Component Publications and Compositions*

Naijun Zhan[1], Eun Young Kang[2], and Zhiming Liu[2]

[1] State Key Lab. of Computer Science, Institute of Software, CAS, Beijing, China
znj@ios.ac.cn
[2] International Institute for Software Technology, United Nations University, Macau
{kang,lzm}@iist.unu.edu

Abstract. One of the major issues in component-based design is how to use a component correctly in different applications according to the given interface specification, called the *publication*, of the component. In this paper we formulate this as the problem of component publication composition and refinement. We define the notion of publications of components that describes how a component can be used by a third party in building their own components or in writing their applications without access to the design or the code of the component. It is desirable that different users of the components can be given different publications according to their need. The first contribution of this paper is to provide a procedure, which calculates a weakest contract of the required interface of a component from the contract of its provided interface and its code. The other contribution, that is more significant from a component-based designer's point of view, is to define composition on publications so that the publication of a composite component can be calculated from those of its subcomponents. For this we define a set of primitive composition operators over components, including *renaming*, *hiding*, *internalizing*, *plugging* and *feedback*. This theory is presented based on the sematic model of rCOS, a refinement calculus of component and object systems.

Keywords: Contracts, Components, Component Publications, and Composition

1 Introduction

The widespread tendency in software and system engineering is towards component-based design [12] by which systems are designed by combining small components into bigger ones. The component-based technique allows a complex design problem to be decomposed by separation of functionality into simpler design problems. It thus helps to decrease the degree of coupling among components and reduce the probability of major accidents caused by combinations of independent component failures [8].

rCOS [5, 4, 1] provides the notions of *interfaces*, *contracts*, *components* and *component publications*. A component is explicitly specified in terms of the contracts of its *provided interface* and *required interface*.

* Naijun Zhan is supported in part by the projects NKBRPC-2002cb312200, NSFC-60493200, NSFC-60721061, NSFC-60573007, NSFC-90718041, and NSFC-60736017, and the other authors are partially supported by the projects HighQSoftD and HTTS funded Macao S&T Fund, STCSM No.08510700300 and NSFC-90718014.

In rCOS, a contract of an interface is a specification of the reactive behavior of the component, including the *interaction protocol* that the environment is assumed to follow, and the data and functionality of each method of the interface [1]. This extends the concept of Meyer's "Design-by-contract" [10], which started out specific to the Eiffel programming language, but is now also used in other languages such as Java and JML [9].

In [3], de Alfaro and Henzinger presented a general theory of composition and refinement of interfaces and components. They also developed a concrete interface theory based on the *interface automata* in [2]. An rCOS contract can be understood as an interface automaton, and a closed rCOS component (i.e. one that does not require services) can be regarded as an I/O automaton of the component. However, a general open component in rCOS has a *provided interface* and a *required interface* and each has a specified contract, meaning that with the *assumption* of the contract for the required interface the component *guarantees* to deliver the specified by the contract of the provided interface. Furthermore, rCOS adopts a declarative approach and denotational semantics. The rCOS contracts also specify rich data structures and functionality of the interface operations in terms of pre- and postconditions in an OO setting. It thus directly supports OO design and implementation of component.

In [1], a procedure is given for an assumed contract of the required interface to calculate a contract of the provided interface. Obviously, it is the *strongest contract of the provided interface* for the given contract of the required interface. However, it is often the case that a component is developed from a specification of its provided services, i.e. a contract of its provided interface. Thus, for a given contract of the provided interface of the component, we need to calculate from the code a contract of its required interface such that the component guarantees the contract of the provided interface. The first contribution in this paper is to give a procedure that for the code of a component and a contract of its provided interface calculates the *weakest contract* of the required interface of the component.

A component vendor normally only provides users with a specification of part of the functionality (i.e. services) according to the users' needs and budgets instead of source code. Such a specification is called a *publication* and is an abstraction of the contracts of the provided and required interfaces. A publication only states the static data functionality of the provided and required methods and an interaction protocol with the environment and it is written in a descriptive style as to serve as a manual for a user to use and for an assembler to assemble it with other components. An assembler composes several simpler components to form a composite component according to their publications. However, a publication of the composite component has to be provided. The other contribution of this paper is to define a set of composition operators on publications. For this, we change the definition of a publication of a component given in [4] such that a publication $(\mathcal{G}, \mathcal{A}, C)$ consists of specifications $\mathcal{G}$ and $\mathcal{A}$ of the data functionality of the provided and required interfaces and an interaction protocol C. The protocol C specifies the interactions with the environments as well as invocation relation of the required methods by the provided methods. In [4], the interactions are separated as provided protocol and the required protocol without the invocation dependency. An invocation dependency oriented protocol C can be represented in different formalisms such as a transition graph, a set of traces, a temporal formula, and a CSP process. In this paper, we use a set of traces of the provided and required methods. In fact, if the set is a regular language, it can also be represented by an automaton [2]. The composition operators we are to define include *renaming, hiding, internalizing , plugging* and *feedback*. We then show that they are consistent with the corresponding operators on components defined in [4, 1] in the sense that the composite publication is indeed a correct publication for the corresponding composite component if the operand publications are correct for the operand components.

Section 2 briefly introduces the unifying theories of programming (UTP) [7] and some basic notions of traces. Section 3 presents the main modeling elements of component based design in rCOS. Section 4 defines the notion of publications. Section 5 introduces an algorithm for calculating the weakest contract of the required interface of a component from its provided contract and source code. Section 6 reviews the composition operators on components and define their counterparts for publications. We will also investigate the correctness of the compositions on publications with respect to the compositions of components. Section 7 discusses future work and concludes the paper.

2 Preliminaries

In UTP, a *sequential program* (but possibly nondeterministic) is represented by a *design* $D = (\alpha, P)$, where

- α denotes the set of state variables (called observables). Each state variable comes in an unprimed and a primed version, denoting respectively the pre- and the post-state value of the execution of the program. In addition to the program variables and their primed versions such as x and x', the set of observables includes two designated Boolean variables, ok and ok', that denotes termination or stability of the program.
- P is a predicate, denoted by $p(x) \vdash R(x, x')$, and defined as $(ok \wedge p(x)) \Rightarrow (ok' \wedge R(x, x'))$. It means that if the program is activated in a stable state, ok, where the *precondition* $p(x)$ holds, the execution will terminate, ok', in a state where the postcondition R holds; thus the post-state x' and the initial state x are related by relation R. We use $pre.D$ and $post.D$ to denote the pre- and post-conditions of D, respectively. If $p(x)$ is $true$, then P is shortened as $\vdash R(x, x')$.

Definition 1. *Let $D_1 = (\alpha, P_1)$ and $D_2 = (\alpha, P_2)$ be two designs with the same alphabet. D_2 is a* refinement *of D_1, denoted by $D_1 \sqsubseteq D_2$, if the following closed implication holds $\forall x, x', ok, ok' \cdot (P_2 \Rightarrow P_1)$. Let $D_1 = (\alpha_1, P_1)$ and $D_2 = (\alpha_2, P_2)$ be two designs with possible different alphabets $\alpha_1 = \{x, x'\}$ and $\alpha_2 = \{y, y'\}$. D_2 is a* data refinement *of D_1 over $\alpha_1 \times \alpha_2$, denoted by $D_1 \sqsubseteq_d D_2$, if there is a relation $\rho(y, x')$ s.t. $\rho(y, x'); D_1 \sqsubseteq D_2; \rho(y, x')$.*

It is proven in UTP that the domain of designs forms a complete lattice with the refinement partial order, and $true$ is the smallest (*worst*) element of the lattice. Furthermore, this lattice is closed under the classical programming constructs, and these constructs are *monotonic operations* on the lattice. These fundamental mathematical properties ensure that the domain of designs is a proper semantic domain for sequential programming languages. There is a nice link from the theory of designs to the theory of predicate transformers with the definition $\mathbf{wp}(p \vdash R, q) \mathrel{\widehat{=}} p \wedge \neg(R; \neg q)$ that defines the *weakest precondition* of a design for a post condition q.

Semantics of *concurrent and reactive programs* is defined by the notion of *reactive designs* with an additional Boolean observable *wait* that denotes suspension of a program. A design P is a *reactive design* if it is a fixed point of $\mathcal{H}$, i.e. $\mathcal{H}(P) = P$, where $\mathcal{H}(p \vdash R) \mathrel{\widehat{=}} (\mathit{true} \vdash \mathit{wait}') \lhd \mathit{wait} \rhd (p \vdash R)$. Here, $P_1 \lhd b \rhd P_2$ is a conditional statement, which means if b holds then P_1 else P_2, where b is a Boolean expression and P_1 and P_2 are designs. We define a *guarded design* $D = (\alpha, g\& P)$, where P is a design, to specify the reactive behavior $\mathcal{H}(P) \lhd g \rhd (\mathit{true} \vdash \mathit{wait}')$, meaning that if the guard g is false, the program stays suspended, otherwise it behaves like $\mathcal{H}(P)$. We use *guard.D* to denote the guard g and *func.D* to denote its functionality P. A reactive design is to ensure that a synchronization of a method invocation by the environment and the execution of the method can only occur when the guard is true and *wait* is false. The domain of reactive designs enjoys the same closure properties as the domain of sequential designs, and also refinement is defined as logical implication. This allows us reactive designs to define the semantics of concurrent programming languages of guarded commands of the form $g\&c$. For details, we refer the reader to our earlier work in [6].

2.1 Notations for traces

Given a set Σ of events, we use Σ^* to denote the set of all finite traces generated out of Σ in which $\langle\,\rangle$ is a special one, i.e. the empty trace, and Σ^∞ the set of all infinite traces generated from Σ. For a trace $s_1 \in \Sigma^*$ and a trace $s_2 \in \Sigma^* \cup \Sigma^\infty$, $s_1{}^\frown s_2$ is the conventional concatenation operation. We use s^k to denote the concatenation of s k times, where $k \in \mathbb{N} \wedge k \geq 0$. If $k = 0$, then s^k is denoted by $\langle\,\rangle$. s^* denotes $\exists k \in \mathbb{N}.k \geq 0 \wedge s^* = s^k$, whereas s^+ denotes $\exists k \in \mathbb{N}.k > 0 \wedge s^+ = s^k$. This operation is also conventionally overloaded to operate on sets $T_1{}^\frown T_2$ and $E^\frown T_2$, where T_1 is a subset of Σ^*, T_2 a subset of $\Sigma^* \cup \Sigma^\infty$, and E a subset of Σ. A trace s_1 is a *prefix* of s_2, denoted by $s_1 \preceq s_2$, if there exists a trace s such that $s_2 = s_1{}^\frown s$, and s is called a *suffix* of s_2. We use $\mathbf{tail}(s)$ and $\mathbf{head}(s)$ to stand for the tail and the head of s, respectively. We use $s[b/a]$ to denote the trace obtained from s by replacing all occurrences of a with b, and $T[b/a]$ the set of traces obtained from T by replacing a with b in each trace of T. The *projection* of a trace s on a set E of events, denoted by $s \downharpoonright E$, is the trace obtained from s by removing from it all events that are not in E, and we write $s \downharpoonright a$ when E contains only one element a. We also overload this operation and extend it to define the projection of a set T of traces on a set E of events $T \downharpoonright E$. The *restriction* of a trace set T on a set of events M, denoted by $T \backslash M$, is defined by $\{s \mid s \downharpoonright M = \langle\rangle \wedge (\exists t \in T \exists a \in M.s^\frown\langle a\rangle \preceq t \vee s \in T)\}$. If M is a singleton $\{m\}$, $T \backslash M$ is shortened by $T \backslash m$. For simplicity, we denote $s_1 + s_2$ as $\{s_1\} \cup \{s_2\}$, and $T_1 + T_2$ as $T_1 \cup T_2$. We now define the synchronization between s_1 and s_2 via a set E of events, denoted $s_1 \|_E s_2$, as follows:

$$s_1 \|_E s_2 = \begin{cases} \langle\,\rangle & \text{if } (s_1 = \langle\rangle \wedge s_2 = \langle\rangle) \vee (s_1 = \langle\rangle \wedge s_2 = \langle a\rangle^\frown s_2' \wedge a \in E) \vee \\ & \quad (s_2 = \langle\rangle \wedge s_1 = \langle a\rangle^\frown s_1' \wedge a \in E), \\ \langle a\rangle^\frown(s_1' \|_E s_2) & \text{if } s_1 = \langle a\rangle^\frown s_1' \wedge a \notin E \\ \langle a\rangle^\frown(s_1 \|_E s_2') & \text{if } s_2 = \langle a\rangle^\frown s_2' \wedge a \notin E \\ \langle a\rangle^\frown(s_1' \|_E s_2') & \text{if } s_1 = \langle a\rangle^\frown s_1' \wedge s_2 = \langle a\rangle^\frown s_2' \wedge a \in E \end{cases}$$

When E is empty, $s_1 \|_E s_2$ represents the interleaving of s_1 and s_2, shortened by $s_1 ||| s_2$.

3 Contract and Component

We provide models of components at different levels of abstraction. A *component* is a unit of software that *implements* a functionality via its *provided interface*. The functionality is specified as a *contract* of the provided interface. A component, to implement the specified contract, may also *require* or *assume* services from other components. The *required services* are specified by a contract of the *required interface* of the component.

3.1 Contract

An interface $I = \langle F, M\rangle$ provides the syntactic declarations of a set of *fields* and a set of *signatures of operations* (or methods). Each field is declared with type $x : T$, and a signature of an operation is given by a method name, some input and output parameters. For theoretical treatment, we assume one operation has only one input parameter and at most one output parameter and is written as $m(in; out)$, where each of in and out declares a variable with a type. We use $field.I$ and $Meth.I$ to refer to the fields and operations of interface I.

A contract of an interface specifies the functionality of the methods declared in the interface, the protocol of the interaction with the environment, and the reactive behavior of the component.

Definition 2. *A* contract *is a tuple $C = (I, \theta, \mathcal{S}, \mathcal{T})$, where*

- *I is an interface, denoted by $IF.C$; and we use $Meth.C$ for $Meth.IF.C$ and $field.C$ for $field.IF.C$,*
- *θ, denoted by $init.C$, is a design $\vdash R \wedge \neg wait'$ that initializes the values of $field.C$, i.e. defines the initial states,*
- *$\mathcal{S}$, denoted by $spec.C$, specifies each method $m(in; out)$ in $IF.C$ by a guarded design $\mathcal{S}(m)$,*
- *$\mathcal{T}$ is called the* protocol *and denoted by $prot.C$, which is a set of traces of the events over $Meth.C$*[1].

Example 1. A contract of one-place buffer $B_1 = (I, \theta, \mathcal{S}, \mathcal{T})$ is described as follows:

$$\begin{array}{rcl}
I & \widehat{=} & \langle \{buff{:}int^*\}, \{put(in\, x{:}int), get(out\, y{:}int)\}\rangle \\
\theta & \widehat{=} & \vdash buff' = \langle\,\rangle \\
\mathcal{S}(put(in\ x{:}int)) & \widehat{=} & buff = \langle\,\rangle \&(\vdash buff'{=}\langle x\rangle^\frown buff) \\
\mathcal{S}(get(\text{out}\ y{:}int)) & \widehat{=} & buff \neq \langle\,\rangle \&(\vdash buff' = \mathbf{tail}(buff) \wedge y' = \mathbf{head}(buff)) \\
\mathcal{T} & \widehat{=} & (\langle put\rangle^\frown\langle get\rangle)^* + (\langle put\rangle^\frown(\langle get\rangle^\frown\langle put\rangle)^*)
\end{array}$$

Given a contract $C = (I, \theta, \mathcal{S}, \mathcal{T})$, let $Meth.C^+ \mathrel{\widehat{=}} \{m(u; v) \mid m(x : T_1; y : T_2) \in Meth.I \wedge u \in T_1 \wedge v \in T_2\}$. The dynamic semantics of C is given by a divergence set $\mathcal{D}(C)$ and a failure

[1] Notice that this is an abstract version of the protocol in our earlier versions [6, 1].

set $\mathcal{F}(C)$. $\mathcal{D}(C)$ is the set of all traces $m_1(u_1;v_1),\ldots,m_k(u_k;v_k)$ over $Meth.C^+$ such that the execution of these invocations of a prefix of the trace in consecution from the initial state enters a diverging state. Also, the *failure set* of C is the set of pairs $\langle s,X\rangle$ such that either after the execution of the trace s over $Meth.C^+$, all the events in $X\subseteq Meth.C^+$ are not enabled, i.e their guards are disabled, or s is a divergence. The failure-divergence semantics of contracts allows us to use the CSP failure-divergence partial order [11] as a refinement relation between contracts [1], denoted by $C_1\sqsubseteq C_2$. C_1 and C_2 are equivalent, denoted $C_1\equiv C_2$, if $C_1\sqsubseteq C_2\wedge C_2\sqsubseteq C_1$. It is noted that this refinement relation requires that the interfaces $Meth.IF.C_1$ and $Meth.IF.C_2$ have exactly the same set of methods.

A contract C has to be *consistent* in the sense that no execution of a trace in the protocol from an initial state may enter a *blocking state* in which *wait* is *true* or a *diverging state* in which *ok'* is false. The notion of consistency is defined in [1] and a theorem of *separation of concerns* is proven there that allows the refinement of a design $func.spec.C(m)$ without violating the consistency, and $C=(I,\theta,\mathcal{S},\mathcal{T}_1+\mathcal{T}_2)$ is consistent iff $C_i=(I,\theta,\mathcal{S},\mathcal{T}_i)$ are both consistent, $i=1,2$. Furthermore, for a triple $(I,\theta,\mathcal{S})$ there exists a *largest protocol* $\mathcal{T}$ such that contract $C=(I,\theta,\mathcal{S},\mathcal{T})$ is consistent, and called a *complete contract*. A complete contract can be simply written as $C=(I,\theta,\mathcal{S})$ by omitting its protocol, and we use $trace.C$ to denote the largest protocol of C.

For a trace tr over $Meth.C^+$, we define an *abstraction* tr^-, that is a trace over the events $Meth.C^+$: $\langle\rangle^-=\langle\rangle$, and $(\langle m(u;v)\rangle^\frown tr)^-=\langle m\rangle^\frown tr^-$. Thus, we have the following theorem of the relation between the traces and the failure set for a complete contract.

Theorem 1. *For a complete contract C, trace.C* $=\{tr^-\mid(tr,X)\in\mathcal{F}(C)\wedge tr\notin\mathcal{D}(C)\}$.

Example 2. For the one-place buffer in Example 1, we can further give the following two contracts $B_2=(IF.B_1,init.B_1,\mathcal{S}_2,\mathcal{T}_2)$ and $B_3=(IF.B_1,init.B_1,\mathcal{S}_3,\mathcal{T}_3)$, where

$$\begin{array}{rcl}
\mathcal{S}_2(put(\text{in } x{:}int)) & \widehat{=} & (\vdash buff'=\langle x\rangle^\frown buff)\lhd buff=\langle\,\rangle\rhd(\vdash buff'=buff)\\
\mathcal{S}_2(get(\text{out } y{:}int)) & \widehat{=} & buff\neq\langle\,\rangle\&(\vdash buff'=\mathbf{tail}(buff)\wedge y'=\mathbf{head}(buff))\\
\mathcal{T}_2 & \widehat{=} & (\langle put\rangle^\frown\langle put\rangle^{*\frown}\langle get\rangle)^*\\
\mathcal{S}_3(put(\text{in } x{:}int)) & \widehat{=} & buff=\langle\,\rangle\&(\vdash buff'=\langle x\rangle^\frown buff)\\
\mathcal{S}_3(get(\text{out } y{:}int)) & \widehat{=} & (\vdash buff'=\mathbf{tail}(buff)\wedge y'=\mathbf{head}(buff))\lhd buff\neq\langle\,\rangle\rhd\\
 & & (\vdash\exists c\in int.buff'=buff\wedge y'=c)\\
\mathcal{T}_3 & \widehat{=} & (\langle get\rangle+(\langle put\rangle^\frown\langle get\rangle))^*
\end{array}$$

We see that B_1,B_2 and B_3 are complete contracts satisfying $B_1\sqsubseteq B_2\wedge B_1\sqsubseteq B_3$, but $B_2\not\sqsubseteq B_3$ and $B_3\not\sqsubseteq B_2$. □

The following theorem indicates that any contract is equivalent to a complete contract.

Theorem 2. *Given a contract* $C=(I,\theta,\mathcal{S},\mathcal{T})$*, let* C' *be* $(\langle field.C\cup\{tr:Meth.C^*\},Meth.C\rangle,$ $init.C\wedge tr'=\langle\,\rangle,\mathcal{S}',\mathcal{T})$*, where* $\mathcal{S}'(m)=(\exists s\in\mathcal{T}.tr^\frown\langle m\rangle\leq s\wedge guard.\mathcal{S}(m))\&(pre.\mathcal{S}(m)\vdash post.\mathcal{S}(m)\wedge tr'=tr^\frown\langle m\rangle)$*. Then,* C' *is complete, and* $trace.C'=\mathcal{T}\wedge C'\equiv C$.

Based on this theorem, in what follows, we will only focus on complete contracts, and therefore all contracts are referred to as complete contracts, if not otherwise stated.

3.2 Component

A component K is an *implementation* of a contract of an interface that provides services to other components. This interface is called the *provided interface*. In order to implement the provided services, K may *use* services provided by other components via an interface, called the *required interface*.

Definition 3. *A* component *is a tuple* $K = (I, M, c_0, \mathbb{C}, J)$*, where*

- I *is an interface, called the* provided interface *of* K *and denoted by* $pIF.K$*. We also write* $pMeth.K$ *for* $Meth.pIF.K$ *and* $pfield.K$ *for* $field.pIF.K$*.*
- M *is a set of method signatures, called the* private methods *of* K *and denoted by* $priMeth.K$*.*
- c_0 *is the initialization statement of the component, denoted by* $init.K$*, that initializes the set of states of the component.*
- $\mathbb{C}$ *is called the* coding function *and denoted as* $code.K$ *that maps each method in* $pMeth.K \cup priMeth.K$ *to a guarded command.*
- J *is an interface, called the* required interface *of* K *and denoted by* $rIF.K$*. We also write* $rMeth.K$ *for* $Meth.rIF.K$ *and* $rfield.K$ *for* $field.rIF.K$*. It is required that* $rMeth.K$ *contains all the methods that occur in the code of the methods given by* $code.K$*, but not declared in* $pMeth.K \cup priMeth.K$*.*

The code in guarded command of each method can be defined as a *reactive design*. For a given contract C_r of the required interface $rIF.K$ of K, a contract C_p of the provided interface $pIF.K$ can be calculated from the code of the methods given by $code.K$. This determines a function $\lambda C_r \cdot spec.K$ such that for a complete contract C_r of $rIF.K$, $spec.K.C_r$ is a complete contract of $pIF.K$. We take the function $spec.K$ as the semantics of component K[6]. This semantics enjoys the following property that for two contracts C_1 and C_2 of $rIF.K$, if $C_1 \sqsubseteq C_2$ then $spec.K.C_1 \sqsubseteq spec.K.C_2$.

We say that a component K *implements* a contract C_p of its provided interface $pIF.K$ with a contract C_r of its provided interface $rIF.K$ if $C_p \sqsubseteq spec.K.C_r$, and K *implements* C_p if there exists such a contract C_r. Obviously, $spec.K.C_r$ is the *strongest contract* that K implements with C_r.

Example 3. The following three components K_1, K_2 and K_3 respectively implement the contracts B_1 in Example 1 and B_2 and B_2 in Example 2. For convenience, we shall rename some method and field in the interface.

$$
\begin{aligned}
&pIF.K_1 &&= IF.B_1,\\
&priMeth.K_1 &&= \emptyset,\\
&init.K_1 &&= buff := \langle\,\rangle,\\
&code.K_1(put) &&= buff{=}\langle\rangle \rightarrow (buff{:=}\langle x\rangle),\\
&code.K_1(get) &&= buff{\neq}\langle\rangle \rightarrow (y{:=}\textbf{head}(buff); buff{:=}\langle\,\rangle)\\
&rIF.K_1 &&= \emptyset
\end{aligned}
$$

$$
\begin{aligned}
&pIF.K_2 &&= IF.B_2[buff_1/buff, get_1/get],\\
&priMeth.K_2 &&= \emptyset,\\
&init.K_2 &&= buff_1 := \langle\,\rangle,\\
&code.K_2(put) &&= buff_1{:=}\langle x\rangle \lhd buff_1{=}\langle\rangle \rhd put_1(\textbf{head}(buff_1)); buff_1 := \langle x\rangle\\
&code.K_2(get_1) &&= buff_1{\neq}\langle\rangle \rightarrow (y{:=}\textbf{head}(buff_1); buff_1{:=}\langle\,\rangle)\\
&rIF.K_2 &&= \langle\{put_1(in\,x{:}int)\}\rangle
\end{aligned}
$$

$$\begin{array}{ll}
pIF.K_3 & = IF.B_3[buff_2/buff, put_1/put], \\
priMeth.K_3 & = \emptyset, \\
init.K_3 & = buff_2 := \langle\,\rangle, \\
code.K_3(put_1) & = buff_2{=}\langle\rangle \rightarrow buff_2{:=}\langle x\rangle \\
code.K_3(get) & = (y{:=}\mathbf{head}(buff_2); buff_2{:=}\langle\rangle) \lhd buff_2{\neq}\langle\rangle \rhd get_1(y) \\
rIF.K_3 & = \langle\{get_1(out\, y{:}int)\}\rangle
\end{array}$$

4 Publications of Components

To compose components and use components to write applications, one does not need to know their code or even their design. However, one needs a specification to some extent about what services are provided and what services are required and the protocol that describes the interactions with the environments. The idea is that the less details specified the better. In this section we define such as specification, called a *publication* of a component. For a generic representation, we first define the notion of a *specification* of a component.

