Typing Non-Uniform Concurrent Objects

António Ravara
Department of Mathematics
Instituto Superior Técnico
Av. Rovisco Pais 1049–001 Lisboa, Portugal
amar@math.ist.utl.pt

Vasco T. Vasconcelos
Department of Informatics
Faculty of Sciences, University of Lisbon
Campo Grande 1749-016 Lisboa, Portugal
vv@di.fc.ul.pt

Draft, October 14, 1999

Abstract

Concurrent objects offer services non-uniformly, as the availability of a service may depend on the internal state of the object. Types-as-interfaces is an approach not suitable to model non-uniform service availability, and the message-not-understood communication error is, in this setting, very restrictive. We propose a looser definition of communication error, demanding only weak fairness for the reception of messages, by not accounting as errors messages that request methods that may become enabled at some time in the future. This new notion of error detects messages that are never accepted, either because the requested method does not exists at all, or because the object is deadlocked and cannot change its state to accept the request. We formalise non-uniform concurrent objects in TyCO, a name-passing object calculus, and we ensure program safety via a type system. Types are terms of a process algebra which describe dynamic aspects of the behaviour of objects. The type system assigns terms of this algebra to the names of processes, and ensures that typable processes do not run into communication errors.

1 Introduction

Herein we study non-uniform concurrent objects in TyCO [31, 28], an asynchronous name-passing object calculus along the trends of the π-calculus [19, 20]. Concurrent objects can offer services non-uniformly, according to synchronisation constraints, i.e., the availability of a service depends on the state of the object (that reflects the state of the system) [22]. Objects with methods enabled or disabled according to its internal state are very common in object-oriented programming (e.g. a stack, a finite buffer, an FTP server, a bank account, a cash machine). Nevertheless, while the typed λ-calculus is a firm ground to study typing for sequential object-oriented languages, the typing of concurrent objects poses particular problems due to the non-uniform availability of their methods. A static notion of typing, like interfaces-as-types, is not powerful enough to capture dynamic properties of the behaviour of concurrent objects. Whence, we aim at a more flexible style of programming and at a type discipline that copes with non-uniform concurrent objects: the interfaces of the objects only have enabled methods, and when a client asks for a disabled method the message is not rejected if the internal state of the object may change such that the method becomes available. A pair object-message—a redex—is not an error if the message may be accepted at some time in the future. Therefore, a process that is not an error guarantees weak fairness in the reception of messages: a permanently enabled object eventually receives a message which participates in a redex with it. Usually a redex that does not reduce is considered an error. To achieve the objectives previously referred, we provide a looser definition of communication error. We call bad redexes those that do not reduce, and we do not consider them as errors if the object may evolve to a state where it can accept the message. A communication error is a process with a persistent bad redex, either because the object is deadlocked, or because it will never have the right method for the message. Therefore, we adopt a behaviours-as-types approach, such that a type is able to characterise all possible life-cycles of an object, thus providing information about how to interact with it. Types are terms of a process algebra, namely higher-order labelled transition systems that characterise all possible life-cycles of concurrent objects. These types are partial specifications of the behaviour of objects, able to cope with non-uniform service availability. They are also suitable to specify protocols, like pop3, since a type can represent sequences of requests to a server.

Traditional type systems assign rigid interface-like types to the names of the objects [19, 31, 16]. With a more flexible notion of error and with these richer types, we develop a type system that ensures safety properties resulting from the absence of persistent bad redexes, namely the absence of some deadlocks (that we call local), and the guarantee that typable processes will not run into communication errors.

Take the typical example of a one-place buffer that only allows read/consume operations when it is full, and write/produce operations when it is empty. We like to specify it as follows, since it shows clearly the behaviour of the buffer, alternating between write and read operations.

```
def Empty(b) = b?{ write(u) = Full[b,u] }
and Full(b,u) = b?{ read(r) = r!val[u] | Empty[b] }
in ...
```

This style of programming is state-dependent, that is, for each object we specify its different interfaces according to the availability of services. The example presented can easily be scaled up to an n-place finite buffer by introducing a third state, where the buffer is neither empty nor full. Objects usually have a finite (and small) number of such states, thus it is not difficult to program in this style. The referred type systems reject the example above, since name b do not have a single interface containing both labels *write* and *read*. An alternative typable implementation, uses the busy-waiting technique to handle non-available operations.

```
def Buf(b,v,empty) =
      b?{ read(r)  = if empty then b!read[r] | Buf[b,v,true]
                     else r![v] | Buf[b,v,true],
          write(u) = if empty then Buf[b,u,false]
                     else b!write[u] | Buf[b,v,false] }
in ...
```

