A Model of Component-Based Programming *

Xin Chen[1,4], He Jifeng[2], Zhiming Liu[1], and Naijun Zhan[1,3]

[1] International Institute for Software Technology, United Nations University, Macau
{chenxin,lzm,znj}@iist.unu.edu
[2] Software Engineering Institute, East China Normal University, Shanghai, China
jifeng@sei.ecnu.edu.cn
[3] Lab. of Computer Science, Institute of Software, CAS, Beijing, China
znj@ios.ac.cn
[4] Department of Computer Science and Technology, Nanjing University, China

Abstract. Component-based programming is about how to create *application programs* from *prefabricated components* with *new software* that provides both *glue* between the components, and *new functionality*. Models of components are required to support *black-box compositionality* and *substitutability* by a third party as well as *interoperability*. However, the glue codes and programs designed by users of the components for new applications in general do not require these features, and they can be even designed in programming paradigms different from those of the components. In this paper, we extend the rCOS calculus of components with a model for glue programs and application programs that is different from that of components. We study the composition of a glue program with components and prove that components glued by a glue program yield a new component.
Keywords: *Components, Contracts, Protocols, Composition, Glue Codes, Application Programs, Refinement.*

1 Introduction

Component-based development (CBD) is about how to create new software by combining *prefabricated components* with *new programs* that provide both glue between the components, and new functionality [1]. Furthermore, there seems to be no disagreement on the following interrelated properties that components enjoy.

1. *Black-box composability, substitutability and reusability:* there is no need to know the design and the implementation when composing a component with other parts of the system, substituting a component with another one or reusing it in another application.
2. *Independent development:* components can be designed, implemented, verified, validated and deployed independently.

* This work is partly supported by the project HighQSoftD funded by Macao Science and Technology Development Fund, and by the projects NSFC-60493200, NSFC-60421001, NSFC-60573007 and NKBRPC-2002cb312200.

3. *Interoperability:* components can be implemented in different programming languages and paradigms, but they can be composed, be glued together and cooperate with each another.

These features require that a component has a black-box specification of what it *provides* to and what it *requires* from its environment. In rCOS [5, 6, 4], the provided services and required service of a component are given by the contract of the *provided interface* and the contract of the *required interface* of the component, respectively. Thus, the contracts together with the interfaces of a component provide a black-box specification of the component. The model of contracts in rCOS also defines the unified semantic model of implementations of interfaces in different programming languages, and thus clearly supports interoperability of components and analysis of the correctness of a component with respect to its interface contract. The theory of refinements of contracts and components in rCOS characterizes component substitutivity, as well as supporting independent development of components. Compositions are defined in rCOS for chaining the provided interface of one component to the required interface of another, renaming and hiding interface operations of a component.

However, there is no precise characterization for the "new program" that provides both "glue" between the components, and "new functionality". In this paper, we introduce the notion of *processes* into rCOS. Like a component, a process has an interface declaring its local variables and methods, and its behavior is specified by a process contract. Unlike a component that passively waits for a client to call its provided services, a process is active and has its own control on when to call out or to wait for a call to its provided services. For such an active process, we cannot have separate contracts for its provided interface and required interface, because we cannot have separate specifications of outgoing calls and incoming calls [5]. For simplicity, but without losing expressiveness, we assume a process like a Java thread does not provide services and only calls operations provided by components. Therefore, processes can only communicate via shared components. The composition of two processes will be by interleaving, and produce a new process.

Let C be the parallel composition of a number of disjoint components C_i, $i = 1, \ldots, k$. A glue program for C is a process P that makes calls to the operations in set X provided by C. The *synchronization composition* $P \underset{X}{\parallel} C$ of C and P is defined similarly to the alphabetized parallel in CSP [7, 12]. The gluing composition is defined by hiding the synchronized methods between the component C and the process P. We show that $(P \underset{X}{\parallel} C)\backslash X$ is a component. We will study the algebraic laws of the composition of processes and components as well.

We also model an application program as a set of parallel processes that make use of the services provided by components. As processes only interact with components via the provided interfaces of the components, interoperability is thus supported as the contracts which define the semantics of the common interface description language (IDL), even though components, glue programs and components are not implemented in the same language. Analysis and verification of an

application program can be performed in the classical formal frameworks, but at the level of contracts of components instead of implementations of components. The analysis and verification can reuse any proved properties about the components, such as divergence freedom and deadlock freedom of the implementation of the components, without the need to reprove them.

The rest of this paper is organized as follows. Section 2 contains a brief summary of rCOS. In section 3, we define the model of process and gluing composition. As well, we prove that gluing components by a process indeed forms a new component and then present a method to calculate the contract of the resulted component. Section 4 presents a comparison between our work to the relative work. Section 5 draws a short conclusion and discusses the future work.

2 Interface, Contracts and Components

The major concern of our model of components in rCOS [5, 6] is *reuse* and *interoperablity* and thus captures the essence of informal definition of components given in Szyperski's book [13]:

> A component is a unit of composition with contractually specified interfaces and fully explicit context dependencies that can be deployed independently and is subject to third party composition.

Thus, a component has a provided interface, and optionally a *required interface*, and each interface is associated with a specification called its *contract*. This section uses examples to introduce the main modelling elements of the component model in rCOS.

2.1 Preliminaries

For convenience, we first introduce some notions of traces. Given an alphabet Σ, Σ^* denotes all finite sequences generated from Σ, while Σ^∞ denotes all infinite sequences generated from Σ. Given a sequence s, we will use $|s|$, $\mathbf{tail}(s)$, and $\mathbf{head}(s)$ to denote the length, tail, and head of s, respectively. $s_1 \bullet s_2$ denotes the concatenation of the sequences s_1 and s_2, and $s_1 \preceq s_2$ denotes that s_1 is a prefix of s_2. $s \upharpoonright \Sigma$ stands for the sequence obtained by removing all events not in Σ from s. If Σ is a singleton $\{a\}$, $s \upharpoonright A$ is abbreviated as $s \upharpoonright a$. $s \downarrow b$ counts the number of occurrences of b in s.

2.2 Interface

An interface $I = \langle FDec,\ MDec \rangle$ declares a set of *fields* and a set of *operation signatures* without providing any semantic information of their designs and implementations. Here, for the sake of encapsulation, all fields declared in an interface are assumed to be *local* to the underpinning contract and component and therefore are not accessible to its environments. The environments can only access the

declared fields via the declared methods[1]. Each field in *FDec* has the form $x : T$ of a variable with its type, and an operation $m(\mathbf{in}\, inx, \mathbf{out}\, outx) \in MDec$ declares a name for the operation and its input parameters and output parameters with their types. For simplicity, we do not deal with data types formally and assume that a method has at most one input parameter and one output parameter and is written in the form $m(\mathbf{in}\, u, \mathbf{out}\, v)$ in what follows.

