Vol. 48 No. SIG 10(PRO 33) IPSJ Transactions on Programming June 2007

Regular Paper

An Equivalence Relation for the Typed Ambient Calculus

Toru Kato[†]

The ambient calculus is a process algebra designed for describing mobile processes. It has a layered structure of ambients that enables us to describe not only mobile processes but also the world in which the processes move around such as computer networks and freight systems. When we describe such a system with ambients, however, malicious processes can destroy nodes of the network or alter the construction of the system. Thus, several mobility types for the ambient calculus have been proposed to enable us to give each node desirable characteristics that prevent malicious processes from acting harmfully. Gordon and Cardelli, the originators of the ambient calculus, also defined an equivalence relation for the untyped ambient calculus. Our previous work pointed out that there exist identified processes up to the relation that have different properties, and it refined the relation so that we can discriminate those processes. This paper shows that the original relation and our previous relation are no longer available for the typed ambient calculus and it presents another relation that is suitable.

1. Introduction

The ambient calculus 3), devised by Cardelli and Gordon of Microsoft Research, was originally invented to model the behavior of network environments and various kinds of mobile entities with a unified framework. It has a layered structure of ambients that enables us to describe not only mobile processes but also the world in which the processes move around such as computer networks and freight systems. Since ambient calculus was developed for modeling such entities, Cardelli gave it the notion of security. The typed ambient calculus 2) is an extension of the ambient calculus that gives types to ambients for security purposes. When we describe a network with ambients, we would like to protect nodes from malicious processes that try to destroy nodes or alter the construction of the network. On the other hand, messenger ambients could be opened to extract the messages when they enter a node. The type system can flexibly control the opening property of ambients.

In the type system, names of ambients are classified in several groups according to their characteristics. For example, some ambients (more precisely, their names) in the Node group are given a type such that these ambients must not move around, and others in the Packet group have a type that can migrate among ambients of the Node group. The ambients in the Node group also have a type such that they must not be opened, while the type for those in the Packet group says that they can be opened in ambients of the Node group.

When we give an ambient a property in accordance with the type system of Ref. 2), however, we have to bind the ambient's name; that is, the ambient becomes invisible from the environment. For example, suppose that we have the process of the untyped ambient calculus $P \overset{def}{=} a[P']$ that provides a global service (meaning any processes of the environment e.g., $m[\mathit{in}\ a.Q]$ can access the ambient a). In order to protect the ambient a from an opening operation by the environment, we give it a type such as $P_{typed} \overset{def}{=} (\nu Node)(\nu a : Node{\curvearrowright}\{\}[{\curvearrowright}\{\}, {\circ}\{\}, T])a[P']$, where $P' : T$. The process P_{typed} does not provide global service any more.

As a solution to the problem, this paper proposes an extension of the syntax of the typed ambient calculus, called the *extended typed ambient calculus* (*ETAC*) by introducing the *type tag*. The type tag gives processes properties without them being bound so that the processes can keep providing global service.

This paper also presents an equivalence relation for ETAC. In our previous work 5), we presented an equivalence relation for the untyped calculus by extending the original one presented in Ref. 4). We explained the need for extension of the original equational relation using choice macro processes that were constructed by using only parallel composition and restriction primitives. An unexpected opening capability,

† Department of Informatics, Kinki University

however, can destroy the construction of choice macro processes. Using the typed ambient calculus, our choice macro always works as we intended. The type system is a desirable feature of the ambient calculus. However, it makes our previous equational relation and even the original one hard to apply. Thus, we need another relation for the typed ambient calculus.

This paper is organized as follows. In Section 2, we review the syntax and operational semantics of the typed ambient calculus. Section 3 explains the intuitive behavior of processes of the typed ambient calculus using our choice macro. In Section 4, we review the original equivalence relation (contextual equivalence) and our previous equivalence (contextually testing equivalence). Section 5 explains why those equivalences are not adequate for the typed framework and Section 6 introduces another equivalence: capability equivalence for the typed ambient calculus.

2. Typed Ambient Calculus

This section reviews the syntax and the semantics of the typed ambient calculus originally defined in Ref. 2). We assume there are infinite sets of *names* ranged over by *m, n, p, q*. *Messages* and *processes* are ranged over by *M,N* and by *P, Q, R*, respectively.

Definition 2.1 (Group sets, Types, Messages and Processes[2)]**)**

$\mathbf{G}, \mathbf{H}$ ::=	finite set of name groups $\{G_1, \ldots, G_k\}$	
W ::=	message type	
	$G^{\frown}\mathbf{G}[F]$	name in group G for ambients that cross $\mathbf{G}$ objectively and contain processes with effects F
	$Cap[F]$	capability unleashing effects F
F ::=	effect	
	$^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$	crosses $\mathbf{G}$, may open $\mathbf{H}$, may exchange T

S, T ::=	exchange type	
	Shh	no exchange
	$W_1 \times \ldots \times W_k$	tuple exchange
M, N ::=	message	
	n	name
	$in\ M$	can enter M
	$out\ M$	can exit M
	$open\ M$	can open M
	ϵ	null
	$M.N$	path

P, Q, R ::=		processes
	$(\nu G)P$	group restriction
	$(\nu n{:}W)P$	name restriction
	0	inactivity
	$P \vert Q$	composition
	$!P$	replication
	$M[P]$	ambient
	$M.P$	action
	$(x_1{:}W_1, \ldots, x_k{:}W_k).P$	input
	$\langle M_1, \ldots, M_k \rangle$	output
	$go(N).M[P]$	make an ambient move N where N must be $in\ M'$ $out\ M'$ or ϵ

□

We use the following abbreviations:

- M for $M.0$
- $M[]$ for $M[0]$
- $(\nu\vec{p})$ for $(\nu p_1{:}W_1), \ldots, (\nu p_k{:}W_k)P$ where $\vec{p} = p_1{:}W_1, \ldots, p_k{:}W_k$.

In Ref. 3), the message in Definition 2.1 is called the capability and, in particular, *in M*, *out M*, *open M* are the essential functionality of processes. Input and output actions (that are restricted in local communication in ambients) are also defined in the ambient calculus. They are the main capabilities of other process algebras such as CCS[8)] or the π–calculus[9)]. The purpose of those kinds of calculus is to express the communicating behavior of concurrent systems while the aim of the ambient calculus is to capture the moving behaviors of processes. Thus, we must concentrate on the faculty of changing the structure of processes.

As message type W and effect F in Definition 2.1 are hard to understand, the following example gives the intuitive explanation of them.

Example 2.2 As message type W appears in the two cases (case 1: name restriction and case 2: input) in Definition 2.1, we give examples for these cases.

case 1: Suppose that we have groups of names $G, G_1, G2$ and the following process: $(\nu n{:}W)go(N).n[P]$, where $W \overset{def}{=} G^{\frown}\mathbf{G}[F]$, $F \overset{def}{=} {}^{\frown}\mathbf{G}, {}^{\circ}\mathbf{H}, T$, $\mathbf{G} \overset{def}{=} \{G_1\}$ and $\mathbf{H} \overset{def}{=} \{G_2\}$. Here, $n : G$ says that name n is a member of G, the first $^{\frown}\mathbf{G}$ says that the ambient $n[P]$ can be objectively moved into and out of only the ambients whose names belong to G_1. Moreover, the second $^{\frown}\mathbf{G}$ in F says that the ambient $n[P]$ subjectively moves according to only the capabilities *in m* and *out m*, where m is a member of G_1.

$^\circ\mathbf{H}$ says that the capabilities *open* l can be executed in the n ambient, where l is a member of G_2. T says that any communication in the n ambient is impossible in the case where $T = \mathit{Shh}$ or it can contain input and output whose types consist of a tuple of some message types in the case where T is a tuple of those message types.

case 2: Suppose that we have process $(x{:}W).P$. If W is defined as in case 1, it can receive the name n, whose type is W. If $W \overset{def}{=} \mathit{Cap}[F]$ and F is defined as case 1, it can receive the capability *open* n, where $n : G^\frown\mathbf{G}[F]$ or the capability *in* m or *out* m for any m. □

An expression $go(M).M[P]$ is called an *objective move*, which used to be defined as a macro as follows:

$$go(N).M[P] \overset{def}{=} (\nu k)k[N.M[\mathit{out}\ k.\mathit{open}\ k.P]] \quad \mathit{where}\ \ k \notin fn(P).$$

Intuitively, the objective move $go(N).M[P]$ means that we make the inactive ambient $M[P]$ move according to N, while the *subjective move* $M[N]$ means that the active ambient moves according to N by itself.

Cardelli 2) made a slight extension in syntax by using objective moves as primitives, as we do in this paper. So our syntax is the extended version.

As we showed in Example 2.2, the name restriction prescribes what is impossible in the process. For example, in case 1 of Example 2.2, *open* m is impossible if $m \notin G_2$, and if P contains such a capability, the typing rules defined in Ref. 2) judge $(\nu n{:}W)n[P]$ as not being a well typed process. In such a case, we use the phrase $(\nu n{:}W)$ *interferes open* m *in* $n[P]$ in this paper.

Definition 2.3 (Free Names and Free Groups 2))

$$\begin{aligned}
&fn(n) \overset{def}{=} \{n\}\\
&fn(\mathit{in}\ M) \overset{def}{=} fn(M)\\
&fn(\mathit{out}\ M) \overset{def}{=} fn(M)\\
&fn(\mathit{open}\ M) \overset{def}{=} fn(M)\\
&fn(\epsilon) \overset{def}{=} \emptyset\\
&fn(M.N) \overset{def}{=} fn(M) \cup fn(N)\\
&fn((\nu G)P) \overset{def}{=} fn(P)\\
&fn((\nu n{:}W)P) \overset{def}{=} fn(P) - \{n\}\\
&fn(0) \overset{def}{=} \emptyset\\
&fn(P|Q) \overset{def}{=} fn(P) \cup fn(Q)\\
&fn(!P) \overset{def}{=} fn(P)\\
&fn(M[P]) \overset{def}{=} fn(M) \cup fn(P)\\
&fn(M.P) \overset{def}{=} fn(M) \cup fn(P)\\
&fn((x_1{:}W_1,\ldots,x_k{:}W_k).P) \overset{def}{=} fn(P) - \{x_1,\ldots,x_k\}\\
&fn(\langle M_1,\ldots,M_k\rangle) \overset{def}{=} fn(M_1)\cup,\ldots,\cup fn(M_k)\\
&fn(goN.M[P]) \overset{def}{=} fn(N) \cup fn(M) \cup fn(P)\\
&fg((\nu G)P) \overset{def}{=} fg(P) - fg(G)\\
&fg((\nu n{:}W)P) \overset{def}{=} fg(W) \cup fg(P)\\
&fg(0) \overset{def}{=} \emptyset\\
&fg(P|Q) \overset{def}{=} fg(P) \cup fg(Q)\\
&fg(!P) \overset{def}{=} fg(P)\\
&fg(M[P]) \overset{def}{=} fg(P)\\
&fg(M.P) \overset{def}{=} fg(P)\\
&fg((x_1{:}W_1,\ldots,x_k{:}W_k).P)\\
&\quad \overset{def}{=} fg(W_1) \cup \ldots \cup fg(W_k) \cup fg(P)\\
&fg(\langle M_1,\ldots,M_k\rangle) \overset{def}{=} \emptyset\\
&fg(goN.M[P]) \overset{def}{=} fg(P)\\
&fg(G[T]) \overset{def}{=} \{G\} \cup fg(T)\\
&fg(\mathit{Cap}[T]\ ]) \overset{def}{=} fg(T)\\
&fg(\mathit{Shh}) \overset{def}{=} \emptyset\\
&fg(W_1 \times \ldots \times W_k) \overset{def}{=} fg(W_1)\cup,\ldots,\cup fg(W_k) \quad \square
\end{aligned}$$

Definition 2.4 (Structural Congruence: $P \equiv Q$ 2))

$$\begin{aligned}
&P|Q \equiv Q|P\\
&(P|Q)|R \equiv P(Q|R)\\
&!P \equiv P|!P\\
&m \neq n \Rightarrow (\nu n{:}W_1)(\nu m{:}W_2)P\\
&\qquad\qquad \equiv (\nu m{:}W_2)(\nu n{:}W_1)P\\
&n \notin fn(P) \Rightarrow (\nu n{:}W)(P|Q) \equiv P|(\nu n{:}W)Q\\
&m \neq n \Rightarrow (\nu n{:}W)m[P] \equiv m[(\nu n{:}W)P]\\
&P|0 \equiv P\\
&(\nu n{:}W)0 \equiv 0\\
&(\nu G)0 \equiv 0\\
&!0 \equiv 0\\
&\epsilon.P \equiv P\\
&go(\epsilon).P \equiv P\\
&(M.M').P \equiv M.M'.P\\
&m \neq n \Rightarrow (\nu n{:}W)m[P] \equiv m[(\nu n{:}W)P]\\
&(\nu G_1)(\nu G_2)P \equiv (\nu G_2)(\nu G_1)P\\
&G \in fg(W) \Rightarrow (\nu G)(\nu n{:}W)P \equiv (\nu n{:}W)(\nu G)P\\
&G \in fg(P) \Rightarrow (\nu G)(P|Q) \equiv P|(\nu G)Q\\
&(\nu G)m[P] \equiv m[(\nu G)P]\\
&P \equiv P\\
&Q \equiv P \Rightarrow Q \equiv P\\
&P \equiv Q, Q \equiv R \Rightarrow P \equiv R\\
&P \equiv Q \Rightarrow (\nu n{:}W)P \equiv (\nu n{:}W)Q
\end{aligned}$$

$P \equiv Q \Rightarrow (\nu G)P \equiv (\nu G)Q$
$P \equiv Q \Rightarrow P|R \equiv Q|R$
$P \equiv Q \Rightarrow !P \equiv !Q$
$P \equiv Q \Rightarrow M[P] \equiv M[Q]$
$P \equiv Q \Rightarrow M.P \equiv M.Q$
$P \equiv Q \Rightarrow (x_1,\ldots,x_k).P \equiv (x_1,\ldots,x_k).Q$
□

The behavior of processes of the ambient calculus is defined by the following reduction rules.

Definition 2.5 (Reduction: $P \rightarrow Q$ 2))
$n[in\ m.P|Q]|m[R] \rightarrow m[n[P|Q]|R]$
$go(in\ m.N).P|m[Q] \rightarrow m[go(N).P|Q]$
$m[n[out\ m.P|Q]|R] \rightarrow n[P|Q]|m[R]$
$m[go(out\ m.N).n[P]|Q] \rightarrow go(N).n[P]|m[Q]$
$open\ n.P|n[Q] \rightarrow P|Q$
$\langle M_1,\ldots,M_k\rangle|(x_1{:}W_1,\ldots,x_k{:}W_k).P$
$\quad \rightarrow P\{x_1 \leftarrow M_1,\ldots,x_k \leftarrow M_k\}$

$P \rightarrow Q \Rightarrow P|R \rightarrow Q|R$
$P \rightarrow Q \Rightarrow (\nu n{:}W)P \rightarrow (\nu n{:}W)Q$
$P \rightarrow Q \Rightarrow (\nu G)P \rightarrow (\nu G)Q$
$P \rightarrow Q \Rightarrow n[P] \rightarrow n[Q]$
$P' \equiv P, P \rightarrow Q, Q \equiv Q' \Rightarrow P' \rightarrow Q'$
□

3. External Choice Operation

This section explains the behavior of processes of the typed ambient calculus having an external choice macro and explains the extension of the syntax.

3.1 Choice Macro for the Untyped Calculus

The ambient calculus and the typed ambient calculus do not have choice primitives because of the difficulty of implementation, especially in distributed environments, and also because they can be simulated by using parallel composition and restriction primitives. Cardelli 3) presents an example of the simulation of choice. That operation is an internal choice and it can be used only for local services. Thus, we defined an external choice by which we can define global services 5).

Definition 3.1 (External Choice for the Untyped Calculus "$+^n_u$") Let B and C be any processes of the ambient calculus. Then, we define $b[B] +^n_u c[C]$ as follows:

$b[B] +^n_u c[C] \overset{def}{=}$
$(\nu trash, sync)($
$\ b[in\ trash$
$\quad |\ go(in\ n.out\ n).sync[out\ trash.B|trash[out\ b]]$
$\quad |\ open\ sync]$
$\ |\ c[in\ trash$
$\quad |\ go(in\ n.out\ n).sync[out\ trash.C|trash[out\ c]]$
$\quad |\ open\ sync]).$
□

The symbol n of $+^n_u$ in Definition 3.1 is a parameter of the operator and u is the identifier of the choice operator (we will show two more choice operators $+^n_p$ and $+^n$ in this paper) We explained the behavior of this choice operation "$+^n_u$" in Ref. 5). When a choice process such as $a[b[]] +^n_u a[c[]]$ is used with a process such as $n[in\ a.\ in\ b]$ (we call it a *traveling ambient*), it works as an ideal choice primitive: after several reductions, only the ambient chosen by the traveling ambient remains visible and the other ambient goes into the ambient *trash*[] and becomes inaccessible.

3.2 Pseudo External Choice for Typed Calculus

When there exists a malicious process with a traveling ambient, for example *open a*.0, the process can destroy the structure of the choice macro and make it useless. This is not only the case for our examples but also for network structures constructed of ambients. The opening control of the type system enables us to design networks that are safer against illegal operations.

Definition 3.2 (Pseudo External Choice "$+^n_p$") Let B and C be any processes of the typed ambient calculus. Then, we define $b[B] +^n_p c[C]$ as follows:

$b[B] +^n_p c[C] \overset{def}{=}$
$(\nu\, Trsh, Sync)$
$(\nu b{:}Node^{\frown}\emptyset[{}^{\frown}Trsh, {}^{\circ}\emptyset, Shh\,])$
$(\nu c{:}Node^{\frown}\emptyset[{}^{\frown}Trsh, {}^{\circ}\emptyset, Shh\,])$
$(\nu trash{:}\,Trsh^{\frown}\{Node, Thread\}[{}^{\frown}Node, {}^{\circ}\emptyset, Shh\,])$
$(\nu sync{:}Sync^{\frown}\{\,Thread\}[{}^{\frown}Trsh, {}^{\circ}Sync, Shh\,])$
$\ b[in\ trash$
$\quad |\ go(in\ n.out\ n).sync[out\ trash.B|trash[out\ b]]$
$\quad |\ open\ sync]$
$\ |\ c[in\ trash$
$\quad |\ go(in\ n.out\ n).sync[out\ trash.C|trash[out\ c]]$
$\quad |\ open\ sync]).$
□

In choice process $b[B] +^n_p c[C]$ above, the bound name *trash* is a member of bound group *Trsh*, the ambients having the name *trash* objectively move into or move out of ambients belonging to groups *Node* and *Thread*, subjectively move into or move out of ambients belonging to the group *Node*, no process can open them, and no message exchange occurs in them.

The bound names b and c are members of the free group *Node*, the ambients having the names b and c objectively move into or move out of no ambient, subjectively move into or move out of ambients belonging to group *Trsh*, no process can open them, and no message exchange occurs in them.

4. Extended Typed Ambient Calculus

The choice process $b[B] +_p^n c[C]$ is, however, not what we would like to define because the names b and c are bound; that is, the ambients having those names cannot be accessed from traveling ambients. We have to bind the name when we intend to give an ambient the locking property or movement property though we only want to gain those properties.

The cause of the problem is the dual binding in this type system. The group binding and the name binding cause an inconsistency: names in a free group must be bound when we try to give them properties, while we have no problem when we give properties to names in a bound group. As a solution to this problem, we propose extending the syntax by introducing *type tags* with the idea that *the name in a free group is free and the name in a bound group is bound.*

According to this idea, we extend the definition of the processes of Definition 2.1 as follows:

Definition 4.1 (Extension of Processes)

$P, Q, R ::=$	processes
$(\nu G)P$	group restriction,
$(\nu n{:}W)P$	restriction,
	where G in W is bound,
$(\gamma n{:}W)P$	type tag, where if $W = G^\frown\mathbf{G}[F]$, then G is free,
$\cdots$	as in Definition 2.1. $\square$

We call the expression $(\gamma n{:}W)$ *type tag* and we call the calculus with type tag the *Extended Typed Ambient Calculus* (*ETAC*). When we write a process $(\gamma n{:}W)P$, the group G must be free, the name n appearing in P is free, and the ambient whose name is n has the property described as W. We extend Definition 2.3 by adding the following rules.

Definition 4.2 (Extension of Free Name)

$fn((\gamma n{:}W)P) \overset{def}{=} fn(P)$

$fg((\gamma n{:}W)P) \overset{def}{=} fg(W) \cup fg(P)$

$\cdots$ as in Definition 2.3. $\square$

The type tag does not create a scope of the name, but only gives properties to ambients. Suppose that we have the following process in which the inner n ambient of the following process gets out from the outer n ambient and gets into the ambient $m[]$:

$$(\gamma n{:}W)n[n[out\ n.in\ m]]|(\gamma m{:}W')m[\ ].$$

Unlike the name restriction, the scope of n is not expanded; instead, the type tag is duplicated as follows:

$$(\gamma n{:}W)n[\ ]|(\gamma m{:}W')m[(\gamma n{:}W)n[\ ]].$$

This leads to the first rule of Definition 4.3. There may be the following case in the way of a reduction:

$$(\gamma n{:}W')n[(\gamma n{:}W)n[out\ n.in\ m]].$$

In this case, ambients obey the nearest inner tag. This leads to the second rule of Definition 4.3. We extend the definition of structural congruence by adding the following rules.

Definition 4.3 (Extension of Structural Congruence)

$n \in fn(P) \Rightarrow (\gamma n{:}W)n[P] \equiv (\gamma n{:}W)n[(\gamma n{:}W)P]$

$(\gamma n{:}W_1)(\gamma n{:}W_2)P \equiv (\gamma n{:}W_2)P$

$m \neq n \Rightarrow (\gamma n{:}W_1)(\gamma m{:}W_2)P$
$\equiv (\gamma m{:}W_2)(\gamma n{:}W_1)P$

$n \notin fn(P) \Rightarrow (\gamma n{:}W)(P|Q) \equiv P|(\gamma n{:}W)Q$

$m \neq n \Rightarrow (\gamma n{:}W)m[P] \equiv m[(\gamma n{:}W)P]$

$P \equiv Q \Rightarrow (\gamma n{:}W)P \equiv (\gamma n{:}W)Q$

$(\gamma n{:}W)0 \equiv 0$

$\cdots$ as in Definition 2.4. $\square$

According to Definition 4.3, we have the following transitions:

(case *out n***)** $(\gamma n{:}W)n[n[out\ n.Q]|P] \equiv (\gamma n{:}W)n[(\gamma n{:}W)n[out\ n.Q]|P] \rightarrow (\gamma n{:}W)n[P]|(\gamma n{:}W)n[Q]$.

(case *go(out n)***)** $(\gamma n{:}W)n[go(out\ n.N).n[P]|Q] \equiv (\gamma n{:}W)n[(\gamma n{:}W)go(out\ n.N).n[P]|Q] \rightarrow (\gamma n{:}W)n[Q]|(\gamma n{:}W)go(N).n[P]$.

Thus, we only need to add the following rule to the definition of reduction.

Definition 4.4 (Extension of Reduction)

$P \rightarrow Q \wedge$

$(\gamma n{:}W)$ does not interfere with the transition in $P \wedge$ for any R and S.

$(P \not\equiv n[m[out\ n.R]|S],$
$P \not\equiv n[go(out\ n.N).m[R]|S]$
where $n \in \{m\} \cup fn(R) \cup fn(N))$

$\Rightarrow (\gamma n{:}W)P \rightarrow (\gamma n{:}W)Q$

$\cdots$ as in Definition 2.4. □

We call the above calculus the *extended typed ambient calculus* (*ETAC*). Using processes with the type tag, we define the choice macro of ETAC as follows.

Definition 4.5 (External Choice "$+^n$") Let B and C be any processes of ETAC. Then, we define $b[B] +^n c[C]$ as follows:

$b[B] +^n c[C] \overset{def}{=}$
$(\nu\, Trsh, Sync)$
$(\gamma b{:}Node^\frown\emptyset[{}^\frown Trsh, {}^\circ\emptyset, Shh\,])$
$(\gamma c{:}Node^\frown\emptyset[{}^\frown Trsh, {}^\circ\emptyset, Shh\,])$
$(\nu trash{:}\, Trsh^\frown\{Node, Thread\}[{}^\frown Node, {}^\circ\emptyset, Shh\,])$
$(\nu sync{:}Sync^\frown\{Thread\}[{}^\frown Trsh, {}^\circ Sync, Shh\,])$
$b[in\ trash$
$\mid go(in\ n.out\ n).sync[out\ trash.B|trash[out\ b]]$
$\mid open\ sync]$
$\mid c[in\ trash$
$\mid go(in\ n.out\ n).sync[out\ trash.C|trash[out\ c]]$
$\mid open\ sync]).$
□

Example 4.6 We explain the behavior of this choice operation "$+^n$" by the following transitions: when the traveling ambient $(\gamma n{:}Thread^\frown\emptyset[{}^\frown\{Node, {}^\circ\emptyset, Shh\,])n[in\ a|in\ b]$ is running parallel to $a[b[]] +^n a[c[]]$ and a malicious process "*open a*", that is,

$(\gamma n{:}Thread^\frown\emptyset[{}^\frown\{Node, {}^\circ\emptyset, Shh\,])$
$(n[in\ a|in\ b])|open\ a|a[b[]] +^n a[c[]],\qquad (*)$

the traveling ambient can enter either the ambient $a[\cdots]$ of $a[b[]]$ or $a[\cdots]$ of $a[c[]]$, the process "*open a*" cannot do anything because of the opening control property of ambient whose name is a. According to Definition 4.5, the expression $(*)$ has the following structure:

$(\gamma n{:}Thread^\frown\emptyset[{}^\frown\{Node, {}^\circ\emptyset, Shh\,])$
$(n[in\ a|in\ b])|open\ a$
$|(\nu\, Trsh, Sync)$
$(\gamma a{:}Node^\frown\emptyset[{}^\frown Trsh, {}^\circ\emptyset, Shh\,])$
$(\gamma b{:}Node^\frown\emptyset[{}^\frown Trsh, {}^\circ\emptyset, Shh\,])$
$(\gamma c{:}Node^\frown\emptyset[{}^\frown Trsh, {}^\circ\emptyset, Shh\,])$
$(\nu trash{:}\, Trsh^\frown\{Node, Thread\}[{}^\frown Node, {}^\circ\emptyset, Shh\,])$
$(\nu sync{:}Sync^\frown\{Thread\}[{}^\frown Trsh, {}^\circ Sync, Shh\,])$
$(a[in\ trash$
$\mid go(in\ n.out\ n).sync\,[out\ trash.b[]|\ trash[out\ a]]$
$|open\ sync]$
$\mid a[in\ trash$
$\mid go(in\ n.out\ n).sync\,[out\ trash.c[]|\ trash[out\ a]$
$|open\ sync]).$

Suppose that the traveling ambient happens to choose the first $a[\ldots]$ ambient. After several reductions, the expression $(*)$ is reduced to the following process:

$(\gamma n{:}Thread^\frown\emptyset[{}^\frown\{Node, {}^\circ\emptyset, Shh\,])|open\ a$
$|(\nu\, Trsh, Sync)$
$(\gamma a{:}Node^\frown\emptyset[{}^\frown Trsh, {}^\circ\emptyset, Shh\,])$
$(\gamma b{:}Node^\frown\emptyset[{}^\frown Trsh, {}^\circ\emptyset, Shh\,])$
$(\gamma c{:}Node^\frown\emptyset[{}^\frown Trsh, {}^\circ\emptyset, Shh\,])$
$(\nu trash{:}\, Trsh^\frown\{Node, Thread\}[{}^\frown Node, {}^\circ\emptyset, Shh\,])$
$(\nu sync{:}Sync^\frown\{Thread\}[{}^\frown Trsh, {}^\circ Sync, Shh\,])$
$(a[b[n[]]]$
$|trash[a[go(in\ n.out\ n)$
$.sync\,[out\ trash.c[]|trash[out\ a]]$
$|open\ sync]]).$

Finally, the traveling ambient can reach the destination, and the contents of *trash* are inaccessible and process *open a* cannot open the ambient $a[\ldots]$. Thus, we find that our "$+^n$" can behave as an ideal choice operator. □

In Ref. 2), the typed ambient calculus does not have variables as the π–calculus 9), while Ref. 4) defines names and variables for the untyped ambient calculus so that contextual equivalence is to be defined. Here, we introduce variables to the syntax and define free variables for ETAC as defined in 4).

Definition 4.7 (Messages)

M, N ::=message		
	x	variables
	$\cdots$	as in Definition 2.1.