In the second implementation, a process containing the redex $\mathrm{Buf}[b, v, \mathit{empty}] \mid b!\mathit{read}[r]$ is not an error, and the presence of a message of the form $b!\mathit{write}[u]$ makes possible the acceptance of the *read* message. Similarly, in the first implementation a process containing the redex $\mathrm{Empty}[b] \mid b!\mathit{read}[r]$ should not be considered an error, as again, the presence of a message like $b!\mathit{write}[u]$ also makes the reception of the *read* message possible. Nonetheless, notice that a process locally deadlocked in name b, like $\nu b\,(\mathrm{Empty}[b] \mid b!\mathit{read}[r])$ is an error. In conclusion, the objects behave similarly, but the first implementation is more informative and more natural regarding the buffer's behaviour. For a thorough discussion on non-uniform objects in TyCO see [24].

We have been working on a theory of types able to accommodate this style of programming [26, 25]. In this work we incorporate and develop the idea of types as graphs [26]. Our first proposal of a looser notion of communication error is there, but it did not incorporate deadlock. We propose herein a syntactic view of the graph types, defining a type algebra that provides a description of some dynamic aspects of the behaviour of objects. Our types characterise the semantics of concurrent objects by representing all their possible life-cycles as a state-transition system. Therefore, the types provide an internal view of the objects that inhabit them, which is synchronous, as a transition corresponds to message reception. Hence, the types enjoy the rich algebraic theory of a process algebra, with an operational semantics defined via a labelled transition system, and a notion of equivalence. The equivalence notion is based on a notion of bisimulation that is incomparable with other notions in the literature, to which we define an axiomatic system, complete at least for image-finite types [25].

2 The calculus of objects

TyCO (**Ty**ped **C**oncurrent **O**bjects) is a name-passing calculus featuring asynchronous communication between concurrent objects via labelled messages carrying names. The calculus is an object-oriented extension of the asynchronous π-calculus [13, 6], where the objects behave according to principles of the actor model of concurrent computation [12, 2] (with exception of the uniqueness of actors' names and the fairness assumption).

Syntax. Consider *names* a, b, v, x, y, and *labels* l, m, n, possibly subscripted or primed, such that the set of names is countable and disjoint from the set of labels. Let $\tilde{v}$ stand for a sequence of names, and $\tilde{x}$ for a sequence of pairwise distinct names; moreover, $\vec{a}$ denotes a pair of sequences of names, the first identified by $\tilde{a}_i$, and the second by $\tilde{a}_o$. Assume a countable set of process variables, denoted by X, Y.

Definition 2.1 (Processes).
The set of *processes* is given by the following grammar.

$$P \quad ::= \quad a?M \quad | \quad a!l[\tilde{v}] \quad | \quad P \mid Q \quad | \quad \nu x\, P \quad | \quad (\mathsf{rec}\, X(\vec{x}).a?M)[\vec{v}] \quad | \quad X[\vec{v}] \quad | \quad \mathbf{0}$$

where $M ::= \{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$, with $n \geq 1$.

Objects, of the form $a?M$, and *messages*, of the form $a!l[\tilde{v}]$, are the basic processes in the calculus. An object is an input-guarded labelled sum, where the name a is the *location* of the object, and M is a finite collection of labelled methods; each method $l(\tilde{x}) = P$ is labelled by a distinct label l, has a finite sequence of names $\tilde{x}$ as formal parameters, and has a process P as body. An asynchronous message has a name a of some object as target, and carries a *labelled value* $l[\tilde{v}]$ that selects the method l with actual parameters $\tilde{v}$. Messages are output prefixes without continuations, thus, non-blocking. Hence, the calculus is asynchronous. The location of an object or the target of a message is the *subject* of the process.

