INFINITE TYPES FOR DISTRIBUTED OBJECT INTERFACES

Elie Najm[1], Abdelkrim Nimour[1] and Jean-Bernard Stefani[2]

[1] École Nationale Supérieure des Télécomunications
[Elie.Najm, Abdelkrim.Nimour]@enst.fr

[2] France Telecom-CNET
jeanbernard.stefani@cnet.francetelecom.fr

Abstract:

In a previous work [12] we presented COB, an object calculus which features objects with dynamically changing service offers. We also defined a type system for interfaces and a typing discipline that guarantees that no object may experience an unexpected service request at run-time. In the present paper, we extend our type system such that it describes "infinite state" types. We define equivalence and subtyping relations over those types based on bisimulation and simulation relations. We also define an algorithm to decide the equivalence or the subtyping relation of two types.

The type system is applied to OL, a new calculus that uses the interface "states" in the behavior of the objects. As in COB, in OL there is a distinction between private and public interfaces. A private interface can have at most one client at a time whereas a public interface can be known by more that one client. Private interfaces can thus propose a non-uniform service offer (the set available services may change during the computation). OL extends the results of COB to infinite types: in spite of non-uniform service offers, in a well-typed

P. Ciancarini et al. (eds.), *Formal Methods for Open Object-Based Distributed Systems*

configuration, there may not occur any "service not understood" error at run-time.

INTRODUCTION

behavior types

Usually type systems for objects or for object interfaces are satisfied by defining the signatures of methods that an object may accept. In [12], we defined a type system where one can define ordering constraints on the messages (methods) that can be handled by an object. For example the behavior type of a one place buffer interface can be written like this:

$$\begin{array}{rcl} \text{OneBuffer} & = & \text{put}(\cdots);\text{Full} \\ \text{Full} & = & \text{get}(\cdots);\text{OneBuffer} \end{array}$$

Interfaces of this type expect to receive repeatedly put messages followed by get messages. To define a two place buffer, one must introduce an intermediate state where both put and get messages are allowable:

$$\begin{array}{rcl} \text{TwoBuffer} & = & \text{put}(\cdots);\text{Intermediate} \\ \hline \text{Intermediate} & = & \begin{array}{l} \text{put}(\cdots);\text{Full} \\ + \\ \text{get}(\cdots);\text{TwoBuffer} \end{array} \\ \hline \text{Full} & = & \text{get}(\cdots);\text{Intermediate} \end{array}$$

The definition of the type of a larger buffer may be very complex because of the multiplication of states. A simple way to avoid this complexity is to introduce a counter in the buffer:

$$\text{TwoBuffer}[n] = \begin{array}{ll} [n < 2] & \text{put}(\cdots);\text{TwoBuffer}[n+1] \\ + & \\ [n > 0] & \text{get}(\cdots);\text{TwoBuffer}[n-1] \end{array}$$

The parameter n represents the difference between the number of put and get messages. Actually, this is the definition of an infinite family of buffers, TwoBuffer[0], TwoBuffer[1], ... The empty two place buffer being TwoBuffer[0]. The put messages are possible only if $n < 2$; i.e., less than two elements in the buffer, while get messages are possible only if $n > 0$; i.e., the buffer is not empty. With this representation it is very easy to define the type of a larger buffer. The put messages have to be guarded with the maximum number of elements allowed in the buffer: $[n < max]\,\text{put}(\cdots);\text{Buffer}[n+1]$. The principal novelty of this representation of behavior types comparing to the one presented in [12] is that it is possible to describe types with unbounded number of states:

$$\text{Buffer}[n] = \begin{array}{ll} & \text{put}(\cdots);\text{Buffer}[n+1] \\ + & \\ [n > 0] & \text{get}(\cdots);\text{Buffer}[n-1] \end{array}$$

The above definition is the behavior type of an unbounded buffer. There is no restriction on the number of elements it can hold. The only constraint is that the number of put's must be greater or equal to the number of get's.

The aim of this type system is to provide a way to record the history of the usage of an interface. Typically, we want to be able to count the messages handled by an interface and compare them. So in our type system after receiving a message the type state may be incremented or decremented by a certain positive integer value (counting) and the receiving of a message can be guarded by a predicate of the form: $n < c$ or $n > c$ where c is a positive integer constant (comparison).

This type system allows to define relations between more than two messages. Suppose that we have a special buffer with two put operations. Of course, we want the total number of put's to be greater or equal to the number of get's:

$$\begin{array}{lll} & & \mathrm{put}_1(\cdots);\mathrm{Buffer}[n+1] \\ & + & \\ \mathrm{Buffer}[n] = & & \mathrm{put}_2(\cdots);\mathrm{Buffer}[n+1] \\ & + & \\ & [n>0] & \mathrm{get}(\cdots);\mathrm{Buffer}[n-1] \end{array}$$

The calculus

We define OL, a calculus that describes configurations of objects running in parallel and communicating with each other by exchanging messages. The syntax of the language is given by the grammar of table 1. In this syntax, the

$$\begin{array}{rcl} I & ::= & u{:}T \\ Dcl & ::= & A[\tilde{I}] = B \\ Guard & ::= & u.n > c \mid u.n < c \\ Recep & ::= & ?u\big([Guard_1]m_1(\tilde{I}_1) = B_1, \cdots, [Guard_n]m_n(\tilde{I}_n) = B_n\big) \\ B & ::= & 0 \\ & \mid & !u.m(\tilde{\rho}\tilde{v}) > B \\ & \mid & \textstyle\sum_{i=1}^{n} Recep_i \\ & \mid & A[\tilde{\rho}\tilde{v}] \\ & \mid & \text{create } A[\tilde{\rho}\tilde{v}] > B \\ & \mid & \text{if } Guard \text{ then } B_1 \text{ else } B_2 \\ C & ::= & B \mid C|C \mid \text{new } u{:}\mu x[a] \text{ in } C \end{array}$$

Table 1 Syntax of OL

terminal T denotes an interface type. An interface type is a pair consisting of a mode (public or private) and a behavior type. A private interface can be known, i.e., its client role held, by only one client at any time whereas a public interface can have multiple clients. As a public interface must be

able to offer services to multiple clients, its service offer has to be uniform; i.e., the set of services available at a public interface does not change during the computation. In contrast, the services available at a private interface can change dynamically depending on the protocol of interaction between the two partners. The mode (noted μ) also determines the kind of interaction on this interface. The communication on a private interface is by rendez-vous. The invocation of a public interface creates a message that will be absorbed by its destination interface in a subsequent step. These two modes of interaction are inspired by the interaction modes of the ODP computational model (see [13] for a formal definition of this model). In this model, communication between objects is possible only if their interfaces are bound. There are two forms of bindings: implicit and explicit. Using an "implicit binding", an object only has to know the reference of one of the interfaces of a server. A service invocation then corresponds to a message creation that has to be transported by the system infrastructure to its destination. The public interfaces are intended to model this form of interaction. The second form of binding in ODP is the "explicit binding". In this case, the two objects, O_1 and O_2, wanting to communicate have to create a binding object. This binding object is distributed; i.e., one of its interfaces will be co-located with the object O_1 and another one co-located with the object O_2. The invocation of a service of O_2 by O_1 is dealt with as follows: O_1 sends the invocation message to the binding object's co-located interface and then the binding object delivers the message to O_2 via the O_2 co-located interface. The interaction between O_1, O_2 and the binding object are synchronous. This is possible because interfaces are co-located and private (the interfaces of the binding object have been created especially for the communication between O_1 and O_2). The "explicit binding" form of interaction of ODP corresponds to the interaction mode of private interfaces.

We briefly introduce the main features of our calculus using the following stack example:

```
Stack[self: private ?Buffer[n], top: private !TCell]=
  ?self[[self.n>0]get(r1: T1)= new r2: T2 in
                  !top.read(!r2) >
                    ?r2[ret(e: T, next: TCell)= !r1.ret(!e) >
                          Stack[?self, !next]
                       ]
                 put(e: T)= new cell: private TCell in
                   create Cell[?cell, !top, !e] >
                     Stack[?self, !cell]
     ]

Cell[self: private ?TCell, next: private !TCell , e: T]=
  ?self[read(r: T2)= !r.ret(!e, !next) > 0]
```

The first object, Stack, has the server role (ability to receive messages) of the private interface self. Its behavior is specified by its parametric behavior type, Buffer[n] introduced in the preceeding section. All the put and get services'

invocations will arrive on this interface. Stack also has the client role (ability to emit messages) of the private interface top of type TCell (interfaces of type TCell can only perform once one read action). The behavior of Stack is to wait for a put or a get message. This behavior is non-uniform in the sense that the get service is guarded with a predicate and thus is not always available. The guard of the get service ensures that the behavior type state (n) is not null which means that the stack is not empty.

get service. the argument r1 is a reference to the interface where to return the top of the stack. The client role of the newly created interface is sent to top as an argument of read. Stack then waits on this interface for e the element on the top the stack and next the reference (client role) of the Cell containing the next element. Stack sends e along r1 and then returns to it initial state with an updated value of its top.

put service. Stack simply creates a new Cell with top as successor and then returns to its initial state with the new cell as the value of top.

This piece of code is very similar to what would have been the coding of a stack in ADA, for example. The advantage of the OL approach is that the "synchronization constraints" are encapsulated in the behavior type. The typing system then ensures that the user (programmer) has checked that the stack accepts get action only if the behavior type parameter is greater than 0. In languages like ADA, this is the responsibility of the programmer.

BEHAVIOR TYPES

A parametric behavior type is a quadruplet (E, x, n, r), noted $E \triangleright rx[n]$, where n is the parameter of the type, r is the set of capabilities of the behavior type: $r \subset \{!, ?\}$[1]. The environment E is a set of equations of the form $x_i[n] = e_i$. Each x_i is a behavior type variable that appears once and only once in the left-hand side of an equation. Each e_i is an expression defined by the following syntax:

$$e ::= \sum_{i=1}^{n} [\text{pred}_i] m_i(\tilde{\rho_i}\tilde{x_i}[a_i]); x_i'[f_i]$$

where:

- each m_i is a method name. We consider only deterministic behavior types i.e. $i \neq j \Rightarrow m_i \neq m_j$
- each $\tilde{x_i}$ is a list of behavior type variables describing the behavior of the method arguments
- each $\tilde{\rho_i}$ is the role, client (!), server (?) or both (!?)

[1]Note that the set of capabilities may be empty whereas a role cannot

- each a_i is a positive integer constant
- each pred_i is a predicate of the form $pred(n) = n < c$ or $pred(n) = n > c$ where c is a positive integer constant
- each f_i is a function of the form: $f(n) = n + c$ or $f(n) = n - c$ where c is a positive constant.