Example 1. Consider a buffer of integers. It has an interface that enables the user to put data in and get data from the buffer:

$$B_1{=}\langle buff{:}seq(int), \{put(\mathbf{in}\, x{:}int), get(\mathbf{out}\, y{:}int)\}\rangle$$

where $seq(int)$ is the type of finite sequences of integers.

Interfaces can be *merged* and *extended* by adding new operations [5, 6].

2.3 Contract

A contract of an interface of a component provides semantic information that specifies how the interface can be used and allows us to define the dynamic behavior of the component on the interface. Here, we are only concerned with components of concurrent and distributed software systems and thus only interested in the *functionality* and *interaction protocols* of components, leaving real-time and other non-functional quality of services (QoS) out of the scope of this paper. Formally, a **contract** is a tuple $Ctr(I, Init, MSpec, Prot)$, where

- I is an interface;
- $Init$ is a predicate that defines the initial values of the fields in $I.FDec$;
- $MSPec$ assigns each operation $m(x; y)$ a *static functionality specification* as pair of *pre* and *postconditions* of the form $p(x, I.FDec) \vdash R(x, I.FDec, y', I.FDec')$, where non-primed and primed variables represents the values of the variables in the pre and post state of the execution of the operation, respectively. If the precondition $p(x, I.FDec)$ is true, the pair will be abbreviated as $\vdash R(x, I.FDec, y', I.FDec')$;
- $Prot$ is called the *protocol* of the interface, which is a set of finite sequences of method call events. Each sequence is of the form $m_1, \ldots, m_k$.

Example 2. For the buffer interface in Example 1, the following contract Ctr_B defines a one-place buffer:

$$\begin{aligned} Init &\stackrel{def}{=} |buff|{=}0 \\ MSpec(put(\mathbf{in}\, x{:}int)) &\stackrel{def}{=} (\vdash buff'{=}\langle x\rangle \bullet buff) \\ MSpec(get(\mathbf{out}\, y{:}int)) &\stackrel{def}{=} (\vdash buff' = \mathbf{tail}(buff) \wedge y' = \mathbf{head}(buff)) \\ Prot &\stackrel{def}{=} (put; get)^*{+}(put; (get; put)^*) \end{aligned}$$

[1] In fact, such an assumption can be relaxed. In many cases, the relaxation will improve the ease in developing complex systems, typically, embedded systems.

In many applications, the protocols can be specified as regular expressions and in such a case protocol compatibility can be automatically checked.

A pair of pre and postconditions is called a *design* in [8]. It is proven there that designs are closed under all imperative programming constructors such as assignment, sequential composition, conditional choice, recursion and so on. These constructors are all monotonic with respect to the *refinement* order among designs. In [3], we showed how to define an object-oriented program as a design too. Therefore, the model of contracts of interfaces can be safely used as a common semantic model of different programming languages and paradigms to support interoperability of components.

For theoretical treatment of contracts and their refinement, the designs of operations and the interaction protocol can be combined by the notion of *guarded designs* [5, 6].

A guarded design is a pair of a *guard* g and a design D, denoted by $g\&D$, and defined by $D \lhd g \rhd \mathit{Idle}$[2], meaning that the caller is forced to wait if the guard condition does not hold when invoking the method, otherwise it behaves as the design D. We have proven in [5, 6] that guarded designs are closed under all programming constructors, and these constructors are all monotonic with respect to the *refinement* order.

A **reactive contract** is a triple $\mathit{Ctr} = (I, \mathit{Init}, \mathit{MSpec})$, where MSpec assigns each operation $m(x; y)$ in the interface I with a *guarded design*. In what follows, we use g_m to denote the guard part of $\mathit{MSpec}(m)$, for any $m \in \mathit{MDec}$.

Example 3. The contract in Example 2 can have an equivalent reactive version:

$$\begin{aligned} \mathit{Init} &\stackrel{def}{=} |\mathit{buff}|{=}0 \\ \mathit{MSpec}(\mathit{put}(\mathbf{in}\, x{:}\mathit{int})) &\stackrel{def}{=} (|\mathit{buff}| = 0)\&(\vdash \mathit{buff}' = \langle x \rangle) \\ \mathit{MSpec}(\mathit{get}(\mathbf{out}\, y{:}\mathit{int})) &\stackrel{def}{=} (|\mathit{buff}| = 1)\&(\vdash \mathit{buff}' = \langle\rangle \wedge y' = \mathbf{head}(\mathit{buff})) \end{aligned}$$

Given a reactive contract $\mathit{Ctr} = (I, \mathit{Init}, \mathit{MSpec})$, its dynamic behavior is defined by its sets of failures and divergences ($\mathcal{F}(\mathit{Ctr})$, $\mathcal{D}(\mathit{Ctr})$). Each method call $m(u, v)$ includes two events $?m(u)$ for receiving an invocation and $m(v)!$ for sending a return to the caller. Therefore, each trace in failures and divergences is of the form $?m_1(u_1), m_1(v_1)!, \ldots, ?m_n(u_n), m_n(v_n)!$ or $?m_1(u_1), m_1(v_1)!, \ldots, ?m_n(u_n)$. The failures and divergences are defined as:

- $\mathcal{D}(\mathit{Ctr})$ consists of the sequences of interactions between Ctr and its environment which lead the contract to a divergent state.
- $\mathcal{F}(\mathit{Ctr})$ is the set of pairs (s, X), where s is a sequence of interactions between Ctr and its environment, and X denotes a set of methods to which the contract may refuse to respond after executing s. A failure (s, X) should be one of the following cases:
  1. $s = \langle ?m_1(x_1), m_1(y_1)!, \ldots, ?m_k(x_k), m_k(y_k)! \rangle$ and $\forall m \in X.\neg g_m$, $k \geq 0$. If $k = 0$ then $s = \langle\rangle$. This corresponds to the case when the system reaches a state where none of the guards of the events in X is true, after executing the sequence of calls.

[2] This is the shorthand of **if** g **then** D **else** Idle.

2. $s = \langle ?m_1(x_1), m_1(y_1)!, \ldots, ?m_k(x_k)\rangle$ and $m_k! \notin X$. This corresponds to the case when the operation m_k is waiting to output its result, performing any of other operations will result in a failure, because it is assumed that the execution of a method is atomic in the sense that the method is either executed completely, or not at all, no other methods can interrupt its execution.
3. $s = \langle ?m_1(x_1), m_1(y_1)!, \ldots, ?m_k(x_k)\rangle$ and X could be any set of methods, where the execution of m_k enters a waiting state.
4. Finally, $s \in \mathcal{D}(Ctr)$ and X can be any set of methods. That is, a divergent trace with any set of methods always forms a failure.