Definition 4. *A* specification *of a component K is a triple $S = (\mathcal{P}, \mathcal{R}, C)$, where*

- *$\mathcal{P}$ is a complete contract of $pIF.K$, denoted by $pCtr.S$;*
- *$\mathcal{R}$ is a complete contract of $rIF.K$, denoted by $rCtr.S$;*
- *$C \subseteq (pMeth.K + rMeth.K)^*$, denoted by $causal.S$, is a protocol that specifies the interactions with the environments as well as invocation relation of the required methods by the provided methods of K, called the* invocation dependency oriented protocol *of S,*

such that the following conditions are satisfied

1. *$\mathcal{P} \sqsubseteq spec.K.\mathcal{R}$; and*
2. *$causal.S \downharpoonright pMeth.K \supseteq trace.pCtr.S \wedge causal.S \downharpoonright rMeth.K \subseteq trace.rCtr.S$.*

The first condition indicates that with K's required contract, K implements its provided contract, while the second condition says that projecting the invocation dependency oriented protocol onto the provided methods results in a protocol of the provided contract that is consistent with the specification of the methods; but projecting the invocation dependency oriented protocol onto the required methods results in a protocol that is a subset of the largest protocol of the required contract. This is just an analog of the law of strengthening the postcondition and weakening the precondition in Hoare logic of programs. Verifying the two conditions can be done by checking a design document of the component that contains the verification of the refinement relation, by verification of the source code [2]. The refinement relation between specifications of component can be defined from the refinement of contracts.

Definition 5. *For two specifications S_1 and S_2 of K, $S_1 \sqsubseteq S_2$, if*

[2] Such a verification should be part of the *certification* of the component.

1. $pCtr.S_2$ is a refinement of $pCtr.S_1$;
2. $rCtr.S_1$ is a refinement of $rCtr.S_2$; and
3. $\forall c \in C_2.c \upharpoonright Meth.\mathcal{P}_2 \in prot.\mathcal{P}_1 \Rightarrow c \in C_1$.

The conditions 1&2 say a refined specification should have a stronger provided contract and a weaker required contract. Condition 3 indicates that a refined specification provides more services to and requires less services from the environment and it is equivalent to

$$causal.S_1 \upharpoonright pMeth.K \subseteq causal.S_2 \upharpoonright pMeth.K \wedge causal.S_1 \upharpoonright rMeth.K \supseteq causal.S_2 \upharpoonright rMeth.K.$$

The complete contracts in a specification are given in terms of reactive designs. Therefore, they are not easy to be used for checking their compatibility with the specifications of other components. Further, the guards in the method specification and the protocol provide duplicated information to the user. Therefore, one does not need to have both when they compose and use components. We thus define the notion of *publication* by removing the guards in the method specification. We first define each part of a component publication as *publication contract*.

Definition 6. *A* publication contract *C is a tuple $(I, \theta, \mathcal{D}, \mathcal{T})$, where*

- *I is an interface and θ is an initialization design,*
- *$\mathcal{D}$ is a function, denoted by spec.C that defines each method m of I with a design (no guard) $\mathcal{D}(m)$,*
- *$\mathcal{T}$ is a set of traces over the Meth.I, denoted by prot.C.*

Definition 7. *A* publication *of component K is $U = (\mathcal{G}, \mathcal{A}, C)$ where*

- *$\mathcal{G}$ is a publication contract of an interface I such that $Meth.I \subseteq pMeth.K$,*
- *$\mathcal{A}$ is a publication contract of an interface J such that $Meth.J \supseteq rMeth.K$, and*
- *C is a causal relation over $Meth.I + Meth.J$, denoted by causal.U, such that*

$$causal.U \upharpoonright Meth.I = prot.\mathcal{G} \wedge causal.U \upharpoonright Meth.J = prot.\mathcal{A}.$$

Definition 7 allows the component vendor to give different publications to different component users. This is characterized by the refinement relation between publications.

Definition 8. *For a component K, let $U_1 = (\mathcal{G}_1, \mathcal{A}_1, C_1)$ and $U_2 = (\mathcal{G}_2, \mathcal{A}_2, C_2)$ be publications of K. U_2 is a refinement of U_1, $U_1 \sqsubseteq U_2$, if*

1. $Meth.pCtr.U_1 \subseteq Meth.pCtr.U_2$, $Meth.rCtr.U_1 \supseteq Meth.rCtr.U_2$,
2. $init.pCtr.U_1 \sqsubseteq init.pCtr.U_2$, and $init.rCtr.U_1 \sqsupseteq init.rCtr.U_2$,
3. $\forall m \in Meth.pCtr.U_1.spec.pCtr.U_2(m) \sqsubseteq spec.pCtr.U_2(m)$, and
$\forall n \in Meth.rCtr.U_2.spec.rCtr.U_1(n) \sqsupseteq spec.rCtr.U_2(n)$,
4. $prot.pCtr.U_1 \subseteq prot.pCtr.U_2$, and $prot.rCtr.U_1 \supseteq prot.rCtr.U_2$,
5. $\forall c \in C_2.c \upharpoonright Meth.\mathcal{G}_2 \in prot.\mathcal{G}_1 \Rightarrow c \in C_1$.

Condition 1 says that a refined publication has more provided methods and less required methods; Condition 2 indicates that a refined publication has a stronger initial condition on the provided fields and a weaker initial condition on the required fields; Condition 3 expresses that a refined publication assigns a stronger specification (design) to each provided method, while a weaker specification (design) to each required method; Conditions 4&5 indicate that a refined publication is more likely to provide services to its environment, but less likely to invoke services provided by environment.

4.1 Specification vs Publication

A publication of a component has to be certified and this is done by the verification of the validity of a specification of the component. This is done by relating a contract and a publication contract.

Definition 9. *We define a mapping $\mathcal{M}$ from the domain of complete contracts to that of publication contracts as: for a given complete contract $C = (I, \theta, \mathcal{S})$, $\mathcal{M}(C)$ is a publication contract defined by*

1. $IF.\mathcal{M}(C) = IF.C = I$;
2. $init.\mathcal{M}(C) = init.C[false/wait, false/wait'] = \theta[false/wait, false/wait']$;
3. $spec.\mathcal{M}(C))(m) = P[false/wait, false/wait']$, *if* $spec.C(m) = g\&P$, *for any* $m \in Meth.C$;
4. $prot.\mathcal{M}(C) = trace.C$.

Then we have the following equivalence relation in terms of contracts.

Theorem 3. *For any complete contract C, we have $C \equiv \mathcal{M}(C)$.*

This theorem indicates that we can use a protocol instead of the guards of the provided methods of a component to control the interaction between the component and its environment.

Definition 10. *Conversely, we define a mapping $\mathcal{L}$ from the domain of publication contracts to the domain of complete contracts as: for a given publication contract $C = (I, \theta, \mathcal{D}, \mathcal{T})$, $\mathcal{L}(C)$ is a complete contract defined by*

1. $IF.\mathcal{L}(C) = \langle field.I \cup \{tr : Meth.C^*\}, Meth.I\rangle$;
2. $init.\mathcal{L}(C) = init.C \wedge tr' = \langle\,\rangle \wedge \neg wait' = \theta \wedge tr' = \langle\,\rangle \wedge \neg wait'$;
3. $spec.\mathcal{L}(C))(m) = (\exists s \in \mathcal{T}.tr^\frown\langle m\rangle \preceq s)\&\mathcal{D}(m) \wedge tr' = tr^\frown\langle m\rangle$, *for any* $m \in Meth.C$.

Note that the idea of this definition is similar to Theorem 2 by strengthening the guard of each method to obtain a complete contract. We also have

Theorem 4. *For any publication contract C, we have $C \equiv \mathcal{L}(C)$.*

This theorem indicates that we can add a guard to each of the provided methods of a component instead of its protocol to control the interaction between the component and its environment.

Theorem 3 and Theorem 4 indicate that $\mathcal{M}$ and $\mathcal{L}$ form a Galois connection, and imply that interaction between a component and its environment can be done either decentralizedly by the guards of its provided methods or centralizedly by a protocol.

Corollary 1. *(***Contract and Publication Contract***)*

1. *$\mathcal{L}(\mathcal{M}(C))$ is a complete contract for any complete contract C, and $C \equiv \mathcal{L}(\mathcal{M}(C))$; and*
2. *$\mathcal{M}(\mathcal{L}(C))$ is a publication contract for any publication contract C, and $C \equiv \mathcal{M}(\mathcal{L}(C))$.*

The connection between specification and publication of component is expressed as follows:

Theorem 5. *(***Specification vs Publication***)*

1. *If $S = (\mathcal{P}, \mathcal{R}, C)$ is a specification of K, then $U = (\mathcal{M}(\mathcal{P}), \mathcal{M}(\mathcal{R}), C)$ is its publication;*
2. *If $U = (\mathcal{G}, \mathcal{A}, C)$ is a publication of K, then $P = (\mathcal{L}(\mathcal{G}), \mathcal{L}(\mathcal{A}), C)$ is a specification of $K[IF.\mathcal{G}/IF.K, IF.\mathcal{A}/rIF.K]$, where $K[IF.\mathcal{G}/IF.K, IF.\mathcal{A}/rIF.K]$ is the component derived from K by restricting its provided methods to $IF.\mathcal{G}$ and extending its required methods to $IF.\mathcal{A}$.*

5 Calculate Weakest Required Contract and Publication

To calculate the weakest required contract, *wrc.K.pCtr* for a component K to implement a given provided contract *pCtr*, we first calculate the invocation dependency oriented protocol *ioprot.K.pCtr* of K from its code and the protocol of *pCtr*. We then derive from this protocol and the functionality specification of the methods in *pCtr* the weakest required contract.