Notation. we will write $(x[n] \rightarrow [\text{pred}]m(\tilde{\rho}\tilde{x}[a]); x'[f]) \in E$ to mean that the environment E contains the equation:
$x[n] = [\text{pred}]m(\tilde{\rho}\tilde{x}[a]); x'[f] + \sum_{i=1}^{n} [\text{pred}_i]m_i(\tilde{\rho}_i\tilde{x}_i[a_i]); x_i'[f_i]$

Behavior type action

Since we want to deal with non-uniform service availability, we have to specify how types evolve. We consider here only instantiated behavior types, ranged over by the meta-variable X. A type evolves by performing a type action. A type action is a method signature annotated with a role: $\rho m(X_1, \ldots, X_n)$. After performing an action a type evolves to another type as shown in the following rule:

$$\frac{(x[n] \rightarrow [\text{pred}]m(\rho_1 x_1[a_1], \cdots, \rho_n x_n[a_n]); y[f]) \in E \quad \text{pred}(b) = \text{TRUE}}{E \triangleright rx[b] \xrightarrow{\rho m(E \triangleright \rho_1 x_1[a_1], \cdots, E \triangleright \rho_n x_n[a_n])} E \triangleright ry[f(b)]} \text{ with } \rho \subset r$$

Which informally reads; If there is an equation containing a method m in the parametric behavior type definition and if the predicate is evaluated to TRUE for a value b then the instantiation of this behavior type with the value b can perform an m action. Let us, for example, consider how the type of an empty unbounded buffer can evolve .[2]

$$?\text{Buffer}[0] \xrightarrow{?put()} ?\text{Buffer}[1]$$

?Buffer[0] can perform a put action and then evolves to ?Buffer[1].

Behavior type equivalence

To define the equivalence of two behavior types we use the well known bisimulation relation (see [9]).

Definition 1 (Bisimulation)
A binary relation β over behavior types is a bisimulation if $(X_1, X_2) \in \beta$ implies:

i) $X_1 \xrightarrow{\rho m(\tilde{Y}_1)} X_1' \Rightarrow X_2 \xrightarrow{\rho m(\tilde{Y}_2)} X_2'$ and $(X_1', X_2') \in \beta$ and $(\tilde{Y}_1, \tilde{Y}_2) \in \beta$

[2] Usually, we consider, without loss of generality, that all the type equations are defined in the same environment E. So, the behavior type $E \triangleright rx[n]$ is written $rx[n]$.

ii) $X_2 \xrightarrow{\rho m(\tilde{Y}_2)} X_2' \Rightarrow X_1 \xrightarrow{\rho m(\tilde{Y}_1)} X_1'$ and $(X_1', X_2') \in \beta$ and $(\tilde{Y}_1, \tilde{Y}_2) \in \beta$

Definition 2 (Type equivalence)
Two types X_1, X_2 are equivalent, noted $X_1 \sim X_2$, iff $(X_1, X_2) \in \beta$ for some bisimulation relation B.

Behavior subtypes

Two behavior types X_1 and X_2 are in a subtyping relation if:

server case: all receiving actions that X_1 can perform can be performed by X_2 too

client case: all sending actions that X_2 can perform can be performed by X_1 too

Definition 3 (Subtyping relation)
A binary relation $\mathcal{R}$ over behavior types is a subtyping relation if $(X_1, X_2) \in \mathcal{R}$ implies:

i) $X_1 \xrightarrow{?m(\tilde{Y}_1)} X_1' \Rightarrow X_2 \xrightarrow{?m(\tilde{Y}_2)} X_2'$ and $(X_1', X_2') \in \mathcal{R}$ and $(\tilde{Y}_1, \tilde{Y}_2) \in \mathcal{R}$

ii) $X_2 \xrightarrow{!m(\tilde{Y}_2)} X_2' \Rightarrow X_1 \xrightarrow{!m(\tilde{Y}_1)} X_1'$ and $(X_1', X_2') \in \mathcal{R}$ and $(\tilde{Y}_1, \tilde{Y}_2) \in \mathcal{R}$

Let us consider the following example where u is a client interface and v a server interface:

$$A_1[u{:}\mu X_1] = B_1 \qquad A_2[v{:}\mu X_2] = B_2$$
$$\text{new } w{:}\mu X \text{ in create } A_1[w^!] > \text{create } A_2[w^?] > 0$$

The interface w instantiates both u and v so X is a subtype of both X_1 and X_2. According to our definition: i) $X_1 \xrightarrow{!m()} X_1'$ implies $X \xrightarrow{!m()} X'$; ii) $X \xrightarrow{?m()} X'$ implies $X_2 \xrightarrow{?m()} X_2'$. We are then sure that all the services invoked by B_1 will be available in B_2.

Definition 4 (Subtyping)
A behavior type X_1 is a subtype of a behavior type X_2, noted $X_1 \preceq X_2$, if $(X_1, X_2) \in \mathcal{R}$ for some subtyping relation $\mathcal{R}$.

Deciding behavior type equivalence

As we have seen before, to prove the equivalence of two types X_1 and X_2 we must exhibit a bisimulation relation β such that $(X_1, X_2) \in \beta$.