Example 4. To continue Example 3, the dynamic behaviour of the buffer can be described by the following *failure/divergence* model:

$$\begin{aligned}
&\mathcal{D} = \emptyset \\
&\mathcal{F} = \{(s, X) \mid (\exists k \in \mathbb{N}.s = \langle S(k)\rangle \wedge X \subseteq \overline{\{?put\}}) \\
&\qquad\qquad \vee (\exists k \in \mathbb{N}.s = \langle S(k), ?put(x_{k+1})\rangle \wedge X \subseteq \overline{\{put!\}}) \\
&\qquad\qquad \vee (\exists k \in \mathbb{N}.s = \langle S(k), ?put(x_{k+1}), !put()\rangle \wedge X \subseteq \overline{\{?get\}}) \\
&\qquad\qquad \vee (\exists k \in \mathbb{N}.s = \langle S(k), ?put(x_{k+1}), put()!, ?get()\rangle \wedge X \subseteq \overline{\{get!\}})\}
\end{aligned}$$

where

$$\begin{aligned}
S(k) &\stackrel{def}{=} ?put(x_1), put()!, ?get(), get(x_1)!, ...?put(x_k), put()!, ?get(), get(x_k)! \\
\overline{Y} &\stackrel{def}{=} \{?put, put!, ?get, get!\} - Y
\end{aligned}$$

The following notion of *refinement* allows us to compare and substitute components according to their contracts.

Definition 1 *Let Ctr_1 and Ctr_2 be two contracts. We say that Ctr_1 is* refined *by Ctr_2, denoted by $Ctr_1 \sqsubseteq Ctr_2$, if*

1. *Ctr_2 provides the same services as that of Ctr_1, i.e. $Ctr_2.MDec = Ctr_1.MDec$,*
2. *Ctr_2 is not easier to diverge than Ctr_1, i.e. $\mathcal{D}(Ctr_2) \subseteq \mathcal{D}(Ctr_1)$, and*
3. *Ctr_2 is not easier to deadlock than Ctr_1, i.e. $\mathcal{F}(Ctr_2) \subseteq \mathcal{F}(Ctr_1)$.*

Ctr_1 and Ctr_2 are equivalent*, denoted by $Ctr_1 \equiv Ctr_2$, if they refine each other.*

For the full refinement calculus of components, we refer the reader to [4].

2.4 Component

A component is an implementation of a contract of its provided interface. To implement such a contract, the component may *use* services provided by other components. These services are called *required services* and are specified as a *contract* of an interface that is called the *required interface*.

Formally, a *component* C is a tuple $(I, Init, MCode, PriMDec, PriMCode, InMDec)$, where

1. I and $Init$ are its interface and initial condition, respectively;
2. $PriMDec$ is a set of method declarations that are internal to the component;

3. *MCode* (*PriMCode*) maps each method m in *I.MDec* (resp. *PriMDec*) to a program of a underlining programming language. However, according to the results of [8], any program can be abstracted as a *guarded command* $g\&c$, further to a *guarded design*. So, without loss of generality, we always assume that the above two functions map each method to a guarded command from now on.
4. *InMDec* denotes a required interface whose operations are called in the implementations of the operations in *PriMCode* and *I.MDec*, but not declared there.

We use *C.I*, *C.Init*, *C.MCode*, *C.PriMDec*, *C.PriMCode* and *C.InMDec* to denote the corresponding parts of C.

According to [8], a guarded command $g\&c$ can always be defined as a guarded design $Dsn(g\&c)$. The command c may contain both invocations to methods in *PriMDec* and *InMDec*. Once the code of the private commands are given, their semantics can be used for the calculation of $Dsn(g\&c)$. However, $Dsn(g\&c)$ also depends on the given contract of the required interface. Therefore, the semantics of component C is defined to be the contract function $[\![C]\!](\cdot)$ such that for any given contract $InCtr$ of the required interface *InMDec*, $[\![C]\!](InCtr)$ is the contract of the provided interface *I.MDec* in which the guarded design of each operation m is calculated by $Dsn(MCode(m))$ from the code of *PriMDec* and the given required contract. A component C is called *closed* if it does not require external services.

Remark 1. As we discussed earlier, the model of contracts is used as the common semantics for different programming languages used to implement components. This is why we say that our model of components supports interoperability. The Object rCOS presented in [3] provides a unified calculus for defining and reasoning both object-oriented and imperative programs, and thus can be used for formal construction of components.

Remark 2. The model of components also supports encapsulation of both data and implementation of components. A component can only be used according to its contract that is an abstract semantics of the component. This strongly supports reuse, not only components but also proofs of properties of the contracts. This is crucial for scaling up the method.

2.5 Chaining components together

It is a natural way to compose components by chaining the provided operations of one component to the required operation of the other.

Definition 2 *Let* C_1 *and* C_2 *be components such that* $C_1.I.FDec \cap C_2.I.FDec = \emptyset$, $C_1.I.MDec \cap C_2.I.MDec = \emptyset$ *and* $C_1.PriMDec \cap C_2.PriMDec = \emptyset$. *Then the* chaining C_1 *to* C_2, *denoted by* $C_1 \rangle\rangle C_2$, *is the component with*

- $(C_1 \rangle\rangle C_2).FDec \stackrel{def}{=} C_1.FDec \cup C_2.FDec$,

- $(C_1\rangle\rangle C_2).InMDec \stackrel{def}{=} (C_2.InMDec \cup C_1.InMDec) - (C_2.MDec \cup C_1.MDec)$,
- $(C_1\rangle\rangle C_2).MDec \stackrel{def}{=} C_1.MDec \cup C_2.MDec$,
- $(C_1\rangle\rangle C_2).Init \stackrel{def}{=} C_1 Init. \wedge C_2.Init$,
- $(C_1\rangle\rangle C_2).Code \stackrel{def}{=} C_1.Code \cup C_2.Code$, *and*
- $(C_1\rangle\rangle C_2).PriCode \stackrel{def}{=} C_1.PriCode \cup C_2.PriCode$.

It is easy to show that the chaining operator is monotonic with respect to the refinement order of components [6]. In the special case when $(C_1.InMDec \cup C_2.InMDec) \cap (C_1.MDec \cup C_2.MDec) = \emptyset$, the chaining C_1 to C_2 is called *disjoint union* and denoted as $C_1||C_2$. Some other operators over components have also been defined in [6] such as *renaming*, *feedback* and *hiding*.