□

Definition 4.8 (Free Variables)

$fv(x) \overset{def}{=} \{x\}\quad fv(n) \overset{def}{=} \emptyset\quad fv(\epsilon) \overset{def}{=} \emptyset$
$fv(in\ M) \overset{def}{=} fv(M)$
$fv(out\ M) \overset{def}{=} fv(M)$
$fv(open\ M) \overset{def}{=} fv(M)$
$fv(M.N) \overset{def}{=} fv(M) \cup fv(N)$
$fv((\nu G)P) \overset{def}{=} fv(P)$
$fv((\nu n{:}W)P) \overset{def}{=} fv(P)$
$fv((\gamma n{:}W)P) \overset{def}{=} fv(P)$
$fv(0) \overset{def}{=} \emptyset$
$fv(P|Q) \overset{def}{=} fv(P) \cup fv(Q)$
$fv(!P) \overset{def}{=} fv(P)$
$fv(M[P]) \overset{def}{=} fv(M) \cup fv(P)$
$fv(M.P) \overset{def}{=} fv(M) \cup fv(P)$
$fv((x_1{:}W_1, \ldots, x_k{:}W_k).P) \overset{def}{=}$
$fv(P) - \{x_1, \ldots, x_k\}$
$fv(\langle M_1, \ldots, M_k\rangle) \overset{def}{=} fv(M_1)\cup, \ldots, \cup fv(M_k)$
$fv(goN.M[P]) \overset{def}{=} fv(N) \cup fv(M) \cup fv(P)$ □

5. Problems of Existing Equivalences

Contextual equivalence for the untyped ambient calculus was presented in Ref. 4) based on the set of names of ambients that are or will be observable from the environment of the processes. We pointed out that there exist two contextual equivalent processes that have different behaviors by defining the choice macro "$+_u^n$", and we proposed another equivalence relation in Ref. 5).

5.1 Contextual Equivalence[☆]

First, we review the original equivalence relation.

Definition 5.1 (P Exhibits a Name n: $P \downarrow n$ [4]))

$$P \downarrow n \overset{def}{\Leftrightarrow} \text{ there are } \vec{m}, P', P'' \text{ such that } n \notin \{\vec{m}\} \text{ and } P \equiv (\nu\vec{m})(n[P']|P''). \quad \Box$$

Definition 5.2 (Convergence to a Name n: $P \Downarrow n$ [4]))

$$\text{(Conv Exh)}\ \frac{P \downarrow n}{P \Downarrow n} \qquad \text{(Conv red)}\ \frac{P \rightarrow Q \quad Q \Downarrow n}{P \Downarrow n} \qquad \Box$$

Definition 5.3 (Context [4])) $\mathcal{C}()$ is a process containing zero or more holes. $\mathcal{C}(P)$ is the outcome of filling all the holes of the context $\mathcal{C}()$ with the process P. $\Box$

Definition 5.4 (Contextual Equivalence: $P \simeq Q$ [4]))

$$P \simeq Q \overset{def}{\Leftrightarrow} \text{ for all } n \text{ and } \mathcal{C}() \text{ with } \mathcal{C}(P) \text{ and } \mathcal{C}(Q) \text{ closed, } \mathcal{C}(P) \Downarrow n \Leftrightarrow \mathcal{C}(Q) \Downarrow n. \quad \Box$$

The expression "$\mathcal{C}(Q)$ closed" means $\mathcal{C}(Q)$ does not have free variables.

Parallel testing equivalence for CCS [8]) is not adequate for the ambient calculus. This is explained in Ref. 4) as follows: for example, two processes $out\ p.0$ and 0 are testing equivalent though in the context $\mathcal{C}() \equiv p[m[()]]$, $\mathcal{C}(out\ p.0) \Downarrow m$ while $\mathcal{C}(0) \not\Downarrow m$. Thus, to capture the mobile properties of processes of the ambient calculus, contextual equivalence was presented.

Furthermore, in Ref. 5), we showed that there exist two processes of the untyped ambient calculus that are identified by contextual equivalence though they have different properties.

Example 5.5 Let P_u and Q_u be the processes of the untyped ambient calculus as follows:

$$P_u \overset{def}{=} a[P_{u1}] +_u^n a[P_{u2}]. \quad P_{u1} \overset{def}{=} a[P_{u2}] +_u^n b[].$$
$$Q_u \overset{def}{=} a[Q_{u1}] +_u^n a[Q_{u2}]. \quad P_{u2} \overset{def}{=} a[P_{u1}] +_u^n c[].$$
$$Q_{u1} \overset{def}{=} a[Q_{u1}] +_u^n b[].$$
$$Q_{u2} \overset{def}{=} a[Q_{u2}] +_u^n c[].$$

According to Definition 3.1 for "$+_u^n$", P and Q have the following structure:

$P_u \equiv (\nu\ trash, sync)$ (
 $a[in\ trash$
  $|\ go(in\ n.out\ n).sync[out\ trash.P_{u1}|trash[out\ a]]$
  $|\ open\ sync]$
 $|\ a[in\ trash$
  $|\ go(in\ n.out\ n).sync[out\ trash.P_{u2}|trash[out\ a]]$
  $|\ open\ sync]$).

$Q_u \equiv (\nu\ trash, sync)$ (
 $a[in\ trash$
  $|\ go(in\ n.out\ n).sync[out\ trash.Q_{u1}|trash[out\ a]]$
  $|\ open\ sync]$
 $|\ a[in\ trash$
  $|\ go(in\ n.out\ n).sync[out\ trash.Q_{u2}|trash[out\ a]]$
  $|\ open\ sync]$).

The behaviors of P_u and Q_u are illustrated in **Fig. 1**. Let $Tr \overset{def}{=} n[!in\ a|in\ c]$ be a traveling ambient. Each labelled arrow in Fig. 1 represents a transition of the process done by exe-

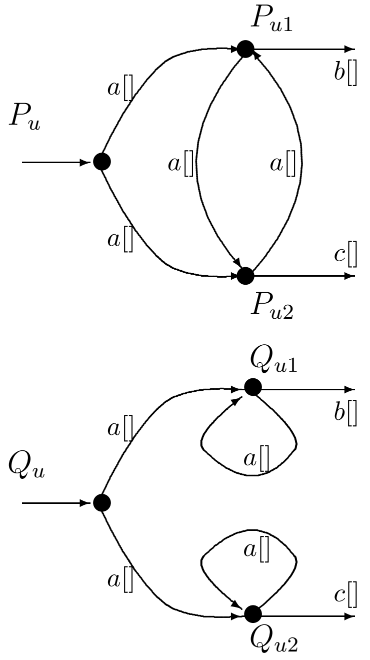

Fig. 1 Process P_u and process Q_u.

☆ All processes in this subsection are that of the untyped ambient calculus.

cuting one of the capabilities *in a*, *in b*, or *in c*, and each node represents a process. For example, the leftmost node of the upper graph is process $P_u|Tr$ and the leftmost arrow is a certain transition that makes process $P_u|Tr$ available. The upper labelled arrow with $a[]$ represents the execution of the capability *in a* that leads the process to the process $a[P_{u1}|Tr]$, which is expressed by the labeled node with P_{u1}.

When the traveling ambient Tr is running parallel to P_u, it cannot fail in reaching the ambient $c[]$, while it may fail when it is running parallel to Q_u. Therefore, P_u and Q_u should be distinguished. However, in Ref. 5), it was proved that contextual equivalence identifies $P_u \simeq Q_u$ by structural induction. □

As is explained in Ref. 4), contextual equivalence is a form of may testing equivalence. In order to define a finer equivalence relation, we defined the testing equivalence (composed of may and must testing) in Ref. 5) as follows.

Definition 5.6 (Hit a name n: $P \downdownarrows n$ 5))

$$\text{(Hit Exh)}\ \frac{P \downarrow n}{P \downdownarrows n} \qquad \text{(Hit Red)}\ \frac{\text{for any } Q \text{ st } P \rightarrow Q. \quad Q \downdownarrows n}{P \downdownarrows n}$$

□

Intuitively, $P \downdownarrows n$ iff ambient $n[]$ will be visible in every possible execution path of P as must testing.

