\mathcal{I}_1 \\
S_2 &=_{df} (Q_1 \,||\, P_{n-21} \,||\, \cdots \,||\, P_{n_2 k_{n-2}})\backslash\mathcal{I}_2 \\
&\cdots\cdots \\
S_{n-1} &=_{df} (Q_{n-2} \,||\, P_{11} \,||\, \cdots \,||\, P_{1k_1})\backslash\mathcal{I}_{n-2}
\end{aligned}
$$

We have $Q = Q_{n-1} = S_{n-1}$.

7 Conclusion and Discussion

Conclusion We have proposed a model for software components and defined composition and refinement of components. In the model, we separate 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. The design may make use of the provided services of existing components. Such a design can be specified in the object-oriented specification notation developed in [8, 9], that supports step-wise construction of a component. The *verifier* of a component has to take the design specification and the specification of the required interfaces to verify the design is correct with respect to its requirement specification.

When composing components that have been already verified, one has to check the matchability of the provided service of one component with the specification of the required service of another, both syntac-

tically 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 a simplified version in the sense that protocols of the interfaces are not described. This will keep the definition simpler and the task of matchability checking easier. There are several possible ways to address the problem of protocols. First, we can introduce control state variables in an interface and thus contracts and components. This will allow us to define a contract as a state machine or statechart, e.g. [17]. 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 specification of the order of an interface methods in a contract, e.g. [1]. Again, matching between protocols in different components has to be checked and deadlock problem needs to be looked at. As we know from the model of CSP, this is not a trivial task either. One may adopt a compromise approach. That is one should provide the protocol of the provided interface and a protocol of its required interface independently. Such a protocol is then described in terms of *regular language* on the method names of the interface. To check the matchability between a provided interface with a required interface is 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. This procedure can be automated.

Although the model of components does allow mutual dependency between two components and we can in theory check live-lock once the design specifications of the components are given, again, all experts in formal methods are well aware that this is difficult in practice. We suggest the designers try to avoid mutual dependencies in their component software development.

We have not introduced special concepts of *connectors*, e.g. [6, 13]. We treat these as components though they may be special components. It is interesting and important to define a special class of components for connectors and study their properties and uses. We include this in our future work.

The communication between components is modelled as synchronous method invocation. Asynchronous communication can be achieved by introducing buffer components.

Future work also includes applying this framework to bigger case studies, combine it with UML, and provide a formalization to CORBA and COM. We will also looking into adding non-functional features, such as timing and resources into the interfaces of components.

Related work We take the views of [5, 10, 6, 18] about a components that a component both provides to and requires from other components. We define a notation for the specification of provided and required services as *contracts*. Our concept of contracts is similar to that of Meyer [16]. The purpose of Meyer' contracts is to support the development of object methods in the context of client-server relationships between two objects. Our contracts are to support the specification and development services of components. A component does not have to be an object and our approach supports both object-oriented and module-oriented development. Thus, our opinion is that our work is a further development of Meyer's notion of contracts.

A notion of contract is given in [11] to model collaboration and behavioural relationships between objects. In our approach, we provide the separation between the specification of a contract for an interface from the specification of the behaviour of the component that realizes the contract. A notion of contract can also be found in [4] that emerged in the model of action systems. It promotes the separation between the specification of what an agent can do in a system and how they need to be coordinated to achieve the required global behaviour of the system. However, the architectural components there are not explored.

A notion of contracts is defined in [3, 2] with a notation for its specification. A contract therein 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 specifies 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. However, it is interesting to investigate how these two notions of contracts can be combined to provide better support to both system construction and evolution.

A formal model of components can be found in [7] where notions of provided and required interfaces are also used. The aim of the required interface therein is only to support the specification of the design and implementation of a component for a given set of provided services. There is no clear notion of contracts of interfaces in that paper, but a provided interface method of a component is specified in terms method calls to methods in the required interface. Therefore, the specification of the provided services reveals unnecessary implementation details. Our simple relational logic also differs from their modal logic too.

References

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

[2] L. F. Andrade and J.L.Fiadeiro. Contracts: Supporting architectural-based evolution.

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

[4] R.J.R. Back, L. Petre, and I.P. Paltor. Formalizing UML use cases in the refinement calculus. Technical Report 279, Turku Centre for Computer Science, Turku, Finland, May 1999.

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

[6] D. D'Souza and A.C. Wills. *Objects, Components and Framework with UML: The Catalysis Approach*. Addison-Wesley, 1998.

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

[8] J. He, Z. Liu, and X. Li. A relational model for object-oriented programming. Technical Report UNU/IIST Report No 231, UNU/IIST, P.O. Box 3058, Macau, March 2001.

[9] J. He, Z. Liu, and X. Li. Towards a refinement calculus for object-oriented systems. To appear at ICCI02 as a Keynote Talk, August 19-20, 2002, Alberta, Canada, 2002.

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

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

[12] C.A.R. Hoare and J. He. *Unifying theories of programming*. Prentice-Hall International, 1998.

[13] J.L.Fiadeiro and A.Lopes. Semantics of architectural connectors. In M.Bidoit and M.Dauchet, editors, *Proc. TAPSOFT'97, LNCS 1214*. Springer-Verlag, 1997.

[14] Z. Liu, X. Li, and J. He. Using transition systems to unify *uml* models. Technical report, Dept. of Maths and Computer Science, the University of leicester, England., May 2002.

[15] Z. Liu, X. Li, J. He, and Y. Chen. A relational model for object-oriented analysis. Technical Report UNU/IIST Repor No 258, UNU/IIST, P.O. Box 3058, Macau, July 2002.

[16] B. Meyer. Applying design by contract. *IEEE Computer*, May 1992.

[17] B. Selic. Using UML for modelling complex real-time systems. In F. Mueller and A. Bestavros, editors, *Language Compilers, and Tools for Embedded Systems, Volume 1474 of Lecture Notes in Computer Science*, pages 250–262. Springer, 1998.

[18] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley.