=8.5truein =11truein

Refinement of Models of Software Components *

[Extended Abstract] *

Zizhen Wang[1,2] and Hanpin Wang[1]
1 School of EECS, PKU &
Key Lab.HCST (PKU), Ministry of Education
Beijing, China
2 UNU/IIST, Macau

Naijun Zhan †
Lab. of Comp. Sci., Institute of Software, CAS
Beijing, China
znj@ios.ac.cn

ABSTRACT

Models of *software components* at different levels of abstraction, *component interfaces, contracts, implementations* and *publications* are important for component-based design. Refinement relations among models at the same level and between different levels are essential for model-driven development of components. Classical refinement theories mainly focus on verification and put little attention on design. Therefore, most of them are not suitable for component-based model-driven development (CB-MDD). To address this issue, in this paper, we propose two refinement relations for CB-MDD, that is a trace-based refinement and a state-based refinement. Both are discussed in the framework of rCOS, which is a formal model of component and object systems. These refinement relations provide different granularity of abstraction and can capture the intuition that a refined component provides "more" and "better" services to the environment. We also show how to extend these refinement relations to allow us to compare contracts, components and publications with different interfaces by exploiting the primitive operator *internalizing* over contracts, components and publications.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/specifications; D.2.4 [**Software/Program Verification**]: Formal methods

General Terms

Theory

Keywords

CB-MDD, data refinement, trace refinement, rCOS

1. INTRODUCTION

Component-based model-driven development (CB-MDD) [16] is regarded as an effective way to develop complex systems and has been successfully applied in industry. Its basic idea is to compose/decompose a complicated system from/into some simpler ones with well-defined interfaces used for communication across the components. Models of *software components* at different levels of abstraction, *component interfaces, contracts, implementations* and *publications* are important for component-based design. Refinement relations among models at the same level and between different levels are essential for CB-MDD.

There have been lots of work on refinement of programs, the so-called reactive systems in particular [1, 4, 3, 14]. To show that a program P_h (at a higher level of abstraction) is refined by another program P_l (at a lower level of abstraction), denoted by $P_h \sqsubseteq P_l$, it is in general to find a "refinement mapping" from the state space of P_h to that of P_l, or the other way around so that the execution of P_h "simulates" that of P_l. The correctness of a refinement is justified by showing that every behavior of the lower-level system is also a behavior of the higher-level system. This means that the classical refinement theories mainly focus on the verification of safety property. Some attempts e.g. [1, 4] to deal with liveness property under certain fairness conditions have been done.

In CB-MDD, components are the first class entities. A component consists of a set of services (methods). Each invocation to a method of a component is controlled by the component through the implementation of a guard. Comparing two components, we should emphasize their reactivity, i.e. their ability of reaction to invocations from the environment. Intuitively, a refined component should provide not only better services (in the sense of functionality), but also more services (in the sense more easily to be invoked). It seems that classical refinement theories are not suitable for such a purpose as most of them can be reduced to trace containment.

Example 1. *Consider the following two components*[1] *adapted from* [6]: Comp *has three methods* msg, ack *and* nack*, where* msg *is used to send messages,* ack *to indicate a successful transmission, and* nack *to indicate a failure. Whenever* msg *is called by a user, the component returns* ok *or* fail *by calling back* ok *or* fail *provided by the user. To perform* msg*, the component invokes* send *provided by other component for sending messages. The two possible return values are* ack *and* nack *designated by which method is called back. When the method* msg *is invoked, the component tries to send the message, and resends it if the first transmission fails.*

*This work is supported partly by NSFC-60721061, NSFC-60970031, NSFC-90718041, NSFC-60736017, NSFC-60873061, and HTTS funded by Macao Science and Technology Development Fund

*A full version of this paper is available as a technical report of UNU/IIST [18] at `www.iist.unu.edu`

†The corresponding author: South Fourth Street, No. 4, Zhong Guan Cun, Beijing, 100080, P.R. China

[1]The interface automata of the two components can be found in [6]

If both transmissions fail, the component reports failure by calling back with fail*; otherwise, it reports success by calling back with* ok*. The above procedure can repeat infinitely. So, the behavior of* Comp *can be specified as the following CSP process:*

$$\text{rec } X.msg?; send!; (ack?; ok! + nack?; send!; (ack?; ok! + nack?; fail!)); X$$

While the behavior of QickComp *is similar to* Comp*'s except that it provides an additional choice, that is a try-once-only* once *designed for messages that are useless when stale. Thus, the behavior of* QickComp *is*

$$\begin{aligned}&\text{rec } X.(msg?; send!; (ack?; ok! + nack?; send!; (ack?; ok! + nack?; fail!)) +\\ &\qquad once?; send!; (ack?; ok! + nack?; fail!)); X\end{aligned}$$

Clearly, we would like to have refinement so that QickComp *is a refinement of* Comp*, because* QickComp *implements all services provided by* Comp*, and is consistent with* Comp *in their implementation. But according to classical refinement theories, we should have the trace set of* QickComp *is contained in* Comp*'s, obviously, it is impossible in this example.* $\square$

Developing appropriate refinement relations on components is a challenge in CB-MDD, and some first attempts have been done. For example, de Alfaro and Henzinger proposed the notion of *alternating simulation* [6] to compare components at the level of interfaces represented by *interface automata*. An interface automaton only describes the execution order of the provided methods and the required methods of a component. A component refines another one if it has weaker input assumptions and stronger output guarantees. Such an idea exactly reflects the intuition that a refined component more easily reacts to invocations to its provided methods. However, in real CB-MDD, comparison between components should include not only the ability of reaction to invocations from the environment, but also the functions of their corresponding methods. He et al gave another try towards this issue by directly introducing the *failure/divergence partial order* of CSP as a refinement relation [9, 11, 10].

In this paper, we will re-investigate this problem and propose two refinement relations for components, i.e. a trace-based refinement and a state-based refinement in the framework of rCOS. rCOS is a formal model of object and component systems, based on Unifying Theories of Programming (UTP) [13]. In rCOS, a component can be represented by different models at different levels of abstraction, *interface*, *contract*, *component* and *publication*. An interface provides the type information for an interaction point of component. A contract of an interface specifies the semantics of the interface, which associates each declared method with a guarded design. A component implements a contract via specific programming language. The implementation could need to call other services provided by other components, which are declared in a required interface. A publication of a component can be seen as its formal manual about what services are provided and what services are required by the component and the interaction between the component and its environment. Detailed comprehensive understanding rCOS can be referred to [11, 10, 5, 19].