Example 5. Define two buffer components C_1 and C_2 as follows

$$
\begin{array}{ll}
C_1.FDec & = \{buff_1{:}Seq(int)\}\text{/* Seq(int) means sequence of integer*/} \\
C_1.MDec & = \{put(\mathbf{in}\, x{:}int), get_1(\mathbf{out}\, y{:}int)\} \\
C_1.Code(put) & = (buff_1{:=}\langle x\rangle) \lhd buff_1{=}\langle\rangle \rhd (put_1(\mathbf{head}(buff_1)); buff_1{:=}\langle x\rangle) \\
 & \quad \text{/*When } C_1 \text{ is full, } C_1 \text{ forwards the input data to } C_2\text{ */} \\
C_1.Code(get_1) & = (buff_1{\neq}\langle\rangle) \longrightarrow (y{:=}\mathbf{head}(buff_1); buff_1{=}\langle\rangle) \\
C_1.InMDec & = \{put_1(\mathbf{in}\, x{:}int)\} \\
C_2.FDec & = \{buff_2{:}Seq(int)\} \\
C_2.MDec & = \{put_1(\mathbf{in}\, x{:}int), get(\mathbf{out}\, y{:}int)\} \\
C_2.Code(put_1) & = (buff_2{=}\langle\rangle) \longrightarrow buff_2{:=}\langle x\rangle \\
C_2.Code(get) & = (y{:=}\mathbf{head}(buff_2); buff_2{:=}\langle\rangle) \lhd buff_2{\neq}\langle\rangle \rhd get_1(y) \\
 & \quad \text{/*When } C_2 \text{ is empty, } C_2 \text{ gets a data from } C_1\text{ */} \\
C_2.InMDec & = \{get_1(\mathbf{in}\, y{:}int)\}
\end{array}
$$

Then, $C_1\rangle\rangle C_2$ is shown in Fig.1 (a), hiding get_1 in $C_1\rangle\rangle C_2$, i.e. $(C_1\rangle\rangle C_2)\backslash\{get_1\}$ is shown in Fig.1 (b).

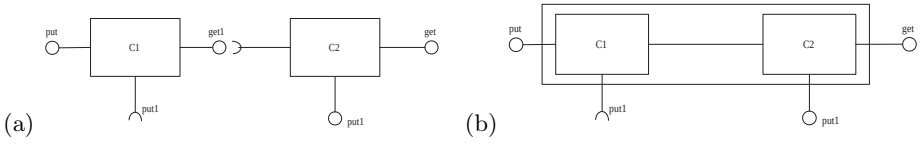

Fig. 1. (a) Chaining Composition, (b) Hiding After Chaining

3 Processes: A Model of Glue and Application Programs

In addition to building new components by applying the component operators defined in the previous section to existing components, we often need to *glue* existing components with a program to form a new component. Because in the most cases, we have to restrict the behaviour of the existing components and coordinate them in order to construct a new component from them. Thus, these component operators will not be applicable any more. For example, it is impossible to simply apply the chaining operator to two one-place buffers with the

same contract defined in Example 3 to produce a two-place buffer as we did in Example 5.

Glue code in general has different characteristics from components and we model it as *a process*. Like a component, a process has an interface declaring its own local variables and methods and its behavior is specified by a process contract. Unlike a component which passively waits for a client to call its provided services, a process is active and has its own flow of control on when to call out or to wait for a call to its provided services. For such an active process, we cannot have separate contracts for the provided interface and required interface, because we cannot have separate specifications of outgoing calls and incoming calls [5].

Glue codes and application programs play different roles in component-based software development. However, their behavior shares common characteristics. Application programs have their own control flows, and carry out their own computation task by using services provided by components, interacting with components in the same way as a glue program.

In this section, we define the model of processes and the glue composition of a process and a component. For simplicity and predictability, we assume that processes do not provide methods to their environment and do not communicate directly with each other. They are loosely coupled and can only communicate via invoking methods of components. The composition of processes is defined by interleaving and yields a new process.

3.1 Processes

The interface of a process is the access point through which the process invokes the operations of components. The process also carries out local computation by changing its local variables.

Definition 3 *A* process interface *I is a pair $\langle FDec, MDec\rangle$, where FDec is a set of field declarations, and MDec is a set of method invocation signatures. Each of them is of the form $!m(\mathbf{in}\, u : U, \mathbf{out}\, v : V)$.*

A process contract *Ctr is a triple $\langle I, Init, MSpec\rangle$, where I is a process interface, Init and MSpec are defined same as in a reactive contract.*

We use the notation $\overline{I.MDec}$ to denote the set $\{m \mid !m(\mathbf{in}\, u : U, \mathbf{out}\, v : V) \in I.MDec\}$.

Example 6. As shown in Fig.2 (a), a three-place buffer is built by gluing two one-place buffers defined in Example 3. The contract of the glue process is

$$
\begin{array}{ll}
I.FDec & = \{tmp : seq(int)\} \\
I.MDec & = \{!put(\mathbf{in}\, u : int), !get(\mathbf{out}\, v : int)\} \\
Init & = |tmp| = 0 \\
MSpec(!put(u)) & = \{u, tmp\} : |tmp| > 0 \,\&\, \vdash u' = \mathbf{head}(tmp) \wedge tmp' = \langle\rangle \\
MSpec(!get(v)) & = \{v, tmp\} : |tmp| = 0 \,\&\, \vdash tmp' = \langle v\rangle
\end{array}
$$

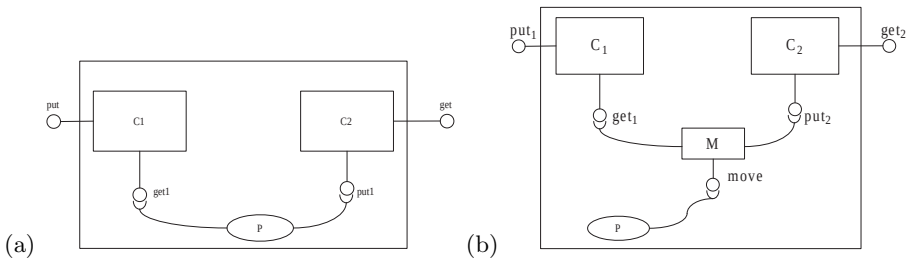

Fig. 2. (a) Gluing Two One-place Buffers Forms a Three-place Buffer, (b) Gluing Two One-place Buffers Forms a Two-place Buffer

As shown in the Fig.2 (b), to construct a two-place buffer, we need a new component that assures the execution of sequence $get_1(x), put_2(x)$ is not interrupted. Here, $M.Code(move) = \{get_1(u); put_2(u)\}$

The dynamic behavior of a process contract is defined on the basis of the observable events of the forms $!m(u)$ for making an invocation and $m(v)?$ for receiving a return from the invoked component. These are the *synchronization complementary events* of $?m(u)$ and $m(v)!$ in the behavior of a component contract.