Definition 5 (Bisimulation construction (I))

- $(X_1, X_2) \in \beta_0$ (init)

- for all $(X_1, X_2) \in \beta_k$ (small step)
  - $(X_1, X_2) \in \beta_{k+1}$
  - case $X_1 \xrightarrow{\rho m(\tilde{Y}_1)} X_1'$: if $X_2 \xrightarrow{\rho m(\tilde{Y}_2)} X_2'$ then $(X_1', X_2') \in \beta_{n+1}$ and $(\tilde{Y}_1, \tilde{Y}_2) \in \beta_{n+1}$ else Fail
  - case $X_2 \xrightarrow{\rho m(\tilde{Y}_2)} X_2'$: if $X_1 \xrightarrow{\rho m(\tilde{Y}_1)} X_1'$ then $(X_1', X_2') \in \beta_{n+1}$ and $(\tilde{Y}_1, \tilde{Y}_2) \in \beta_{n+1}$ else Fail
- $\beta_k = \beta_{k+1}$ (success)

Property. The fixed-point relation constructed using the rules of the Bisimulation construction (I) is a bisimulation relation over behavior types.

This property is due to the fact that the rules of the relation construction (Definition 5) are directly inspired by the bisimulation definition.
Let us consider the following example:

$B1[n] =$	$[n > 0]$	get(); $B1[n-1]$	$B2[n] =$	$[n > 0]$	get(); $B2[n-2]$
	$+$	put(); $B1[n+1]$		$+$	put(); $B2[n+2]$

If we want to check the equivalence between !B1[0] and !B2[0] we have to start from a relation $\beta_0 = \{(!B1[0], !B2[0])\}$. Since both !B1[0] and !B2[0] can perform a put action, we have $\beta_1 = \{(!B1[0], !B2[0]), (!B1[1], !B2[2])\}$. It is easy to see that it is not possible to reach a fixed-point (in a finite number of steps) because both types can always perform a put action and then evolve to new behavior types with a greater parameter. So $\beta_2 = \beta_1 \cup \{(!B1[2], !B2[4])\}$, $\beta_3 = \beta_2 \cup \{(!B1[3], !B2[6])\}$ and so on. A simple way to avoid this kind of infinite sequence is to add all the elements of the infinite sequence in one step.

To achieve this goal, we add to definition of the bisimulation construction (I) another rule called: big step.

Definition 6 (Bisimulation construction (II))
The bisimulation construction (II) is defined with the rules of the definition of bisimulation construction (I) and with the following one:

- for all $(E_1 \triangleright rx_1[a_1], E_2 \triangleright rx_2[a_2]) \in \beta_k$ (big step)

  if $(x_1[n] \rightarrow [n > a_1']m(\rho_1 x_1'[n + c_1]) \in E_1$ and $(x_2[n] \rightarrow [n > a_2']m(\rho_1 x_2'[n + c_2]) \in E_2$ and $a_1 > a_1'$ and $a_2 > a_2'$

  then $(E_1 \triangleright rx_1[(c_1 \times n) + a_1], E_2 \triangleright rx_1[(c_2 \times n) + a_2]) \in \beta_{k+1}$ for $n > 0$

Let us apply this last rule to our preceeding example. We have $\beta_0 = \{(!B1[0], !B2[0])\}$. Both types have the same set of capabilities $\{!\}$. Both corresponding parametric types can perform a put action for every value of their parameter greater

than 0. So $\beta_1 = \beta_0 \cup \{(!B1[1 \times n + 0], !B2[2 \times n + 0]) \text{ for } n > 0\}$. It is easy to see that the relation $B_1 = \{(!B1[n], !B2[2n]) \text{ for } n \geq 0\}$ is a fixed-point and thus a bisimulation relation.

Remark. what we are saying here is that the sequence $\beta_0, \beta_1, \ldots, \beta_n$, where β_n is a fixed-point, is finite. Of course, each of the β_i is possibly infinite. For a full mechanization of the bisimulation construction, one will have to deal with operations on infinite sets. It is easy to have a finite representation of sets obtained by our construction rules since all the constraints on states are linear. This method can be applied similarly to the subtyping relation construction.

STATIC SEMANTICS

Definitions and notations

In order to facilitate the expression of the typing rules, we introduce the notion of constrained behavior types. A constrained behavior type is a special form of parametric behavior types noted $E \triangleright \rho x[\text{inf sup}[$ where inf and sup are the parameter bounds: inf $\leq n <$ sup, n representing the parameter of the (constrained) behavior type, $inf \in \mathcal{N}$ and $sup \in \mathcal{N} \cup \{\infty\}$.

Remark. Note that parametric behavior types and (instantiated) behavior type are special cases of constraint types: $E \triangleright \rho x[a \ \ a+1[$ is the behavior type $E \triangleright \rho x[a]$ and $E \triangleright \rho x[0 \ \infty[$ is the parametric behavior type $E \triangleright \rho x[n]$

A constrained behavior type can perform an action if its corresponding behavior type can perform this action for all the values between inf and sup.

$$\frac{\begin{array}{l}(x[n] \rightarrow [\text{pred}]m(\rho_1 x_1[a_1], \cdots, \rho_n x_n[a_n]); y[f]) \in E \\ n \in [\text{inf sup}[\ \Rightarrow pred(n) = \text{TRUE}\end{array}}{E \triangleright rx[\text{inf sup}[\xrightarrow{\rho m(E \triangleright \rho_1 x_1[a_1], \cdots, E \triangleright \rho_n x_n[a_n])} E \triangleright ry[f(\text{inf}) \ (\text{sup})[} \text{ with } \rho \subset r$$

For example ?Buffer[0 ∞[cannot perform a *get* action whereas ?Buffer[1 ∞[does.