In summary, the contributions of this paper include:

- Firstly, a trace-based refinement is defined. We first propose *trace refinement* on contracts and components by combining the trace containment refinement of CSP [15] and data refinement of designs in UTP [13]. Intuitively, a contract C_1 is *trace refined* by C_2 if each execution sequence of C_1 must be one of C_2's and each method of C_1 is *data refined* by its counterpart in C_2. Since a component (publication) could be open, that is in which there may be invocations to other components, similar to the definition of alternating simulation, we then define *alternating trace refinement* on publications based on the trace refinement.
- Secondly, we propose a state-based refinement relation, which is finer than the trace-based refinement relation. By revisiting the notion of data refinement in classical data refinement theories for action systems [3] and guarded designs [11], we define *data refinement* on contracts and components. According to our definition, a refined method has a weaker guard in contrast to the condition that the guard of a method to be refined and that of its refinement should be equivalent in classical data refinement theories [2, 17, 11][2]. Similarly, based on the data refinement, we can define *alternating data refinement* on publications.
- Finally, we show how these refinement relations together with the internalization of provided methods allow us to compare contracts, components and publications with different interfaces by exploiting the primitive operator *internalizing* over contracts, components and publications defined in [19].

The rest of this paper is organized as: We in Section 2 review some basic notions; Section 3 presents trace refinement and data refinement on contracts. Section 4 considers the notions of trace refinement and data refinement on components. In Section 5, alternating trace refinement and alternating data refinement on publications are proposed. Section 6 is devoted to extending these refinement relations to compare contracts, components and publications with different interfaces. Section 7 concludes this paper.

Because of the limit of space, we will omit the detailed proofs which can be found in the full version of the paper [18].

2. BASIC NOTIONS

In this section, we review the basic notions of UTP which will be used later, including design, guarded design, data refinement and so on. Comprehensive understanding of UTP can be referred to [13].

2.1 Design

UTP takes an approach to model the execution of a program in terms of a relation between the *states* of the program. For a sequential program, each state variable in the alphabet of the program comes in a unprimed and a primed versions, 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 alphabet also includes a boolean variable ok to denote whether a program is started properly and its primed version ok' to represent whether the execution has terminated. Notice the observables ok and ok' only help defining the semantics of the program and do not appear in the program texts.

A program can be defined as a predicate over a given alphabet α, called a *design*, denoted by D, which characterizes the functionality of the program, and of the form

$$p(x) \vdash R(x, x') \overset{\text{def}}{=} (ok \wedge p(x)) \Rightarrow (ok' \wedge R(x, x'))$$

It means that if the program is activated in a stable state ($ok = true$), where the *precondition* $p(x)$ holds, the execution will terminate ($ok' = true$), in a state where the *postcondition* $R(x, x')$ holds.

In what follows, if $p(x)$ is *true*, then $p(x) \vdash R(x, x')$ will be abbreviated as $\vdash R(x, x')$.

[2]In Back's Refinement Calculus [3], a variant of the condition is presented.

The classical programming operators on designs can be defined in a standard way, e.g. $D_1 \triangleleft b \triangleright D_2$ stands for conditional choice, where b is a boolean condition on program variables, and means if b holds then D_1 else D_2; while $D_1; D_2$ stands for the sequential composition of D_1 and D_2, defined by

$$\begin{aligned} D_1; D_2 \quad &\stackrel{\text{def}}{=} \quad \exists v_0, \ldots, v_n . D_1[v_0, \ldots, v_n / x_1', \ldots, x_n'] \wedge \\ & \qquad D_2[v_0, \ldots, v_n / x_1, \ldots, x_n] \end{aligned}$$

where $\alpha(D_1) = \alpha(D_2) = \{x_1, \ldots, x_n\} \cup \{x_1', \ldots, x_n'\}$, and $\phi[x/y]$ stands for replacing any occurrence of y in ϕ by x.

2.2 Refinement of design

The refinement relation between designs with the same alphabet is defined as logical implication. That is, let $D_1 \stackrel{\text{def}}{=} p_1 \vdash R_1$ and $D_2 \stackrel{\text{def}}{=} p_2 \vdash R_2$ be two designs over the alphabet α, D_2 is a *refinement* of D_1, denoted by $D_1 \sqsubseteq D_2$ if $\forall x, x', \ldots, ok, ok'.(D_2 \Rightarrow D_1)$, where $\alpha = \{x, x', \ldots, ok, ok'\}$. From the definition, we can conclude

$$(D_1 \sqsubseteq D_2) \quad \equiv \quad \forall x, x', \ldots, ok, ok'.(((p_1 \wedge R_2) \Rightarrow R_1) \wedge (p_1 \Rightarrow p_2))$$

I.e., a refined design should have a weaker precondition and a stronger postcondition.

If two designs do not have the same alphabet, we can use *data refinement* [12, 8, 3], which uses a relation(mapping) to relate their state spaces, as well as their behavior. Data refinement can be classified into *forward* and *backward*. In this paper, we consider only forward data refinement, which describes how the state space of abstract level are related to the one of concrete level. Similar results can be established for backward data refinement.

Definition 1 (data refinement). *Let D_1 be a design over alphabet α_1, D_2 be a design over alphabet α_2. D_1 is* data refined *by D_2, denoted by $D_1 \sqsubseteq_d D_2$, if there is a relation $\rho(y, x) \subseteq \alpha_2 \times \alpha_1$, satisfying*

$$((\vdash \rho(y, x')); D_1) \sqsubseteq (D_2; (\vdash \rho(y, x'))) \tag{1}$$

D_1 and D_2 are called data equivalent*, denoted by $D_1 =_d D_2$, iff $D_1 \sqsubseteq_d D_2 \wedge D_2 \sqsubseteq_d D_1$.*

In the above definition, if the relation ρ is fixed, we will denote $D_1 \sqsubseteq_d D_2$ by $D_1 \sqsubseteq_\rho D_2$ for clarity.

It is proven in [13] the domain of designs with this order forms a complete lattice which is closed under the classical programming operators. These operators are *monotonic* on the lattice. The property ensures that the domain of designs is a proper semantic domain for sequential programming languages.

The linking between designs $p \vdash R$ and Dijkstra's weakest precondition transformer wp, is

$$wp(p \vdash R, r) \stackrel{\text{def}}{=} p \wedge \neg(R; \neg r)$$

2.3 Reactive design and guarded design

A reactive program interacts with its environment, usually engages in alternative periods of computation and periods of stability. In UTP, in order to model such reactive programs, a boolean observable *wait* and its primed version *wait'* are introduced into the alphabet. $wait' = true$ means the system enters the *blocking (deadlock)* state. The introduction of intermediate observations has implication for sequential composition: a reactive program should not start until its predecessor has properly terminated. This rule can be formulated as a healthiness condition, and a design satisfying this healthiness condition is called to be *reactive*. Formally,

Definition 2 (reactive design). *A design D over alphabet α is* reactive *if D is a fixed point of the mapping $\mathcal{H}$, where $\mathcal{H}(D) \stackrel{\text{def}}{=} (\vdash wait' \wedge v = v') \triangleleft wait \triangleright D$.*

A reactive design satisfying the above condition keeps idle for ever, i.e. enters a blocking state. Clearly, for any design D, $\mathcal{H}(D)$ is a reactive design.

