Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 160 (2006) 173–195
www.elsevier.com/locate/entcs

A Theory of Reactive Components

He Jifeng[1]

Software Engineering Institute, East China Normal University, Shanghai, China

Xiaoshan Li[1]

Faculty of Science and Technology, University of Macau, Macao SAR, China

Zhiming Liu[1]

International Institute for Software Technology
The United Nations University, Macao SAR, China

Abstract

We present a theory of reactive components. We identify a component by its provided services, and specify the individual services by a guarded-design, which enables one to separate the responsibility of clients from the commitment made by the component, and model the behaviour of a component by a set of failures and divergences. Protocols are introduced to coordinate the interactions between a component and its environment. We adopt the notion of process refinement to formalise the substitutivity of components, and provide a complete proof method based on the notion of simulations. We also study the algebraic properties of component combinators.

Keywords: Interfaces, Contracts, Components, Protocols, Composition

1 Introduction

Software elements are components if they are composable [27]. Composition enables prefabricated components to be reused by rearranging them in ever new composites. To facilitate correct integration and updating, a component description usually includes the following ingredients

- A precise specification of the services that it offers to its clients
- A specification of all its dependencies.

[1] Email: jifeng@sei.ecnu.edu.cn, xsl@umac.mo, lzm@iist.unu.edu. This work is partially supported by the NSF project 60573085, the 211 Key project of the Ministry of Education, the 973 project 2002CB312001 of the Ministry of Science and Technology of China, and the project HighQSoftD funded by Macao Science and Technology Fund.

doi:10.1016/j.entcs.2006.05.022

The specification of a component serves both users and developers. For users, the specification provides a definition of its interface. As an encapsulated module, a component must be usable on the sole basis of its specification, without access to non-interface information such as the source code even though it is available. In principle, the interface is the only document that users can access. Therefore, the specification of interface must be precise and complete. For developers, the specification of a component also provides an abstract definition of its internal structure. Although it is invisible to users, it is useful to developers, at least as documentation of the component. To accommodate these requirements, this paper develops a specification mechanism which integrates the event-based model (serving clients) and the state-based model (serving developers).

This paper presents the following notions for component-based systems that serve different purposes for different people at different system development stages:

- **Reactive design calculus**. We introduce the notion of a *guarded design* (represented by $g\&D$) to specify the behaviours of a service,
  - the guard g is used to describe the *availability* of the service, i.e., the service can be invoked only when g holds initially.
  - the design D specifies the behaviours of execution of the service once it is activated successfully.
- **Contract calculus**. Contracts are specifications of interfaces. A contract associates an interface to an abstract data model plus a set of functional specifications of its services, and as well as an interaction protocol that imposes the order on use of services of the interface.
- **Component calculus**. A component is a software module with a set of *provided services* and a set of *required services*. Semantically it can be identified as a *monotonic* mapping from the contracts of its required services to the contracts of its provided services.

The specifications of components used in practical software development today are limited primarily to what we will call syntactic specifications. This form of specification includes the specifications used with technologies such as COM [22], the Object Management Group's CORBA [23], and Sun's ENTERPRISE JavaBeans [24]. Several techniques for designing component-based systems include semantic specifications. In [5], UML and the Object Constraint Language [29] are used to write component specifications. Another well-known method that uses the same notations is Catalysis [6]. In these frameworks, an interface consists of a set of operations. In addition, a pair of precondition and postcondition is associated with each operation. Note that the idea of pre- and postcondition is widely used in a variety of software techniques, such as the Vienna Development Method [16] and Design by Contract [21].

The novel features of this paper are

(i) It integrates a state-based model of functional behaviour and an event-based model of inter-component interaction. The state-based model is for white-box specification in support of component design, and the event-based model is for

black-box specification used when composing components.

(ii) Our approach facilitates assurance of global refinement by local refinement via integration of the event-based simulation and the state-based refinement. Global refinement is usually defined as a set containment of system behaviours, and can be verified deductively within a theorem prover. Local refinement is based on specification of individual operations, and can be established by simulation techniques using a model checker.

2 Reactive Design Calculus

A component provides a set of services to its environment. The signature of a service is of the form

$$m(in : T_1, out : T_2)$$

where m is the name of the service, and variables *in* and *out* represent the value and result parameters of the service respectively, and T_1 and T_2 are the types of parameters.

2.1 Design

When a service is available, and activated successfully, its behaviour can be described by a *design* [15]

$$D = (\alpha, p \vdash R)$$

where α is the set of all free variables used by predicates p and R, and

(i) p, called the *precondition*, is the assumption on which the service can rely when it is activated.

(ii) R, called the *postcondition*, relates the initial states of the component to its final states. It is the commitment made by the service to its client.

The notation $p \vdash R$ denotes the predicate

$$(ok \wedge p) \Rightarrow (ok' \wedge R)$$

where the boolean variables ok and ok' are present to describe the termination property of execution, and the execution diverges when ok' is $false$.

Design $D_1 =_{df} (\alpha, P_1)$ is refined by design $D_2 =_{df} (\alpha, P_2)$, denoted by $D_1 \sqsubseteq D_2$, if all the observation one can make over the execution of P_2 is permitted by P_1

$$[P_2 \Rightarrow P_1]$$

where the square brackets $[P_2 \Rightarrow P_1]$ denotes universal quantification over all variables of the set α. For notational simplicity, we abbreviate $(\alpha, P_1) \sqsubseteq (\alpha, P_2)$ by $P_1 \sqsubseteq P_2$ in later discussion.

Two designs are *equivalent* if they refine each other

$$D_1 \equiv D_2 \ =_{df} \ (D_1 \sqsubseteq D_2) \wedge (D_2 \sqsubseteq D_1) \qquad \square$$

Lemma 2.1 $(p_1 \vdash R_1) \sqsubseteq (p_2 \vdash R_2)$ *iff*

$$[p_1 \Rightarrow p_2] \quad \textit{and} \quad [(p_1 \wedge R_2) \Rightarrow R_1] \qquad \square$$

Designs are closed under the programming operators [15]. In particular, we define the condition choice and sequential composition as where

$$(P \lhd b \rhd Q) =_{df} (b \wedge P) \vee (\neg b \wedge Q)$$

$$P(s'); Q(s) =_{df} \exists m \bullet (P(m) \wedge Q(m))$$

Lemma 2.2 *(1)* $(p_1 \vdash R_1) \vee (p_2 \vdash R_2) \equiv (p_1 \wedge p_2) \vdash (R_1 \vee R_2)$

(2) $(p_1 \vdash R_1) \lhd b \rhd (p_2 \vdash R_2) \equiv (p_1 \lhd b \rhd p_2) \vdash (R_1 \lhd b \rhd R_2)$

(3) $(p_1 \vdash R_1); (p_2 \vdash R_2) \equiv (p_1 \wedge \neg(R_1; \neg p_2)) \vdash (R_1; R_2),$ $\square$