The restriction of a constrained behavior type by a guard is defined by the following equations and is undefined otherwise:

$$\begin{aligned} rx[inf \ sup[\backslash[n > c] &= rx[c \ sup[\text{ if } c < sup \\ rx[inf \ sup[\backslash[n < c] &= rx[inf \ c[\text{ if } c > inf \end{aligned}$$

We define also $possible(X)$ as the set of all the method names that can be performed by any restriction of X.

$possible(rx[inf \ sup[) = \{m \in Meth, \exists a \in [inf \ sup[, \rho \in r\tilde{X}, X' | rx[a] \xrightarrow{\rho m(\tilde{X})} X'\}$

An interface type is then a pair noted μX where μ is the interface mode and X its constrained behavior type. The meta-variable T will range over interface types. A typing context Γ is a list of bindings of the form: $u:T$ or $u:(\tilde{T})$. The set of interface names appearing in a context Γ is called its domain, noted: $Dom(\Gamma)$. The context extension, noted $\Gamma, u:T$, is defined such that $\Gamma, u:T \vdash u:T$. The static semantics is given using the following judgments:

judgment	meaning
$\Gamma \vdash u:T$	in the context Γ the interface u has type T
$\Gamma \vdash u:(\tilde{T})$	in the context Γ the object A has type $(\tilde{T})$
$\Gamma \vdash B$	in the context Γ the behavior B is well typed
$\Gamma \vdash C$	in the context Γ the configuration C is well typed

In order to check the non-duplication of server roles and private client roles, we define a partial function, noted $T_1 \oplus T_2$ over interface types such that a non-duplicable role cannot be present in both T_1 and T_2.

Definition 7 (Interface type addition)
The interface type addition is defined by the following equations and is undefined otherwise:

$$\oplus \begin{array}{l}(\text{public } r_1 x[inf\,sup[) \\ (\text{public } r_2 x[inf\,sup[)\end{array} = \text{public } (r_1 \cup r_2)x[inf\,sup[\text{ if } ? \notin (r_1 \cap r_2)$$

$$\oplus \begin{array}{l}(\text{private } r_1 x[inf\,sup[) \\ (\text{private } r_2 x[inf\,sup[)\end{array} = \text{private } (r_1 \cup r_2)x[inf\,sup[\text{ if } (r_1 \cap r_2) = \emptyset$$

We extend $\oplus$ to contexts as follows: $(\Gamma, u:T_1) \oplus (u:T_2)$ denotes the context $\Gamma, u:(T_1 \oplus T_2)$. We will write $\Gamma, u:\mu r x[inf\ sup[\oplus r' u$ as a short hand for $\Gamma, u:\mu r x[inf\ sup[\oplus \mu' r' x[inf\ sup[$.

Definition 8 (Subtyping over interface types)
An interface type $\mu r x[inf\,sup[$ is a subtype of an interface type $\mu' r' x'[inf'\,sup'[$ if $\mu = \mu'$ and

- if $inf = sup + 1$ then $inf' = sup' + 1$ and $rx[inf] \preceq rx[inf']$
- if $inf = 0$ and $sup = \infty$ then $inf' = 0$, $sup' = \infty, r = r'$ and $x = x'$.

Typing rules

The basic idea underlying our typing rules is to guarantee that each object use the interfaces in a way compatible with their declared behavior type. Our rules ensure also that there is no duplication of the roles of the private interfaces.

$$\frac{\Gamma \vdash u{:}\mu rx[inf\ sup[\Rightarrow? \notin r}{\Gamma \vdash 0}$$

this rule introduces the notion of receiving obligation. If an object still has the server role of an interface then it cannot stop.

$$\frac{\begin{array}{l}\Gamma, u{:}T_2 \vdash B \\ T_1 \xrightarrow{!m(\tilde{T})} T_2 \\ \Gamma \vdash \tilde{v}{:}\tilde{T}' \\ \tilde{T}' \preceq \tilde{T} \\ \Gamma \oplus \tilde{\rho}\tilde{v} \text{ defined}\end{array}}{\Gamma \oplus \tilde{\rho}\tilde{v}, u{:}T_1 \vdash !u.m(\tilde{\rho}\tilde{v}) > B}$$

This rule ensures that the type of u allows an m action and that the emitter cannot use anymore in B the non-duplicable roles it has sent.

$$\frac{\begin{array}{l}\Gamma, u{:}T_1', \tilde{v_1}{:}\tilde{T_1} \vdash B_1 \ldots \Gamma, u{:}T_n', \tilde{v_n}{:}\tilde{T_n} \vdash B_n \\ T \setminus Guard_1 \xrightarrow{?m_1(\tilde{T_1})} T_1' \ldots T \setminus Guard_n \xrightarrow{?m_n(\tilde{T_n})} T_n' \\ possible(T) = \{m_1, \ldots, m_n\}\end{array}}{\Gamma, u{:}T \vdash ?u[[Guard_1]m_1(\tilde{v_1}{:}\tilde{T_1}) = B_1 \cdots [Guard_n]m_n(\tilde{v_n}{:}\tilde{T_n}) = B_n]}$$