By associating a guard with a design, an invocation to the service specified by the design then can be controlled by the guard. Only if the guard is *true*, the service is available. A design together with a guard forms a guarded design, i.e.

Definition 3 (guarded design). *Let g be a guard and D be a design over α, the notation $g\&D$ denotes the* guarded design *$D \triangleleft g \triangleright (\vdash wait' \wedge v = v')$.*

For convenience, we denote its guard by *guard.D*, its design by *func.D* for a given guarded design $D = g\&D'$. If g does not hold, the guarded design will enter blocking state and keep idle for ever, otherwise execute its design part.

Refinement between guarded designs is defined similar to Definition 1, details can be found in [3, 17, 8, 7].

3. TRACE REFINEMENT AND DATA REFINEMENT OF CONTRACTS

An (provided) *interface I* only provides the syntactic type information for an interaction point of a component. Formally, I is a pair $\langle FDec, MDec \rangle$, where $FDec$ declares a set of variables (fields), denoted by $FDec.I$, and $MDec$ declares a set of operation (method) signatures, denoted by $MDec.I$.

A contract of an interface specifies the semantics of the services declared in the interface. In particular, it specifies each method by a guarded design, where the design characterizes the functionality of the method and the guard controls the availability of the method to the environment. In addition, a contract provides a protocol to specify the permitted order of method calls and indicate the interaction behaviors with the environment.

Definition 4. *A* contract *is a tuple $C = (I, Init, Spec, Prot)$, where*

- *I, written as $IF.C$, is an interface, its fields declaration is denoted by $FDec.C$, its methods declaration is denoted by $MDec.C$;*
- *Init, denoted by $Init.C$, is a design that initializes the values of $FDec.I$, and is of the form $\vdash R(v') \wedge \neg wait'$, where R is the initial condition;*
- *Spec, written as $Spec.C$, maps each method $m(in, out)$ in $MDec.I$ to a guarded design $Spec(m) = g\&D$ with the alphabet $\alpha = FDec.I \cup FDec.I' \cup \{in, out\} \cup \{ok, ok', wait, wait'\}$;*
- *Prot, called the protocol, written as $Prot.C$, is the set of sequences of call events. Each sequence is of the form $\langle ?m_1(x_1), \ldots, ?m_k(x_k) \rangle$, standing for the interaction protocol between the contract and its environment, where $?m_i(x_i)$ represents an invocation of method m with an input value x_i.*

We call a sequence of call events as a *trace*. A *legal* trace is one whose execution from an initial state can not enter a blocking nor a diverging state. We can formally define it using weakest precondition transformer as

Definition 5. *A legal trace is a trace $\langle ?m_1(x_1), \ldots, ?m_k(x_k) \rangle$ satisfying*

$$\begin{aligned} wp(Init; g_1\&D_1[x_1/in_1]; \ldots; g_k\&D_k[x_k/in_k], \\ \neg wait \wedge \exists m \in MDec.C \bullet guard.Spec(m)) = true \end{aligned}$$

A contract is *consistent*, if it will never enter a blocking nor diverging state when its environment interacts with it complying with its protocol. From the definition of legal trace, we can easily draw a conclusion that a contract is consistent *iff* each trace in its protocol is legal.

Therefore, for a tuple $C = (I, Init, Spec)$, there could be more than one protocol together with it to form a consistent contract, among which the largest one is called the *weakest consistent protocol*, i.e. the set of all legal traces. It is obvious that a contract is consistent *iff* its protocol is a subset of its weakest consistent protocol. A contract whose protocol equals to the weakest consistent protocol is called *complete contract*. A complete contract is simply written as $C = (I, Init, Spec)$.

By the result given in [19], for any consistent contract, we can always find another complete contract whose protocol is same as the former's such that they are equivalent. So, hereafter all contracts are referred to complete contracts if not otherwise stated.

EXAMPLE 2. *Consider a one-place buffer of integers. The buffer provides two services put and get. The user can put an integer in via put and get an integer via get from the buffer. In the following, the field buff is an integer list, but the guards of two provided methods make this buffer has one-place capacity. The contract is specified as follows:*

$$\begin{array}{rcl} C_1 & = & (I \stackrel{\text{def}}{=} \langle\{buff{:}int^*\}, \{put(\mathbf{in}\, x{:}int), get(\mathbf{out}\, y{:}int)\}\rangle \\ & & Init \stackrel{\text{def}}{=} \vdash buff' = \langle\,\rangle \\ & & Spec(put(\mathbf{in}\, x{:}int)) \stackrel{\text{def}}{=} buff = \langle\,\rangle \& (\vdash buff' = \langle x\rangle^\frown buff) \\ & & Spec(get(\mathbf{out}\, y{:}int)) \stackrel{\text{def}}{=} buff \neq \langle\,\rangle \& (\vdash buff' = \mathbf{tail}(buff) \\ & & \qquad\qquad \wedge y' = \mathbf{head}(buff)) \\ & & Prot \stackrel{\text{def}}{=} \langle (put; get)^* + (put; (get; put)^*)\rangle) \end{array}$$

where **head** *and* **tail** *are standard operators of lists. Clearly, C_1 is also complete.* □

Failure/divergence refinement.

In previous work of rCOS [11, 10], the failure/divergence model of CSP [15] is used for describing dynamic behaviour of contracts, i.e. the semantics of contract C is described by its divergence set $\mathcal{D}(C)$ and failure set $\mathcal{F}(C)$. $\mathcal{D}(C)$ contains all *interaction* traces that lead to divergence, while $\mathcal{F}(C)$ is the set of all pairs (s, X) where s is an interaction trace and X is a set of methods such that after the execution of the trace s, all methods in X are refused (disabled).

In [11, 10], the CSP failure/divergence partial order [15] was introduced as a refinement relation between contracts as: Given two contracts C_1 and C_2 with $MDec.C_1 = MDec.C_2$, it is said C_1 is *failure/divergence refined* by C_2, denoted by $C_1 \sqsubseteq C_2$, if $\mathcal{F}(C_2) \subseteq \mathcal{F}(C_1)$ and $\mathcal{D}(C_2) \subseteq \mathcal{D}(C_1)$.

Failure/divergence refinement facilitates checking deadlock and livelock by using CSP tools such as FDR. However, the disadvantages of the refinement is also obvious. First, it is impossible to apply the refinement to compare two contracts with different interfaces; Second, what's more, C_1 can be compared with C_2 only if their guards are equivalent, and this restriction makes this method not able to reflect the intuition that a refined contract could more easily react to the environment and provide more services. This is shown by the following example.

