GUARANTEEING LIVENESS IN AN OBJECT CALCULUS THROUGH BEHAVIORAL TYPING

Elie Najm [a,1], Abdelkrim Nimour [b], Jean-Bernard Stefani [c,1]

[a]Ecole Nationale Superieure des Telecomunications
[b]Schlumberger - Test & Transactions - Smart Cards Products
[c]France Telecom-CNET

[a]Elie.Najm@enst.fr, [b]knimour@slb.com, [c]jeanbernard.stefani@cnet.francetelecom.fr

Abstract: A typing discipline for concurrent objects is defined. A decidable type system is constructed which features a liveness property of well-typed object configurations: all pending request messages are treated within a finite number of steps (modulo fairness). The type system is exhibited on *OL2*, a variant of the asynchronous π-calculus. The present paper builds on previous work. It extends the safety results of [11]. It also provides for the incremental extension of well-typed (and well-behaved) configrations: when the code of an object is type checked in the context of an interface repository, it can be added a existing configuration while preserving the liveness property.

INTRODUCTION

We are concerned with guaranteeing liveness properties in configurations of objects. We advocate a now well-established approach based on behavioral typing. The aim is to define a typing discipline and an associated type system such that well-typed programs yield well-behaving programs. We base our analysis on an object calculus *OL2*, akin to the asynchronous π-calculus [16, 10] or the actors model [1]. In our setting, the problem we tackle can be stated as follows: considering an *OL2* well-typed closed object configuration C, then C satisfies a liveness property whereby for any configuration C' derived from C, and for any pending message M contained in C', there is configuration C'' derived from C' where M has been absorbed by its target object. Furthermore, we handle configuration extensions. The type system is such that

[1]This work has been partially supported by RNRT project MARVEL

J. Wu et al. (eds.), *Formal Methods for Protocol Engineering and Distributed Systems*

well-typed objects are safely added to running configurations and liveness is preserved in the extended configuration. The paper is structured as follows: section 2 is an informal introduction to *OL2*, section 3 presents the operational semantics of *OL2*, section 4 discusses the syntax and semantics of the interface type system, section 5 presents is devoted to a small example, section 6 introduces the static semantics, the rules of which are given in the appendix, section 7 presents the main results of the paper and section 8 is the conclusion.

THE *OL2* CALCULUS

OL2 is an object-based calculus, i.e., it provides for the description of objects with features such as interfaces, clients, servers, service offers, method invocation, etc. It is now a well-established result that object calculi are based on more foundational theories. Indeed, *OL2* is an avatar of the asynchronous π-calculus with the usual, implementation oriented [16, 4] restrictions: (i) restriction of choice to input-prefix processes; (ii) specialization of gates to input or output; (iii) tagging of messages (and actions) with constant labels. In object terminology, such message labels are called methods names; they act as input selectors. *OL2* can also be seen as extension of the Actors language whereby actors may have multiple addresses.

We start with an informal presentation of the calculus, gradually introducing the different constructs, then we will turn to the formal syntax and semantics. *OL2* describes configurations of objects running in parallel and exchanging messages. Each object offers services on a set of interfaces. Each object may invoke the services of other objects by sending messages to the interfaces providing these services. The syntax of a service offer on an interface of an object is as follows:

$$u[m() = B_1] \qquad \text{(Ex-1)}$$

The behavior of this object is to wait for the message m on the interface u and then proceed as specified in B_1. The invocation of this service m by an object is written:

$$u.m() > B_2 \qquad \text{(Ex-2)}$$

We will so distinguish between two roles that can be possessed by an object for a given interface:

- client role: an object that has the client role of an interface can invoke the services offered by this interface.
- server role: an object that has the server role of an interface can accept messages (and thus it offers services) on this interface.

For instance, the behavior expression of the example Ex-1 describes an object which has the server role on u whereas the one of example Ex-2 has the client role. Note that an object may have both roles for the same interface: a typical case for newly created interfaces. Interface roles may be exchanged between objects. For instance:

$$u.m(!v) > B \qquad \text{(Ex-3)}$$

is the behavior of sending the client role of v to the object holding the server role of u. In a second example:

$$u.m(?v) > B \qquad \text{(Ex-4)}$$

we describe the behavior of sending the server role of v. An object can also send both roles:

$$u.m(!?v) > B \qquad \text{(Ex-5)}$$

The static semantics of *OL2* enforces certain restrictions on passing roles between objects. A general restriction, which is independent of the type of the interfaces, is that server roles can be held by at most one object at a time. Thus, when a server role is sent to another object, the sending object loses this role. In Ex-4, object B is not allowed to offer services on interface v (or else it is not well-typed). There are additionnal restrictions on interfaces that are imposed by their types. An interface type, T, is a tuple (μ, k, ρ, x) where:

- μ is the public/private mode (or mode for short). An interface can be declared public or private. For a private interface, the client role can, at a given instant, be possessed by at most one object, i.e., a sender of a client role of a private interface loses that role (in Ex-3, if v has been declared private then the static semantics ensures that there can be no invocation of v in B). For public interfaces there is no such restriction.
- k is the mobility kind of the interface (or kind for short). An interface can be declared mobile or stabile. For stabile interfaces, the server role cannot be sent. There is no such restriction for mobile interfaces. Ex-4 is not well-typed if v is stabile, and is well-typed otherwise.
- ρ is the role of the interface: client (noted !), server (noted ?) or both (noted !?). The role type of a newly created interface is !?.
- x is the behavior automaton of the interface. It prescribes the allowable succession of messages that the interface can handle. Interface behavior types are discussed in a subsequent section.