Definition 5.7 (Contextually Testing Equivalence: $P \simeq_{test} Q$ 5))

$$P \simeq_{test} Q \overset{def}{\Leftrightarrow}$$
$$\text{for all } n, \mathcal{C}() \text{ with } \mathcal{C}(P) \text{ and } \mathcal{C}(Q) \text{ closed,}$$
$$\mathcal{C}(P) \Downarrow n \Leftrightarrow \mathcal{C}(Q) \Downarrow n$$
$$\text{and } \mathcal{C}(P) \downdownarrows n \Leftrightarrow \mathcal{C}(Q) \downdownarrows n.$$

□

Let $\mathcal{C}_u$ be the context as follows:

$$\mathcal{C}_u() \overset{def}{=} n[!in\ a] \mid !open\ a \mid open\ b \mid - .$$

In a usual parallel testing scenario, P_u in Example 5.5 would not pass must testing. This is because by using even a context (tester) that has infinite opening a action such as $\mathcal{C}_u$, from P_{u2} position of Fig. 1, we may not open b because of the infinite path of opening the a ambients; this means that $\mathcal{C}_u(P_u)$ may fail to open b and so does $\mathcal{C}_u(Q_u)$.

In a contextual testing scenario, however, we will **<u>observe</u>** the name b from the P_{u2} position of Fig. 1 even in that infinite path; this means that $\mathcal{C}_u(P_u)$ never fails to Hit b while $\mathcal{C}_u(Q_u)$ does from the Q_{u2} position. Consequently, $P_u \not\simeq_{test} Q_u$.

5.2 Problems

In ETAC, equivalences based on observable names in Subsection 5.1 are no longer adequate; that is, even contextually testing equivalence cannot distinguish the processes of ETAC that are similar to P_u and Q_u. We show the problems by defining the processes P and Q of ETAC in Example 5.8 such that the ambients whose names are a cannot be opened by any processes.

Example 5.8 Let P and Q be the processes as follows:

$$P \overset{def}{=} a[P_1] +^n a[P_2].$$
$$Q \overset{def}{=} a[Q_1] +^n a[Q_2].$$
$$P_1 \overset{def}{=} a[P_2] +^n b[]. \quad P_2 \overset{def}{=} a[P_1] +^n c[].$$
$$Q_1 \overset{def}{=} a[Q_1] +^n b[]. \quad Q_2 \overset{def}{=} a[Q_2] +^n c[].$$

According to **Definition 4.5** for "$+^n$", P and Q have the following structures:

$P \equiv$
$(\nu\, Trsh, Sync)$
$(\nu\ trash{:}Trsh^\frown\{Node, Thread\}[{}^\frown Node, {}^\circ\emptyset, Shh\ ])$
$(\gamma a, b, c{:}Node^\frown\emptyset[{}^\frown Trsh, {}^\circ\emptyset, Shh\ ])$
$(\nu sync{:}Sync^\frown\{Thread\}[{}^\frown Trsh, {}^\circ Sync, Shh\ ])($
$\quad a[in\ trash$
$\quad\ \mid go(in\ n.out\ n).sync[out\ trash.P_1|trash[out\ a]]$
$\quad\ \mid open\ sync]$
$\quad \mid a[in\ trash$
$\quad\ \mid go(in\ n.out\ n).sync[out\ trash.P_2|\ trash[out\ a]]$
$\quad\ \mid open\ sync]).$

$Q \equiv$
$(\nu\, Trsh, Sync)$
$(\nu\ trash{:}Trsh^\frown\{Node, Thread\}[{}^\frown Node, {}^\circ\emptyset, Shh\ ])$
$(\gamma a, b, c{:}Node^\frown\emptyset[{}^\frown Trsh, {}^\circ\emptyset, Shh\ ])$
$(\nu sync{:}Sync^\frown\{Thread\}[{}^\frown Trsh, {}^\circ Sync, Shh\ ])($
$\quad a[in\ trash$
$\quad\ \mid go(in\ n.out\ n).sync[out\ trash.Q_1|trash[out\ a]]$
$\quad\ \mid open\ sync]$
$\quad \mid a[in\ trash$
$\quad\ \mid go(in\ n.out\ n).sync[out\ trash.Q_2|\ trash[out\ a]]$
$\quad\ \mid open\ sync]).$

In an untyped case, contextually testing equivalence distinguished P_u from Q_u by opening the ambients whose name is a and exhibiting the name c, though opening control prevents us from using the same strategy. □

We can prove that contextually testing equivalence identifies P and Q in Example 5.8.

Lemma 5.9 Let P and Q be the processes in Example 5.8. For all n and $\mathcal{C}()$ with $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ closed, $\mathcal{C}(P) \Downarrow n \Rightarrow \mathcal{C}(Q) \Downarrow n$ and $\mathcal{C}(P) \downdownarrows n \Rightarrow \mathcal{C}(Q) \downdownarrows n$. □

Proof: Suppose that $\mathcal{C}(P) \Downarrow n$ and $\mathcal{C}(P) \downdownarrows n$. There are two cases: n can be the name of the ambient in P (case (1)) or in $\mathcal{C}(P)$ (case (2)).

case (1) As the ambient a cannot be opened, any reduction does not expose the ambients in P_1, P_2, P_3, P_4. Thus, obviously, the only name n that is and will be visible from the environment of P and Q is a. Thus, for all n (appearing in P) and $\mathcal{C}()$ with $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ closed, $\mathcal{C}(P) \Downarrow n \Rightarrow \mathcal{C}(Q) \Downarrow n$ and $\mathcal{C}(P) \downdownarrows n \Rightarrow \mathcal{C}(Q) \downdownarrows n$.

case (2) consists of case (i), where n (appearing in $\mathcal{C}()$) is in the top level; that is, $\mathcal{C}(P) \downarrow n$, case(ii) n (appearing in $\mathcal{C}()$) will be visible without any interaction with P, and case (iii) n (appearing in $\mathcal{C}()$) will be visible after several interactions with P. In cases (i) and (ii), it is obvious that $\mathcal{C}(P) \Downarrow n \Rightarrow \mathcal{C}(Q) \Downarrow n$ and $\mathcal{C}(P) \downdownarrows n \Rightarrow \mathcal{C}(Q) \downdownarrows n$.

Case (iii). From the structure of P and the types of a, b, c in P, the only executable capabilities of the ambient in $\mathcal{C}()$ are subjective or objective moves concerned with a, b, c. Suppose that after several reductions, $m[S|M.go(out\ m).n[R]]$ appearing in $\mathcal{C}()$ goes into P and comes back from P and $n[R']$ appears in the top level for some m, S, M, R, R' with R reduced to R'.

- Let M be $in\ a.out\ a$. Obviously $\mathcal{C}(P) \Downarrow n \Rightarrow \mathcal{C}(Q) \Downarrow n$ and $\mathcal{C}(P) \downdownarrows n \Rightarrow \mathcal{C}(Q) \downdownarrows n$.
- Let M be $in\ a.in\ b.out\ b.out\ a$. $\mathcal{C}(P) \not\downdownarrows n$.
- Let M be $in\ a.in\ c.out\ c.out\ a$. $\mathcal{C}(P) \not\downdownarrows n$.
- Let M be $in\ a * .in\ b.out\ b.out\ a*$, where $*$ means more than 0 times iteration. There is a computation path that has infinite $in\ a$ that prevents us from executing $in\ b$. Thus, $\mathcal{C}(P) \not\downdownarrows n$.
- Any other combination of those capabilities gives the same results in these cases because of the structure of P. □

Lemma 5.10 Let P and Q be the processes in Example 5.8. For all n and $\mathcal{C}()$ with $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ closed, $\mathcal{C}(Q) \Downarrow n \Rightarrow \mathcal{C}(P) \Downarrow n$ and $\mathcal{C}(Q) \downdownarrows n \Rightarrow \mathcal{C}(P) \downdownarrows n$ □

The proof is just the same. Consequently, we prove $P \simeq_{test} Q$.