5.1 Calculating Invocation Dependency Oriented Protocol

Let K be a component and assume $pMeth.K = \{m_1, \cdots, m_k\}$, $priMeth.K = \{n_1, \cdots, n_\ell\}$ and $rMeth.K = \{r_1, \cdots, r_e\}$. For any method $m \in pMeth.K$, we calculate the set X_m of sequences of invocations to methods in *rMeth.K* which are the possible invocation sequences to required methods in the execution of the code $code.K(m)$ of m. We define a function that for a program command c computes the set $\mathcal{T}r(c)$ of invocation sequences in the execution of c:

$$\mathcal{T}r(c) \mathrel{\widehat{=}} \begin{cases} \langle m \rangle & \text{if } c = m(x,y) \text{ or } c = z := m(x,y), \text{ where } m \in rMeth.K \\ X_m & \text{if } c = m(x,y) \text{ or } c = z := m(x,y), \\ & \text{where } m \in pMeth.K \cup priMeth.K \\ \mathcal{T}r(c_1)^\frown\mathcal{T}r(c_2) & \text{if } c = c_1; c_2 \\ \mathcal{T}r(B)^\frown(\mathcal{T}r(c_1) + \mathcal{T}r(c_2)) & \text{if } c = \textbf{if } B \textbf{ then } c_1 \textbf{ else } c_2 \\ (\mathcal{T}r(B)^\frown\mathcal{T}r(c_1))^* & \text{if } c = \textbf{while } B \textbf{ do } c_1 \\ \langle\rangle & \text{otherwise} \end{cases}$$

Using function $\mathcal{T}r$, we define the following $k + \ell$ trace equations for the provided and private methods of component K:

$$\begin{array}{l} X_{m_1} = \mathcal{T}r(code.K(m_1)), \ldots, X_{m_k} = \mathcal{T}r(code.K(m_k)), \\ X_{n_1} = \mathcal{T}r(code.K(n_1)), \ \ldots, \ X_{n_\ell} = \mathcal{T}r(code.K(n_\ell)). \end{array}$$

Note that since a provided method could call required methods implicitly via calling private methods of K, we have to consider the provided methods together with the private methods in the trace equations (1). It is easy to see that these trace equations contain recursion because a provided or a private method can call any other provided or private methods, including themselves. Since all trace operations used in $\mathcal{T}r$ are monotonic w.r.t the set containment. The least fixed points of the above trace equations exist and are taken to be the solutions to the variables X_{m_i} and X_{n_j} for $1 \le i \le k$ and $1 \le j \le l$, and each of them is a subset of $(rMeth.K)^*$.

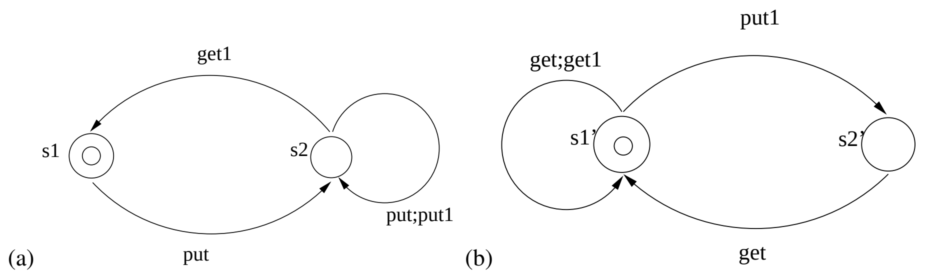

Fig. 1. (a) Transition Graph of *code*.K_2, (b) Transition Graph of *code*.K_3

For example, for K_2 and K_3 in Example 3, we have $X_{put} = \langle\rangle + put_1$, $X_{get_1} = \langle\rangle$ for K_2 and $X_{get} = \langle\rangle + get_1$, $X_{put_1} = \langle\rangle$ for K_3.

For each provided method m, the solution to X_m contains all possible sequences of the invocations to the methods in *rMeth.K* in the code of m. However, in an invocation sequence of the provided methods, each execution (occurrence) of a provided method m might contain only part of the invocation sequences in X_m. This is due to, for instance, a conditional. Thus, we have to calculate this subset X_m^i of X_m for each occurrence m^i of each provided method m in *prot.pCtr* according to the transition graph of the component. The invocation dependency oriented protocol *ioprot.K.pCtr* can be obtained by replacing each occurrence of a provided method m in *prot.pCtr* with $m\hat{\ }X_m^i$ that is the event m concatenated with the invocation sequences of this occurrence of m.

Example 4 According to the code of *put*, we know the fact that if $buff_1 = \langle\rangle$ then *put* does not invoke put_1, otherwise it indeed invokes put_1. On the other hand, we have $X_{put} = \langle\rangle + \langle put_1\rangle$ and if we directly replace *put* in *prot*.B_2 with $\langle put\rangle\hat{\ }X_{put}$, the resulting execution trace *prot*.$B_2[\langle put\rangle/X_{put}]$ will violate the above fact.

The transition graphs for *code*.K_2 and *code*.K_3 in Example 3 are given in Fig.1. According to Fig.1. (a), we know in the cases of its first execution and each execution following get_1, *put* does not invoke put_1, and in other cases it invokes put_1. Therefore, we have to replace the first occurrence of *put* by *put*, and any other occurrences of *put* with $put; put_1$ in the subexpression $\langle put\rangle; \langle put\rangle^*; \langle get_1\rangle$ of *prot*.B_2. This derives *ioprot.K*.B_2 as $(\langle put\rangle\hat{\ }\langle put; put_1\rangle^*\hat{\ }\langle get_1\rangle)^*$, written as C_{B_2}. Similarly, the invocation dependent protocol *ioprot*.K_3.$B_3$of K_3 for B_3 is $((\langle put_1\rangle\hat{\ }\langle get\rangle) + (\langle get; get_1\rangle)^*)^*$. □

5.2 Calculating Weakest Required Contract

After calculating the invocation dependency oriented protocol *ioprot.K.pCtr*, we can easily obtain the protocol of the required contract *wrc.K.pCtr* by projecting it onto the required methods: $prot.wrc.K.pCtr \mathrel{\widehat{=}} ioprot.K.pCtr \downharpoonright rMeth.K$. With this protocol and Theorem 2, we only need to calculate the specification of the data functionality of the required methods. In other words, if we obtain the unguarded designs of the required methods in *rMeth.K*, together with *ioprot.K.pCtr* we form a publication contract, and then *wrc.K.pCtr* is the contract of the required interface obtained from Theorem 2.

Let $rMeth.K = \{r_1, \ldots, r_t\}$, and for each $r \in rMeth.K$ let D_r represent the design for r. Then, $D_{r_1}, \ldots, D_{r_t}$ are calculated as the weakest solution to the following equation family.

$$\begin{aligned}
[\![n_1]\!] &\equiv [\![code.K(n_1)]\!][D_{r_1},\cdots,D_{r_t}/r_1,\cdots,r_k],\\
&\vdots\\
[\![n_\ell]\!] &\equiv [\![code.K(n_\ell)]\!][D_{r_1},\cdots,D_{r_t}/r_1,\cdots,r_t],\\
spec.pCtr(m_1) &\sqsubseteq [\![code.K(m_1)]\!][D_{r_1},\cdots,D_{r_t}/r_1,\cdots,r_t],\\
&\vdots\\
spec.pCtr(m_k) &\sqsubseteq [\![code.K]\!](m_k)[D_{r_1},\cdots,D_{r_t}/r_1,\cdots,r_t]
\end{aligned} \tag{1}$$

Solving the equations (1) is essentially equivalent to the problem of decomposition of sequential programs in the denotational setting. In general, the equations are not solvable. Nevertheless under some special restrictions, such as each occurrence of D_{r_i}, $i = 1,\cdots,t$ is linear, the equations can be solvable. The initial condition of $rCtr$ can be derived from $init.pCtr$ and $init.K$.