Unlike in the actor model and in most object-oriented languages, in TyCO there can be several objects sharing the same location, and locations without associated objects. A distributed object is implemented by parallel composition of several objects with the same location, possibly with different sets of methods, reflecting parts of the object in different states. However, the type system ensures that *only the output capability of names may be transmitted*, e.g., in a method $l(\tilde{x}) = P$ the formal parameters $\tilde{x}$ are not allowed to be locations of objects in the body P (i.e., we rule out processes like $a?\{val(x) = x?\{m\}\}$). Such a restriction, henceforth called the *locality condition*, is present in object-oriented languages where the creation of a new name and of a new object are tightly coupled (e.g. Java). This restriction is also present in recent versions of the asynchronous π-calculus [3, 5, 17] and in the join-calculus [11]. The authors claim that this has an object-oriented flavour, since a name has a clear scope where all the inputs are defined; outside that scope other objects can access the its methods, but cannot create new objects with that name.

The process $(\mathsf{rec}\, X(\vec{x}).a?M)[\vec{v}]$ denotes a *persistent object*. It is a recursive definition, which is a template for the creation of an object, together with two sequences of actual parameters, input and output, to instantiate the formal parameters. We adapted this constructor from [3].

The remaining constructors of the calculus are fairly standard in name-passing process calculi: process $P \mid Q$ denotes the *parallel composition* of processes; process $\nu x\, P$ denotes the *scope restriction* of the name x to the process P (often seen as the creation of a new name, visible only inside the process); *inaction*, with denotation $\mathbf{0}$, is the terminated process. The process $X[\vec{v}]$ is an application to a process variable of a sequence of actual parameters that substitute the formal ones in the recursive definition.

Example 2.2 (One-place Cell). A very simple example of a persistent object is the *one-place buffer cell*: it stores a value and offers two methods: *read*, with a parameter that is the returning address; and *write*, also with a parameter that is the new value to store. Take the following template.

$$\mathsf{rec}\,\mathrm{Cell}(c, v).c?\{read(r) = r!val[v] \mid \mathrm{Cell}[c, v]\,,\ write(u) = \mathrm{Cell}[c, u]\}$$

A request to Cell$[a, w]$ for method *write* triggers a computation that stores the value b in the Cell a, overwriting the previous value w. (where '$\longrightarrow$' stands for reduction [31]).

$$\mathrm{Cell}[a, w] \mid a!\mathit{write}[b] \longrightarrow \mathrm{Cell}[a, b]$$

The original formulation of TyCO used replication to have persistent objects (thus, unbounded computation power). Despite its simplicity, replication is unwieldy. The latest formulation, called core-TyCO [29], has a form of process-declaration of which the recursive definition used herein is a particular case. Defining replication as an abbreviation in core-TyCO is straightforward. To enforce the unfolding of the recursive definition only when necessary, we define transition rules for persistent objects.

Notation. When a sequence of parameters is empty we omit the symbol ε; when a method has an empty sequence of parameters we omit the parentheses, writing $l = P$ instead of $l() = P$. Furthermore, we abbreviate to l a method $l = \mathbf{0}$ or a labelled value carrying an empty sequence of names, $l[]$, we abbreviate to $\nu\tilde{x}\, P$ a process $\nu x_1 \cdots \nu x_n\, P$, consider that the operator 'ν' extends as far to the right as possible, and that it binds tighter than the operator '$|$'. Let $\{\tilde{x}\}$ denote the set of names of the sequence $\tilde{x}$, and $\{\vec{x}\}$ denote the set of names of the sequences $\tilde{x}_i$ and $\tilde{x}_o$.

A labelled transition system. We define the operational semantics of TyCO via a labelled transition relation in an early style, instead of the reduction relation used in all previous work, as this formulation expresses more naturally the behavioural aspects of the calculus. In spite of the simplicity of the reduction relation, it turns out to be more difficult to study the notion of communication error.

DEFINITION 2.3.