In this rule it is important that $possible() = \{m_1, \ldots, m_n\}$. This ensures that all the messages that can be processed by an interface of type T are handled by the reception action. The *possible* function defined on constrained behavior types is extended to interface types.

$$\frac{\Gamma \vdash Recep_1 \ldots \Gamma \vdash Recep_n}{\Gamma \vdash \sum_{i=1}^{n} Recep_i}$$

In a multiple interfaces' reception, the typing context is propagated as it is in all the branches of the choice.

$$\frac{\begin{array}{c}\Gamma, u{:}T \setminus Guard \vdash B_1 \\ \Gamma, u{:}T \setminus \neg Guard \vdash B_2\end{array}}{\Gamma, u{:}T \vdash \text{ if } Guard \text{ then } B_1 \text{ else } B_2}$$

The Guard constraint is propagated in the then branch whereas its negation is propagated in the else branch

$$\frac{\Gamma, u{:}\mu\, !?x[0\ \infty[\vdash B}{\Gamma \vdash \text{ new } u{:}\mu\, x[n] \text{ in } B} \qquad \frac{\Gamma, u{:}\mu\, !?x[a\ a+1[\vdash B}{\Gamma \vdash \text{ new } u{:}\mu\, x[a] \text{ in } C}$$

A newly created interface has both roles: client and server. If its type is parametric then it should be unconstrained ($n \in [0\ \infty[$).

$$\frac{\Gamma, A{:}(\tilde{T}), \tilde{u}{:}\tilde{T} \vdash B}{\Gamma, A{:}(\tilde{T}) \vdash A[\tilde{u}{:}\tilde{T}] = B}$$

For an object declaration, we simply check that the object behavior is well-typed under the assumption that its interfaces has their declared type.

$$\frac{\begin{array}{l}\Gamma \vdash A{:}(\tilde{T}) \\ \Gamma \vdash \tilde{u}{:}\tilde{T}' \\ \tilde{T}' \preceq \tilde{T}\end{array}}{\Gamma \vdash A[\tilde{\rho}\tilde{u}]}$$

An interface having a subtype of another can replace it in an object instantiation.

$$\frac{\begin{array}{l}\Gamma \vdash B \Gamma \vdash A{:}(\tilde{T}) \\ \Gamma \vdash \tilde{u}{:}\tilde{T}' \\ \tilde{T}' \preceq \tilde{T} \\ \Gamma \oplus \tilde{\rho}\tilde{u}\end{array}}{\Gamma \oplus \tilde{\rho}\tilde{u} \vdash A[\tilde{\rho}\tilde{u}] > B}$$

This is the same case as the precedent except that here we must be careful about how B is going to use the interfaces $\tilde{u}$.

$$\frac{\begin{array}{l}\Gamma_1 \vdash C_1 \\ \Gamma_2 \vdash C_2 \\ \Gamma_1 \oplus \Gamma_2 \text{ defined}\end{array}}{\Gamma_1 \oplus \Gamma_2 \vdash C_1 | C_2}$$

Here again we must be sure that there is no duplication of the roles of a client interface. We must be sure also that there is only one server for a given interface.

DYNAMIC SEMANTICS

We present the operational semantics of configurations in two steps. We first define a structural congruence relation and then we give a reduction relation that specifies how the configurations evolve. To distinguish between the interactions on public and private interfaces, the emitting actions on public interfaces will be noted !! whereas the emitting actions on private interfaces will remain noted !.

Structural Congruence

Let us briefly define our scoping rules. An interface u if bound in a behavior B if it appears in an object declaration ($A[\ldots, u, \ldots] = B$) or if it appears in the scope of a new operator (new u in B) otherwise it is free. The notation C[v/u] denotes the substitution of all the free occurrences of u by v in C.

The equations defining the structural congruence are given in table 2.

$$Recep_1 + Recep_2 \equiv Recep_2 + Recep_1$$
$$(Recep_1 + Recep_2) + Recep_3 \equiv Recep_1 + (Recep_2 + Recep_3)$$
$$C_1|C_2 \equiv C_2|C_1\ ,\ (C_1|C_2)|C_3 \equiv C_1|(C_2|C_3)\ ,\ C|0 \equiv C$$
$$\text{new } u_1 \text{ in } (\text{new } u_2 \text{ in } C) \equiv \text{new } u_2 \text{ in } (\text{new } u_1 \text{ in } C)$$
$$(\text{new } u \text{ in } C_1)|C_2 \equiv \text{new } u \text{ in } (C_1|C_2) \quad \text{if } u \text{ is not free in } C_2$$
$$\text{new } u \text{ in } C_1 \equiv \text{new } v \text{ in } C_2 \text{ if } C_1[w/u] = C_2[w/v] \text{ for some fresh interface name } w$$

Table 2 Structural Congruence

Reduction Rules

We define now the reduction rules that specify how a configuration can evolve by making a single and atomic step. Our reduction rules are annotated with information that will help us maintaining the private interfaces' types during the computation. The evolution of configurations may generate messages. The syntax of a message is similar to the syntax of the method invocation except that the message has no continuation. To avoid any ambiguity messages will be written between brackets: $[!u.m(\tilde{v})]$.

The reduction relation is defined by the following rules.