EXAMPLE 3. *Let m_1 and m_2 be two simple stateless methods, without divergence. 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 three complete contracts, therefore with the protocols $Prot.C_1 = \emptyset$, $Prot.C_2 = \{m_1\}^*$, and $Prot.C_3 = \{m_1, m_2\}^*$, respectively. Their* divergence sets *and* failure sets *are respectively $\mathcal{D}(C_1) = \mathcal{D}(C_2) = \mathcal{D}(C_3) = \emptyset$, and $\mathcal{F}(C_1) = \{(\langle\rangle, \{m_1, m_2\})\}$, $\mathcal{F}(C_2) = \{(s, \{m_2\}) \mid s \in \{m_1\}^*\}$, and $\mathcal{F}(C_3) = \{(s, \emptyset) \mid s \in \{m_1, m_2\}^*\}$*[3]. *Obviously, we would like to have $C_1 \sqsubseteq C_2$, as well as $C_1 \sqsubseteq C_3$ and $C_2 \sqsubseteq C_3$, because C_3 accepts more method invocations than C_1 and C_2, and C_2 accepts more method invocations than C_1. But none of them holds as their failure sets are not comparable.* □

[3]Note that here we just list the maximal failure of each contract. In fact, the failure set should be closed w.r.t. the inclusion of refusal sets, i.e. if (s, X) is a failure, so is (s, Y) for any $Y \subseteq X$.

3.1 Trace refinement

The following two principles towards refinement of components in CB-MDD should be followed, that is:

I. A refined contract should more easily react to the environment;

II. A refined contract should also provide "better" services in the sense of functionality.

The principle **I** has been implemented in the notion of alternating simulation [6], according to which a component refines another one if it has weaker input assumptions and stronger output guarantees. In fact, contracts can be seen as a special kind of interface automata without output guarantees, i.e. without invocations to methods provided by other components. However, in interface automata, the functionality of methods is abstracted away, and therefore the principle **II** was not taken into account in alternating simulation.

He et al gave another try towards this problem by directly introducing the *failure/divergence partial order* of CSP as a refinement relation [9, 11, 10]. As we explained above, such an approach is not a good solution to the problem too, as it does not meet the principle **I**.

In the following, by combining trace refinement of CSP [15] and data refinement of designs in UTP [13], we define a new refinement relation between contracts, still called *trace refinement*. According to our definition, a refined contract is more easily invoked by and also provides better services (in the sense of functionality) to its environment. Formally,

DEFINITION 6. *Given two contracts C_1 and C_2 with same interface, C_1 is* trace refined *by C_2, denoted by $C_1 \sqsubseteq_{tr} C_2$, if*
i. C_1 and C_2 both are consistent;
ii. $Init.C_1 \sqsubseteq_d Init.C_2$;
iii. $func.(Spec.C_1(m)) \sqsubseteq_d func.(Spec.C_2(m))$, for each $m \in MDec.C_1$;
iv. $Prot.C_1 \subseteq Prot.C_2$.

The distinctness between trace refinement defined above and trace refinement in CSP lies in: firstly, the former focuses on legal traces, whereas the latter focuses on general traces; secondly, trace containment in the former is forward direction in contrast to backward trace containment in the latter. Compared with alternating simulation, trace refinement reflects not only a refined component provides more services, but also a refined component provides better services.

The following theorem indicates that trace refinement defined above and failure/divergence refinement are not comparable, i.e.

THEOREM 1. • *There exist contracts C_1 and C_2 such that $C_1 \sqsubseteq_{tr} C_2$, but $C_1 \not\sqsubseteq C_2$;*
• *There exist contracts C_1 and C_2 such that $C_1 \sqsubseteq C_2$, but $C_1 \not\sqsubseteq_{tr} C_2$.*

3.2 Data refinement

Trace refinement is thought as the coarsest refinement relation in process algebra. It is desirable to look for a finer one. In particular, according to Definition 6, combination of trace refinement in CSP and data refinement in UTP makes it very complicated to check whether two contracts are in a trace refinement relation. Fortunately, we can define a finer refinement relation between contracts, called *data refinement* by revisiting data refinement of guarded designs.

3.2.1 Data refinement of guarded designs revisited

In classical data refinement theories, it is required to guarantee the consistency between *data refinement* and *trace refinement* or *failure/divergence refinement*. For example, the consistency between data refinement and trace refinement was investigated in [2, 1]; while the consistency between data refinement and failure/ divergence refinement was studied in [17, 7]. So, if system A_1 is refined by system A_2, we require:

• On one hand, every trace of A_2 is contained in the trace set of A_1. This implies that the guard of every refined action should be stronger;

• On the other hand, every refusal of A_2 is also a refusal of A_1. This implies that the guard of every refined action should be weaker.

So in the definition of *data refinement* between actions in [17] and between guarded designs in [11], it is required that the guard of an action (a guarded design) to be refined and that of its refinement are equivalent. A variant of the condition is presented in the definition of *data refinement* in Back's Refinement Calculus [2].

For our purpose, we revise the notion of data refinement between guarded designs by allowing the guard of the refined guarded design is weaker than that of the refining guarded design, so that the refined guarded design is much easier to be invoked.

Definition 7 (data refinement of guarded designs). *Let $g_1\&D_1$ be a guarded design over α_1, $g_2\&D_2$ be a guarded design over α_2. $g_1\&D_1$ is* data refined *by $g_2\&D_2$, denoted by $g_1\&D_1 \sqsubseteq_d g_2\&D_2$, if there is a relation $\rho(y,x) \subseteq \alpha_2 \times \alpha_1$ such that $\rho(y,x) \Rightarrow (g_1 \Rightarrow g_2)$ and $((\vdash \rho(y,x')); D_1) \sqsubseteq (D_2; (\vdash \rho(y,x')))$.*

Also, if the relation ρ is fixed, we will denote $g_1\&D_1 \sqsubseteq_d g_2\&D_2$ by $g_1\&D_1 \sqsubseteq_\rho g_2\&D_2$ for clarity.

3.2.2 Data refinement of contracts

By exploiting the above revised data refinement over guarded designs, we can define a data refinement relation over contracts as follows:

Definition 8 (Data Refinement). *Given two contracts C_1 and C_2 with same provided methods, C_1 is* data refined *by C_2, denoted $C_1 \sqsubseteq_d C_2$, if there is a relation ρ on $FDec.C_2 \times FDec.C_1$ such that*

• *$Init.C_1 \sqsubseteq_\rho Init.C_2$; and*

• *$\forall m \in MDec.C_1 \bullet Spec.C_1(m) \sqsubseteq_\rho Spec.C_2(m)$.*

The second condition requires each method in C_1 is data refined by the corresponding one in C_2. According to Definition 7, this means each C_2's method has a weaker guard and a stronger functionality. In fact, the possibility of reaction to its environment is directly related to the protocol set of the contract. Obviously, a contract with a larger protocol set will provide more services and more easily react to the environment.