- An occurrence of the process variable X in a process P is *bound* if it occurs in the part M of a subterm $(\mathsf{rec}\, X(\vec{x}).a?M)[\vec{v}]$ of P; otherwise the occurrence of X is *free*. We define accordingly the set $\mathrm{fv}(P)$ of the *free variables* in a process P.
- An occurrence of a name x in a process P is *bound* if it is in a part of P of the form $l(\tilde{w}x\tilde{y}) = Q$ (a method), or if it is in a subterm of P of one of the forms $(\mathsf{rec}\, X(\vec{x}).a?M)[\vec{v}]$ where $x \in \{\vec{x}\}$, or $\nu x\, Q$; otherwise the occurrence of x is *free*. We define accordingly the set $\mathrm{fn}(P)$ of the *free names* in a process P.
- *Alpha-conversion*, denoted by $\equiv_\alpha$, affects bound names and bound process variables.
- The process $P[\tilde{v}/\tilde{x}]$ denotes the *simultaneous substitution* in P of the free occurrences of each name in $\tilde{x}$ by the respective names in $\tilde{v}$; it is defined only when $\tilde{x}$ and $\tilde{v}$ are of the same length. Similarly, the process $P[\vec{v}/\vec{x}]$ denotes the the simultaneous substitution in P of the pair of sequences in $\vec{x}$ by the respective sequences in $\vec{v}$. Finally, the process $P[Q/X[\vec{v}]]$ denotes the simultaneous substitution in P of the occurrences of $X[\vec{v}]$, where X is free, by the process Q.

Henceforth we assume the standard name convention on process variables. Furthermore, we consider that process variables only occur bound in a process, and that in a recursive definition like $(\mathsf{rec}\, X(\vec{x}).a?M)[\vec{v}]$ we have $\mathrm{fn}(a?M) \subseteq \{\vec{x}\}$.

DEFINITION 2.4. The *structural congruence* relation is inductively defined by the following rules.

$$P \equiv Q\,,\ \text{if } P \equiv_\alpha Q$$

$$a?M \equiv a?M' \text{ and } (\mathsf{rec}\, X(x\vec{x}).x?M)[a\vec{v}] \equiv (\mathsf{rec}\, X(x\vec{x}).x?M')[a\vec{v}]\,,\ \text{if } M \text{ is a permutation of } M'$$

$$P \mid \mathbf{0} \equiv P \qquad P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid R \equiv P \mid (Q \mid R)$$

$$\nu x\, \mathbf{0} \equiv \mathbf{0} \qquad \nu xy\, P \equiv \nu yx\, P \qquad \nu x\, P \mid Q \equiv \nu x\, (P \mid Q) \text{ if } x \notin \mathrm{fn}(Q)$$

DEFINITION 2.5 (MESSAGE APPLICATION). The result of applying a labelled value $l[\tilde{v}]$ to a collection of methods M, denoted by $M \bullet l[\tilde{v}]$, is the process $P[\tilde{v}/\tilde{x}]$ if $l(\tilde{x}) = P$ is a method in M and the substitution is defined.

Table 1: The *asynchronous transition relation.*

$$\text{(Out)}\quad a!l[\tilde v] \xrightarrow{a!l[\tilde v]} \mathbf{0} \qquad \text{(In)}\quad a?M \xrightarrow{a?l[\tilde v]} M \bullet l[\tilde v] \qquad \text{(Com)}\quad a?M \mid a!l[\tilde v] \xrightarrow{\tau} M \bullet l[\tilde v]$$

$$\text{(RIn)}\qquad (\mathsf{rec}\, X(x\vec x).x?M)[a\vec v] \xrightarrow{a?l[\tilde v]]} M[\mathsf{rec}\, X(x\vec x).x?M/X, a\vec v/x\vec x] \bullet l[\tilde v]$$

$$\text{(Rec)}\qquad (\mathsf{rec}\, X(x\vec x).x?M)[a\vec v] \mid a!l[\tilde v] \xrightarrow{\tau} M[\mathsf{rec}\, X(x\vec x).x?M/X, a\vec v/x\vec x] \bullet l[\tilde v]$$

$$\text{(Par)}\quad \frac{P \xrightarrow{m} Q}{P \mid R \xrightarrow{m} Q \mid R}\ (\mathrm{bn}(m) \cap \mathrm{fn}(R) = \emptyset) \qquad \text{(Res)}\quad \frac{P \xrightarrow{m} Q}{\nu x\, P \xrightarrow{m} \nu x\, Q}\ (x \notin \mathrm{fn}(m) \cup \mathrm{bn}(m))$$

$$\text{(Open)}\quad \frac{P \xrightarrow{\nu \tilde x\, a!l[\tilde v]} Q}{\nu x\, P \xrightarrow{\nu x \tilde x\, a!l[\tilde v]} Q}\ (a \notin \{x\tilde x\}) \qquad \text{(Struct)}\quad \frac{P \equiv P' \quad P' \xrightarrow{m} Q' \quad Q' \equiv Q}{P \xrightarrow{m} Q}$$