Corollary 2.3 *If* $\neg(D_1; D_2)[true, false/ok, ok']$, *then*

$$(D_1; D_2)[true, true/ok, ok'] \equiv D_1[true, true/ok, ok']; R_2$$

Proof. From the assumption and Lemma 2.2(3). $\square$

We can treat a design as a predicate transformer by defining

$$\mathbf{wp}(p \vdash R, q) =_{df} p \wedge \neg(R; \neg q)$$

2.2 Reactive Design

A service is not always available to its environment. We introduce a guard to specify the case when a service can be invoked. Because of requirement for this type of synchronisation, a service will usually engage in alternative periods of computation and periods of stability, while it is waiting for interaction from its environment. We therefore introduce into the alphabet of a service a variable $wait'$, which is true just during these stable periods. Its main purpose is to distinguish intermediate observations made on the waiting states from the observations made on terminations. The introduction of intermediate observations has implications for sequential composition: all the intermediate observations of P are of course also intermediate observations of $P; Q$. Rather than change the definition of sequential composition given in Lemma 2.2, we enforce these rules by a healthiness condition. If Q is asked to start in a waiting state of its predecessor, it leaves the state unchanged

$$(R) \qquad Q = (true \vdash wait' \wedge v = v') \lhd wait \rhd Q$$

Definition 2.4 [Reactive design] A design (α, P) is *reactive* if P is a fixed point of the mapping $\mathcal{H}$

$$\mathcal{H}(P) =_{df} (true \vdash wait' \wedge v = v') \lhd wait \rhd P \qquad \square$$

The mapping $\mathcal{H}$ maps each design P to a reactive design and thus for any P, $\mathcal{H}(P)$ satisfies the healthiness condition (R).

Theorem 2.5 $\mathcal{H}^2(P) \equiv \mathcal{H}(P)$ $\square$

Because $\mathcal{H}$ is monotonic, we conclude that reactive designs form a complete lattice.

Theorem 2.6

(1) $\mathcal{H}(P_1) \vee \mathcal{H}(P_2) \equiv \mathcal{H}(P_1 \vee P_2)$

(2) $\mathcal{H}(P_1) \lhd b \rhd \mathcal{H}(P_2) \equiv \mathcal{H}(P_1 \lhd b \rhd P_2)$

(3) $\mathcal{H}(P_1); \mathcal{H}(P_2) \equiv \mathcal{H}(P_1; \mathcal{H}(P_2))$

(4) $\exists x \bullet \mathcal{H}(P) \equiv \mathcal{H}(\exists x \bullet P)$

(5) $\exists x' \bullet \mathcal{H}(P) \equiv \mathcal{H}(\exists x' \bullet P)$ □

Definition 2.7 [Guarded design] Let g be a guard and $D = (\alpha, P)$ a design, the notation $g \& D$ denotes the design

$$(\alpha,\ P \lhd g \rhd (true \vdash wait' \wedge v' = v))$$

Theorem 2.8 *If D is a reactive design, so is $g \& D$* □

Theorem 2.9

(1) $g\&D_1 \vee g\&D_2 \equiv g\&(D_1 \vee D_2)$

(2) $(g_1\&D_1) \lhd b \rhd (g_2\&D_2) \equiv (g_1 \lhd b \rhd g_2)\&(D_1 \lhd b \rhd D_2)$

(3) $(g_1\&D_1 \,;\, g_2\&D_2) \equiv g_1\&(D_1; g_2\&D_2)$ □

In this paper we use a guarded design as the specification of a service.

2.3 Programs as Reactive Design

To verify that a service meets its specification, we are going to embed programs into the domain of reactive designs by providing a denotational semantics for programs.

The execution of *skip* terminates successfully, leaving all program variable unchanged.

$$skip \ =_{df}\ \mathcal{H}(true \vdash (\neg wait' \wedge x' = x \wedge ... \wedge z' = z))$$

Program *stop* stays in the waiting state forever.

$$stop \ =_{df}\ \mathcal{H}(true \vdash wait')$$

chaos is the worst program, i.e., its behaviour is totally unpredictable.

$$chaos \ =_{df}\ \mathcal{H}(false \vdash true)$$

Let x be a variable and e an expression. If both x and e are well-defined, and the type of e matches that of x, then the execution of $x := e$ assigns the value of e to x and leaves other program variables intact.

$$x := e \ =_{df}\ \mathcal{H}(WF(x := e) \vdash (\neg wait' \wedge x' = e \wedge y' = y \wedge ... \wedge z' = z))$$

where the predicate $WF(x := e)$ holds whenever x and e possess the same type, and e is well-defined.

Conditional construct is defined in a usual way

$$\texttt{if}\ b\ \texttt{then}\ P\ \texttt{else}\ Q \ =_{df}\ P \lhd b \rhd Q$$

Sequential composition and non-deterministic choice have the same definitions as presented in Lemma 2.2. Theorem 2.6 indicates that all the combinators are

closed in the domain of reactive designs.

The declaration command $\texttt{var}\, x$ introduces a new variable x

$$\texttt{var}\, x \ =_{df} \ \exists x \bullet skip$$

Undeclaration $\texttt{end}\, x$ ends the scope of variable x.

$$\texttt{end}\, x \ =_{df} \ \exists x' \bullet skip$$

The iteration construct $\texttt{while}\ b\ \texttt{do}\ P$ is defined as the worst fixed point of the recursive equation in the complete lattice of reactive designs

$$X \ = \ (P; X) \lhd b \rhd skip$$

Assume that the service $m(in : T_1, out : T_2)$ has the pair $g\&D$ as its specification. Then the execution of service request $m(e, v)$ can be identified with the following reactive design

$$m(e, v) \ =_{df} \ \left(\begin{array}{c} \texttt{var}\ in, out; in := e; \\ \\ g\&D; \\ \\ v := out; \texttt{end}\ in, out \end{array} \right)$$

3 Contracts

A component usually provides a collection of access points, called an *interface*, to its environment.

3.1 Interfaces

An *interface* is a syntactic specification of the access point of a component [27]. When a component has multiple access points, each representing a different service offered by the component, then it is expected to have multiple interfaces. An interface offers no implementation of any of its services. Instead it merely names a collection of services and provides only their signatures:

$$\langle FDec, \ MDec \rangle$$

where $FDec$ denotes a set of fields $x : T$ where field x is declared to have type T, and $MDec$ denotes a set of services

Two interfaces are *composable* if they do not use the same filed name with different type. If I and J are composable, then the notation $I \uplus J$ denotes the composite interface with

$$FDec =_{df} I.FDec \cup J.FDec \qquad MDec =_{df} I.MDec \cup J.MDec$$

Like the set union operator, the interface combinator $\uplus$ is idempotent, symmetric and associative.

Definition 3.1 [Interface inheritance] Let I and J be interfaces. Assume that no field of J is redefined in I. We use the notation $I\,\textbf{extends}\,J$ to represent the

interface with the following field and service sectors

$$FDec =_{df} I.FDec \cup J.FDec$$
$$MDec =_{df} I.MDec \cup \{m(x : U, y : V) \mid m(x : U, y : V) \in J.MDec \wedge m \notin I.MDec\}$$

Theorem 3.2

(1) $I \textbf{ extends } (J_1 \uplus J_2) = (I \textbf{ extends } J_1) \uplus (I \textbf{ extends } J_2)$

(2) $(I_1 \uplus I_2) \textbf{ extends } J = (I_1 \textbf{ extends } J) \uplus (I_2 \textbf{ extends } J)$ □

Definition 3.3 [Hiding] Let I be an interface and S a set of service names. The notation $I\backslash S$ denotes the interface after removal of the services of S from I.

$$I\backslash S.FDec =_{df} I.FDec \quad I\backslash S.MDec =_{df} I.MDec \setminus S$$

Theorem 3.4

(1) $(I\backslash S_1)\backslash S_2 = I\backslash(S_1 \cup S_2)$

(2) $I\backslash\emptyset = I$

(3) $(I_1 \uplus I_2)\backslash S = (I_1\backslash S) \uplus (I_2\backslash S)$

(4) $(I \textbf{ extends } J)\backslash S = (I\backslash S) \textbf{ extends } (J\backslash S)$

3.2 Contracts

The specification of the provided services of a component is given by a contract.