The synchronization on a private interface is by rendez-vous:

$$?u[m(\tilde{u}{:}\tilde{T}) > B + \Sigma m_i(\tilde{u_i}{:}\tilde{T_i}) > B_i]|!u.m(\tilde{v}) > B' \xrightarrow{u,m} B[\tilde{v}/\tilde{u}]|B'$$

The invocation of a public interface generates a message ($[!u.m(\tilde{v})]$) whose behavior is to synchronize with this interface:

$$!!u.m(\tilde{v}) > B \longrightarrow B|[!u.m(\tilde{v})]$$

A message is absorbed by the appropriate (public) interface and then vanishes:

$$?u[m(\tilde{u}{:}\tilde{T}) > B + \Sigma m_i(\tilde{u_i}{:}\tilde{T_i}) > B_i]|[!u.m(\tilde{v})] \longrightarrow B[\tilde{v}/\tilde{u}]$$

As the interface type information are maintained during the computation, the evaluation of the guard is straightforward:

$$\frac{eval(Guard) = TRUE}{\text{if } Guard \text{ then } B_1 \text{ else } B_2 \longrightarrow B_1} \qquad \frac{eval(Guard) = FALSE}{\text{if } Guard \text{ then } B_1 \text{ else } B_2 \longrightarrow B_2}$$

We simply replace the instantiation of the object by the corresponding behavior:

$$\frac{A[\tilde{u}{:}\tilde{T}] \stackrel{Dcl}{=} B}{A[\tilde{v}] \longrightarrow B[\tilde{v}/\tilde{u}]}$$

The created object runs is parallel with the continuation of the creating behavior:

$$\frac{A[\tilde{u}{:}\tilde{T}] \stackrel{Dcl}{=} B}{\text{create } A[\tilde{v}] > B' \longrightarrow B[\tilde{v}/\tilde{u}]|B'}$$

This rule states that if a sub-configuration can evolve to a new one then the whole configuration can evolve too:

$$\frac{C_1 \xrightarrow{u,m} C_1'}{C|C_1 \xrightarrow{u,m} C|C_1'}$$

When there is an interaction on a private interface we update its type:

$$\frac{\begin{array}{c} C \xrightarrow{u,m} C' \\ T \xrightarrow{\rho m(\tilde{T})} T' \end{array}}{\text{new } u{:}T \text{ in } C \longrightarrow \text{new } u{:}T' \text{ in } C'} \text{ with } \rho \in \{!, ?\}$$

The type of the variable u is not affected by an interaction on an other interface u:

$$\frac{C \xrightarrow{v,m} C'}{\text{new } u{:}T \text{ in } C \xrightarrow{v,m} \text{new } u{:}T \text{ in } C'} \text{ with } v \neq u$$

A non annotated reduction does not affect the new operator:

$$\frac{C \longrightarrow C'}{\text{new } u{:}T \text{ in } C \longrightarrow \text{new } u{:}T \text{ in } C'}$$

The following rule states that configurations that are equivalent (according to $\equiv$) behave equally:

$$\frac{C_1 \equiv C_1' \xrightarrow{u,m} C_2' \equiv C_2}{C_1 \xrightarrow{u,m} C_2}$$

Run-time safety

We obtain for the infinite type system of OL the same results as for the finite type system of COB [12]. The static and dynamic semantics we have defined ensure the run-time safety of well-typed OL programs. We consider here only well-typed closed configurations ; i.e., well typed-under in the empty context ($\emptyset \vdash C$). this means that all the interfaces of the configuration have been introduced by the new operator and thus both the client and server roles exist in the configuration.

Theorem 1 (Subject reduction) If $\Gamma \vdash C$ and $C \xrightarrow{u,m} C'$ then there exists a context Γ' such that $\Gamma' \vdash C'$

The careful use of the roles of the interface and the definition of the $\oplus$ function that ensures that the roles of private interfaces are not duplicated ensures that

the client and server role of private a interface evolve concomitantly. As the public interface roles are uniform, the typing is preserved by the reduction relation.

Theorem 2 (Run-time safety)
A well-typed closed configuration ($\emptyset \vdash C$) contains no immediate possibility for communication failure.

This property follows directly from the preceeding theorem as in a well typed configuration both client and server types of an interface can perform the same actions.

CONCLUSION AND RELATED WORK

We defined a calculus endowed with a typing system that guarantees a run-time safety property in well typed object configurations. The type system of OL describes interfaces that may offer non-uniform services. We defined a semantic of behavior types based on CCS process semantics ([9]). We also defined equivalence and subtyping relations based on bisimulation and simulation relations ([9]). Although behavior types can have an infinite number of states we showed how the bisimilarity (the principle is the same of the simulation relation) of two behavior types is checked. There is a lot of work done in the field of the verification of infinite state systems. Our work can be compared to to be the work of Sergio Yovine ([24]) and Colin Stirling et al. ([19, 20]). In [24], the author provide a mechanisms for verifying modal properties of timed automata. Timed automata is a typical case of infinite state system verification. Our types use the same kind of constraints (guards) used for the clocks of timed automata. The proof method presented in [24]is based on analytic tableau. In [19], the authors give the proof that "bisimulation equivalence is decidable for all context-free processes" and in [20] the author does the same for normed pushdown processes. We did not formally study the expressiveness of our type system, but it seems that it includes the context free languages. Even more, our types include non context free types like the unbounded buffer ($put^n get^m$ with $n > m$). We believe that our algorithm for checking bisimilarity is simpler and more tractable.