DEFINITION 2.6. The set of *action labels* is given by the following grammar, where $\{\tilde x\} \subseteq \{\tilde v\} \backslash \{a\}$.

$$m \quad ::= \quad \tau \quad | \quad a?l[\tilde v] \quad | \quad \nu \tilde x\, a!l[\tilde v]$$

The *silent action* τ denotes internal communication in the process; the *input action* $a?l[\tilde v]$ represents the reception in the name a of an l-labelled message carrying names $\tilde v$; the *output action* $\nu \tilde x\, a!l[\tilde v]$ represents the emission to the name a of an l-labelled message carrying names $\tilde v$, some of them bound (those in $\tilde x$; the name a is free to allow the message to be received). In the last two action labels, the name a is the *subject* of the label, and the names $\tilde v$ are their *objects*.

DEFINITION 2.7. The set $\mathrm{fn}(m)$ of the *free names* of the action label m is defined as follows.

$$\mathrm{fn}(\tau) = \emptyset, \qquad \mathrm{fn}(a?l[\tilde v]) = \{a\} \cup \{\tilde v\}, \qquad \mathrm{fn}(\nu \tilde x\, a!l[\tilde v]) = \{a\} \cup (\{\tilde v\} \setminus \{\tilde x\})$$

The set $\mathrm{bn}(m)$ of the *bound names* of the action label m is defined accordingly.

We define the operational semantics of TyCO via an *asynchronous transition relation* that is inspired by the labelled relation defined by Honda and Yoshida for the ν-calculus [14], and by Amadio *et al.* for the asynchronous π-calculus [4]. The asynchronous transition relation is the smallest relation on processes generated by the rules in the Table 1. The relation expresses the behaviour of a process in terms of its possible exchanges of messages with the environment, what emphasises its asynchronous nature.

3 Error processes

A communication error in TyCO is an object-message pair, such that message application is not defined. Two different reasons may cause the error: the message requests a method of the object but with a wrong number of parameters, or the message requests a method that does not exist in the target object. To deal with non-uniform service availability in concurrent objects, a static notion of error is not suitable. We propose a looser notion of process with a communication error, which is dynamic, such that the situations referred above are no longer considered as errors if the request may be accepted by the object at some time in the future (after changing its state).

Notation. Assume the following abbreviations.

- Let $\Longrightarrow$ denote $\xrightarrow{\tau}{}^*$, $\xRightarrow{m}$ denote $\Longrightarrow\xrightarrow{m}\Longrightarrow$, and $\xRightarrow{\widehat m}$ denote $\xRightarrow{m}$ if $m \neq \tau$, otherwise $\Longrightarrow$. Moreover, $\xRightarrow{\widetilde m}$ and $\xRightarrow{\widehat{\widetilde m}}$ denote sequences of $\xRightarrow{m}$ and $\xRightarrow{\widehat m}$, respectively.
- Let $\mathrm{obj}(\widetilde m)$ denote the set of objects of the sequence $\widetilde m$, and $\mathrm{subj}(\widetilde m)$ the set of subjects of $\widetilde m$. Moreover, let $\{\widetilde m\}$ denotes the set of action labels in the sequence $\widetilde m$.

In the remainder of this section consider that m denotes only input actions.

Redexes. The new notion of process with a communication error needs an auxiliary notion of *redex*. A redex is a pair object/message sharing the same subject. If message application is defined then the redex may reduce. Since an object's location is not unique—what allows the implementation of distributed objects by parallel composition—there may be several redexes with the same message. We call *bad redex* a pair of a distributed object and a message where none of possible redexes may reduce.

DEFINITION 3.1 (REDEXES).