Definition 3.5 [Contract] A *contract* Ctr is a quadruple $(I, Init, Spec, Prot)$ where

(i) I is an interface.

(ii) $Spec$ maps each service m of I to its specification $g_m \& D_m$,

(iii) $Init$ is a design characterising the initial states of fields.

$$Init = true \vdash (init(x') \wedge \neg wait')$$

where x represents the fields of interface I.

(iv) $Prot$ is a set of sequences of service requests $\langle ?m_{i_1}(x_{i_1}), ..., ?m_{i_k}(x_{i_k})\rangle$, standing for the interaction protocol between the contract with its the environment, where $?m_i(x_i)$ represents an invocation event of service m_i with the input value x_i.

3.3 Dynamic Behaviour

A component interacts with its environment via its access points. Its behaviours can be recorded by a sequence of service invocations. Furthermore, we also want to model the potential failures exhibited during its execution: *deadlock* and *livelock*. In a summary, the dynamic behaviours of a contract can be described by the triple

$$(\mathcal{D}, \mathcal{F}, Prot)$$

The set $\mathcal{D}$ consists of the sequences of invocations of services whose execution ends with a divergent state

$$\mathcal{D} =_{df} \{\langle ?m_1(x_1), !m_1(y_1), \ldots, ?m_k(x_k)\rangle \cdot s\} \mid \exists \begin{pmatrix} v,\, v', \\ y_k,\, wait' \end{pmatrix} \bullet \begin{pmatrix} Init; \\ (g_{m_1} \& D_{m_1})[x_1,\, y_1/x,\, y']; \\ \ldots; \\ (g_{m_k} \& D_{m_k})[x_k,\, y_k/x,\, y'] \end{pmatrix} \begin{bmatrix} true/ok, \\ false/ok' \end{bmatrix} \}$$

where $!m_i(y_i)$ represents the output event which delivers the return value y_i to the caller of service m, v and v' are the field variables and their primed versions, x and y are the formal input and output parameters of the methods.

$\mathcal{F}$ is the set of pairs $(s,\, X)$, which describes the situation when the component may refuse to engage in the events of set X after performing all services recorded in sequence s. It consists of the following five cases

(i) s is a divergent trace

$$\{(s,\, X) \mid s \in \mathcal{D}\}$$

(ii) In the initial state the service of set X are not available, i.e., their guards are false

$$\{(\langle\rangle,\, X) \mid \exists v' \bullet Init[true,\, false,\, true,\, false/ok,\, wait,\, ok',\, wait'] \ \wedge \ \forall ?m \in X \bullet \neg g_m[v'/v]\}$$

(iii) On completion of execution of s, the services of set X become inaccessible.

$$\{\langle ?m_1(x_1),\, !m_1(y_1), ..., ?m_k(x_k),\, !m_k(y_k)\rangle,\, X) \mid \exists v' \bullet (Init; (g_{m_1} \& D_{m_1})[x_1, y_1/x, y']; \ldots; (g_{m_k} \& D_{m_k})[x_k,\, y_k/x,\, y']) [true,\, false,\, true,\, false/ok,\, wait,\, ok',\, wait'] \ \wedge \ \forall ?m \in X \bullet \neg g_m[v'/v]\}$$

(iv) The execution of service m_k is ready to deliver its outcome

$$\{\langle ?m_1(x_1),\, !m_1(y_1), ..., ?m_k(x_k)\rangle,\, X) \mid \exists v',\, y_k \bullet (Init; (g_{m_1} \& D_{m_1})[x_1, y_1/x, y']; ...; (g_{m_k} \& D_{m_k})[x_k, y_k/x, y']) [true,\, false,\, true,\, false/ok,\, wait,\, ok',\, wait'] \ \wedge \ !m_k(y_k) \notin X\}$$

(v) The execution of service m_k enters a waiting state

$$\{\langle ?m_1(x_1),\, !m_1(y_1), ..., ?m_k(x_k)\rangle,\, X) \mid \exists v',\, y_k \bullet \begin{pmatrix} Init; \\ (g_{m_1} \& D_{m_1})[x_1, y_1/x, y']; \\ ...; \\ (g_{m_{k-1}} \& D_{m_{k-1}})[x_{k-1},\, y_{k-1}/x,\, y'])[true,\, false/ok',\, wait']; \\ (g_{m_k} \& D_{m_k})[x_k, y_k/x, y'] \end{pmatrix} [W]\}$$

where, $W =_{df} [true, false, true, true/ok, wait, ok', wait']$

We define $Traces =_{df} \{s \mid \exists X \bullet (s, X) \in \mathcal{F}\}$

Definition 3.6 [Consistency] A contract Ctr is consistent (denoted by $Consistent(Ctr)$), if it will never get stuck unless its environment violates the protocol, i.e., for all $\langle ?m_1(x_1), ..., ?m_k(x_k)\rangle$ in $Prot$

$$\exists y_k \bullet \mathbf{wp}\left(\begin{pmatrix} Init; \\ (g_{m_1} \& D_{m_1})[x_1, y_1/x, y']; \\ ...; \\ (g_{m_k} \& D_{m_k})[x_k, y_k/x, y'] \end{pmatrix}, \neg wait \wedge \exists m \bullet g_m\right) = true$$

Lemma 3.7 *Ctr is consistent iff for all $tr \in Prot$ the following holds*

$$\exists s \in Traces \bullet s \downarrow \{?\} = tr \wedge$$

$$(\forall (s, X) \in \mathcal{F} \bullet s \downarrow \{?\} \preceq tr \Rightarrow X \neq \{?m, !m \mid m \in MDec\})$$

where $s \downarrow \{?\}$ denotes the subsequence of s consisting of its input events, and $s \preceq t$ indicates that s is a prefix of t. □

Theorem 3.8

(1) Let $Ctr_i = (I, Init, Spec, Prot_i)$ for $i = 1, 2$. Then
$Consist(Ctr_1) \wedge Consist(Ctr_2) \Rightarrow Consist(I, Init, Spec, Prot_1 \cup Prot_2)$

(2) If Ctr is consistent and $Prot_1 \subseteq Prot$, then $(I, Init, Spec, Prot_1)$ is also consistent.

(3) Let $Ctr_i = (I, Init_i, Spec_i, Prot)$ and $Spec_i(m) = g_m \& D_{i,m}$.
If $(D_{1,m} \sqsubseteq D_{2,m})$ for all $m \in MDec$ and $Init_1 \sqsubseteq Init_2$,
then $Consist(Ctr_1) \Rightarrow Consist(Ctr_2)$ □

Given the specifications of component services and initial state, it is possible to find a corresponding protocol such that they form a consistent contract.

Theorem 3.9 (Weakest protocol) *$(I, Init, Spec, Prot)$ is consistent iff $(Prot \subseteq WProt)$, where*

$WProt =_{df} \{\langle ?m_1(x_1), ..., ?m_k(x_k)\rangle \mid$

$$\exists y_k \bullet \mathbf{wp}\left(\begin{pmatrix} Init; \\ (g_{m_1} \& D_{m_1})[x_1, y_1/x, y']; \\ ...; \\ (g_{m_k} \& D_{m_k})[x_k, y_k/x, y'] \end{pmatrix}, \neg wait \wedge \exists m \bullet g_m\right) = true\}$$ □

Corollary 3.10 *$WProt$ is prefix-closed.* □

We will use $(I, Init, Spec)$ to stand for the contract $(I, Init, Spec, WProt)$ in later discussion.