Example 5. We apply the above calculation procedure to Example 4. First from Theorem 2, $IF.wrc.K_2.B_2 = \langle\{tr : put_1^*\}, \{put_1(in\, x{:}int)\}\rangle$, and the protocol $prot.wrc.K_2.B_2 = put_1^*$, the design of the method put_1 is $\vdash tr' = tr^\frown\langle put_1\rangle$, and the initial condition is $tr = \langle\rangle$.

Similarly, for K_3 and B_3, the interface $IF.wrc.K_3.B_3 = \langle\{tr : get_1^*\}, \{get_1(out\, y{:}int)\}\rangle$, the initial condition $init.wrc.K_3.B_3 = tr = \langle\rangle$, the design of get_1 is $\vdash tr' = tr^\frown\langle get_1\rangle$, and protocol $prot.wrc.K_3.B_3 = get_1^*$. □

6 Compositions of Components and Their Publications

Composing a composite component from existing simpler ones via connectors plays a key role in component-based methods. In this section, we first review and revise the compositions on component given in [4, 1]. However, the main contribution in this section is to define composition on component publications and present their relation to the compositions of components.

6.1 Compositions of Components

We define the basic operators of components including *renaming*, *hiding*, *internalizing*, *plugging* and *feedback*.

Renaming. Renaming an interface method of a component is a simple connector defined as follows.

Definition 11. *Let K be a component and $m(x : T_1; y : T_2)$ a method signature that does not occur in priMeth.K.*

1. *Renaming a provided $n(u : T_1, v : T_2)$ in pMeth.K gives a component $K[m/n]$ such that*
   - *$pIF.K[m/n] = \langle field.pIF.K,\ Meth.pIF.K + \{m\} - \{n\}\rangle$, $priMeth.K[m/n] = priMeth.K$ and $rIF.K[m/n] = rIF.K$;*
   - *$init.K[m/n] = (init.K)[m/n]$;*

  - *$code.K[m/n](m) = (code.K(n))[m/n]$, and $code.K[m/n](op) = (code.K(op))[m/n]$ for any other op in $pMeth.K[m/n] \cup priMeth.K[m/n]$.*
2. *Renaming a required method $n(u : T_1, v : T_2)$ in $rMeth.K$ gives the component $K[m/n]$ such that*
  - *$pIF.K[m/n] = pIF.K$, $priMeth.K[m/n] = priMeth.K$ and $rIF.K[m/n] = \langle field.rIF.K, Meth.rIF.K - \{n\} + \{m\}\rangle$;*
  - *$init.K[m/n] = (init.K)[m/n]$;*
  - *$code.K[m/n](op) = code.K(op)[m/n]$ for any op in $pMeth.K[m/n] \cup priMeth.K[m/n]$.*

Where $c[m/n]$ is the command obtained from c by replacing each occurrence of n with m.

Notice that in the above definition, the code of a provided method, a private method, or the initiation statement may contain some invocations to n, so we have to rename n to m in the corresponding code of the renamed component. Besides, $K[m/n] = K$ if n does not occur in K.

A renamed component $K[m/n]$ can be easily implemented by using a connector, which is a component that provides the method with the fresh name m and the body of m calls the provided method n of K.

Hiding. We sometimes restrict a user from using some provided methods of a component by hiding these methods ($K\backslash m$). Hiding is semantically the same as moving the hidden methods from the provided interface to the set of private methods of the component. Formally,

Definition 12. *Let $K = (I, M, c_0, \mathbb{C}, J)$ be a component, $m \in Meth.I$. Hiding m in K is denoted by $K\backslash m$ and defined as $(\langle Meth.I - \{m\}, field.I\rangle, M \cup \{m\}, c_0, \mathbb{C}, J)$.*

The hiding operator is associative. Therefore hiding a set of provided methods is same as hiding them one by one. Notice that hiding should be used carefully as hiding a method may result in a dead component. E.g. in Example 3, $K_1\backslash put$ results in a deadlock.

$K\backslash m$ can be implemented by renaming each provided method n of K to a fresh method n_1, and by adding a connector component that provides all the methods n that K provides except for m, and each n calls its code, which is the renamed method n_1 of the renamed component.

Internalizing. Similar to hiding, *internalizing* a method m in a component K is to remove it from the provided interface of K and add it into the private method set, denoted by $K\swarrow\!\!/\, m$. However, unlike hiding, internalizing just changes all explicit invocations to the internalized method to implicit invocations to the method. For example, in Example 3, internalizing get in K_1 results in a new component that provides only put, but every execution of put will implicitly be followed by an execution of get, therefore it allows any number of put operations on consecution. Formally,

Definition 13. *For a component K and a set of methods M in its provided interface, we define the component $K\swarrow\!\!/\, M$ as*

- *$pIF.K\swarrow\!\!/\, M = (pIF.K)\backslash M$, $priMeth.K\swarrow\!\!/\, M = priMeth.K + M$, $rIF.K\swarrow\!\!/\, M = rIF.K$,*

– $init.K \swarrow\!\!\!/ M = init.K$,
– $(code.K \swarrow\!\!\!/ M)(n)$ *can be defined in different ways:*
  - $(code.K \swarrow\!\!\!/ M)(n) = \sqcap_{s \in M*} code.K(s); code.K(n); \sqcap_{s \in M^*} code.K(s)$ *for* $n \in pMeth.K \swarrow\!\!\!/ M$, *where* $code.K(s)$ *stands for the sequential execution of the methods in sequence* s *and* $\sqcap$ *is the* nondeterministic choice operator *in the program language,*
  - $(code.K \swarrow\!\!\!/ M)(n) = code.K(n)$ *for* $n \in priMeth.K$
  - $(code.K \swarrow\!\!\!/ M)(m) = code.K(m)$ *for* $m \in M$

The above definition indicates internalizing essentially changes all explicit invocations to the internalized methods to implicit invocations. This is semantically equivalent to reprogramming all provided methods in $pMeth.K - M$ by adding possible sequences of invocations to M before and after the code of n, i.e. $code.K(n)$, for each $n \in pMeth.K - M$. However, instead of changing the code, the internalizing connector is implemented by programming a scheduling processes that synchronizes with the component on the internalized methods.

In most cases, the number of invocations to these internalized methods before and after a noninternalized method should be finite; otherwise internalizing must give rise to an divergence, i.e. livelock. We can see this by further investigating the example given in the above. After internalizing get in K_1, it is clear that put can execute infinite many times. Thus, internalizing put in $K_1 \swarrow\!\!\!/ \{get\}$ will cause a divergence.

Plugging The most often used composition in component construction is to plug the provided interface of a component K_1 into the required interface of another K_2, denoted by $K_1 \bowtie K_2$. A component can plug into another component only if they have no name conflicts.

Definition 14. *A component* K_1 *is* pluggable *to a component* K_2 *if the following conditions hold:*

1. $(field.pIF.K_1 \cap field.pIF.K_2) = \emptyset$, *and* $(pMeth.K_1 \cap pMeth.K_2) = \emptyset$;
2. $(priMeth.K_1 \cap priMeth.K_2) = \emptyset$;
3. $(field.rIF.K_1 \cap field.rIF.K_2) = \emptyset$, *and* $(rMeth.K_1 \cap rMeth.K_2) = \emptyset$;
4. $priMeth.K_i \cap (pMeth.K_j + rMeth.K_j + priMeth.K_j) = \emptyset$, *where* $i \neq j$ *and* $i, j = 1, 2$.

Notice that the above conditions can always be guaranteed by renaming conflicting names.

Definition 15. *Let* K_1 *be a component that is pluggable to a component* K_2. *Then plugging* K_1 *to* K_2, *denoted* $K = K_1 \bowtie K_2$, *is defined as follows:*

– $filed.pIF.K = filed.pIF.K_1 + field.pIF.K_2$;
– $pMeth.K = pMeth.K_1 + pMeth.K_2 - rMeth.K_1 - rMeth.K_2$;
– $priMeth.K = priMeth.K_1 + priMeth.K_2 + (pMeth.K_1 \cap rMeth.K_2) + (rMeth.K_1 \cap pMeth.K_2)$;
– $filed.rIF.K = filed.rIF.K_1 + field.rIF.K_2$;
– $rMeth.K = rMeth.K_1 + rMeth.K_2 - pMeth.K_1 - pMeth.K_2$;
– $init.K = init.K_1 \wedge init.K_2$; *and*
– $code.K(m) = (code.K_1 \oplus code.K_2)(m)$ *for each* $m \in \sum_{i=1}^{2} pMeth.K_i + priMeth.K_i$, *where* $\oplus$ *is the union of two functions.*

Notice that we do not allow calling circles in the above definition.

The above definition indicates that the provided methods of K_1 that are plugged to the required methods of K_2 become private and not available to the environment anymore, and vice versa.