6. Capability Equivalence

As we cannot rely on the equivalences based on the names visible from environments any more, we concentrate our attention on what processes can do after they have performed some actions.

Definition 6.1 (Labels *cap*) Letting n be a name, we define a set of labels ranged over by *cap* as follows:

cap ::=	label
in n	enter n ambient
out n	exit n ambient
open n	open n ambient
go(*in n*)	enter n ambient objectively
go(*out n*)	enter n ambient objectively

□

We define the labelled transition system only for *cap* of Definition 6.1 that changes the structure of the processes.

Definition 6.2 (Labelled Transition $\xrightarrow{cap}$)

$$\frac{}{n[in\ m.P|Q]|m[R] \xrightarrow{in\ m} m[n[P|Q]|R]}$$

$$\frac{}{m[n[out\ m.P|Q]|R] \xrightarrow{out\ m} n[P|Q]|m[R]}$$

$$\frac{}{open\ m.P|m[Q] \xrightarrow{open\ m} P|Q}$$

$$\frac{}{go(in\ m.N).n[P]|m[Q] \xrightarrow{go(in\ m)} m[go(N).n[P]|Q]}$$

$$\frac{}{m[go(out\ m.N).n[P]|Q] \xrightarrow{go(out\ m)} go(N).n[P]|m[Q]}$$

$$\frac{P \xrightarrow{cap} Q}{P|R \xrightarrow{cap} Q|R} \qquad \frac{P \xrightarrow{cap} Q}{m[P] \xrightarrow{cap} m[Q]}$$

$$\frac{P \xrightarrow{cap} Q,\ (\nu n{:}W) \text{ dose not interfere } cap \text{ in } P}{(\nu n{:}W)P \xrightarrow{cap} (\nu n{:}W)Q}$$

$$\frac{P \xrightarrow{cap} Q}{(\nu G)P \xrightarrow{cap} (\nu G)Q}$$

$$\frac{n \in \{m\} \cup fn(Q)}{(\ \gamma n{:}W)n[m[out\ n.Q]|P] \xrightarrow{out\ n} (\gamma n{:}W)n[P]|(\gamma n{:}W)m[Q]}$$

$$\frac{n \in \{m\} \cup fn(Q) \cup fn(N)}{(\gamma n{:}W)n[go(out\ n.N).m[P]|Q] \xrightarrow{go(out\ n)} (\gamma n{:}W)n[P]|(\gamma n{:}W)go(N).m[P]}$$

$$\frac{\begin{array}{l} P \xrightarrow{cap} Q,\ (\gamma n{:}W) \text{ dose not interfere } cap \text{ in } P, \\ \text{for any } R \text{ and } S. \\ \quad (P \not\equiv n[m[out\ n.R]|S], \\ \quad\ P \not\equiv n[go(out\ n.N).m[R]|S] \\ \quad\ \text{where } n \in \{m\} \cup fn(R) \cup fn(N)) \end{array}}{(\gamma n{:}W)P \xrightarrow{cap} (\gamma n{:}W)Q}$$

□

Definition 6.3 (Predicate $\mathit{Can}(P, \mathit{cap})$) Let P be a process of ETAC and *cap* be any label defined in Definition 6.1. We define the predicate $\mathit{Can}(P, \mathit{cap})$ as follows: $\exists P'.P \overset{*}{\rightarrow}\overset{cap}{\rightarrow} P'$, where '$\overset{*}{\rightarrow}$' expresses a reflexive transitive closure of reduction '$\rightarrow$' defined in Definition 4.4. □

Intuitively, P can execute capability *cap* as the last step of the transition that reduces P to P'.

Definition 6.4 (Process R_P^{cap}) Let P be a process and *cap* be any label defined in Definition 6.1 such that $\mathit{Can}(P, \mathit{cap})$. From the definition of *Can*, the following condition holds:

$$\exists R, P'.P \overset{*}{\rightarrow} R \wedge R \overset{cap}{\rightarrow} P'.$$

We define R_P^{cap} to be any process R satisfying the condition

$$R_P^{cap} \in \{R \mid \exists P'.P \overset{*}{\rightarrow} R \overset{cap}{\rightarrow} P'\}.$$

□

Definition 6.5 (Binary Relation $P \leq_{can} Q$)

$P \leq_{can} Q \overset{def}{=}$

$\forall \mathit{cap}, \mathcal{C}()$. if $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ are closed, then

$$\mathit{Can}(\mathcal{C}(P), \mathit{cap}) \Rightarrow \mathit{Can}(\mathcal{C}(Q), \mathit{cap})$$
$$\wedge\ \forall \mathit{cap}',\ \forall R_{\mathcal{C}(P)}^{cap}, \exists R_{\mathcal{C}(Q)}^{cap}.$$
$$\mathit{Can}(R_{\mathcal{C}(P)}^{cap}, \mathit{cap}') \Rightarrow \mathit{Can}(R_{\mathcal{C}(Q)}^{cap}, \mathit{cap}').$$

□

Definition 6.6 (Capability Equivalence)

$$P \simeq_{can} Q \overset{def}{=} P \leq_{can} Q \wedge Q \leq_{can} P.\ \Box$$

We explain the intuitive notion of capability equivalence in the following example.

Let $\mathcal{C}()$ be the context as follows:

$$\mathcal{C}() \overset{def}{=} (\gamma n{:}\mathit{Thread}{\curvearrowright}\emptyset[{\curvearrowright}\mathit{Node}, {}^{\circ}\emptyset, \mathit{Shh}\ ])$$
$$(n[!\mathit{in}\ a|\mathit{in}\ c]|-).$$

and let P and Q be the processes defined in Example 5.8. Then, $\mathit{Can}(\mathcal{C}(P), \mathit{in}\ c) \wedge \mathit{Can}(R_{\mathcal{C}(P)}^{in\ c}, \mathit{in}\ b)$ holds. This means that when process $\mathcal{C}(P)$ is reduced to a process that can directly enter the ambient $c[]$, there is still a possibility that the ambient $b[]$ is accessible. On the other hand, when process $\mathcal{C}(Q)$ is reduced to the same stage, it will not be able to access the ambient $b[]$. Consequently, capability equivalence can distinguish P from Q.

Proposition 6.7 Capability equivalence is a congruence. □

Proof: We can apply the strategy of the proof for contextual equivalence[4)].

(Equivalence part) Reflexivity and symmetry are trivial. We show the proof for only transitivity. Suppose that $P_1 \leq_{can} P_2$ and $P_2 \leq_{can} P_3$. Let $\mathcal{C}()$ be any context such that $\mathcal{C}(P_1)$ and $\mathcal{C}(P_3)$ are closed, let θ be a closing substitution for $\mathcal{C}(P_2)$, and let $\mathcal{D}() \overset{def}{=} \mathcal{C}()\theta$. Suppose that *cap* is any label of Definition 6.1 such that $\mathit{Can}(\mathcal{C}(P_1), \mathit{cap})$ and let $R_{\mathcal{C}(P_1)}^{cap}$ be any processes satisfying $\exists \mathit{cap}'.\mathit{Can}(R_{\mathcal{C}(P_1)}^{cap}, \mathit{cap}')$.