3.4 Contract Refinement

Definition 3.11 [Contract refinement] Contract Ctr_1 *is refined* by contract Ctr_2, denoted by $Ctr_1 \sqsubseteq_{ctr} Ctr_2$, if

(i) $MDec_1 = MDec_2$

(ii) $\mathcal{D}(Ctr_1) \supseteq \mathcal{D}(Ctr_2)$

(iii) $\mathcal{F}(Ctr_1) \supseteq \mathcal{F}(Ctr_2)$

Ctr_1 and Ctr_2 are equivalent (denoted by $Ctr_1 \equiv_{ctr} Ctr_2$), if they refine each other.

To establish a refinement relation between contracts it is required to construct the divergent traces and failures sets of contracts. The following theorems show how to compare the dynamic behaviours of contracts by linking the specifications of their services via simulations.

Theorem 3.12 (Downward simulation implies refinement) *Ctr^1 is refined by Ctr^2 if there exists a total mapping $\rho(u, v')$ from $FDec^1$ onto $FDec^2$ satisfying the following conditions*

(i) $[init^2 \Rightarrow (init^1; \rho)]$

(ii) $(g^1_m \wedge D^1_m); \mathbf{sim} \sqsubseteq \mathbf{sim}; (g^2_m \wedge D^2_m)$, *and*

(iii) $[\rho \Rightarrow (g^1_m \equiv g^2_m)]$

where $\mathbf{sim} =_{df} true \vdash (\rho(u, v') \wedge wait' = wait)$.

Proof. From (1) it follows that for $i = 1, 2$.

$$(Init^1; \mathbf{sim}) \sqsubseteq Init^2$$

Let

$$OP^i =_{df} (g_{m_1} \& D^i_{m_1}); ...; (g_{m_k} \& D^i_{m_k})$$

From (ii) and (iii) one can show that

$$(Init^1; OP^1; \mathbf{sim}) \sqsubseteq (Init^2; OP^2)$$

First we are going to show that $\mathcal{D}(Ctr_2) \subseteq \mathcal{D}(Ctr_1)$:

From definition of $\mathcal{D}$, we have

$\langle ?m_1(x_1), !m_1(y_1), ..., ?m_k(x_k), !m_k(y_k)\rangle \cdot s \in \mathcal{D}(Ctr_2)$

{Definition of $\mathcal{D}$}

$\Longrightarrow \exists v,\, v',\, wait' \bullet (Init^2; ...; (g_{m_k} \& D^2_{m_k}[x_k/x]))[true,\, false/ok,\, ok']$

$\{(g \& P)[false/wait'] = (g \wedge P)[false/wait']\}$

$\Longrightarrow \exists v,\, v',\, wait' \bullet (Init^2; ...; (g_{m_k} \wedge D^2_{m_k}[x_k/x]))[true,\, false/ok,\, ok']$

{Definition of OP^i}

$\Longrightarrow \exists u,\, u',\, wait' \bullet (Init^1; ...; (g_{m_k} \wedge D^1_{m_k}[x_k/x]); \mathbf{sim})[true,\, false/ok,\, ok']$

{Definition of **sim**}

$\Longrightarrow \exists u,\, u',\, wait' \bullet (Init^1; ...; (g_{m_k} \wedge D^1_{m_k}[x_k/x])[true,\, false/ok,\, ok']$

$\{(g \& P)[false/wait'] = (g \wedge P)[false/wait']\}$

$\Longrightarrow \exists u,\, u',\, wait' \bullet (Init^1; ...; (g_{m_k} \wedge D^1_{m_k}[x_k/x])[true,\, false/ok,\, ok']$

{Definition of $\mathcal{D}$}

$\Longrightarrow \langle ?m_1(x_1), !m_1(y_1), ..., ?m_k(x_k), !m_k(y_k)\rangle \cdot s \in \mathcal{D}(Ctr_1)$

Now we are going to prove $\mathcal{F}(Ctr_2) \subseteq \mathcal{F}(Ctr_1)$. Let us examine the following three cases:

Case 1 $(\langle\rangle,\, X) \in \mathcal{F}(Ctr^2)$ {Definition of $\mathcal{F}$}

$\Longrightarrow$ $\exists v' \bullet (Init^1[true,\, false,\, true,\, false/ok,\, wait,\, ok',\, wait'] \wedge$

$\forall ?m \in X \bullet \neg g^2_m[v'/v]$ $\{(a)\}$

$\Longrightarrow$ $\exists u',\, v' \bullet Init^1[true,\, false,\, true,\, false/ok,\, wait,\, ok',\, wait'] \wedge$

$\rho(u',\, v') \wedge \forall ?m \in X \bullet g^2_m[v'/v])$ $\{(3)\}$

$\Longrightarrow$ $\exists u' \bullet Init^1[true,\, false,\, true,\, false/ok,\, wait,\, ok',\, wait'] \wedge$

$\forall ?m \in X \bullet g^1_m(u'))$ {Definition of $\mathcal{F}$}

$\Longrightarrow$ $(\langle\rangle,\, X) \in \mathcal{F}(Ctr^1)$

Case 2 $(\langle ?m_1(x_1), !m_1(y_1), ..., ?m_k(x_k), !m_k(y_k)\rangle, X) \in \mathcal{F}(Ctr^2)$

$\{(g\&P)[false/wait'] = (g \wedge P)[false/wait']\}$

$\Longrightarrow$ $\exists v' \bullet ((Init^2; ...; (g^2_{m_k} \wedge D^2_{m_k}))[x_k, y_k/x, y'])$

$[true, false, true, false/ok, wait, ok', wait'] \wedge$

$\forall ?m \in X \bullet \neg g^2_m[v'/v]$ $\{(b)$ and Corollary of Lemma 2.2$\}$

$\Longrightarrow$ $\exists u', v' \bullet (Init^1; ...; (g^1_{m_k} \wedge D^1_{m_k})[x_k, y_k/x, y'])$

$[true, false, true, false/ok, wait, ok', wait'] \wedge$

$\rho(u', v') \wedge \forall ?m \in X \bullet g^2_m[v'/v])$ $\{(3)\}$

$\Longrightarrow$ $\exists u' \bullet (Init^1; ...; (g^1_{m_k} \wedge D^1_{m_k})[x_k, y_k/x, y'])$

$[true, false, true, false/ok, wait, ok', wait'] \wedge$

$\forall ?m \in X \bullet g^1_m(u'))$ $\{(g\&P)[false/wait'] = (g \wedge P)[false/wait']\}$

$\Longrightarrow$ $(\langle ?m_1(x_1), !m_1(y_1), ..., ?m_k(x_k), !m_k(y_k)\rangle, X) \in \mathcal{F}(Ctr^1)$

Case 3 $\langle ?m_1(x_1), !m_1(y_1), ..., ?m_k(x_k)\rangle, X) \in Failures(Ctr_2)$

$\{(g\&P)[false/wait'] = (g \wedge P)[false/wait']\}$

$\Longrightarrow$ $\exists v', y' \bullet (Init^2; ...; (g^2_{m_k} \wedge D^2_{m_k})[x_k/x])$