An interface creation is given by the syntax:

$$\textbf{new } u\!:\!T \textbf{ in } B$$

the behavior of this expression is to create a new interface u of type T and then to behave like B. An example of interface creation:

$$\textbf{new } u\!:\!\textbf{public mobile } !?x \textbf{ in } B$$

When offering a service on an interface, an object must declare the type of the interfaces that it expects to receive. An example of service offer:

$$u[m(w\!:\!T) = B]$$

The behavior of this expression is to wait, on interface u, for the message m containing an interface of type T. The received interface is bound to w.

In *OL2* the behavior of objects can be named. This is achieved as in the following example (where we consider a behavior parametrized with two interfaces):

$$A[u{:}T_1, v{:}T_2] = B_A$$

An object can then be instantiated by providing the actual parameters. Name instantiation is similar to the *become* operation in Actors. Object creation is another feature of *OL2*. It is different from object instantiation in that the creating object has its proper continuation that runs in parallel with the created object. For instance, the expression:

$$\textbf{create } A[?w_1, !w_2] > B$$

evolves to a new one where A and B run in parallel.

In all the examples we presented until now only one message was handled by reception actions. In the general case, more than one service may be available on an interface and a reception action is written:

$$u[m_1(\tilde{v}_1) = B_1, \ldots, m_n(\tilde{v}_n) = B_n]$$

It is possible for an object to offer services on more than one interface at the same time. This is written :

$$u_1[\ldots] + \ldots + u_n[\ldots]$$

At any given instant, for any given object, all messages that have been sent by other objects and that are waiting to be processed are stored in a collection of buffers. This is represented as follows:

$$\left\langle B \ , \ Q \right\rangle$$

where Q is a set of buffers indexed by the set of server interfaces of object B:

$$\left\langle B \ , \ \{u_1 = q_1 \ ; \ \cdots \ ; \ u_k = q_n\} \right\rangle$$

each of the buffers q_i is a list of messages of the form: $m_1(\tilde{v}_1), \ \cdots \ , \ m_n(\tilde{v}_n)$. Depending on the mode of the interface, the associated list of messages is either unordered (when the interface is public) or ordered (when the interface is private). This reflects the semantics of message transfer between a client and a server interface: the transfer is order preserving (FIFO) in the case of private interfaces and non order preserving (Bag) in the case of public interfaces. Finally, a running configuration of objects, R, is expressed using the commutative and associative operator $|$:

$$\left\langle B_1 \ , \ Q_1 \right\rangle \mid \left\langle B_2 \ , \ Q_2 \right\rangle \mid \ldots \mid \left\langle B_n \ , \ Q_n \right\rangle$$

and interfaces that are internal to a configuration, R, are expressed by the new operator:

$$\textbf{new } u_1 \ \textbf{in} \ \cdots \textbf{new } u_n \ \textbf{in } R$$

$$
\begin{array}{rcl}
T & ::= & \mu\, k\, \rho\, x \\
I & ::= & u{:}T \\
Dcl & ::= & A[\widetilde{I}] = B \\
B & ::= & a > B \;\Big|\; \sum_{i=1}^{n} u_i[Abs_i] \;\Big|\; A[\widetilde{\rho u}] \;\Big|\; \textbf{new } I \textbf{ in } B \;\Big|\; \mathbf{0} \\
a & ::= & u.m(\widetilde{\rho v}) \;\Big|\; \textbf{create } A[\widetilde{\rho u}] \\
Abs & ::= & m_1(\widetilde{I}_1) = B_1, \ldots, m_n(\widetilde{I}_n) = B_n
\end{array}
$$

Terminals	Meaning
A	behavior name
u, v, w	interface names
T	interface type
$m, m_1, \ldots, m_n$	method names
μ	interface mode: **public** or **private**
k	interface kind: **mobile** or **stabile**
x	interface behavior type
ρ	interface role: client (!), server (?) or both (!?)

Summary of the Syntax of *OL2* objects

$$
\begin{array}{rcl}
R & ::= & \langle B \,,\, Q \rangle \\
 & | & R|R \\
 & | & \textbf{new } u \textbf{ in } R \\
Q & ::= & \{u_1 = q_1 \,;\, \cdots \,;\, u_n = q_n\}
\end{array}
$$

Summary of the Syntax of *OL2* running configurations

OPERATIONAL SEMANTICS

We need first to define how names are bound in *OL2*. Variables u, $u_1, \ldots, u_n$ occurring in an object or a running configuration are bound if they occur in a sub-expression having one of the forms: **new** $u : T$ **in** B, **new** u **in** R, $m(u_1, \ldots, u_n) = B$, $A(u_1 : T_1, \ldots, u_n : T_n) = B$; otherwise the occurences are free. The sets $fn(B)$ and $fn(R)$ are defined accordingly, and so is *alpha-conversion*, denoted $\equiv_\alpha$.

We present the operational semantics of object configurations in two steps. We first define a structural congruence relation and then we give a reduction relation that specifies how configurations evolve. *OL2* is statically typed so will omit all the information about types in the dynamic semantics.