Example 6. From the above definition, we know that the components K_2 and K_3 in Example 3 are pluggable, $K_2 \bowtie K_3$ can be defined as:

$$\begin{array}{ll}
pIF.K_2 \bowtie K_3 & = \langle \{buff_1, buff_2{:}int^*\}, \{put(in\,x{:}int), get(out\,y{:}int)\}\rangle \\
priMeth.K_2 \bowtie K_3 & = \{put_1(in\,x{:}int), get_1(out, y{:}int)\} \\
init.K_2 \bowtie K_3 & = buff_1 := \langle\,\rangle; buff_2 := \langle\,\rangle \\
code.K_2 \bowtie K_3(put) & = (buff_1{:=}\langle x\rangle) \lhd buff_1{=}\langle\rangle \rhd (put_1(\mathbf{head}(buff_1))) \\
code.K_2 \bowtie K_3(get) & = (y{:=}\mathbf{head}(buff_2); buff_2{:=}\langle\rangle) \lhd buff_2{\neq}\langle\rangle \rhd get_1(y) \\
code.K_2 \bowtie K_3(get_1) & = (buff_1{\neq}\langle\rangle) \rightarrow (y{:=}\mathbf{head}(buff_1); buff_1{:=}\langle\,\rangle) \\
code.K_2 \bowtie K_3(put_1) & = (buff_2{=}\langle\rangle) \rightarrow buff_2{:=}\langle x\rangle \\
rIF.K_2 \bowtie K_3 & = \emptyset.
\end{array}$$

The transition graph of $K_2 \bowtie K_3$ is given in Fig.2, which is a two-place buffer. □

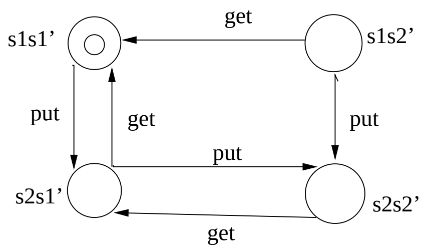

Fig. 2. Transition Graph of $K_2 \bowtie K_3$

Feedback Let K be a component, suppose its provided method m has the same signature as the required method n. We use the notion $K[m \hookrightarrow n]$ to represent the component which feeds back its provided service m to the required one n such that whenever n is invoked in K, m is invoked in $K[m \hookrightarrow n]$. This *feedback* can be defined by using the plugging operator. Let F be the component, which only provides n, and the code of n be $n()\{m()\}$, i.e. F only requires m. Then $K[m \hookrightarrow n] \;\widehat{=}\; K \bowtie F$.

6.2 Composition of Publications

In this subsection, we investigate these operators at the level of publications from the user's point of view.

Renaming. Since a publication contains two publication contracts, renaming a method in a publication definitely involves renaming a method in a publication contract. So, we first define renaming in (publication) contract.

Definition 16. *Given a contract C and n ∈ Meth.C, renaming n to m in C, denoted* $C[m/n]$*, is defined as*

- *$IF.C[m/n] = (IF.C)[m/n]$;*
- *$init.C[m/n] = init.C$;*
- *$spec.C[m/n](n) = spec.C(n)$, and $spec.C[m/n](op) = spec.C(op)$ for any other method of the interface;*
- *$prot.C[m/n] = (prot.C)[m/n]$,*

where m is a fresh method name, with the parameter type as n.

Definition 17. *Let $U = \langle \mathcal{G}, \mathcal{A}, C\rangle$. Renaming a method $n \in Meth.\mathcal{G} \cup Meth.\mathcal{A}$ gives the publication $U[m/n] = \langle \mathcal{G}[m/n], \mathcal{A}[m/n], C[m/n]\rangle$.*

Hiding. Hiding a provided method in a publication is semantically equivalent to removing this method from its provided interface. Formally,

Definition 18. *Let $U = (\mathcal{G}, \mathcal{A}, C)$ be a publication, and $m \in Meth.\mathcal{G}$. Hiding m in U, denoted $U\backslash m$, is defined by $U\backslash m = \langle \mathcal{G}\backslash m, \mathcal{A}, C\backslash m)\rangle$, where $\mathcal{G}\backslash m$ is defined by*

- *$IF.\mathcal{G}\backslash m = \langle field.\mathcal{G}, Meth.\mathcal{G} - \{m\}\rangle$,*
- *$init.\mathcal{G}\backslash m = init.\mathcal{G}$,*
- *$spec.\mathcal{G}\backslash m(n) = spec.\mathcal{G}(n)$ for each method n in $Meth.\mathcal{G} - \{m\}$, and*
- *$prot.\mathcal{G}\backslash m = (prot.\mathcal{G})\backslash m$.*

Note that hiding required methods in a component or publication does not make sense as required methods can be looked as bound variables (names) from a logical point of view.

Internalizing. Internalizing a set of methods in a publication is via internalizing these methods in its provided contract and hiding them in its invocation dependency oriented protocol. Internalizing methods in a publication contract is quite similar to internalizing methods in a component by changing all explicit invocations to these internalized methods to implicity invocations. Thus, from outside, these methods are invisible, but their impacts are still there.

Given a publication contract $C = (I, \theta, \mathcal{D}, \mathcal{T})$, let $M \subseteq Meth.I$ be internalized in C and $m \in Meth.I - M$. Then, all possible sequences of invocations to these internalized methods in M before and after each execution of m can be calculated according to $\mathcal{T}$ as follows:

$$maxT(\mathcal{T}, m, M) \mathrel{\widehat{=}} \{\ell\hat{\ }e\hat{\ }r \mid \ell \in M^* \wedge r \in M^* \wedge \exists tr_1, tr_2 \in Meth.I^*.tr_1\hat{\ }(\ell\hat{\ }e\hat{\ }r)\hat{\ }tr_2 \in \mathcal{T}\}$$

Definition 19. *Let $\mathcal{G}$ be a publication contract and $M \subseteq Meth.\mathcal{G}$. Internalizing M in $\mathcal{G}$, denoted $\mathcal{G}\swarrow\!\!/ M$, is the publication such that*

- *$IF.\mathcal{G}\swarrow\!\!/ M = (IF.\mathcal{G})\backslash M$,*
- *$init.\mathcal{G}\swarrow\!\!/ M = init.\mathcal{G}$,*
- *$spec.\mathcal{G}\swarrow\!\!/ M(n) = \sqcap_{s \in maxT(prot.\mathcal{G}, n, M)} spec.\mathcal{G}(s)$ for each method n in $Meth.\mathcal{G} - M$, and*
- *$prot.\mathcal{G}\swarrow\!\!/ M = prot.\mathcal{G} \downharpoonleft (Meth.\mathcal{G} - M)$*

Then, internalizing on a publication can be defined

Definition 20. *For a publication $U = (\mathcal{G}, \mathcal{A}, C)$, $U\swarrow\!\!/ M = (\mathcal{G}\swarrow\!\!/ M, \mathcal{A}, C \downharpoonleft (Meth.\mathcal{G} - M))$.*

Plugging. We now define the plugging operator on publications. A publication U_1 can plug into another publication U_2 only if on one hand, U_1 and U_2 have no naming conflicts; on the other hand, if a method m is respectively specified in U_1's provided contract and U_2's required contract, then the former must be a refinement of the latter, and vice versa. Formally,

Definition 21. *Let U_i, $i = 1, 2$, be publications. U_1 and U_2 are* pluggable *if*

1. $(\mathit{field.pCtr.U_1} \cap \mathit{field.pCtr.U_2}) = \emptyset$, *and* $(\mathit{Meth.pCtr.U_1} \cap \mathit{Meth.pCtr.U_2}) = \emptyset$;
2. $(\mathit{field.rCtr.U_1} \cap \mathit{field.rCtr.U_2}) = \emptyset$, *and* $(\mathit{Meth.rCtr.U_1} \cap \mathit{Meth.rCtr.U_2}) = \emptyset$;
3. $\mathit{spec.pCtr.U_i}(m) \sqsupseteq \mathit{spec.rCtr.U_j}(m)$, *for each* $m \in \mathit{pMeth.U_i} \cap \mathit{rMeth.U_j}$, *where* $i, j = 1, 2$ *and* $j \neq i$.