$[true, false, true, false/ok, wait, ok', wait'] \wedge !m_k \notin X$

$\{(b)$ and Corollary of Lemma 2.2$\}$

$\Longrightarrow$ $\exists u', v', y' \bullet (Init^1; ...; (g^1_{m_k} \wedge D^1_{m_k})[x_k/x])$

$[true, false, true, false/ok, wait, ok', wait'] \wedge$

$\rho(u', v') \wedge !m_k \notin X)$

$\{(g\&P)[false/wait'] = (g \wedge P)[false/wait']\}$

$\Longrightarrow$ $\langle ?m_1(x_1), !m_1(y_1), ..., ?m_k(x_k)\rangle, X) \in \mathcal{F}(Ctr_2)$

□

Theorem 3.13 (Upward simulation implies refinement) *Ctr^1 is refined by Ctr^2 if there exists a total surjective mapping $\rho(v, u')$ from $FDec^2$ onto $FDec^1$ satisfying the following conditions*

(1) $[(init^2; \rho) \Rightarrow init^1]$

(2) $(g^2_m \wedge D^2_m); (true \vdash \rho \wedge wait' = wait) \sqsupseteq (true \vdash \rho \wedge wait' = wait); (g^1_m \wedge D^1_m)$, *and*

(3) $[\exists u \bullet \rho(v, u) \wedge \forall m \bullet (g^1_m(u) \equiv g^2_m(v))]$

Proof. Similar to Theorem 3.12. □

In [10] it is shown that downward simulation together upward simulation forms

a complete proof method for contract refinement.

Theorem 3.14 (Completeness of simulations) *If $Ctr_1 \sqsubseteq_{ctr} Ctr_2$, then there exists a contract Ctr' such that*

(1) There is a upward simulation from Ctr' to Ctr_1

(2) There is a downward simulation from Ctr' to Ctr_2 □

3.5 Contract Composition

Definition 3.15 [Service hiding] Let Ctr be a contract, and S a subset of $MDec$. $Ctr\backslash S$ removes the services of S from the contract Ctr

$\mathcal{D}(Ctr\backslash S) =_{df}$

$\{s \mid s \in \mathcal{D}(Ctr) \wedge s \in \{?m(x), !m(y) \mid m \in MDec \setminus S\}^*\}$

$\mathcal{F}(Ctr\backslash S) =_{df}$

$\{(s, X) \mid (s, X) \in \mathcal{F}(Ctr) \wedge s \in \{?m(x), !m(y) \mid m \in MDec \setminus S\}^* \wedge$

$X \subseteq \{?m, !m \mid m \in MDec \setminus S\}\}$

Theorem 3.16 *If $Ctr_1 \sqsubseteq_{ctr} Ctr_2$ then $Ctr_1\backslash S \sqsubseteq_{ctr} Ctr_2\backslash S$.*

Proof. We are going to show $\mathcal{D}(Ctr_2\backslash S) \subseteq \mathcal{D}(Ctr_2\backslash S)$

$s \in \mathcal{D}(Ctr_2\backslash S)$ {Def 3.6}

$\Longrightarrow s \in \mathcal{D}(Ctr_2) \wedge$

$s \in \{?m(x), !m(y) \mid m \in MDec_2 \setminus S\}^*\}$ $\{Ctr_1 \sqsubseteq_{ctr} Ctr_2\}$

$\Longrightarrow s \in \mathcal{D}(Ctr_1) \wedge$

$s \in \{?m(x), !m(y) \mid m \in MDec_2 \setminus S\}^*\}$ $\{MDec_1 = MDec_2\}$

$\Longrightarrow s \in \mathcal{D}(Ctr_1\backslash S)$

Similarly we can show $\mathcal{F}(Ctr_2\backslash S)$ is a subset of $\mathcal{F}(Ctr_1\backslash S)$. □

Theorem 3.17

(1) $Ctr\backslash S_1\backslash S_2 \equiv_{ctr} Ctr\backslash S_2\backslash S_1$

(2) $Ctr\backslash\emptyset \equiv_{ctr} Ctr$

Proof. We will show that $\mathcal{D}(Ctr\backslash S_1\backslash S_2) = \mathcal{D}(Ctr\backslash S_2\backslash S_1)$.

$$
\begin{array}{ll}
& s \in \mathcal{D}(Ctr \backslash S_1 \backslash S_2) \qquad \{\text{Def 3.6}\} \\
\equiv & s \in \mathcal{D}(Ctr \backslash S_1) \wedge \\
& s \in \{?m(x),\, !m(y) \mid m \in (Ctr \backslash S_1).MDec \setminus S_2\}^*\} \qquad \{\text{Def 3.6}\} \\
\equiv & s \in \mathcal{D}(Ctr) \wedge \\
& s \in \{?m(x),\, !m(y) \mid m \in Ctr.MDec \setminus S_1\}^* \wedge \\
& s \in \{?m(x),\, !m(y) \mid m \in Ctr.MDec \setminus (S_1 \cup S_2)\}^*\} \quad \{S_1 \subseteq (S_1 \cup S_2)\} \\
\equiv & s \in \mathcal{D}(Ctr) \wedge \\
& s \in \{?m(x),\, !m(y) \mid m \in Ctr.MDec \setminus (S_1 \cup S_2)\}^* \\
\equiv & s \in \mathcal{D}(Ctr \backslash S_2 \backslash S_1)
\end{array}
$$

□

All the services declared by an interface are *public*, i.e, they are accessible by the environment of the interface. In the following we introduce the notion of *private services*, which are only visible within the contract.

Definition 3.18 [General contracts] A general contract $GCtr$ is a contract extended with a set $PriMDec$ of private services and their specifications $PriSpec$.

$$(Ctr,\, PriMDec,\, PriSpec)$$

where we assume that $Ctr.MDec$ and $PriMDec$ are disjoint. Define

$$\mathcal{D}(GCtr) =_{df} \mathcal{D}(Ctr)$$
$$\mathcal{F}(GCtr) =_{df} \mathcal{F}(Ctr)$$

Theorem 3.19 (General contract refinement) $GCtr_1 \sqsubseteq_{ctr} GCtr_2$ *iff* $Ctr_1 \sqsubseteq_{ctr} Ctr_2$ □

Definition 3.20 [Hiding] Let $GCtr = (Ctr,\, PriMDec,\, PriMSpec)$ be a general contract, and S a subset of its public services. The notation $GCtr \backslash S$ represents the general contract

$$(Ctr \backslash S,\, PriMDec \cup S,\, PriSpec \cup (S \lhd Spec))$$

For the notational convenience we define

$$Ctr \Downarrow S =_{df} Ctr \backslash (Ctr.MDec \setminus S)$$

Theorem 3.21 *If* $GCtr_1 \sqsubseteq_{ctr} GCtr_2$, *then* $GCtr_1 \Downarrow S \sqsubseteq_{ctr} GCtr_2 \Downarrow S$

Proof. From Theorem 3.16. □

Theorem 3.22

(1) $GCtr \backslash S_1 \backslash S_2 \equiv_{ctr} GCtr \backslash S_2 \backslash S_1$

(2) $GCtr \backslash \emptyset \equiv_{ctr} GCtr$

Proof. From Theorem 3.17. □

Definition 3.23 [Composition] Let $GCtr_1$ and $GCtr_2$ be general contracts. Assume that

(1) all the shared fields have the same type.

(2) all the shared methods have the same definition.

(3) $Init_1$ and $Init_2$ are consistent; i.e. $(Init_1 \wedge Init_2)$ is a feasible design.