Structural Congruence

The first structural rules state that the choice operator between receptions is commutative and associative:

$$\Big\langle u[Abs_1] + v[Abs_2]\ ,\ Q\Big\rangle \equiv \Big\langle v[Abs_2] + u[Abs_1]\ ,\ Q\Big\rangle$$

$$\Big\langle (u[Abs_1] + v[Abs_2]) + w[Abs_3]\ ,\ Q\Big\rangle \equiv \Big\langle u[Abs_1] + (v[Abs_2] + w[Abs_3])\ ,\ Q\Big\rangle$$

The parallel operator is also commutative and associative and 0 is its neutral element:

$$R_1 \mid R_2 \equiv R_2 \mid R_1\ ,\ (R_1 \mid R_2) \mid R_3 \equiv R_1 \mid (R_2 \mid R_3)\ ,\ R \mid 0 \equiv R$$

The order of the introduction of the interfaces is meaningless:

$$\textbf{new}\ u_1\ \textbf{in}\ (\textbf{new}\ u_2\ \textbf{in}\ R) \equiv \textbf{new}\ u_2\ \textbf{in}\ (\textbf{new}\ u_1\ \textbf{in}\ R)$$

The two last rules are the π-calculus scope extrusion and the alpha-conversion:

$$(\textbf{new}\ u\ \textbf{in}\ R_1) \mid R_2 \equiv \textbf{new}\ u\ \textbf{in}\ (R_1 \mid R_2) \quad \text{if } u \text{ is not free in } R_2$$

$$R_1 \equiv R_2 \text{ if } R_1 \equiv_\alpha R_2$$

Reduction Rules

We now define the reduction rules that specify how a configuration can evolve by making a single and atomic step. In our reduction rules we apply, on the buffers of pending messages, the operations *push*, *pop* and *top*. The semantics of these operations depend on the type of the buffer (queue or bag).

The reduction relation is defined by the following rules:

$$\frac{\begin{array}{l} R_1 = \Big\langle B_1\ ,\ \{u = q\} \cup Q_1 \Big\rangle \\ R_2 = \Big\langle u.m(\rho_1 v_1, \cdots, \rho_k v_k) > B_2\ ,\ Q_2 \Big\rangle \\ J = \{j \mid ? \in \rho_j\} \\ Q_2 = \bigcup_{j \in J} \{v_j = q_j\} \cup Q_2' \\ q' = push(m(\rho_1 v_1, \cdots, \rho_k v_k), q) \end{array}}{R_1 \mid R_2 \longrightarrow \Big\langle B_1\ ,\ \{u = q'\} \cup Q_1 \bigcup_{j \in J} \{v_j = q_j\} \Big\rangle \mid \Big\langle B_2\ ,\ Q_2' \Big\rangle}$$

The invocation of the service m of the interface u located in the object R_1 puts this message (m) in the buffer corresponding to this interface in R_1. The buffers associated to the servers that are migrating (q_i) move also from R_2 (the invoker) to R_1 (the invoked). Note that these buffers are transfered to the invoked object R_1 before the message m is absorbed by this invoked object R_1.

$$\frac{\begin{array}{c} Q = \{u = q\} \cup Q' \\ top(q) = m(\widetilde{\rho v}) \\ q' = pop(q) \end{array}}{\left\langle u[m(\widetilde{u}) = B + \Sigma m_i(\widetilde{u}_i) = B_i] \, , \, Q \right\rangle \longrightarrow \left\langle B[\widetilde{v}/\widetilde{u}] \, , \, \{u = q'\} \cup Q' \right\rangle}$$

A reception action consists in consuming the first message of the corresponding queue.

$$\frac{A[\widetilde{u}] \stackrel{Def}{=} B}{\left\langle A[\widetilde{v}] \, , \, Q \right\rangle \longrightarrow \left\langle B[\widetilde{v}/\widetilde{u}] \, , \, Q \right\rangle}$$

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

$$\frac{A[\widetilde{u}] \stackrel{Def}{=} B}{\left\langle \textbf{create } A[\widetilde{v}] > B' \, , \, Q \right\rangle \longrightarrow \left\langle B[\widetilde{v}/\widetilde{u}] \mid B' \, , \, Q \right\rangle}$$

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

$$\frac{R_1 \longrightarrow R_1'}{R \mid R_1 \longrightarrow R \mid R_1'}$$

This rule states that if a subconfiguration can evolve to a new one then the whole configuration can evolve too.

$$\frac{R \longrightarrow R'}{\textbf{new } u \textbf{ in } R \longrightarrow \textbf{new } u \textbf{ in } R'}$$

The reduction rule concerning the interface creation operator (equivalent to the restriction operator of the π-calculus) is straightforward. In a labeled-transition semantics it would have been more complicated.

$$\frac{R_1 \equiv R_1' \longrightarrow R_2' \equiv R_2}{R_1 \longrightarrow R_2}$$

This rule states that configurations that are equivalent (according to $\equiv$) behave equally.