Example 4. *Consider the following buffer*

$$
\begin{array}{rcl}
C_2 & = & (I \stackrel{\text{def}}{=} \langle\{buff{:}int^*\}, \{put(\mathbf{in}\, x{:}int), \\
 & & get(\mathbf{out}\, y{:}int)\}\rangle \\
Init & \stackrel{\text{def}}{=} & \vdash buff' = \langle\,\rangle \\
Spec(put(\mathbf{in} x{:}int)) & \stackrel{\text{def}}{=} & |buff| \le 1\&(\vdash buff'{=}\langle x\rangle\hat{\ }buff) \\
Spec(get(\mathbf{out} y{:}int)) & \stackrel{\text{def}}{=} & |buff| \ge 1\&(\vdash buff' = \mathbf{tail}(buff) \\
 & & \wedge y' = \mathbf{head}(buff)) \\
Prot & \stackrel{\text{def}}{=} & \langle((put;get)^*;(\varepsilon + put;(put;get)^* + \\
 & & put;(put;get)^*;get))^*\rangle.
\end{array}
$$

where $|buff|$ stands the size of buff, ε for empty sequence.

C_2 shares the same interface of C_1 defined in Example 2 and each provided method of C_2 has the same functionality as that of its counterpart in C_1, but with a weaker guard. Thus, it follows $C_1 \sqsubseteq_d C_2$ according to Definition 8. In fact, we can see C_2 provides two-places capacity. In addition, $Prot.C_1 \subseteq Prot.C_2$, so $C_1 \sqsubseteq_{tr} C_2$. □

The following theorem indicates that data refinement is finer than trace refinement.

Theorem 2. *For any two complete contracts C_1 and C_2 with the same interface, $C_1 \sqsubseteq_d C_2$ implies $C_1 \sqsubseteq_{tr} C_2$.*

4. TRACE REFINEMENT AND DATA REFINEMENT OF COMPONENTS

A component is an implementation of a contract. Besides the methods declared in the interface, a component maybe contains some methods which are not available to the public, but used by the component itself. So a component needs to declare a set of *private methods* and give their implementations. Furthermore, the implementations of the provided and private methods may call methods provided by other components. Therefore, a component should have a *required interface* to declare the services which are provided by other components.

Definition 9. *A component is a tuple $K = (I, PriMDec, Init, Code, InMDec)$, where*

• *I, denoted pIF.K, is a provided interface. Its method declaration is denoted by pMDec.K;*

• *PriMDec, denoted PriMDec.K, is a set of method signatures which are private to the component;*

• *Init, denoted Init.K, is the initialization statement that initializes the variables of the component;*

• *Code, denoted Code.K, maps each method in $MDec.I \cup PriMDec$ to a piece of program of a underlying programming language, possibly containing invocations to methods of other components;*

• *InMDec, denoted rIF.K, declares a set of methods which are implemented in other components, but invoked in Code, called the* required interface. *Its method declaration is denoted by rMDec.K.*

The code of each method can be defined as a *guarded reactive design*. Given a component K, we denote by $[\![K]\!]$ the abstraction of K by abstracting each of its provided and private methods to a guarded reactive design. For a given contract C_r of the required interface *rIF.K*, a contract C_p of the provided interface *pIF.K* can be calculated from $[\![K]\!]$ and C_r. This determines a function $\lambda C_r \cdot [\![K]\!]$ such that for a complete contract C_r of *rIF.K*, $[\![K]\!](C_r)$ is a complete contract of *pIF.K*. Obviously, $[\![K]\!](C_r)$ is the strongest contract of the provided interface w.r.t. C_r in the sense that w.r.t. C_r, K can provide any services refined by $[\![K]\!](C_r)$. We take the function $\lambda C_r \cdot [\![K]\!]$ as the semantics of component K [11, 5]. The semantic function is monotonic w.r.t. any of the three refinement relations. For instance, given two contracts C_1 and C_2 of *rIF.K*, if $C_1 \sqsubseteq C_2$ then $[\![K]\!](C_1) \sqsubseteq [\![K]\!](C_2)$.

In [11], a refinement relation between components, still called *failure/divergence refinement*, is defined in terms of the failure/ divergence refinement between contracts.

Accordingly, in terms of *trace refinement* and *data refinement* between contracts, respectively, we can define *trace refinement* and *data refinement* between components as follows:

Definition 10. *Given two components K_1 and K_2 with $pMDec.K_1 = pMDec.K_2$ and $rMDec.K_1 \supseteq rMDec.K_2$, K_1 is said to be trace refined by K_2, denoted by $K_1 \sqsubseteq_{tr} K_2$, if $[\![K_1]\!](C_r) \sqsubseteq_{tr} [\![K_2]\!](C_r)$ for any required contract C_r of $rIF.K_1$.*

Definition 11. *Given two components* K_1 *and* K_2 *with* $pMDec.K_1 = pMDec.K_2$ *and* $rMDec.K_1 \supseteq rMDec.K_2$, K_1 *is said to be data refined by* K_2*, denoted by* $K_1 \sqsubseteq_d K_2$*, if* $[\![K_1]\!](C_r) \sqsubseteq_d [\![K_2]\!](C_r)$ *for any required contract* C_r *of* $rIF.K_1$.

Example 5. *Consider the following two components* K_1 *and* K_2 *that respectively implement the contract* C_1 *in Example 2 and* C_2 *in Example 4.*

$$
\begin{array}{rcl}
pIF.K_1 & = & IF.C_1, \\
PriMeth.K_1 & = & \emptyset, \\
Init.K_1 & = & buff := \langle\,\rangle, \\
Code.K_1(put) & = & \mathrm{length}(buff) = 0 \rightarrow (buff{:=}\langle x\rangle^\frown buff), \\
Code.K_1(get) & = & \mathrm{length}(buff) \neq 0 \rightarrow (y{:=}\mathrm{head}(buff); \\
& & buff{:=}\mathrm{tail}(buff)) \\
rIF.K_1 & = & \langle\{\mathrm{length}(\mathbf{in}\, x : int^*, \mathbf{out}\, y : int), \mathrm{head}(\mathbf{in}\, x : int^*, \\
& & \mathbf{out}\, y : int), \mathrm{tail}(\mathbf{in}\, x : int^*, \mathbf{out}\, y : int^*)\}\rangle \\
pIF.K_2 & = & IF.C_2, \\
PriMeth.K_2 & = & \emptyset, \\
Init.K_2 & = & buff := \langle\,\rangle, \\
Code.K_2(put) & = & \mathrm{length}(buff) < 2 \rightarrow (buff{:=}\langle x\rangle^\frown buff), \\
Code.K_2(get) & = & \mathrm{length}(buff) \neq 0 \rightarrow (y{:=}\mathrm{head}(buff); \\
& & buff{:=}\mathrm{tail}(buff)) \\
rIF.K_2 & = & \langle\{\mathrm{length}(\mathbf{in}\, x : int^*, \mathbf{out}\, y : int), \mathrm{head}(\mathbf{in}\, x : int^*, \\
& & \mathbf{out}\, y : int), \mathrm{tail}(in\, x : int^*, out\, y : int^*)\}\rangle
\end{array}
$$

It is easy to check that $K_1 \sqsubseteq_{tr} K_2$ *and* $K_1 \sqsubseteq_d K_2$. □

According the above definitions, by Theorem 2, we can easily show that data refinement on components implies trace refinement on components.