Since $\mathcal{C}(P_1)$ are closed, $\mathcal{C}(P_1) \equiv \mathcal{D}(P_1)$. Thus,

$$\mathit{Can}(\mathcal{D}(P_1), \mathit{cap}) \text{ and } \exists \mathit{cap}', \forall R_{\mathcal{D}(P_1)}^{cap}.$$
$$\mathit{Can}(R_{\mathcal{D}(P_1)}^{cap}, \mathit{cap}').$$

With this condition, the fact that $\mathcal{D}(P_2)$ is closed, and the assumption $P_1 \leq_{can} P_2$, we are led to the following condition:

$$\mathit{Can}(\mathcal{D}(P_2), \mathit{cap}) \wedge \exists R_{\mathcal{D}(P_2)}^{cap}.\mathit{Can}(R_{\mathcal{D}(P_2)}^{cap}, \mathit{cap}').$$

As $\mathcal{C}(P_3)$ is closed, $\mathcal{C}(P_3) \equiv \mathcal{D}(P_3)$ and $\mathcal{D}(P_3)$ is also closed. These conditions and the assumption $P_2 \leq_{can} P_3$ lead us to the following condition:

$$\mathit{Can}(\mathcal{D}(P_3), \mathit{cap}) \wedge \exists R_{\mathcal{D}(P_3)}^{cap}.\mathit{Can}(R_{\mathcal{D}(P_3)}^{cap}, \mathit{cap}').$$

As $\mathcal{C}(P_3) \equiv \mathcal{D}(P_3)$,

$$\mathit{Can}(\mathcal{C}(P_3), \mathit{cap}) \wedge \exists R_{\mathcal{C}(P_3)}^{cap}.\mathit{Can}(R_{\mathcal{C}(P_3)}^{cap}, \mathit{cap}').$$

Therefore, $P_1 \leq_{can} P_3$. The symmetric procedure proves that $P_3 \leq_{can} P_1$.

(Precongruence part) We can apply a similar proof to the one in Ref. 4). Let P and Q be any capability equivalent processes, *cap* be any label of Definition 6.1, and $\mathcal{C}()$ be any context such that $\mathcal{C}(P)$ and $\mathcal{C}(Q)$ are closed. Suppose that $\mathcal{D}()$ is any context such that $\mathcal{D}(\mathcal{C}(P))$ and $\mathcal{D}(\mathcal{C}(Q))$ are closed and $\mathit{Can}(\mathcal{D}(\mathcal{C}(P)), \mathit{cap})$ and $\exists \mathit{cap}'.\mathit{Can}(R_{\mathcal{D}(\mathcal{C}(P))}^{cap}, \mathit{cap}')$.

As $\mathcal{D}(\mathcal{C}())$ is a context and $P \leq_{can} Q$, $\mathit{Can}(\mathcal{D}(\mathcal{C}(Q)), \mathit{cap})$ and $\mathit{Can}(R_{\mathcal{D}(\mathcal{C}(Q))}^{cap}, \mathit{cap}')$ by Definition 6.6 and 6.5. Consequently, we have $\mathcal{C}(P) \leq_{can} \mathcal{C}(Q)$. The symmetric way proves $\mathcal{C}(Q) \leq_{can} \mathcal{C}(P)$.

Because we have proved $\mathcal{C}(P) \simeq_{can} \mathcal{C}(Q)$ for any context $\mathcal{C}()$ and for any two processes P and Q such that $P \simeq_{can} Q$, capability equivalence "$\simeq_{can}$" is a precongruence. □

7. Conclusions

The primary result of this paper is that it showed the problems of the type system defined in Ref. 2) and presented an extension of the type system. It also presented an equivalence relation for the extended typed ambient calculus.

The type system in Ref. 2) gives type to an ambient according to the group to which the name of the ambient belongs and to the groups among which the ambient can move around. In that system, a restriction is defined on names and groups. When we try to give an ambient a property (e.g., cannot be opened, cannot move around), however, we have to bind the name of the ambient even if the name belongs to a free group. That is, bound names are members of free groups. For processes providing global service, however, this is inconvenient.

To solve this problem, we extended the definitions of free names and processes by adding the type tag. The type tag resembles name restriction, though it does not bind names but only gives ambients restrictions on opening and moving properties. This method gives us a flexible way to specify entities on networks that have global names.

An equational relation, called contextual equivalence, was given to the untyped ambient calculus based on the names observed from environments. But the type system, especially the opening control of ambients, makes the equivalence hard to apply to the typed ambient calculus. Contextual equivalence identifies or discriminates processes by their possibility of exhibiting the names of the ambients that compose the processes. For security, however, the ambients that express nodes of a network should not be opened. As a result, the network becomes invisible and contextual equivalence can only tell trivial differences.

As the opening control prevents us from distinguishing processes by exhibiting names of ambients, we defined the capability equivalence based on what the processes can do after they have performed several actions. We did not use the notion of what the process can do because there are processes that the method cannot distinguish them though they have different properties. We showed such a phenomenon by defining choice macro processes in Ref. 5). We proposed capability equivalence using the should testing idea 1) originally presented for CCS processes to distinguish such processes.

We need to compare our capability equivalence with bisimulations. For example, a bisimulation is defined for safe ambients that is also a congruent relation in Ref. 7). We also intend to check if the choice macro is to be defined by using the type system defined in Ref. 6): the dynamic types.

References

1) Brinksma, E., Rensink, A. and Vogler, W.: Fair Testing, *LNCS*, Vol.962, pp.313–327 (1995).

2) Cardelli, L., Ghelli, G. and Gordon, A.D.: Types for the Ambient Calculus, *Information and Computation*, Vol.177, pp.60–194 (2002).

3) Cardelli, L. and Gordon, A.D.: Mobile Ambients, *LNCS*, Vol.1378, pp.140–155 (1998).

4) Gordon, A.D. and Cardelli, L.: Equational Properties of Mobile Ambients, *Mathematical Structures in Computer Science*, Vol.13, No.3, pp.371–408 (2003).

5) Kato, T.: An Equational Relation for Ambient Calculus, *Trans. Information Processing Society of Japan*, Vol.46, No.12, pp.3016–3029 (2005).

6) Lhoussaine, C. and Sassone, V.: A Dependently Typed Ambient Calculus, *LNCS*, Vol.2986, pp.171–187 (2004).

7) Merro, M. and Hennessy, M.: Bisimulation Congruences in Safe Ambients, *Proc. 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp.71–80 (2002).

8) Milner, R.: *Communication and Concurrency*, Prentice Hall (1989).

9) Parrow, J.: An Introduction to the π-Calculus, Bergstra, J.A., Ponse, A. and Smolka, S.A. (Eds.), *Handbook of Process Algebra*, North-Holland (2001).

(Received December 16, 2006)
(Accepted March 25, 2007)

Toru Kato received his Ph.D. degree from Okayama University in 1997. Since 1998 he had been a research fellow of the Japan Society for the Promotion of Science. Since 2000 he has been in Kinki University as a lecturer. His current research interests are the theoretical study of concurrent processes, implementation of process algebras, and formal semantics of concurrent logic programming languages. He is a member of IPSJ, IEICE, and JSSST.