INTERFACE TYPES

Syntax of Interface Behavior Types

A behavior type is a triple (E, x, r), noted $E \triangleright rx$, where E is a finite state automaton describing the behavior of the interface, x is the initial state of the automaton and r is the set of capabilities of the interface ($r \subset \{?, !\}$). Following the definition introduced by Nierstrasz ([14]) we define E by a set of equations of the form $x_i = e_i$ where each x_i is a type variable that appears once and only once in the left-hand side of a equation and each e_i is an expression defined by the following syntax:

$$e ::= \sum_{i=1}^{n} m_i(\widetilde{r}_i\widetilde{x}_i); y_i \text{ where } i \neq j \Rightarrow m_i \neq m_j \text{ for all } i, j \in [1..n]$$

Note that interface behavior types are deterministic. We denote ε the empty summation ($n = 0$). The variables $X, Y, \ldots$ range over the set IBTypes of interface behavior types.

Note. In the following we will consider, without loss of generality[1], that all our behavior types are defined in the same environment, E. We will sometimes omit this environment when writing a behavior type, i.e., $E \triangleright rx$ will be written rx. We also sometimes refer to this environment as a type repository.

Semantics of Interface Behavior Types

As in [11], we define a transition relation that specifies the behavior of a behavior type:

$$\frac{(x = m(r_1x_1, \cdots, r_nx_n); y + e) \in E}{rx \xrightarrow[E]{\rho\, m(r_1x_1, \cdots, r_nx_n)} ry} \text{ for some } \rho \in r$$

Example. Let us consider the memory cell case. We have an environment E defined as follows:

$$\begin{array}{lcl} \text{Free\_cell} & = & write(!x); \text{Cell} \\ \text{Cell} & = & read(!y); \text{Cell} \\ & + & release(); \text{Free\_cell} \end{array}$$

An free cell (server) can perform an action "write" and then behave like a cell: ?Free_cell $\xrightarrow[E]{?write(x)}$?Cell. The Cell can then perform either a "read" action - without changing its state or a "release" action - which puts it in the initial state: ?Cell $\xrightarrow[E]{?release()}$?Empty_cell

We define $Succ(X)$ as the set of all the types that X can evolve to in one step:

Definition 1 $Succ(X) = \{Y \in \text{IBTypes}, \exists\, \rho, m, \widetilde{X} \textit{ such that } X \xrightarrow[E]{\rho\, m(\widetilde{X})} Y\}$

Interface Type Equivalence

Two behavior types are equivalent iff they behave equally, i.e., they are bisimilar:

Definition 2 *(Interface type equivalence)*
Two interface types X_1 and X_2 are equivalent, noted $X_1 \sim X_2$ iff:

i) $X_1 \xrightarrow[E]{\rho\, m(\widetilde{Y_1})} X_1'$ *implies* $X_2 \xrightarrow[E]{\rho\, m(\widetilde{Y_2})} X_2'$ *and* $X_1' \sim X_2'$, $\widetilde{Y_1} \sim \widetilde{Y_2}$

ii) $X_2 \xrightarrow[E]{\rho\, m(\widetilde{Y_2})} X_2'$ *implies* $X_1 \xrightarrow[E]{\rho\, m(\widetilde{Y_1})} X_1'$ *and* $X_2' \sim X_1'$, $\widetilde{Y_2} \sim \widetilde{Y_1}$

[1]Two behavior types defined in two distinct environments can be simply redefined in the union of these environments with an appropriate renaming of type variables.

Although this definition is recursive, it is well-formed.
A behavior type is uniform if it has always the same behavior.

Definition 3 *X is uniform iff:*

$$Y \in Succ(X) \Rightarrow X \sim Y \text{ and } Y \text{ is uniform.}$$

In *OL2* all the interfaces declared public are checked by the static semantics to have this property which ensures that all pending messages are (or will be) processable, although they may come from different clients and in an arbitrary order.

Interface behavior subtyping

Informally, an interface type X_1 is a subtype of a type X_2 iff:

client case: the processable messages of X_1 are a subset of the processable messages of X_2.

server case: the processable messages of X_1 are a superset of the processable messages of X_2.

This is to guaranty the safe substitution of a supertype by its subtype.

Definition 4 *(Interface subtyping)*
An interface type X_1 is a subtype of an interface type X_2, noted $X_1 \preceq X_2$ iff:

i) $X_1 \xrightarrow[E]{!m(\widetilde{Y_1})} X_1'$ *implies* $X_2 \xrightarrow[E]{!m(\widetilde{Y_2})} X_2'$ *and* $X_1' \preceq X_2'$, $\widetilde{Y_1} \preceq \widetilde{Y_2}$

ii) $X_2 \xrightarrow[E]{?m(\widetilde{Y_2})} X_2'$ *implies* $X_1 \xrightarrow[E]{?m(\widetilde{Y_1})} X_1'$ *and* $X_1' \preceq X_2'$, $\widetilde{Y_1} \preceq \widetilde{Y_2}$

Note that if X_1 and X_2 have both client and server roles then they are equivalent.

EXAMPLE

We present a small buffer example intended to show how interface types can be defined and used in a configuration of objects. We demonstrate also how coordination can be modeled using private interfaces.