The failure $\mathcal{F}(Ctr)$ and divergence $\mathcal{D}(Ctr)$ of a process contract Ctr are defined as:

- $\mathcal{D}(Ctr)$ consists of the sequences of interactions between Ctr and its environment which lead the contract to a divergent state. Each of such sequences is of the form $\langle !m_1(x_1), m_1(y_1)?, \ldots, !m_k(x_k), m_k(y_k)?, !m_{k+1}(x_{k+1})\rangle \cdot s$, where s is any sequence of method calls and the execution of m_{k+1} diverges.
- $\mathcal{F}(Ctr)$ is the set of pairs (s, X) where s is a sequence of interactions between Ctr and its environment, and X denotes a set of methods that the contract may refuse to respond to after engaging all events in s. Any $(s, X) \in \mathcal{F}$ should be one of the following cases:
  1. $s = \langle !m_1(x_1), m_1(y_1)?, \ldots, !m_k(x_k), m_k(y_k)?\rangle$ and $\forall m \in X.\neg g_m,\ k \geq 0$. If $k = 0$ then $s =<>$. This case represents that each method in X cannot be engaged after executing the sequence of calls, because their guards do not hold in the state.
  2. $s = \langle !m_1(x_1), m_1(y_1)?, \ldots, !m_k(x_k)\rangle$ and $m_k? \notin X$. This corresponds to the case where the contract is waiting for the return.
  3. $s = \langle !m_1(x_1), m_1(y_1)?, \ldots, !m_k(x_k)\rangle$ and X could be any set of methods. Here the execution of m_k enters a waiting state.
  4. Finally, $s \in \mathcal{D}(Ctr)$ and X can be any set of methods. That is, a divergent trace with any set of methods always forms a failure.

For a divergence free contract, case (4) will disappear. We can combine $!m(x)$ and $m(y)?$ into $m(x, y)$ and describe the failures in terms of sequences over events $m(x, y)$ by removing $!m_k(x_k)$ from the traces in cases (2) and (3) and put the event $m(x, y)$ into the refusal set. Thus, $\mathcal{F}(Ctr)$ can be simply defined as:

1. $s = \langle m_1(x_1, y_1), \ldots, m_k(x_k, y_k)\rangle$ and $\forall m \in X.\neg g_m$; or

2. $s = \langle m_1(x_1, y_1), \ldots m_k(x_k, y_k)\rangle$ and $\forall m \in X$ if m is executed following s, then m must reach a waiting state.

It is worth noting that the difference of failures and divergences of processes and contracts lies in the forms of sequences of method calls, the former's is of the form $!m_1(x_1), m_1(y_1)?, \cdots, !m_k(x_k), m(y_k)?, \cdots$, while the latter's is of the form $?m_1(x_1), m_1(y_1)!, \cdots, ?m_k(x_k), m(y_k)!, \cdots$.

Example 7. The dynamic behaviour of the process given in the Example 6 can be described by the following *failure/divergence* model:

$$\begin{aligned}
&\mathcal{D} = \emptyset\\
&\mathcal{F} = \{(s, X) \mid (\exists k \in \mathbb{N}.s = \langle S(k)\rangle \wedge X \subseteq \overline{\{!get_1\}})\\
&\qquad \vee (\exists k \in \mathbb{N}.s = \langle S(k), !get_1()\rangle \wedge X \subseteq \overline{\{get_1?\}})\\
&\qquad \vee (\exists k \in \mathbb{N}.s = \langle S(k), !get_1(), get_1(x_{k+1})?\rangle \wedge X \subseteq \overline{\{!put_2\}})\\
&\qquad \vee (\exists k \in \mathbb{N}.s = \langle S(k), !get_1(), get_1(x_{k+1})?, !put_2(x_{k+1})\rangle \wedge X \subseteq \overline{\{put_2?\}})\}
\end{aligned}$$

where

$$\begin{aligned}
S(k) &\stackrel{def}{=} !get_1(), get_1(x_1)?, !put_2(x_1), put_2()?, ..., !get_1(), get_1(x_k)?, !put_2(x_k), put_2()?\\
\overline{Y} &\stackrel{def}{=} \{!get_1(), get_1()?, !put_2(), put_2()?\} - Y
\end{aligned}$$

In fact, a process can be seen as a special component without provided services. Therefore, we can apply the chaining operator of components to processes to produce new processes. However, all application of the operator to any two processes P_1 and P_2 will be degenerated to the *disjoint union* of P_1 and P_2, i.e. $P_1 \| P_2$, as P_1 and P_2 both have no provided services. On the other hand, the other operators such as *renaming* and *hiding* can not apply to processes, because from a logical point of view, the names of the required services of a process are bound to the process.

3.2 Composing a component with a process

We consider the glue composition of a closed component and a process. If there are a number of closed components to be glued by a process, the disjoint union of these components forms another closed component.

Definition 4 *Let C be a closed component and P be a process that only calls methods provided by C, then the failures and divergences of the* synchronization composition $C \underset{X}{\|} P$*, denoted as $\mathcal{F}(C \underset{X}{\|} P)$ and $\mathcal{D}(C \underset{X}{\|} P)$ respectively, similarly to [12], are defined as:*

$$\begin{aligned}
\mathcal{D}(C \underset{X}{\|} P) &= \{a \bullet b \mid \exists s \in \mathcal{T}(C), t \in \mathcal{T}(P).a \in (s \underset{X}{\|} t) \cap \Sigma^* \wedge (s \in \mathcal{D}(C) \vee t \in \mathcal{D}(P))\}\\
\mathcal{F}(C \underset{X}{\|} P) &= \{(a, Y \cup Z) \mid Y \setminus X = Z \setminus X \wedge\\
&\qquad \exists s \in \mathcal{T}(C), t \in \mathcal{T}(P).(s, Y) \in \mathcal{F}(C) \wedge (t, Z) \in \mathcal{F}(P) \wedge a \in (s \underset{X}{\|} t)\}\\
&\quad \cup \{(a, Y) \mid a \in \mathcal{D}(C \underset{X}{\|} P)\}
\end{aligned}$$

where $\mathcal{T}(Q)$ stands for the set of traces of Q, where Q is either a component or a process; X is the set of synchronized methods; $\Sigma = \{?m(x_i), m(y_i)! \mid m \in C.MDec\}$, $b \in \Sigma^*$ and $s \underset{X}{\parallel} t$ denotes the parallel operation over traces, e.g. $abc \underset{\{b,c\}}{\parallel} a'bcd = \{aa'bcd, a'abcd\}$.