We use u and v to stand the variables in the sets $FDec_1 \setminus FDec_2$ and $FDec_2 \setminus FDec_1$ respectively, and define

$$(g\&D)_{+x} =_{df} g\&(D \wedge true \vdash (x' = x))$$

Then the notation $GCtr_1 \parallel GCtr_2$ denotes the general contract

$FDec =_{df} FDec_1 \cup FDec_2$

$MDec =_{df} MDec_1 \cup MDec_2$

$Init =_{df} Init_1 \wedge Init_2$

$$Spec(m) =_{df} \begin{cases} GCtr_1.Spec(m)_{+v} & m \in MDec_1 \\ GCtr_2.Spec(m)_{+u} & m \in MDec_2 \end{cases}$$

$PriMDec =_{df} (PriMDec_1 \cup PriMDec_2) \setminus MDec$

$$PriSpec(m) =_{df} \begin{cases} GCtr_1.PriSpec(m)_{+v} & m \in PriMDec_1 \setminus MDec \\ GCtr_2.PriSpec(m)_{+u} & m \in PriMDec_2 \setminus MDec \end{cases}$$

Theorem 3.24 *If* $GCtr_1 \sqsubseteq_{ctr} GCtr_2$
and $(GCtr_1.FDec \cup GCtr_2.FDec) \cap GCtr_3.FDec = \emptyset$
then $(GCtr_1 \parallel GCtr_3) \sqsubseteq_{ctr} (GCtr_2 \parallel GCtr_3)$ □

Theorem 3.25

(1) $(GCtr_1 \parallel GCtr_2)\backslash S \equiv_{ctr}$
$(GCtr_1\backslash(S \cap GCtr_1.MDec)) \parallel (GCtr_2\backslash(S \cap GCtr_2.MDec))$

(2) $(GCtr_1 \parallel GCtr_2) \equiv_{ctr} (GCtr_2 \parallel GCtr_1)$

(3) $(GCtr_1 \parallel GCtr_2) \parallel GCtr_3 \equiv_{ctr} GCtr_1 \parallel (GCtr_2 \parallel GCtr_3)$ □

4 Component

A component C is a tuple

$$(I, Init, Code, PriMDec, PriCode, InMDec)$$

where

(1) I is an interface listing all the provided services of C.

(2) The tuple (I, $Init$, $Code$, $PriMDec$, $PriCode$) has the same structure as a generalised contract, except that the functions $Code$ and $PriCode$ map a service m to its code

$$(\alpha_m,\ g_m,\ code_m)$$

(3) $InMDec$ denotes the set of all the required services.

4.1 Dynamic Behaviours of a Component

During the execution of a provided service, it may invoke some required services. As a result, the dynamic behaviours of a component depend on the specification of its required services.

Definition 4.1 [Behaviours of a component] Let C be a component, and $InCtr$ be a contract satisfying

$$C.InMDec = InCtr.MDec$$

Without loss of generality, we assume that $C.I$ and $In.I$ are disjoint. The notation $C(InCtr)$ represents the general contract ($OutCtr$, $PriMDec$, $PriSpec$)

$$\begin{aligned}
OutCtr.FDec &=_{df} C.FDec \cup InCtr.FDec\\
OutCtr.MDec &=_{df} C.MDec \cup InCtr.MDec\\
OutCtr.Init &=_{df} C.Init \wedge InCtr.Init\\
OutCtr.Spec &=_{df} C.MDec \lhd \Phi \cup InCtr.Spec\\
OutCtr.PriMDec &=_{df} C.PriMDec\\
OutCtr.PriSpec &=_{df} C.PriMDec \lhd \Phi
\end{aligned}$$

where the function Φ maps every (provided and internal) service m of the component to a guarded design $g_m\&\mathbf{Beh}(code_m)$:

$$\mathbf{Beh}(m(inexp;\ outvar)) =_{df} \left(\begin{array}{l}\mathbf{var}\ in,\ out;\ in := inexp;\\ g\&P;\\ outvar := out;\mathbf{end}\ in,\ out\end{array}\right)$$

if $m(in : U;\ out : V) \in InMDec$

and $InSpec(m) = g\&P$

$$\mathbf{Beh}(m(inexp,\ outvar)) =_{df} \left(\begin{array}{l}\mathbf{var}\ in,\ out;\ in := inexp;\\ g\&\mathbf{Beh}(code);\\ outvar := out;\\ \mathbf{end}\ in,\ out\end{array}\right)$$

$$\text{if } m \in MDec \cup PriMDec$$
$$\text{and } MCode(op) = (\alpha,\ g,\ code)$$

$$\mathbf{Beh}(c) =_{df} c \quad \text{if } c \text{ is a program}$$

In this way, a component maps a contract of its required services to a contract of its provided services.

A component is a monotonic mapping with respect to the refinement ordering of contracts.

Theorem 4.2 (Monotonicity) *If* $InCtr_1 \sqsubseteq_{ctr} InCtr_2$, *then* $C(Ctr_1) \sqsubseteq_{ctr} C(Ctr_2)$.

Proof. Assume that there is a (downward or upward) simulation $\rho(u, v')$ between $InCtr_1$ and $InCtr_2$. Let x be the field variables declared in the interface of C. It can be shown that the mapping $\rho(u, v') \wedge x = x'$ is also a simulation between the contracts $C(Ctr_1)$ and $C(Ctr_2)$, i.e., $C(Ctr_1)$ is refined by $C(Ctr_2)$. The conclusion follows from Theorem 3.12. □

Definition 4.3 [Component refinement] A component C_1 is refined by C_2 (denoted by $C_1 \sqsubseteq_{comp} C_2$), if

(1) $C_1.I.MDec = C_2.I.MDec$,

(2) $C_1.InMDec = C_2.InMDec$, and

(3) For all the input contracts $InCtr$

$$C_1(InCtr) \sqsubseteq_{ctr} C_2(InCtr)$$

Theorem 4.4 $\sqsubseteq_{comp}$ *is transitive.* □

4.2 Component Composition

Definition 4.5 [Hiding] Let C be a component, and $S \subseteq C.MDec$, then the notation $C\backslash S$ represents the component after removal of service of S from its provided service:

$$I =_{df} (C.FDec,\ C.MDec \setminus S)$$
$$Init =_{df} C.Init$$
$$Code =_{df} (C.MDec \setminus S) \lhd C.Code$$
$$PriMDec =_{df} C.PriMDec \cup S$$
$$PriCode =_{df} C.PriCode \cup S \lhd C.Code$$
$$InMDec =_{df} C.InMDec$$

Theorem 4.6 (Component hiding vs contract projection)
$(C\backslash S)(InCtr) \equiv_{ctr} C(InCtr)\backslash S$

Proof. From the definition of $C(Ctr)$ it follows that for all $m \in C.MDec \setminus S$

$$C(InCtr).OutCtr.Spec(m) = (C\backslash S)(InCtr).OutCtr.Spec(m)$$

and for all $m \in C.MDec \cap S$

$$C(InCtr).OutCtr.Spec(m) = (C\backslash S)(InCtr).PriSpec(m)$$

which leads to the conclusion. □

The hiding operator is monotonic.