Our initial step is to declare the interface type for a one place buffer that stores elements (interfaces) of type `elem`:

```
x\_buff = put(!x\_elem)     ; x\_full
x\_full = get(!x\_r\_elem)  ; x\_buff
```

Interfaces of type `x_r_elem` return elements of type `elem`:

```
x_r_elem = ret(!x_elem) ; 0
```

A buffer and a client

An *OL2* object that manages a one place buffer can be now written:

```
Buffer[buf: private stabile ?x_buff] =

  buf [ put(e: private stabile !x_elem) =
                                Full_Buffer[?buf, !e]
      ]
```

This object starts with an initial server interface `buf` which is ready to accept a `put` method. Method `put` takes an argument `e` which is the client role of an interface of type `x_elem`. After accepting method `put` the object becomes a `Full_Buffer` which has the continuation of the server role for `buf` and a client role for interface `e`. `Full_Buffer` is ready for method `get` which takes an interface `r` that is used as a target for returning the value `e`:

```
Full_Buffer[buf: private stabile ?x_full,
               e: private stabile !x_elem
           ] =
              buf [ get(r: private stabile !r_x_elem) =
                                r.ret(!e) > Buffer[?buf]
                  ]
```

Note how, at the initial state of `Full_Buffer`, the behavior type of interface `buf` is `?x_full`. We can now define a client that uses our buffer:

```
Buffer_Client [ buf: private stabile !x_buff,
                   e: private stabile !x_elem
                ]=
                 buf.put(!e) >
                 new r: private stabile !? r_x_elem in
                 buf.get(!r) >
                 r [ ret(e: private stabile !x_elem) =
                     Buffer_Client[!buf, !e]
                   ]
```

Object `Buffer_Client` is parametrized with the (client) interface of the buffer and the (client) interface of the element that is to be stored in the buffer. It starts by invoking method `put` and then it creates a new interface that is used in the argument of the invocation of method `get`.

A buffer and two clients

We turn now to modelling two coordinated clients for our buffer: a producer and a consumer. We will take advantage of the private interface `buf` to synchronize these

two objects: the client role for `buf` is passed between the two objects - the producer after storing an element in the buffer, and the consumer after getting this element. The `Producer` object is as follows:

```
Producer [ buf: private stabile !x_buff,
             e: private stabile !x_elem,
          prod: private stabile ?x_empty,
          cons: private stabile !x_full
         ]=
           buf.put(!buf) >
           prod [ token(buf: private stabile !x_buff) =
                  Producer[!buf,!e,?prod,!cons]
                ]
```

and the `Consumer` object is given hereafter:

```
Consumer [ cons: private stabile ?x_full,
           prod: private stabile !x_empty
         ]=
           cons [ token(buf:private stabile !x_full) =
                      new r: private stabile !? r_x_elem in
                      buf.get(!r) >
                      r [ ret(e: private stabile !x_elem) =
                          prod.token(!buf) >
                          Consumer[?cons, !prod]
                        ]
                ]
```

The previous objects make use of interface behavior types `x_full` and `x_empty` which are simply defined by the following equations:

```
x_full = token(!x_full) ; x_empty
x_empty = token(!x_buff) ; x_full
```

The running configuration for our three objects can now be given:

new `prod` **in new** `cons` **in new** `buf` **in new** `e` **in**

$$\langle \texttt{Buffer[?buf]} , \emptyset \rangle$$

$$\langle \texttt{Producer[!buf,!e,!prod,?cons]} , \emptyset \rangle \mid \langle \texttt{Consumer[?cons, !prod]} , \emptyset \rangle$$

STATIC SEMANTICS

The type system that we propose enforces the following properties:

- interfaces that are declared to be private are privately used. That means that when a role (either client or server) is sent, it can no more be used by the sender.
- the service offered by public interfaces is uniform.
- the server role of stabile interfaces do not migrate.
- server roles of mobile interfaces migrate only towards stabile interfaces.
- objects behave according to the behavior types of their interfaces.
- when at a reception state objects are ready to receive on all of their interfaces.

It is the conjunction of these properties that ensures the liveness result stated in a subsequent. Let us consider, for instance, the tricky deadlock where a server interface is stuck in a buffer because the target server interface is stuck also in a buffer. This can occur in the configuration, R:

$$\left\langle u.m(?u') > B \ , \ \emptyset \right\rangle \mid \left\langle u'.m'(?u) > B' \ , \ \emptyset \right\rangle$$

which will evolve to the deadlocked configuration, R':

$$\left\langle B \ , \ \{u' = m'(?u)\} \right\rangle \mid \left\langle B' \ , \ \{u = m(?u')\} \right\rangle$$

Thanks to the typing rules, configuration R is not well typed, and thus the deadlock is avoided. The static semantics rules are given in the annex.

RUN-TIME LIVENESS

In the present section we only consider those configurations where all objects possess the so called input-well-guarded recursion, i.e., where recursive definitions of objects must go through at least one receiving action. In fact, this property could be dealt with in the type system. Furthermore, we consider closed configurations, i.e., configurations with no free interfaces.

Theorem 1 *(Subject reduction)*
Let R and R' be object configurations. If $\Gamma \vdash R$ and $R \longrightarrow R'$ then there exists an environment Γ' such that $\Gamma' \vdash R'$

Outline of the proof: It is an extension of the subject reduction property introduced in [11]. This result is essentially due to the following invariants that are ensured by the typing rules:

- The services available at the public servers are uniform.
- There is no duplication of server roles (public and private) and no duplication of private clients. So a private client interface type and its server interface type will evolve concomitantly.