We can also apply the hiding operator of CSP to a component C and make any action in X become internal and invisible, denoted as $C \backslash X$. Its dynamic behavior is defined as:

$$\mathcal{D}(C \setminus X) = \{(s \upharpoonright X) \bullet t \mid s \in \mathcal{D}(C) \wedge t \in \mathcal{T}(C) \upharpoonright X\} \cup \{(a \upharpoonright X) \bullet t \mid t \in \mathcal{T}(C) \upharpoonright X \wedge a \in \Sigma^{\infty} \wedge |a \upharpoonright X| < \infty \wedge \forall s \preceq a.s \in \mathcal{T}(C)\}$$

$$\mathcal{F}(C \upharpoonright X) = \{(s \upharpoonright X, Y - X) \mid (s, Y) \in \mathcal{F}(C)\} \cup \{(s, Y) \mid s \in \mathcal{D}(C \setminus X)\}$$

where $\Sigma = \{?m(x_i), m(y_i)! \mid m \in C.MDec\}$.

Definition 5 *Let C be a closed component, P a process s.t. $\overline{P.MDec} \subseteq C.MDec$, the* gluing composition *$C \odot P$ is defined as: $C \odot P \stackrel{def}{=} (C \underset{\overline{P.MDec}}{\parallel} P) \setminus \overline{P.MDec}$.*

In our framework, components and processes are treated differently. So we have to answer what is the entity obtained by glue composition. Theorem 1 answer the question.

Theorem 1. *Suppose a closed component C and a process P satisfying the condition $\overline{P.MDec} \subseteq C.MDec$, then $C \odot P$ is a closed component.*

Proof: According to the above definitions, we can get $\mathcal{T}(C \odot P)$, $\mathcal{F}(C \odot P)$ and $\mathcal{D}(C \odot P)$. Therefore, by Woodcock and Morgan's results [14], we can construct a guarded command (guarded design) for each method m of $(C \odot P)$ and we will use $\mathcal{S}$ to denote the set of all these guarded commands (guarded designs).

For each method m, the corresponding guarded design in $\mathcal{S}$ is defined over two variables tr and $\mathcal{R}$. The variable tr stands a trace that m has engaged in, while $\mathcal{R}$ is a set of refusals. Formally, the guarded design of m can be represented as:

$$(m \notin \mathcal{R} \wedge (tr \bullet \langle m \rangle \in \mathcal{T}(C \odot P))) \& \\ (\{tr\} : \ \vdash tr' = tr \bullet \langle m \rangle); (\{\mathcal{R}\} : \ tr \notin \mathcal{D}(C \odot P) \vdash (tr, \mathcal{R}') \in \mathcal{F}(C \odot P)).$$

Let the initial condition be

$$(\{tr\} : \ \vdash tr' = \langle\rangle); (\{\mathcal{R}\} : \ tr \notin \mathcal{D}(C \odot P) \vdash (tr, \mathcal{R}') \in \mathcal{F}(C \odot P)),$$

and for each m, $(C \odot P).MSpec(m)$ be defined as its corresponding guarded design in $\mathcal{S}$. Thus, we can conclude that $C \odot P$ is a closed component whose provided contract is given as above. □

Similarly, we can prove that the glue composition applying to an open component and a process produces an open component. That is,

Theorem 2. *If C is an open component with a required interface $InMDec$ and P is a process that only calls the provided methods of C, then $(C \odot P)$ is an open component with the required interface $InMDec$.*

The semantics of the open component $(C \odot P)$ is defined as a function that given a contract of the required interface, returns a contract of the provided interface, denoted as $\lambda\, InCtr.(C \odot P)(InCtr)$. It is easy to see that

$$(C \odot P)(InCtr) = C(InCtr) \odot P$$

Example 8. Consider the component given in Fig.2 (a). Its dynamic behaviour is given by the following failures since it is divergence free.

$$\mathcal{F} = \{(tr, X) \mid tr \in \{put_1, get_2\}^* \wedge X \in \mathbb{P}\{put_1, get_2\} \wedge \forall tr_1 \preceq tr. \begin{pmatrix} tr_1 \downarrow put_1 - tr_1 \downarrow get_2 \leq 3 \\ \wedge\ vals(tr_1 \upharpoonright get_2) \preceq vals(tr_1 \upharpoonright put_1) \end{pmatrix} \wedge \begin{pmatrix} (tr \downarrow put_1 = tr \downarrow get_2 \wedge X \subseteq \{get_2\}) \\ \vee\ (tr \downarrow put_1 - tr \downarrow get_2 \leq 2 \wedge X = \emptyset) \\ \vee\ (tr \downarrow put_1 = tr \downarrow get_2 + 2 \wedge X \subseteq \{put_1\}) \end{pmatrix}\}$$

where $vals(s)$ returns the parameters occuring in the sequence s, and $(tr \downarrow put_1 - tr \downarrow get_2)$ is used to compute the number of items stored in the buffer.

3.3 The state-based reactive contract of a glued component

When proving Theorem 1, we use tr and $\mathcal{R}$ as field variables. In this section, we study how to calculate the "state-based" reactive contract of a glued component in terms of the field variables of its subcomponent and process.

The approach is based on the observation that if there is a sequence of methods $s = \langle m, m_1, \ldots, m_k, n\rangle$ occurring in a trace of $C \underset{\overline{P.MDec}}{\|} P$, where $m, n \notin \overline{P.MDec}$ and $m_1, \ldots, m_k \in \overline{P.MDec}$, the behaviour $[\![m]\!];[\![m_1]\!];\ldots;[\![m_n]\!]$ can be considered as a possible behaviour of m in the glued component, where ";" means the sequential composition of guarded designs [8]. The reason is because $m_1, \ldots, m_k$ are hidden and therefore become invisible in the glued component. Thus, for an observable method $m \notin \overline{P.MDec}$, its guarded design is the *non-deterministic* choice [8] of all those possible behaviour. However, it is easy to see that this approach only works when the glued component does not diverge. The divergence freedom can be proved by the theory of CSP and the FDR model checking tool.

Whenever a divergence free trace of $C \underset{\overline{P.MDec}}{\|} P$ has a prefix of the form $\langle m_1, \ldots, m_n, m\rangle$, where $m \notin \overline{P.MDec}$ and $m_1, \ldots, m_n \in \overline{P.MDec}$, we put the behaviour of the invisible sequence $\langle m_1, \ldots, m_n\rangle$ to be part of the initial condition.