- A process of the form $\Pi_{i\in I}\, a?M_i \mid a!l[\tilde{v}]$ is a redex. The objects and the message *participate* in the redex. The redex is *good* if some contractum $M_i \bullet l[\tilde{v}]$ is defined; it is *bad* otherwise.
- A process of the form $\Pi_{i\in I}\, a?M_i \mid a!l[\tilde{v}] \mid Q$, where there is no process Q' nor input action m with subject a such that $Q \stackrel{m}{\Longrightarrow} Q'$, is an $al\tilde{v}$-redex.
- A process P *has* a (bad) redex R if P has a subterm R not under a prefix.
- A name a *instances* a name x in P if $a{=}x$ or else,
  1. $P \Longrightarrow Q$, Q has a redex $b?\{l(\tilde{w}y\tilde{z}){=}\, R, \dots\} \mid b!l[\tilde{u}a\tilde{v}]$, and
  2. $x = y$ or else, a instances x in Q', where Q' is Q with the redex contracted.
- A sequence $\widetilde{m}$ of input actions *generates* an $al\tilde{v}$*-redex* R in P if
  1. $P \stackrel{\widetilde{m}}{\Longrightarrow} Q$ and R is a redex in Q, and $\exists_{\widetilde{m}'\neq\widetilde{m}}\, P \stackrel{\widetilde{m}'}{\Longrightarrow} Q'$ and R is not a redex in Q',
  2. P has a subterm $x!l[\tilde{u}]$ and $a\tilde{v}$ instances $x\tilde{u}$ in $P \mid \widetilde{m}$.

Notion of Error. Errors are processes able to generate bad redexes that persist throughout a computation. An occasional bad redex is not enough to make the process an error if further computation can consume (at least) one of the parties of the bad redex. Thus, messages that may be accepted at some time in the future do not cause errors: an error is a *persistent bad redex*.

EXAMPLE 3.2. Consider the one-place buffer defined in the introduction[1].

1. Process Empty$[b] \mid b!think$ is an error, since the object b never has the method *think*.
2. Processes $R \stackrel{\text{def}}{=}$ Empty$[b] \mid b!read[r]$ and $S \stackrel{\text{def}}{=}$ Empty$[b] \mid b!write[u] \mid b!write[v]$ are not errors, since the bad redexes can disappear (if the environment provides the right messages).
3. Processes $\nu b\, R$ and $\nu b\, S$ are erroneous, since both processes are locally deadlocked (the scope of the object's name is restricted); hence the bad redexes become persistent (cf. [7]). However, $\nu b\,(R \mid a!l[b])$ is not an error, since the name b can be extruded.
4. Process $P \stackrel{\text{def}}{=} a?\{l = a!m\}$ is not an error as the object is ephemeral, but the process $(\mathsf{rec}\, X(a,a).P \mid X[a,a])[a,a]$ is.
5. Process $\nu a\,(a?\{l{=}\, b!m\}) \mid b?\{n\}$ is a deadlock, but not an error.
6. Process $(\mathsf{rec}\, X(x,y).x?\{l_1 = y!l_1 \mid x?\{l_2 = y!l_2 \mid X[x,y]\}\})[a,a] \mid a!l_1 \mid a!l_2$ is not an error, although there is always a bad redex, but with different messages (alternates between label l_1 and l_2). The bad redex is recurring, but not persistent.
7. We do not want to reject processes that compute erroneously due only to the incorrect behaviour of the environment. The following processes are not errors, even though the environment can incorrectly interact with them provoking errors.
   (a) $Q \stackrel{\text{def}}{=}$ Empty$[b] \mid c?\{l(x) = x!think\}$ is not an error, but $Q \xrightarrow{c?l[b]}$ Empty$[b] \mid b!think$, which is an error;
   (b) $P \stackrel{\text{def}}{=} (\mathsf{rec}\, X(x).x?\{l\})[a] \mid b?\{l(x) = \nu v\, x!l[v] | v?\{l(x) = x!l\} \mid c?\{l(x) = x!l[a]\}$ is not an error, and neither is $P \xrightarrow{b?l[c]}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau}\equiv (\mathsf{rec}\, X(x).x?\{l\})[a]$, but $P \xrightarrow{c?l[b]}\xrightarrow{\tau}\xrightarrow{\tau}\equiv$ $\nu v\,(\mathsf{rec}\, X(x).x?\{l\})[a] \mid v?\{l(x){=}\, x!l\} \mid a!l[v]$, which is an error.

[1]The translation of Empty$[b]$ into the syntax of this section is the following: $(\mathsf{rec}\, X(x).x?\{write(u){=}\, x?\{read(r){=}\, X[x] \mid r!val[u]\}\})[b]$.