Theorem 2 *(Liveness)*
Let R_1 be a well-typed closed configuration ($\emptyset \vdash R_1$). For all configurations, R_2, such that $R_1 \xrightarrow{*} R_2$, if R_2 contains a non-empty buffer for a server interface u ($u = q$) then there exits a configuration R_3 such that $R_2 \xrightarrow{*} R_3$ where the message top(q) has been consumed.

Outline of the proof: By induction on the length of the derivation $R_1 \xrightarrow{*} R_2$. This result is due to the conjunction of the static semantic rules:

- rule 1 enforces progress as prescribed by the interface behavior type,
- rule 5 enforces that all server interfaces are opened when the object is in a ready to receive state,
- input-well-guardedness guarantees that all objects either come to a stop or come to a ready to receive state within a finite number of steps,
- rule 2 avoids deadlocks due to cross sending of stabile server interfaces: it avoids that servers become indefinitely unavailable when they migrate in buffers towards other migrating server interfaces,

Liveness preserving configuration extension. The liveness theorem may seem restrictive to closed configurations. In fact it is also valid when dealing with configuration extension. Indeed, the theorem implies the nice feature that any well-typed running configuration can be extended at any time with the addition of well-typed objects (which may hold the client role of some public interfaces of the running configuration) and whereby the resulting new configuration possesses the liveness property stated above. This feature follows from a corollary to theorems 1 and 2.

Corollary 1 *(Object introduction)*
Let R_1 be a well-typed closed running configuration possessing the public interface u:

$$\emptyset \vdash \left\langle \textbf{\textit{new}}\ u : (\textbf{\textit{public}}, k, !?, x)\ \textbf{\textit{in}}\ B_1\ ,\ Q \right\rangle$$

Let R_2 be a well-typed initial configuration possessing a client role of the public interface u:

$$u : (\textbf{\textit{public}}, k, !, z) \vdash \left\langle B_2\ ,\ \emptyset \right\rangle \textit{ and } !z \preceq !x$$

Then the extension of the configuration R_1 with the configuration R_2 is also well-typed and so enjoys the liveness property:

$$\emptyset \vdash \left\langle \textbf{\textit{new}}\ u : (\textbf{\textit{public}}, k, !?, x)\ \textbf{\textit{in}}\ (B_1 \mid B_2)\ ,\ Q \right\rangle$$

Liveness preserving extension is a valuable feature which allows the smooth extension of running configurations. Note that the operation of *introducing an object* in a *running configuration* assumes the existence of two repositories, namely, a

type repository and an interface repository. The type repository is a public database defining all behavior type identifiers. The interface repository associates to each public interface of the running configuration its (uniform) type. The interface repository is a database that can be consulted by new objects that are candidates for extending the running configuration (a broker can be used as an intermediate object for accessing the interface type repository, but this is beyond the scope of the present paper). Note that it is sufficient to check that the newly introduced object is well-typed using contextual information from the type and interface repositories. There is no need to re-inspect the code of the objects in the running configuration and to check the well-typedness of the parallel composition of the new enlarged configuration.

CONCLUSION

We defined a calculus endowed with a typing system that guarantees a progress property in well-typed object configurations. The eminent feature of *OL2* resides in the way its interfaces are declared and consistently managed. Thus, interfaces can be either public or private, mobile or stabile, client or server. The discipline enforced on the use of the interfaces is such that, modulo fairness, in all well-typed configurations, all pending messages are eventually processed by their target interfaces. The typing system of *OL2* can be adapted to a wide variety of languages that have the same communication paradigm. In another paper [13], we extended the safety result of a previous work [11] to infinite automata type systems. Besides the liveness guarantees brought by the present type system, another result is achieved which allows for the incremental, liveness preserving, introduction of new well-typed objects to existing configurations. This provides for the extension of configurations the code of which is no more available. Only the interface types are considered. This last property has important implications in open software design.

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 [10] and the actors [1] paradigms. Concerning the former, a wide variety of typing systems have been proposed that deal with the problem of channel typing. The simplest one [9] just checks the arity of the channels. This type system has been extended such that it can handle polymorphism and type inference [5, 23, 21] 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). Our work extends [14] in two ways: by formally introducing the concept of privacy of interfaces and by allowing message types to include parameter types.

Takeuchi et al. [20] define a typed process calculus called $\mathcal{L}$ with the notion of *session*: "a semantically atomic chain of communication actions" between two processes. In *OL2* a session is represented by a communication between two objects using a private interface. But unlike *OL2*, the session channels are static and the roles of the partners of a session cannot be passed. In *OL2*, an object can pass its client or

server role of a private interface and so delegate to another object the continuation of a "session".

In [17] the author provides a typing system which handles the dynamic behavior of actors. [17] features the concept of type spliting allowing the introduction of a form of parallelism in the types. The type language is based on traces, but the possibility of having recursion and non terminating behavior are not explicitly treated.

The work that is the closest to ours is perhaps [8]. It is based on the asynchronous π-calculus and has a typing system that ensures a certain form of deadlock freeness. The main difference between [8] and our approach is that only a restricted form of behaviors, without loops, is allowed in the liveness fragment of the calculus proposed in [8].

TYCO [22, 18] is another interesting endeavor which is worth mentionning. TYCO is a calculus that is built for the purpose of experimenting behavioral type issues. In [19] a type discipline is exhibited on TYCO which is less restrictive than the one presented here but which guarantees weaker properties.