Definition 22. *Given two publications U_1 and U_2, which are pluggable. Then plug U_1 to U_2 is denoted by $U_1 \bowtie U_2$, and defined as $U_1 \bowtie U_2 = \langle \mathcal{G}, \mathcal{A}, C \rangle$, where*

– $\mathit{field}.\mathcal{G} = \mathit{filed.pCtr.U_1} + \mathit{field.pCtr.U_2}$,
– $\mathit{Meth}.\mathcal{G} = \mathit{pMeth.U_1} + \mathit{pMeth.U_2} - \mathit{rMeth.U_1} - \mathit{rMeth.U_2}$,
– $\mathit{Meth}.\mathcal{A} = \mathit{rMeth.U_1} + \mathit{rMeth.U_2} - \mathit{pMeth.U_1} - \mathit{pMeth.U_2}$,
– $\mathit{spec}.\mathcal{G}(m) = \mathit{spec.pCtr.U_1}(m) \oplus \mathit{spec.pCtr.U_2}(m)$, *for* $m \in \mathit{Meth}.\mathcal{G}$,
– $\mathit{spec}.\mathcal{A}(m) = \mathit{spec.rCtr.U_1}(m) \oplus \mathit{spec.rCtr.U_1}(m)$, *for* $m \in \mathit{Meth}.\mathcal{A}$,
– $\mathit{prot}.\mathcal{G} = \mathit{causal.U_1} \bowtie U_2 \downharpoonright (\mathit{pMeth.U_1} - \mathit{rMeth.U_2})$,
– $\mathit{prot}.\mathcal{A} = \mathit{causal.U_1} \bowtie U_2 \downharpoonright (\mathit{rMeth.U_1} - \mathit{pMeth.U_2})$,
– $\mathit{causal.U_1} \bowtie U_2 = \mathit{causal.U_1} \parallel_{(\mathit{rMeth.U_1} \cap \mathit{pMeth.U_2}) \cup (\mathit{pMeth.U_1} \cap \mathit{rMeth.U_2})} \mathit{causal.U_2}$.

From the above definition, you can see that once a required method of a publication is provided by another publication, then the method does not appear in the plugging of the two publications. This is consistent with plugging two components makes a method required by one and provided by the other private to the composite component.

Example 7. From the above definition, we know that the publications U_{B_2} and U_{B_3} in Example 5 are pluggable, and $U_{B_2} \bowtie U_{B_3}$ is:

$$\begin{aligned}
\mathit{pIF}.U_{B_2} \bowtie U_{B_3} &= \langle \{\mathit{buff}_1, \mathit{buff}_2 : \mathit{int}^*\}, \{\mathit{put}(\mathit{in}\, x : \mathit{int}), \mathit{get}(\mathit{out}\, y : \mathit{int})\} \rangle \\
\mathit{rIF}.U_{B_2} \bowtie U_{B_3} &= \emptyset, \\
\mathit{spec.pCtr}.U_{B_2} \bowtie U_{B_3}(\mathit{put}(\text{in } x : \mathit{int})) &= (\vdash \mathit{buff}_1' = \langle x \rangle ^\frown \mathit{buff}_1) \lhd \mathit{buff}_1 = \langle\, \rangle \rhd (\vdash \mathit{buff}_1' = \mathit{buff}_1) \\
\mathit{spec.pCtr}.U_{B_2} \bowtie U_{B_3}(\mathit{get}(\text{out } y : \mathit{int})) &= (\vdash \mathit{buff}_2' = \mathbf{tail}(\mathit{buff}_2) \wedge y' = \mathbf{head}(\mathit{buff}_2)) \lhd \mathit{buff}_2 \neq \langle\, \rangle \rhd \\
&\quad (\vdash \mathit{buff}_2' = \mathit{buff}_2 \wedge \exists c \in \mathit{int}. y' = c) \\
\mathit{causal}.U_{B_2} \bowtie U_{B_3} &= C_{B_2} \parallel_{\{\mathit{get}_1 ; \mathit{put}_1\}} C_{B_3} \\
&= [(\langle \mathit{put}; \mathit{get} \rangle)^* {}^\frown (\varepsilon + \langle \mathit{put} \rangle ^\frown (\langle \mathit{put}; \mathit{get} \rangle)^* + \\
&\quad \langle \mathit{put} \rangle ^\frown (\langle \mathit{put}; \mathit{get} \rangle)^* {}^\frown \langle \mathit{get} \rangle)]^*.
\end{aligned}$$

We can see $U_{B_2} \bowtie U_{B_3}$ is exactly a publication of $K_2 \bowtie K_3$. □

Feedback. Feedback for publications can be defined similarly to the definition for components.

A publication of a component tells the user what component does and how to use it. Therefore, it must be certified that the component does indeed do what is said in its publication. The following theorem shows that if the subcomponents conform to their publications, a composition of them will conform to the composition of their publications.

Theorem 6. (**Certification of Publication**) *All the operators defined above are compositional. That is,*

1. *if U is a publication of K, then $U[m/n]$ is a publication of $K[m/n]$;*
2. *if U is a publication of K, then $U\backslash m$ is a publication of $K\backslash m$;*
3. *if U is a publication of K, $U \swarrow M$ is a publication of $K \swarrow M$;*
4. *if U_1 is a publication of K_1 and U_2 is a publication of K_2, then $U_1 \bowtie U_2$ is a publication of $K_1 \bowtie K_2$;*
5. *if U is a publication of K, then $U[m \hookrightarrow n]$ is a publication of $K[m \hookrightarrow n]$.*

7 Conclusion and future work

This paper presents our further investigation on component publications and compositions of rCOS. We proposed a general approach on how to calculate the weakest required contract of a component according for a given provided contract. Then, we defined a set of composition operators on components and on their publications, and we studied the relation between compositions of components and their publications.

We hope the definitions and theorems in this paper set up the foundation for our ongoing research on the following problems

- **Decomposition.** The semantic equation (1) in Section 5 is in general unsolvable (undecidable). We have shown that the equations is solvable under some special cases. We will study further conditions under which it is solvable. This is significant to the general problem of program decomposition.
- **Refinement Theories.** The refinement relation between contracts and components in rCOS [5, 1] is essentially the *failure/divergence* partial order of CSP [11]. The disadvantages of such a refinement relation include: 1) it can only be used to compare two components with the same interface; 2) it mainly concerns safety property, but in component-based methods, we have to consider the reactivity to invocations of services from the environment, which is liveness property. To illustrate the second disadvantage, consider the example: let m_1 and m_2 be two simple stateless methods, without divergence and deadlock. Let $C_1 = \{false\&m_1, false\&m_2\}$, $C_2 = \{true\&m_1, false\&m_2\}$, $C_3 = \{true\&m_1, true\&m_2\}$ be complete contracts. Then, $prot.C_1 = \emptyset$, $prot.C_2 = \{m_1\}^*$, and $prot.C_3 = \{m_1, m_2\}^*$. It is easy to get $\mathcal{D}(C_1) = \mathcal{D}(C_2) = \mathcal{D}(C_3) = \emptyset$, and $\mathcal{F}(C_1) = \{(\langle\rangle, \{m_1, m_2\})\} \wedge \mathcal{F}(C_2) = \{(\langle m_1^n\rangle, \{m_2\}) \mid n \geq 0\} \wedge \mathcal{F}(C_3) = \emptyset$. According to the definition, obviously, $C_1 \sqsubseteq C_3$ and $C_2 \sqsubseteq C_3$, but C_1 cannot be compared with C_2. But from a user's point of view, C_2 should be better than C_1.

  We often need to support incremental design by extending a component to provide more services. Therefore, we are currently studying a more general refinement theory with the concepts of *strongest provided contracts* and *weakest required contracts*.
- **Glue Theory.** We are interested in developing a coordination model for specification and verification of glue code in the framework of rCOS.

Acknowledgments

We thank Prof. Jifeng He for so much fruitful discussions on the topic with him and his useful comments on early version of this paper. We are also grateful to the anonymous referees for their criticisms and constructive comments that has led to substantial improvements of the presentation of this paper.

References

1. X. Chen, J. He, Z. Liu, and N. Zhan. A model of component-based programming. In *International Symposium on Fundamentals of Software Engineering (FSEN'07)*, volume 4767 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2007.
2. L. de Alfaro and T. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering (FSE'01)*. ACM Press, January 2001.
3. L. de Alfaro and T. Henzinger. Interface theories for component-based design. In *Proceedings of the First International Workshop on Embedded Software*, volume 2211 of *Lecture Notes in Computer Science*, pages 148–165. Springer, January 2001.
4. J. He, X. Li, and Z. Liu. Component-based software engineering. In *Second International Colloquium on Theoretical Aspects of Computing (ICTAC'05)*, volume 3722 of *Lecture Notes in Computer Science*, pages 70–95. Springer, 2005.
5. J. He, X. Li, and Z. Liu. rCOS: a refinement calculus of object systems. *Theor. Comput. Sci.*, 365(1-2):109–142, 2006.
6. J. He, X. Li, and Z. Liu. A theory of reactive components. *Electr. Notes Theor. Comput. Sci.*, 160:173–195, 2006.
7. C.A.R. Hoare and J.He. *Unifying Theories of Programming*. Prentice-Hall, 1998.
8. G.J. Holzmann. Conquering complxity. *Software technology*, pages 111–113, December, 2007.
9. G. T. Leavens, K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000*, pages 105–106. ACM Press, 2000.
10. D. Mandrioli and B. Meyer. *Design by Contract*. Prentice Hall, 1991.
11. A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.
12. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.