Type systems for concurrent object oriented languages is an active research topic. Many authors have tackled this issue in the realm of the π-calculus [11] and the actors [1] paradigms. Concerning the latter, a wide variety of typing systems have been proposed that deal with the problem of channel typing. The simplest one [10] just checks the arity of the channels. This type system has been extended such that it can handle polymorphism and type inference [4, 23, 22] and subtyping [15, 16]. None of these typing systems handle dynamic service behavior.

The importance of distinguishing public from private interfaces has been identified by [14], but, without giving it a formal treatment. [14] has also introduced the concept of non-uniform service availability and has used traces

to specify the constraints on the ordering of the messages that can be handled by a channel (an interface).

A lot of ongoing work is about type systems for parallel/distributed object languages and calculi (see [21, 17, 8, 2, 3, 6]). The work reported in [18] is the closest to ours. The authors define types based on graphs and an equivalence relation based on bisimilarity of types. We believe that unlike the type system presented in the present paper, the type system defined in [18] is not able to represent "infinite types" like the unbounded buffer type, for example. In addition the ability to use parametric interface type and to test their state in the behavior of OL objects allows us to write very flexible code without sacrificing the safety.

The technical treatment of the contexts in the static semantic and the reduction rule of OL have been inspired from [7]. In this version of the π-calculus the authors use the linear capabilities of some special channel to ensure that they are used (at most) once. We use a similar mechanism to ensure that there is no duplication of the roles of private interfaces.

References

[1] G. A. Agha, I. A. Mason, S. F. Smith and C. L. Talcott, A Foundation for Actor Computation, J. Functional Programming 1 (1), 1993.

[2] Gérard Boudol. Typing the use of resources in a concurrent calculus. ASIAN'97, the Asian Computing Science Conference, Kathmandu, Nepal, LNCS 1345 (1997) 239-253.

[3] Colaço, Pantel, and Sallé. A set constraint-based analyses of actors in proceedings of Second conference on Formal Methods for Object-based Open Systems, Chapman and Hall, 1997.

[4] Simon J. Gay. A sort inference algorithm for the polyadic π-calculus. Twentieth ACM Symposium on Principles of Programming Languages, January 1993.

[5] Kohei Honda. Types for Dyadic Interaction. CONCUR'93, LNCS 612, Springer-Verlag.

[6] Kohei Honda, Vasco T. Vasconcelos and Makoto Kubo. Language primitives and type disciplines for structured communication-based programming ESOP'98, LNCS 1381, Springer-Verlag.

[7] Naoki Kobayashi, Benjamin C. Pierce, David N. Turner. Linearity and the Pi-Calculus. Technical report, Department of Information Science, University of Tokyo and Computer Laboratory, University of Cambridge, 1995.

[8] Naoki Kobayashi A Partially Deadlock-free Typed Process Calculus Twelfth IEEE Symposium on Logic in Computer Science (LICS'97).

[9] Robin Milner. Communication and Concurrency. Prentice-Hall, 1989.

[10] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK, October 1991.

[11] Robin Milner, Joachim Parrow, David Walker. A calculus of mobile processes (Part I and Part II). Information and Computation, 100:1-77, 1992.

[12] E. Najm, A. Nimour A Calculus of Object Bindings. in proceedings of Second conference on Formal Methods for Object-based Open Systems, Chapman and Hall, 1997.

[13] Elie Najm, Jean-Bernard Stefani. A formal semantics for the ODP computational model. Computer Networks and ISDN Systems , Vol 27, 1995.

[14] Oscar Nierstrasz. Regular Types for Active Objects. Object-Oriented Software Composition. O. Oscar Nierstrasz, D.Tsichitzis (Ed.), Prentice Hall, 1995

[15] Benjamin C. Pierce, David Sangiorgi. Typing and subtyping for mobile process. Mathematical Structures in Computer Science, 1995.

[16] Benjamin C. Pierce, David N. Turner. PICT Language Definition. Available electronically, 1995.

[17] Franz Puntigam. Types for active objects based on Trace Semantics. FMOODS'96, Chapmann and Hall.

[18] Antsnio Ravara, Vasco T. Vasconcelos. Behavioural Types for a Calculus of Concurrent Objects. Euro-Par'97, LNCS. Springer-Verlag, 1997.

[19] Soren Christensen, Hans Hüttel and Colin Stirling. Bisimulation Equivalence is Decidable for all Context-Free Processes LFCS report ECS-LFCS-92-218, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK.

[20] Colin Stirling. Decidability of Bisimulation Equivalence for Normed Pushdown Processes LFCS report LFCS report ECS-LFCS-97-352, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, UK.

[21] Kaku Takeuchi, Kohei Honda, Makoto Kubo. An Interaction-based Language and its Typing System. PARLE'94, LNCS 818, Sringer-Verlag.

[22] David N. Turner. The π-calculus: Types, polymorphism and implementation. Ph.D. Thesis, LFCS, University of Edinburgh, 1995.

[23] Vasco T. Vasconcelos, Kohei Honda. Principal typing schemes in a polyadic π-calculus. CONCUR'93, July 1993.

[24] Sergio Yovine. Méthodes et Outils pour la Vérification Symbolique de Systèmes Temporisés. Ph. D thesis, IMAG Grenoble-France, March 1992.