A recent interesting deadlock freeness result was achieved in [3] on a process calculus which is more expressive than *OL2* and which unifies the π and λ calculi. This result, in fact, is more about verifying complex configurations using typing techniques. Our aim is different and we are more concerned with open configurations and issues of extending applications at run time.

The technical treatment of the contexts in the static semantic of *OL2* has 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] R. Amadio and L. Cardelli, *Subtyping recursive types*, ACM-TOPLAS, 15(4):575--631, 1993.

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

[4] Cedric Fournet, Georges Gonthier. *The reflexive chemical abstract machine and the join-calculus.* POPL'96.

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

[6] Kohei Honda. *Types for Dyadic Interaction.* CONCUR'93, LNCS 612, 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. *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.

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

[11] 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.

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

[13] Elie Najm, Abdelkrim Nimour, Jean-Bernard Stefani. *Infinite Types for Distributed Objects Interfaces.* Proceedings of third IFIP conference on Formal Methods for Open Object-based Distributed Systems - FMOODS'99 - Firenze, Italy, February 99. Kluwer.

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

[19] Antonio Ravara, Pedro Resende, and Vasco T. Vasconcelos. *Towards an Algebra of Dynamic Object Types.* In Workshop on Semantics of Objects as Processes, BRICS Notes Series, NS-98-5. 1998.

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

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

[22] Vasco T. Vasconcelos. *Typed Concurrent Objects* ECOOP'94, LNCS 821, Springer Verlag.

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

Appendix

STATIC SEMANTICS RULES

Preliminary definitions and notations

In the treatment of the static semantics we need to associate a type to the interfaces of the objects being considered. A context Γ is a set of bindings of the form: $u : T$ or $A : (T_1, \ldots, T_n)$ where each interface types is defined in a "Type Repository", $\mathbb{TR}$, which is the global set of behavior type equations. The static semantics is given using the judgments given in table 1.

judgment	**meaning**
$\Gamma \vdash u : T$	in the context Γ, the interface u has type T
$\Gamma \vdash A : (T_1, \ldots, T_n)$	in the context Γ, the object A has type $(T_1, \ldots, T_n)$
$\Gamma \vdash \langle B \,,\, Q \rangle$	in the context Γ, the behavior $\langle B \,,\, Q \rangle$ is well-typed
$\Gamma \vdash R$	in the context Γ,the configuration R is well-typed
$\Gamma \vdash Dcl \Rightarrow \Gamma'$	in the context Γ, the declaration Dcl yields context Γ'

Table 1 Judgments of the typing rules

The context extension, noted "$\Gamma, u:T$", is defined such that $\Gamma, u:T \vdash u:T$.

In order to avoid interface server role duplication and private interface client role duplication we define a partial function, $\oplus$, over interface types. The following equations gives the summation ($\oplus$) of two interface types. The function $\oplus$ is undefined in all the other cases.

$$\begin{array}{lcl} (\textbf{public}, k, x, r_1) \oplus (\textbf{public}, k, x, r_2) & = & (\textbf{public}, k, x, r_1 \cup r_2) \text{ if } ? \notin (r_1 \cap r_2) \\ (\textbf{private}, k, x, r_1) \oplus (\textbf{private}, k, x, r_2) & = & (\textbf{private}, k, x, r_1 \cup r_2) \text{ if } (r_1 \cap r_2) = \emptyset \end{array}$$

We extend $\oplus$ to contexts as follows: $(u_1 : T_1, ..., u_n : T_n) \oplus (u_1 : T'_1, ..., u_n : T'_n)$ denotes the context $(u_1 : T_1 \oplus T'_1, ..., u_n : T_n \oplus T'_n)$ We will write $(\Gamma, u : (\mu, k, r, x)) + r'u$ as a shorthand for $\Gamma, u : ((\mu, k, r, x) \oplus (\mu, k, r', x))$

Typing Rules

The basic idea underlying our typing rules is to guaranty that each object uses the interfaces in a way compatible with their declared behavior type. Our rules ensure also that there is no duplication of the roles of interfaces, except for the client role of the public interfaces.

$$\frac{\{u \mid \Gamma \vdash u : (\mu, k, r, x) \text{ with } ? \in r \text{ and } x \neq \varepsilon\} = \emptyset}{\Gamma \vdash \langle \mathbf{0} \,,\, \emptyset \rangle} \qquad (1.A.1)$$

Rule 1 introduces the notion of receiving obligation. If the behavior type of a server can perform an action (is different from ε) then this server cannot stop.

$$\frac{\begin{array}{l}\Gamma,\ u:(\mu,k,r,x_2)\ \vdash\ \left\langle B\ ,\ Q\right\rangle \\ \Gamma\ \vdash\ \tilde{v}:(\tilde{\mu},\tilde{k},\tilde{r},\tilde{y}) \\ rx_1 \xrightarrow[\mathbb{R}]{!m(r_1y_1,\cdots,r_ny_n)} rx_2 \\ \Gamma + \tilde{\rho}\tilde{v} \text{ defined} \\ ? \in r_j \Rightarrow k = \text{stabile, for } j \in [1..n]\end{array}}{\Gamma,\ u:(\mu,k,rx_1) + \tilde{\rho}\tilde{v}\ \vdash\ \left\langle u.m(\tilde{\rho}\tilde{v}) > B\ ,\ Q\right\rangle} \qquad (1.A.2)$$