Theorem 4.7

(1) If $C_1 \sqsubseteq_{comp} C_2$, *then* $(C_1\backslash S) \sqsubseteq_{comp} (C_2\backslash S)$

(2) $C\backslash\emptyset \equiv_{comp} C$

(3) $(C\backslash S_1)\backslash S_2 \equiv_{comp} C\backslash(S_1 \cup S_2) \equiv_{comp} (C\backslash S_2)\backslash S_1$

Proof. The conclusion (1) follows from Theorem 4.6 and Theorem 3.21. The conclusion (2) follows from Theorem 4.6 and Theorem 3.22(2). The conclusion (3) comes from Theorem 3.22(1). □

Definition 4.8 [Chaining] Let C_1 and C_2 be components satisfying the following conditions

(1) None of the provided or private methods of C_2 appears in C_1.

(2) C_1 and C_2 have disjoint field sectors and required service sectors.

In this case, the notation $C_1\rangle\rangle C_2$ represents the composite component connects the provided services of C_1 to the input services of C_2, and is defined by

$$\begin{aligned}
FDec &=_{df} C_1.FDec \cup C_2.FDec \\
MDec &=_{df} C_1.MDec \cup C_2.MDec \\
Init &=_{df} C_1.Init \wedge C_2.Init \\
Code &=_{df} C_1.Code \cup C_2.Code \\
PriMDec &=_{df} C_1.PriMDec \cup C_2.PriMDec \\
PriCode &=_{df} C_1.PriCode \cup C_2.PriCode \\
InMDec &=_{df} (C_2.InMDec \setminus C_1.MDec) \cup C_1.InMDec
\end{aligned}$$ □

Theorem 4.9 $(C_1\rangle\rangle C_2)(InCtr) \equiv_{ctr} (C_1(InCtr_1) \parallel C_2(InCtr_2))\backslash(C_1.MDec \cap C_2.InMDec)$ *where*

$$\begin{aligned}
InCtr_1 &=_{df} InCtr \Downarrow C_1.InMDec \\
InCtr_2 &=_{df} InCtr \Downarrow (C_2.InMDec \setminus C_1.MDec) \parallel \\
&\qquad C_1(InCtr_1) \Downarrow (C_1.MDec \cap C_2.InMDec)
\end{aligned}$$

Proof. The conclusion follows from the following facts

$(C_1\rangle\rangle C_2)(InCtr).OutCtr.Spec(m) = C_1(InCtr_1).OutCtr.Spec(m)$

$$\text{if } m \in C_1.MDec \setminus C_2.InMDec$$

$$(C_1\rangle\rangle C_2)(InCtr).PriSpec(m) = C_1(InCtr_1).PriSpec(m)$$

$$\text{if } m \in C_1.PriMDec$$

$$InCtr_2.Spec(m) = InCtr.Spec(m)$$

$$\text{if } m \in C_2.InMDec \setminus C_1.MDec$$

$$InCtr_2.Spec(m) = C_1(InCtr_1).OutCtr.Spec(m)$$

$$\text{if } m \in C_2.InMDec \cap C_1.MDec$$

$$(C_1\rangle\rangle C_2)(InCtr).OutCtr.Spec(m) = C_2(InCtr_2).OutCtr.Spec(m)$$

$$\text{if } m \in C_2.MDec$$

□

The chain operator is monotonic.

Theorem 4.10 *If* $C_1 \sqsubseteq_{comp} C_2$, *then*

(1) $(C_1\rangle\rangle C_3) \sqsubseteq_{comp} (C_2\rangle\rangle C_3)$

(2) $(C_0\rangle\rangle C_1) \sqsubseteq_{comp} (C_0\rangle\rangle C_2)$

Proof. From Theorem 4.7 and 3.24. □

The hiding operator commutes with the chain operator.

Theorem 4.11 *If* $S_1 \subseteq MDec_1 \setminus InMDec_2$, *and* $S_2 \subseteq MDec_2$, *then* $(C_1\rangle\rangle C_2)\backslash(S_1 \cup S_2) \equiv_{comp} (C_1\backslash S_1)\rangle\rangle(C_2\backslash S_2)$

Proof. From Theorem 4.7 and 3.25. □

Definition 4.12 [Disjoint parallel] Let C_1 and C_2 be components satisfying the following conditions

(i) $FDec_1 \cap FDec_2 = \emptyset$, and

(ii) C_1 and C_2 do not share services.

In this case, the notation $C_1 \otimes C_2$ represents the composite component which has the provided services of C_1 and C_2 as its provided services, and the required services of C_1 and C_2 as its required services:

$$I =_{df} I_1 \uplus I_2$$

$$Init =_{df} Init_1 \cup Init_2$$

$$Code =_{df} Code_1 \cup Code_2$$

$$PriMDec =_{df} PriMDec_1 \cup ProMDec_2$$

$$PriMCode =_{df} PriMCode_1 \cup PriMCode_2$$

$$InMDec =_{df} InMDec_1 \cup InMDec_2$$

Theorem 4.13 $(C_1 \otimes C_2)(InCtr) \equiv_{ctr} C_1(InCtr_1) \parallel C_2(InCtr_2)$, *where*

$$InCtr_1.Spec =_{df} (InMDec_1 \lhd InCtr.Spec)$$

$$InCtr_2.Spec =_{df} (InMDec_2 \lhd InCtr.Spec)$$

Proof. Similar to Theorem 4.7. □

The disjoint parallel operator is monotonic, symmetric and associative.

Theorem 4.14

(1) *If* $C_1 \sqsubseteq_{comp} C_2$ *then* $(C_1 \otimes C) \sqsubseteq_{comp} (C_2 \otimes C)$

(2) $(C_1 \otimes C_2) \equiv_{comp} (C_2 \otimes C_1)$

(3) $C_1 \otimes (C_2 \otimes C_3) \equiv_{comp} (C_1 \otimes C_2) \otimes C_3$.

(4) $(C_1 \otimes C_2)\backslash S \equiv_{comp} (C_1\backslash(S \cap C_1.MDec)) \otimes (C_2\backslash(S \cap C_2.MDec)$

Proof. The conclusion (1) follows from Theorem 4.11 and 3.24. (2) follows from Theorem 4.13 and 3.25(2). From Theorem 3.25(3) it follows the conclusion (3). The conclusion (4) comes from Theorem 4.13 and 3.25(1). □

Definition 4.15 [Feedback] Let C be a component. Suppose its public method m has the same number and types of parameters as the imported method n. We use the notation $C[m \hookrightarrow n]$ to represent the component which feeds back its provided service m to the required service n.

$$\begin{aligned}
I &=_{df} C.I\backslash\{m\} \\
Init &=_{df} C.Init \\
Code &=_{df} (C.MDec \setminus \{m\}) \lhd C.Code \\
PriMDec &=_{df} (C.PriMDec \cup \{m\}) \\
PriCode &=_{df} C.PriCode \cup \{m \mapsto C.Code(m)\} \\
InMDec &=_{df} C.InMDec \setminus \{n\}
\end{aligned}$$

Theorem 4.16 $C[m \hookrightarrow n](InCtr) \equiv_{ctr} C(InCtr_1)\backslash\{m\}$, *where* $InCtr_1$ *is defined*

$$InCtr_1(s) =_{df} \begin{cases} InCtr(s) & s \neq n \\ D & s = n \end{cases}$$

where the guarded design D *is defined as the worst fixed point of the equation* $X = C(InCtr.Spec \cup \{n \mapsto X\}).Spec(m)$

Proof. Similar to Theorem 4.7. □

Theorem 4.17

(1) *If* $C_1 \sqsubseteq_{comp} C_2$, *then* $C_1[m \hookrightarrow n] \sqsubseteq_{comp} C_2[m \hookrightarrow n]$