Formally, we present our approach as follows: Let C be a closed component and P a process with $\overline{P.MDec} \subseteq C.MDec$. Then the contract for $(C \odot P)$ can be calculated as follows:

$$\begin{aligned}
&(C \odot P).FDec && \stackrel{def}{=} C.FDec \cup P.FDec \\
&(C \odot P).MDec && \stackrel{def}{=} C.MDec - \{\overline{P.MDec}\} \\
&(C \odot P).Init && \stackrel{def}{=} (C.Init \wedge P.Init) \wedge \sqcap_{tr \in \mathcal{G}}(C.Init \wedge P.Init); [\![tr]\!] \\
&(C \odot P).MSpec(m) && \stackrel{def}{=} C.MSpec(m) \sqcap_{tr \in \mathcal{Q}(m)} [\![tr]\!], \\
& && \qquad \text{for any } m \in (C \odot P).MDec
\end{aligned}$$

where

– $\mathcal{G} \stackrel{def}{=} \{h\tau \mid \exists s \in \Sigma^*, \exists n \in (C \odot P).MDec. (h\tau \in \overline{P.MDec}^+ \wedge h\tau \bullet \langle n \rangle \bullet s \in \mathcal{LT})\}$, which is the set of maximal invisible prefixes of legal traces.
– $\mathcal{Q}(m) \stackrel{def}{=} \{\langle m \rangle \bullet h\tau \mid \exists r, s \in \Sigma^*, \exists n \in (C \odot P).MDec. (h\tau \in \overline{P.MDec}^+ \wedge r \bullet \langle m \rangle \bullet h\tau \bullet \langle n \rangle \bullet s \in LT)\}$. $\mathcal{Q}(m)$ contains all the sequences of the form $\langle m, m_1, \ldots, m_n \rangle$ in each of the divergence free traces of $C \odot P$, where $m_1, \ldots, m_n \in \overline{P.MDec}$.
– $\mathcal{LT} \stackrel{def}{=} \{t \in \mathcal{T}(C) \mid \exists X \in \mathbb{P}(C.MDec).\ (t, X) \in \mathcal{F}(C \underset{X}{\parallel} P) \wedge t \notin \mathcal{D}(C \underset{X}{\parallel} P) \wedge (t \upharpoonright X) \notin \mathcal{D}((C \underset{X}{\parallel} P) \setminus X)\}$. That is, the legal traces of $C \odot P$ are those that themselves and their projections on $\Sigma - X$ are not divergent .
– $[\![tr]\!]$ maps each sequence tr to a guarded design which is calculated by sequentially composing the guarded design of each method of tr in turn. The guarded design of each method is defined by the following rules:
  1. $[\![m^g]\!]$ is $C.MSpec(m)$ if $m \notin \overline{P.MDec}$, otherwise $C.MSpec(m) \wedge P.MSpec(\overline{m})$. It is easy to see that $[\![m^g]\!]$ is a guarded design, for any $m \in C.MDec$;
  2. if $tr = \langle m_1, m_2, \ldots, m_n \rangle$, then $[\![tr]\!] = [\![m_1^g]\!]; [\![m_2^g]\!]; \ldots; [\![m_n^g]\!]$. Here, ";" means the sequential composition of (guarded) designs (see [8]).

Here, we have to point out that there may be different way to construct the possible behaviour of an observable method and the initial condition, it can therefore result in different contracts. For example, for the sequence $\langle m \rangle \bullet \tau_1 \bullet \tau_2 \bullet \langle n \rangle$, instead of defining their guarded design as $MSpec(m) \stackrel{def}{=} [\![m; \tau_1; \tau_2]\!]$ and $MSpec(n) \stackrel{def}{=} [\![n]\!]$, we can define them as $MSpec(m) \stackrel{def}{=} [\![m; \tau_1]\!]$ and $MSpec(n) \stackrel{def}{=} [\![\tau_2; n]\!]$. However, it is easy to prove that all these contracts should refine each other since they share the same failures and divergences as that of $(C \underset{\overline{P.MDec}}{\parallel} P) \backslash \overline{P.MDec}$.

Example 9. Calculate the contract of the component given in Fig.2 (a) from its dynamic behaviour in Example 8, and the contract of the process and one place buffer given in Example 6 and Example 3 respectively.

$$
\begin{aligned}
I.FDec &= \{tmp, buff_1, buff_2 : seq(int)\} \\
I.MDec &= \{put_1(\mathbf{in}\, u : int;),\ get_2(\mathbf{out}\, v : int)\} \\
Init &= tmp' = \langle\rangle \wedge buff_1' = \langle\rangle \wedge buff_2' = \langle\rangle \\
MSpec(put_1) &= C_1.MSpec(put_1) \sqcap [\![put_1; get_1]\!] \sqcap [\![put_1; get_1; put_2]\!] \sqcap [\![put_1; put_2]\!] \\
&\quad \sqcap [\![put_1; put_2; get_1]\!] \sqcap [\![put_1; get_1]\!] \\
&= \quad \{buff_1\} : |buff_1| = 0 \& \vdash buff_1' = \langle u \rangle \\
&\quad \sqcap \{tmp\} : |buff_1| = 0 \wedge |tmp| = 0 \wedge |buff_2| = 0 \& \vdash tmp' = \langle u \rangle \\
&\quad \sqcap \{buff_2\} : |buff_1| = 0 \wedge |tmp| = 0 \wedge |buff_2| = 0 \& \vdash buff_2' = \langle u \rangle \\
&\quad \sqcap \{buff_1, tmp, buff_2\} : |buff_1| = 0 \wedge |tmp| \neq 0 \wedge |buff_2| = 0 \& \\
&\qquad\qquad \vdash buff_1' = \langle u \rangle \wedge tmp' = \langle\rangle \wedge buff_2' = tmp \\
&\quad \sqcap \{tmp, buff_2\} : |buff_1| = 0 \wedge |tmp| \neq 0 \wedge |buff_2| = 0 \& \\
&\qquad\qquad \vdash tmp' = \langle u \rangle \wedge buff_2' = tmp \\
&\quad \sqcap \{tmp, buff_2\} : |buff_1| = 0 \wedge |tmp| = 0 \wedge |buff_2| \neq 0 \& \\
&\qquad\qquad \vdash tmp' = \langle u \rangle \wedge buff_2' = tmp
\end{aligned}
$$

Similarly, we can calculate $MSpec(get_2)$. Due to space, we omit it.

This example shows that the calculation of the failures and divergences is quite tedious. However it could be aided by the CSP tool FDR [12].