Theorem 3. *For arbitrary two components* K_1 *and* K_2, $K_1 \sqsubseteq_d K_2$ *implies* $K_1 \sqsubseteq_{tr} K_2$.

5. ALTERNATING TRACE (DATA) REFINEMENT OF PUBLICATIONS

At the product level of component, a component should be regarded as a *black box specification* and the source code is impossible to be provided to the user. In order to more flexibly and easily use a matured component, the notion of publication was proposed in [9, 19]. A publication of component declares a subset of the provided services and requires a superset of the required services of the component. Thus, the vender of a component can flexibly provide different services (different subsets of the provided methods) to different users according to their demands and payments. Moreover, each of the methods provided and required by the component is documented as a design rather than a guarded design in order to ease the use and the compatibility checking. In addition, a publication also provides a protocol that represents the invocation dependency between provided methods and required methods in the code of the component so that the assembler can compose a new composite component (in fact, a publication of the new component) according to the publications of the existing components. Thus, a publication can be seen as a pair of publication contracts defined below together with an invocation protocol.

We first introduce the notion of publication contract, which can be seen as a special contract in which all guards of the declared methods are *true*.

Definition 12. *A* Publication contract *is a tuple* $(I, Init, Func, Prot)$*, where*

- *I is an interface;*
- *$Init$ is an initialization design;*
- *$Func$ is a function mapping each method m in $MDec.I$ to a design (no guard or with a guard true);*
- *$Prot$ is the protocol, a set of traces over $MDec.I$, which tells the environment how to use the methods declared in $MDec.I$.*

In [19], we defined two functions to link the domain of publication contracts and that of complete contracts: one function $\mathcal{M}$ mapping each complete contract to a publication contract by removing the guard in each method specification and taking the set of calculated legal traces as its protocol; the other function $\mathcal{L}$ mapping each publication contract to a complete contract where the guards are calculated from the protocol of the given publication contract, meanwhile the static functionality and protocol keep unchanged. In [19] it was proved that $\mathcal{L}$ and $\mathcal{M}$ form a Galois connection, which means interaction between a component and its environment can be controlled either by the guards of each of its provided methods or by a protocol.

Using the mapping $\mathcal{L}$, all refinement relations between contracts can be easily extended to between publication contracts. For instance, we say a publication contract C_1 is data refined by C_2 if $\mathcal{L}(C_1) \sqsubseteq_d \mathcal{L}(C_2)$; Similarly, *failure/divergence refinement* and *trace refinement* between publication contracts can be defined.

Then, publication is formally defined as:

Definition 13. *A* publication *of component* K *is* $U = (\mathcal{G}, \mathcal{A}, C)$ *where*

- *$\mathcal{G}$ is a publication contract of I_1 such that $MDec.I_1 \subseteq pMDec.K$, denoting the provided publication contract of a provided interface.*
- *$\mathcal{A}$ is a publication contract of interface I_2 such that $MDec.I_2 \supseteq rMDec.K$, denoting the required publication contract of a required interface.*
- *C is a invocation protocol over $(MDec.I_1 + MDec.I_2)$, such that $C \downharpoonright MDec.I_1 = Prot.\mathcal{G} \wedge C \downharpoonright MDec.I_2 = Prot.\mathcal{A}$.*

In the above definition, we can see that the user of the component via holding the publication can only use the services declared in I_1, i.e. part of the services provided by the component; while the user has to provide no less than the services required by the component in order to use the provided services.

A publication can be seen as an interface automaton [6] naturally, if we abstract away the functionality of methods. $\mathcal{A}$ assumes the order in which the component calls the required methods, while $\mathcal{G}$ guarantees the order in which the provided methods are called. Note that we here interchange the meaning of *assume/guarantee* given in [6].

In [9], the comparison between publications is given by *failure/divergence* too, defined as: let $U_1 = (\mathcal{G}_1, \mathcal{A}_1, C_1)$ and $U_2 = (\mathcal{G}_2, \mathcal{A}_2, C_2)$ be two publications, then U_2 is a *failure/divergnece refinement* of U_1, denoted by $U_1 \sqsubseteq U_2$, if $\mathcal{G}_1 \sqsubseteq \mathcal{G}_2$ and $\mathcal{A}_1 \sqsupseteq \mathcal{A}_2$.

Obviously, *alternating failure/divergence refinement* between publications has the inherent drawbacks of *failure/divergence refinement* between contracts because of the same reasons. Respectively based on *trace refinement* and *data refinement* between contracts, we can define *alternating trace refinement* and *alternating data refinement* between publications accordingly.

To the end, we need some notations first. Given two sequences of methods s_1 and s_2, we say s_1 approximates s_2 up to M, or s_2 is approximated by s_1 up to M, denoted by $s_1 \preceq_M s_2$, if s_1 can be obtained by removing some occurrences of some methods in M from s_2, where M is a set of methods. For example, $\langle a, a, b, c\rangle \preceq_{\{e,f\}} \langle a, e, a, f, b, e, e, c, e\rangle$.

Definition 14 (alternating trace refinement). *Let* $U_1 = (\mathcal{G}_1, \mathcal{A}_1, C_1)$ *and* $U_2 = (\mathcal{G}_2, \mathcal{A}_2, C_2)$ *be two publications.* U_1 *is* alternating trace refined *by* U_2*, denoted by* $U_1 \sqsubseteq_{tr} U_2$*, if*

i. $\mathcal{G}_1 \sqsubseteq_{tr} \mathcal{G}_2$;
ii. $\mathcal{A}_2 \sqsubseteq_{tr} \mathcal{A}_1$;
iii. $\forall s \in C_2 \bullet s \downharpoonright MDec.\mathcal{G}_2 \in Prot.\mathcal{G}_1 \Rightarrow \exists s' \in C_1 . s \preceq_{MDec.\mathcal{A}_1} s'$.

Conditions *i* and *ii* express that a refined publication provides more service to and requires less services from the environment. While

Condition *iii* says that for a sequence of provided services, if it is available in two publications U_1 and U_2 with the assumption that U_2 refines U_1, then U_1 more easily provides the service sequence to the environment. This is because U_1 is refined by U_2 and therefore requires more required services.