(2) $(C[m_1 \hookrightarrow n_1])[m_2 \hookrightarrow n_2] \equiv_{comp} (C[m_2 \hookrightarrow n_2])[m_1 \hookrightarrow n_1]$

(3) *If* m *is not a member of* S, *then*

$$(C\backslash S)[m \hookrightarrow n] \equiv_{comp} C[m \hookrightarrow n]\backslash S$$

(4) If $m \in MDec_1$ *and* $n \in InMDec_1 \setminus InMDec_2$, *then*

$$(C_1 \otimes C_2)[m \hookrightarrow n] \equiv_{comp} C_1[m \hookrightarrow n] \otimes C_2.$$

(5) If $m \in MDec_2$ *and* $n \in InMDec_1 \setminus MDec_1$, *then*

$$(C_1 \rangle\rangle C_2)[m \hookrightarrow n] \equiv_{comp} C_1 \rangle\rangle (C_2[m \hookrightarrow n])$$

(6) If $m \in MDec_1 \setminus InMDec_2$ *and* $n \in InMDec_1 \setminus InMDec_2$, *then*

$$(C_1 \rangle\rangle C_2)[m \hookrightarrow n] \equiv_{comp} (C_1[m \hookrightarrow n]) \rangle\rangle C_2$$

Proof. (1) follows from Theorem 4.2 and 3.16. (2) and (3) come from Theorem 4.16 and 3.17. (4) follows from Theorem 4.14(4). From Theorem 4.11 it follows the conclusions (5) and (6). □

4.3 Multiple Interface

In general a component can be equipped with several provided service interfaces and required service interfaces:

$(\{(I_i, Init_i, Code_i) \mid i \in L\}, PriMDec, PriCode, \{InMDec_j \mid j \in M\})$

where we assume that

(i) The required service interfaces are composable, i.e.,

$$x : T_1 \in I_i \wedge x : T_2 \in T_j \wedge i \neq j \Longrightarrow T_1 = T_2$$

(ii) The initialisations are consistent, i.e. $\wedge_{i \in L} Init_i$ is a feasible design.

(iii) The mappings $Code_i$ are consistent, i.e.

$$m(in : T_1, out : T_2) \in I_i.MDec \cap I_j.MDec \Longrightarrow Code_i(m) = Code_j(m)$$

(iv) $InMDec_i$ and $InMDec_j$ are disjoint whenever $i \neq j$

In such case this component can be seen as one with single provided service interface

$$I =_{df} \oplus_{i \in L} I_i \quad Init =_{df} \wedge_{i \in L} Init_i \quad Code =_{df} \cup_{i \in L} Code_i$$

and single required service interface $InMDec =_{df} \cup_{j \in M} InMDec_j$.

The component combinators introduced in the previous section are applicable to components with multiple provided and required interfaces.

5 Conclusion

A contract can be employed to describe use cases discussed in the conceptual level of the RUP. One can also realise a conceptual diagram in the UML by introducing service subsystems that are invisible to the outsiders. The contract calculus supports system splitting and verification.

This paper introduces a theory of reactive component calculus, whereby a system can be divided into a number of interconnected components. We model the behaviour of individual service by a guarded design, which enables one to separate the responsibility of clients from the commitment made by the system. We adopt

the notion of process refinement to formalise the substitutivity of components, and provide a complete proof method for refinement.

In future we will investigate the notions of connectors, coordinators, configurations of components, and discuss their specification and design techniques. Application of the this theory to case studies is also a focus of our future work.

Acknowledgement

We would like to thank colleagues Xin Chen, Vladimir Mencl, Rodrigo Ramos, and Wei Zhang for their reading and comments on earlier versions of the paper.

References

[1] R. Back and L.J. von Wright. Refinement Calculus. Springer, 1998.

[2] R. Back, *et al.* Class refinement as semantics of correct object substitutability. Formal Aspect of Computing 2, 18-40, 2000.

[3] P. Borba, A. Sampaio and M. Cornelio. A refinement algebra for object-oriented programming. In Proc of ECOP'2003, Lecture Notes in Computer Science 2743, 457–482, 2003.

[4] A. Cavalcanti and D. Naumann. A weakest precondition semantics for an object-oriented language. Lecture Notes in Computer Science 1709, 1439–1460, 1999.

[5] J. Cheesman and J. Danies, "UML Components – A Simple Process for Specifying Component-Based Software," Reading, MA: Addison-Wesley, 2000.

[6] D'Souza and A.C. Wills, "Objects, Components and Frameworks: The Catalysis Approach," MA: Addison-Wesley, 1998.

[7] Martin Fowler. Refactoring, Improving the Design of Existing Code. Addison-Wesley, 2000.

[8] E. Gamma, *et al.* Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[9] M. Gudgin, "Essential IDL: Interface Design for COM," Reading, MA: Addison-Wesley, 2001.

[10] J. He. Simulation and Process Refinement. Formal Aspect of Computing, 229–241, 1989

[11] J. He. Linking simulation with refinement. In Proc of the 25th Anniversary of CSP, Lecture Notes in Computer Science 3525, 61–75, 2005.

[12] J. He and C.A.R. Hoare. Unifying theories of concurrency, in Proc of ICTAC'2005, 2005.

[13] J. He, Z. Liu, X. Li and S. Qin. A relational model of object oriented programs. In Proc of APLAS'2004, Lecture Notes in Computer Science 3302,415-437, 2004.

[14] C.A.R. Hoare, *et al.* Laws of Programming. Communications of the ACM 30: 672–686, 1987

[15] C.A.R. Hoare and He Jifeng, "Unifying Theories of Programming," Prentice Hall, (1998).

[16] C.B. Jones, "Systematic Software Development Using VDM," Upper Saddle River, NJ: Prentice Hall, 1990.

[17] C. Larman. Applying UML and Patterns. Prentice-Hall International, 2001

[18] X. Li, Z. Liu, J. He and Q. Long. Generating prototypes from a UML model of requirements. In Proc of ICDIT'2004, Lecture Notes in Computer Science 3347, 255–265, 2004.

[19] Z. Liu, J. He and X. Li. Contract-oriented development of component systems. In Proc of IFIP WCC-TCS'2004, 349–366, 2004.

[20] Z. Liu, J. He, X. Li and Y. Chen. A relational model for object-oriented requirements in UML. In Proc of ICFEM'2003, Lecture Notes in Computer Science 2885, 641–665, 2003.

[21] B. Meyer, "Object-Oriented Software Construction," Upper Saddle River, NJ: Prentice Hall, 1997.

[22] Microsoft, "The Component Object Model Specification," Report v0.99, Microsoft Standards, Redmond, WA: Microsoft, 1996.

[23] OMG, "The Common Object Request Broker: Architecture and Specification," Report v2.5, OMG Standards Collection, OMG, 2000.

[24] Sun Microsystems, "JavaBeans 1.01 Specification," http://java.sun.com/beans.

[25] D.B. Roberts. Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana Champain, 1999.

[26] K. Rustan and M. Leino. Recursive object types in a logic of object-oriented programming. Lecture Notes in Computer Science 1381, 1998.

[27] C. Szyperski, "Component Software – Beyond Object-Oriented Programming," Reading, MA: Addison-Wesley, 1998.

[28] L.A. Tokuda. Evolving Object-Oriented Designs with Refactoring. PhD thesis, University of Texas at Austin, 1999.

[29] J. Warmer and A. Kleppe, "The Object Constraint Language," Reading, MA: Addison-Wesley, 1999.