4 Relative Work

In CBD, how to construct composite components from existing ones is a challenging problem. In the object-oriented programming community, there has been extensive research on attacking this issue. For example, SuperGlue [11], Jiazzi [10], the calculus of assemblages [9] and so on. SuperGlue is a connection-based asynchronous programming model. In SuperGlue, a component is either SuperGlue code or Java code with a set of signals (possibly infinite many), and composing existing components is via connection rules over the signals of the subcomponents defined by SuperGlue Code. While Jiazzi [10] can be used to construct large-scale binary components in Java. Jiazzi components can be thought of as generalizations of Java packages with added support for external linking and separate compilation. Existing Java classes and Jiazzi components can be composed by Jiazzi linker to a new Jiazzi component. The linking is similar to the chaining operator in rCOS. Comparing with SuperGlue and Jiazzi, in our approach, each component is equipped with a provided interface and its contract, optionally as well as a required interface and its contract. Thus, components can be more easily reused across different applications, as the provided interfaces and contracts together with the required interfaces and contracts encapsulate their designs and implementations, as well as their data structures. Furthermore, the interoperability of components is well established in our model, since rCOS acts as the underlying theory of component designs which unifies semantic models of different programming languages and paradigms into the notion of interface contracts. What's more, our approach provides more means to compose new components from existing ones, either by the component operators or by the gluing code.

SuperGlue, Jiazzi and rCOS all cope with composing (gluing) components statically in the sense that all method names used for composing must be resolved in the moment these components are composed (glued). Whereas the calculus of assemblages [9] can handle the composing (gluing) dynamically. However, there is no the notion of contracts in within it as well.

[15] investigated the notions of components, composition of components and verification of composed components in an asynchronous interleaving event-based model, called Asynchronous Interleaving Message-passing computation model (AIM), with which the composition of components is interpreted as asynchronous parallel, analogous remark is applied to the composition of properties of components. In fact, we believe what was handled in [15] exactly corresponds to what the chaining operator can do in rCOS. However, rCOS is a combination of event-based model and state-based model, whose event-based model is a

synchronous concurrent model in contrast to that of [15], an asynchronous concurrent model. So, rCOS allows different notations and methods for modelling and analysing different aspects of components and processes, such as pre and post conditions for functionality, traces of events for interaction protocols, failures and divergences for the denotational view of dynamic behavior and guarded designs for operational views of dynamic behavior. This supports the separation of concerns and gives the hope of integrating different verification techniques and tools via this common model. In fact, the assume-guarantee proof style used in [15] can also be easily applied to our framework. However, our work is not only about assume-guarantee verification in the original setting. When chaining components together, the verification and calculation of the composed components are different from the case when components are glued together. Using verified properties in our framework is more about substitution of proof obligations by theorems proved about services that are used in components or application programs.

There are also various approaches to handle the composition of components in the formal methods community. In [2], a component is defined as a stream process function which maps the input streams of actions to the output streams of actions. The refinement relation between components is defined over a pair of input streams and output streams. rCOS clearly divides the provided contract(input actions) and the required contract(output actions) and can treat them separately, which greatly ease the composition of components. Like rCOS, Reo[1] treats components and glue codes(connectors) as distinct types. The two types build on a common formal foundation, the Abstract Behaviour Types. The Abstract Behaviour Types is very expressive for specification, but it is hard to be linked to implementation language. The notion of guarded design in rCOS can link specifications and OO languages very smoothly.

5 Conclusions and Future Work

We have proposed a model supporting component-based programming. The model unifies the component model developed earlier in [6, 5] and the process model defined here. Processes are introduced to model application programs and glue programs which help developers to build new components from existing ones.

In the proposed model, a typical component-based application consists of a family of components and a number of parallel application processes. Some of the components are reused from a component repository while others are newly built using gluing processes as well as component operators (chaining, service renaming, and service hiding).

As for future work, we need to investigate the following issues:

– In this paper, the method to calculate the resulted contract of the gluing of a component and a process is very complicated and difficult to track. Therefore, as a future work, on one hand, we need to simplify the procedure; on the other hand, we will look into automating the calculation.

– It will be interesting research topic to investigate how different verification techniques and tools can be applied to rCOS.
– We are also interested in investigating on how rCOS can be applied to web service systems, and to deal with quality of services (QoS) of components, such as time and resource constraints.
– Case studies of realistic component systems such as CORBA.

Acknowledgements

We are grateful to Prof. Anders P. Ravn for pointing out many features in the design of our model. We also thank Dr. Volker Stolz and Lu yang for their comments. Special thanks are also due to the anonymous referees for their valuable suggestions and comments which help us to improve this paper including its contents as well as its presentation so much.

References

1. F. Arbab. Abstract behavior types: A foundation model for components and their composition. In *Proc. of the FMCO 2002*, volume 2852 of *LNCS*, pages 33–70. Springer, 2003.
2. M. Broy and K. Stølen. *Specification and Development of Interactive Systems: FOCUS on Streams, Interfaces, and Refinement*. Springer, 2001.
3. J. He, X. Li, and Z. Liu. rcos: A refinement calculus of object systems. *Theoretical Computer Science*, 365(1-2):109–142, 2006.
4. J. He, X. Li, and Z. Liu. A theory of reactive components. In *Proc. of FACS'05*, volume 160 of *ENTCS*, pages 173–195. Elsevier, 2006.
5. J. He, Z. Liu, and X. Li. Component software engineering. In *Proc of ICTAC'05*, volume 3722 of *LNCS*, pages 269–276. Springer, 2005.
6. J. He, Z. Liu, and X. Li. A theory of contracts. *Technical Report UNU-IIST Report No 327*, July 2005.
7. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
8. C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice-Hall International, 1998.
9. Y. Liu and S. Smith. Modules with interfaces for dynamic linking and communication. In *ECOOP*, volume 3086 of *LNCS*, pages 414–439. Springer, 2004.
10. S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New-age components for old-fashioned java. In *Proc. of OOPSLA 2001*, pages 211–222. ACM, 2001.
11. S. McDirmid and W. Hsieh. Superglue: Component programming with object-oriented signals. In *ECOOP*, volume 4067 of *LNCS*, pages 206–229. Springer, 2006.
12. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1997.
13. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
14. J. Woodcock and C. Morgan. Refinement of state-based concurrent systems. In *Proc. of VDM Europe'90*, volume 428 of *LNCS*, pages 340–351. Springer, 1990.
15. F. Xie and J. Browne. Verified systems by composition from verified components. In *Proc. of ESEC/SIGSOFT FSE 2003*, pages 277–286. ACM, 2003.