Similarly, *alternating data refinement* is defined as:

Definition 15 (alternating data refinement). *Let $U_1 = (\mathcal{G}_1, \mathcal{A}_1, C_1)$ and $U_2 = (\mathcal{G}_2, \mathcal{A}_2, C_2)$ be two publications. U_1 is alternating data refined by U_2, denoted by $U_1 \sqsubseteq_d U_2$, if*
i. $\mathcal{G}_1 \sqsubseteq_d \mathcal{G}_2$;
ii. $\mathcal{A}_1 \sqsupseteq_d \mathcal{A}_2$;
iii. $\forall s \in C_2 \bullet s \downharpoonright MDec.\mathcal{G}_2 \in Prot.\mathcal{G}_1 \Rightarrow \exists s' \in C_1.s \preceq_{MDec.\mathcal{A}_1} s'$.

According to the definitions above, using Theorem 2, we can easily prove that the alternating data refinement is finer than the alternating trace refinement , i.e.

Theorem 4. *For any two publications ($U_1 = (\mathcal{G}_1, \mathcal{A}_1, C_1)$ and $U_2 = (\mathcal{G}_2, \mathcal{A}_2, C_2)$, $U_1 \sqsubseteq_d U_2$ implies $U_1 \sqsubseteq_{tr} U_2$.*

6. DIFFERENT INTERFACES

In this section, we first briefly review the primitive operators over components and publications defined in [19]. Then we prove these operators except for *internalizing* preserve these refinement relations, and finally we show how to exploit the *internalizing* operator to extend the refinement relations defined in the previous sections to compare contracts, components and publications with different interfaces.

6.1 Primitive operators

In this subsection, we briefly review the set of primitive operators on components and publications, including *renaming, hiding, plugging, feedback* and *internalizing* defined in [19], their formal definitions can be found in [19].

Renaming.

Given a contract C, renaming a method $n \in IF.C$ to a fresh method m with the same type forms a new contract $C[m/n]$ by replacing each occurrence of n in C with m. Similarly, renaming a provided or required method n to a fresh method m with the same type in a component (publication) forms a new component (resp. publication) by replacing each occurrence of n with m in the component (resp. publication).

Hiding.

Hiding a set of methods M in a contract C is essentially equal to removing these methods in M from C, denoted by $C\backslash M$, where $M \subseteq IF.C$; While hiding a set of provided methods in a component K is implemented by changing these provided methods to private methods in K, denoted by $K\backslash M$; Hiding a set of provided methods M in a publication $U = (\mathcal{G}, \mathcal{A}, C)$ can be realized by hiding these methods in G and projecting C onto $(MDec.\mathcal{G} - M) \cup MDec.\mathcal{A}$, denoted by $U\backslash M$.

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, and vice versa, denoted by $K_1 \bowtie K_2$. A component can plug into another component only if they have no name conflicts. Accordingly, a publication U_1 can plug into another publication U_2, denoted by $U_1 \bowtie U_2$, only if on one hand, U_1 and U_2 have no name 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.

Feedback can be seen as a special case of plugging.

Internalizing.

Similar to hiding, *internalizing* a set of provided methods M in a component K is to remove them from the provided interface of K and add them into the private method set, denoted by $K \swarrow M$. However, unlike hiding, internalizing just changes all explicit invocations to the internalized methods to implicit invocations to the methods. This is semantically equivalent to reprogramming all provided methods in $pMDec.K - M$ by adding possible sequences of invocations to M before and after the execution of n, for each $n \in pMDec.K - M$.

Internalizing a set of methods in a publication is via internalizing these methods in its provided publication contract and hiding them in its invocation 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 $C = (I, Init, Func, Prot)$, let $M \subseteq MDec.C$ be internalized in C and $n \in MDec.C - M$. Then, all possible sequences of invocations to these internalized methods in M before and after each execution of n can be calculated according to $Prot$ as follows:

$$maxT(Prot, n, M) \stackrel{\text{def}}{=} \{\ell\hat{\ }n\hat{\ }r \mid \ell \in M^* \wedge r \in M^* \wedge \exists tr_1, tr_2 \in MDec.C^*.tr_1\hat{\ }(\ell\hat{\ }n\hat{\ }r)\hat{\ }tr_2 \in Prot.C\}$$

Definition 16. *Let $\mathcal{G}$ be a publication contract and $M \subseteq MDec.\mathcal{G}$. Internalizing M in $\mathcal{G}$, denoted $\mathcal{G} \swarrow M$, is the publication contract 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 $MDec.\mathcal{G} - M$; and $Prot.\mathcal{G} \swarrow M = Prot.\mathcal{G} \downharpoonright (MDec.\mathcal{G} - M)$.*

Thus, internalizing on publication can be defined as

Definition 17. *For a publication $U = (\mathcal{G}, \mathcal{A}, C)$, $U \swarrow M = (\mathcal{G} \swarrow M, \mathcal{A}, C \downharpoonright (MDec.\mathcal{G} - M + MDec.\mathcal{A}))$.*

6.2 Preserving these refinement relations

In what follows, for brevity, let $\sqsubseteq_r \in \{\sqsubseteq, \sqsubseteq_d, \sqsubseteq_{tr}\}$.

The following theorem indicates that *renaming*, *hiding*, *plugging* and *feedback* preserve the three refinement relations.

Theorem 5. • *For contracts C_1 and C_2, let $n \in IF.C_1$ and $M \subseteq IF.C_1$. If $C_1 \sqsubseteq_r C_2$, then $C_1[m/n] \sqsubseteq_r C_2[m/n]$ and $C_1\backslash M \sqsubseteq_r C_2\backslash M$;*
• *For components K_1 and K_2, let $n \in pMDec.K_1 \cup rMDec.K_1$ and $M \subseteq pMDec.K_1$. If $K_1 \sqsubseteq_r K_2$, then $K_1[m/n] \sqsubseteq_r K_2[m/n]$, $K_1\backslash M \sqsubseteq_r K_2\backslash M$, and $K_1 \bowtie K \sqsubseteq_r K_2 \bowtie K$ for any component K;*
• *For publications U_1 and U_2, let $n \in MDec.\mathcal{G}.U_1 \cup MDec.\mathcal{A}.U_1$ and $M \subseteq MDec\mathcal{A}.U_1$. If $U_1 \sqsubseteq_r U_2$, then $U_1[m/n] \sqsubseteq_r U_2[m/n]$, $U_1\backslash M \sqsubseteq_r U_2\backslash M$, and $U_1 \bowtie U \sqsubseteq_r U_2 \bowtie U$ for any publication U.*

In general, *internalizing* does not preserve the three refinement relations neither on components nor on publications.