Rule 2 could be read like this: if $u.m(\tilde{\rho}\tilde{v}) > B$ is well-typed in a context where u has the client role of an interface of type X and where the interfaces v_i have the roles $\{\rho_i\}$ then B is well-typed in a context where the type of u has evolved to Y and where the roles of the interfaces v_i have been updated.

$$\frac{\begin{array}{l}\Gamma, u:(\mu,k,X_1), \tilde{v}_1:(\tilde{\mu}_1,\tilde{k}_1,\tilde{Y}_1)\ \vdash\ \left\langle B_1\ ,\ Q\right\rangle \\ \vdots \\ \Gamma, u:(\mu,k,X_n), \tilde{v}_n:(\tilde{\mu}_n,\tilde{k}_n,\tilde{Y}_n)\ \vdash\ \left\langle B_n\ ,\ Q\right\rangle \\ X \xrightarrow[\mathbb{R}]{\text{messages}(Q,u)^*} X' \\ X' \xrightarrow[\mathbb{R}]{?m_1(\tilde{Y}_1)} X_1 \qquad \cdots \qquad X' \xrightarrow[\mathbb{R}]{?m_n(\tilde{Y}_n)} X_n \\ Succ(X) = \{X_1,\ldots,X_n\}\end{array}}{\Gamma, u:(\mu,k,X')\ \vdash\ \left\langle u?[m_1(\tilde{v}_1:(\tilde{\mu}_1,\tilde{k}_1,\tilde{Y}_1)) = B_1 \ldots m_n(\tilde{v}_n:(\tilde{\mu}_n,\tilde{k}_n,\tilde{Y}_n)) = B_n]\ ,\ Q\right\rangle} \qquad (1.A.3)$$

In rule 3, the condition $X \xrightarrow[\mathbb{R}]{\text{messages}(Q,u)^*} X'$ where $\text{messages}(Q,u)$ is the ordered list of messages for the interface u in the the buffer Q is here to take into account the delay of the server interface comparing to the client interface. This delay is induced by the message buffering at the server. In this rule it is important that $Succ(X) = \{Y_1,\ldots,Y_n\}$. This ensures that all the messages that can be processed by an interface of type X are handled by the reception action. Having the status of all the v_i(s) to *false* ensures that the servers are not immediately sent after their reception.

$$\frac{\Gamma, u:(\mu,k,!?,x)\ \vdash\ R}{\Gamma\ \vdash\ \textbf{new}\ u:(\mu,k,!?,x)\ \textbf{in}\ R} \qquad (1.A.4)$$

Rule 4 states that a newly created interface has both roles: client and server.

$$\frac{\Gamma_1 \vdash u_1[Abs_1] \ldots \Gamma_n \vdash u_n[Abs_n] \quad J = \{j \mid \Gamma \vdash u_j : (\mu, k, r, x) \text{ and } ? \in r\}}{\Gamma_1 \oplus \cdots \oplus \Gamma_n \vdash \sum_{j \in J} u_j[Abs_j]} \tag{1.A.5}$$

Rule 5 ensures that the object opens up for all interfaces for which it possesses the server role.

$$\frac{\Gamma \vdash (A[\tilde{u} : (\tilde{\mu}, \tilde{k}, \tilde{X})] = B) \Rightarrow \Gamma \quad \Gamma \vdash B}{\Gamma \vdash (A[\tilde{u} : (\tilde{\mu}, \tilde{k}, \tilde{X})] = B) \Rightarrow \Gamma, \tilde{u} : (\tilde{\mu}, \tilde{k}, \tilde{X}), A : ((\tilde{\mu}, \tilde{k}, \tilde{X}))} \tag{1.A.6}$$

In rule 6, the first condition is here to allow recursive definition. It ensures that the behavior B is typed in a context that already contains the bindings needed for the instantiation of A.

$$\frac{\Gamma \vdash \tilde{u} : (\tilde{\mu}, \tilde{k}, \tilde{X}) \quad \Gamma \vdash A : ((\tilde{\mu}, \tilde{k}, \tilde{Y})) \quad \tilde{X} \preceq \tilde{Y} \quad \Gamma + \widetilde{\rho u} \text{ defined}}{\Gamma \vdash A[\widetilde{\rho u}]} \tag{1.A.7}$$

Rule 7 states states that an interface having a subtype of another can replace it in an object instantiation.

$$\frac{\Gamma \vdash B \quad \Gamma \vdash \tilde{u} : (\tilde{\mu}, \tilde{k}, \tilde{X}) \quad \Gamma \vdash A : ((\tilde{\mu}, \tilde{k}, \tilde{Y})) \quad \tilde{X} \preceq \tilde{Y} \quad \Gamma + \widetilde{\rho u} \text{ defined}}{\Gamma + \widetilde{\rho u} \vdash A[\widetilde{\rho u}] > B} \tag{1.A.8}$$

Rule 8 is similar to the precedent except that here we must be careful about how B is going to use the interfaces $\tilde{u}$.

$$\frac{\Gamma_1 \vdash R_1 \quad \Gamma_2 \vdash R_2 \quad \Gamma_1 \oplus \Gamma_2 \text{ defined}}{\Gamma_1 \oplus \Gamma_2 \vdash R_1 | R_2} \tag{1.A.9}$$

Finally, rules 9 states that there is no duplication of the roles of a client interface when composing configurations.