6.3 Extending the refinement relations

The notions of *(alternating) trace refinement* and *(alternating) data refinement* proposed in this paper, as well as *failure/divergence refinement* defined in [9] all are subject to the condition that components, contracts and publications to be compared should be with the same interface. However, by exploiting the *internalizing operator*, these refinement relations could be extended by allowing comparison between contracts, components and publications with different interfaces. The idea is to internalize the underpinning contracts (resp. components and publications) with the uncommon parts, respectively, and then use the given refinement relation to compare the resulting contracts (resp. components and publications) with same interface. Formally,

Definition 18. *Given two contracts C_1 and C_2 possibly with different interfaces, we say C_1 is refined by C_2 up to a refinement relation $\sqsubseteq_r \in \{\sqsubseteq, \sqsubseteq_d, \sqsubseteq_{tr}\}$, denoted by $C_1 \sqsubseteq_r^u C_2$, if $C_1 \swarrow (MDec.C_1 - MDec.C_2) \sqsubseteq_r C_2 \swarrow (MDec.C_2 - MDec.C_1)$.*

Similarly, we can extend the three refinement relations to compare components and publications with different interfaces, respectively.

Example 6. *Consider the following two contracts:*

$$
\begin{aligned}
C_3 \;=\;& (I \stackrel{\text{def}}{=} \langle\{tr : put_1^*\}, \{put_1(\mathbf{in}\, x{:}int)\}\rangle, Init \stackrel{\text{def}}{=} tr = \langle\,\rangle,\\
& \quad Spec(put_1(x)) \stackrel{\text{def}}{=} tr^\frown\langle put_1\rangle \in put_1^* \& (\vdash tr' = tr^\frown\langle put_1\rangle),\\
& Prot \stackrel{\text{def}}{=} put_1^*)\\
C_4 \;=\;& (I \stackrel{\text{def}}{=} \{buff{:}int^*\}, \{put_1(\mathbf{in}\, x{:}int), get(\mathbf{out}\, y{:}int)\}\rangle,\\
& Init \stackrel{\text{def}}{=} \vdash buff' = \langle\,\rangle\\
& Spec(put_1(\text{in}\, x{:}int)) \stackrel{\text{def}}{=} buff = \langle\,\rangle \& (\vdash buff' = \langle x\rangle^\frown buff)\\
& Spec(get(\mathbf{out}\, y{:}int)) \stackrel{\text{def}}{=} buff \neq \langle\,\rangle \& (\vdash buff' = \mathbf{tail}(buff)\\
& \qquad\qquad\qquad\qquad \wedge y' = \mathbf{head}(buff))\\
& Prot \stackrel{\text{def}}{=} (\langle put_1\rangle^\frown\langle get\rangle)^{*\frown}(\langle\rangle + \langle put_1\rangle)).
\end{aligned}
$$

C_3 declares a buffer which can only provide the user the service put_1 to put an item of datum into the buffer. Note that the method can be invoked infinite many times; While C_4 declares a buffer with the capability of C_3 that provides the user two services put_1 and get, which means to put a datum into and get a datum from the buffer, respectively. The two services can only be invoked interchangeably starting with put_1. Obviously, $C_3 \not\sqsubseteq C_4$, $C_3 \not\sqsubseteq_d C_4$ and $C_3 \not\sqsubseteq_{tr} C_4$ as C_4 has an additional method get; but $C_3 \sqsubseteq C_4 \swarrow \{get\}$, $C_3 \sqsubseteq_d C_4 \swarrow \{get\}$ and $C_3 \sqsubseteq_{tr} C_4 \swarrow \{get\}$. That is, $C_3 \sqsubseteq^u C_4$, $C_3 \sqsubseteq_d^u C_4$ and $C_3 \sqsubseteq_{tr}^u C_4$.

7. CONCLUSION AND FUTURE WORK

Inspired by the work in [6], we proposed two refinement relations on components, i.e. a trace-based refinement and a state-based refinement. These refinement relations provide different granularity of abstraction and can capture the intuition that a refined component provides "more" and "better" services to the environment. We also proved the state-based refinement is finer than the trace-based one. In addition, we proposed an approach by exploiting the *internalizing operator* to extend the refinement relations to compare contracts, components and publications with different interfaces.

The ongoing and future work include:

- How to guarantee the proposed refinement relations preserve safety property as classical refinement theories do is very important.
- Connectors are another kind of first-class entities in CB-MDD. It does deserve to investigate refinement of connectors. As one of our future work, we will first focus on how to define connectors with the primitive operators defined in [19] and then consider refinement relations on connectors.
- Another interesting problem is to investigate under which conditions the *internalizing operator* preserves these established refinement relations.

Acknowledgments

We thank Prof. Jifeng He and Dr. Zhiming Liu for introducing us to this area, as well as so much fruitful discussions on the topic. We are grateful to Prof. Bill Roscue, Prof. Tony Hoare, Prof. Anders P. Ravn and Dr. Jeff Sanders for their suggestions and discussions on this work. We also thank the anonymous referees for their comments which improve the presentations of this paper and Dr. Jiaqi Zhu for his proof-reading of this paper.

8. REFERENCES

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[2] R. J. Back. Refinement calculus, part ii: parallel and reactive programs. In *REX workshop'90*, pages 67–93, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[3] R.J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Springer, 1998.

[4] J. Bengt. Simulations between specifications of distributed systems. In *CONCUR '91*, pages 346–360, London, UK, 1991. Springer-Verlag.

[5] X. Chen, J. He, Z. Liu, and N. Zhan. A model of component-based programming. In *FSEN'07*, volume 4767 of *LNCS*, pages 191–206. Springer, 2007.

[6] L. de Alfaro and T. A. Henzinger. Interface automata. In *ESEC / SIGSOFT FSE*, pages 109–120, 2001.

[7] J. He. Process simulation and refinement. *Form. Asp. Comput.*, 1(3):229–241, 1989.

[8] J. He, C. A. R Hoare., and J. Sanders. Data refinement refined. In *ESOP '86*, pages 187–196, London, UK, 1986. Springer-Verlag.

[9] J. He, X. Li, and Z. Liu. Component-based software engineering. In *ICTAC'05*, volume 3722 of *LNCS*, pages 70–95. Springer, 2005.

[10] J. He, X. Li, and Z. Liu. rCOS: A refinement calculus of object systems. *Theoretical Computer Science*, 365(1-2):109–142, November 2006.

[11] J. He, X. Liu, and Z. Liu. A theory of reactive components. *Electronic Notes in Theoretical Computer Science*, 160:173–195, 2006.

[12] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1:271–281, 1972.

[13] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall, 1998.

[14] N. A. Lynch and M.R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC '87*, pages 137–151, New York, NY, USA, 1987. ACM.

[15] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[16] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[17] J. Woodcock and C. Morgan. Refinement of state-based concurrent systems. In *VDM '90*, pages 340–351, London, UK, 1990. Springer-Verlag.

[18] H. Wang Z. Wang and N. Zhan. Towards a theory of refinement of component-based systems. Technical report, UNU-IIST Report No.427, 2009.

[19] N. Zhan, E. Y. Kang, and Z. Liu. Component publications and compositions. In *UTP08*, 2008.