We are now in a position to define the set of processes with a communication error.

DEFINITION 3.3 (ERROR PROCESS). The set ERR is the set of processes P such that,

$$\exists_{\widetilde{m},Q}\,(P \stackrel{\widehat{\widetilde{m}}}{\Longrightarrow} Q, \text{ where } Q \text{ has a bad } al\tilde{v}\text{-redex not generated by } \widetilde{m}, \text{ and}$$

$$\forall_{\widetilde{n},R}\,(Q \stackrel{\widehat{\widetilde{n}}}{\Longrightarrow} R, \text{ where } R \text{ has a bad } al\tilde{v}\text{-redex not generated by } \widetilde{n})).$$

The condition ensures that the message participating in a bad distributed $al\tilde{v}$-redex is neither comes from nor is generated by the environment, since an environment can always interact incorrectly with a process and produce errors. Otherwise, we would have $\forall_P\,(P \stackrel{m}{\Longrightarrow}$ implies $P{\in}\text{ERR})$, i.e., any process with free input-names would be an error.

Whence, communication errors are processes containing a message that can never be accepted by a persistent object, i.e., persistent bad distributed redexes. We can distinguish two cases:

1. *message-never-understood*: the object does not have the method requested by the message, and it will never have (cf. the first example above);
2. *local deadlock*: the object is blocked, and therefore can never reach the state where it could accept the message (cf. the third example above).

4 The Algebra of Behavioural Types

We propose a process algebra, the Algebra of Behavioural Types (ABT [25]), where terms denote types which provide a description of some dynamic aspects of the behaviour of objects. Types are terms of a process algebra, namely higher-order labelled transition systems that characterise all possible life-cycles of concurrent objects. States represent interfaces of objects; state transitions model the dynamic change of the interfaces of objects. Therefore, the types provide an internal view of the objects that inhabit them, which is synchronous since detects the arrival of messages (by performing transitions). The types enjoy the rich algebraic theory of a process algebra, with an operational semantics and a notion of equivalence based on a bisimulation. The syntax and operational semantics are similar to a fragment of CCS [18], the class of Basic Parallel Processes (BPP) of Christensen [8], where communication is not present (parallel composition is merge). For a more thorough discussion on ABT see [25], where we present an axiomatisation of the notion of equivalence, complete at least for image-finite types.

Syntax. Assume a countable set of *method names*, $l, m, n, \ldots$, possibly subscripted, and a countable set of type variables, denoted by t. Consider that the sets disjoint.

DEFINITION 4.1. The set T of *behavioural types* is defined by the following grammar.

$$\alpha \quad ::= \quad \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i \quad | \quad \sum_{i\in I} \upsilon.\alpha_i \quad | \quad \alpha \parallel \alpha \quad | \quad t \quad | \quad \mu t.\alpha$$

where I is a finite indexing set, and each $\widetilde{\alpha}_i$ is a finite sequence of types.

A term of the form $l(\widetilde{\alpha})$ is a *message type*: it corresponds to the action of invoking a method with name l and parameters of type $\widetilde{\alpha}$. A term of the form $l(\widetilde{\alpha}).\alpha$ is a *method type*. The label in the prefix l stands for the name of the method, which has parameters of type $\widetilde{\alpha}$; the type α of the method's body prescribes the behaviour of the method. The term $\upsilon.l(\widetilde{\alpha}).\alpha$ is a *blocked type*, the type of an unavailable method. The sum '$\sum$' is the type composition operator:

1. it puts various method types together to form the type of an object that offers the corresponding collection of methods: the labelled sum $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$;
2. associates various blocked types in the (blocked) sum $\sum_{i\in I} \upsilon.\alpha_i$, which after being released behaves according to one of the types α_i.

Table 2: The *labelled transition relation* of the Algebra of Behavioural Types.

$$\text{ACT}\quad \textstyle\sum_{i\in I} \pi_i.\alpha_i \xrightarrow{\pi_j} \alpha_j \quad (j \in I)$$

$$\text{RPAR}\quad \frac{\alpha \xrightarrow{\pi} \alpha'}{\alpha \parallel \beta \xrightarrow{\pi} \alpha' \parallel \beta} \qquad \text{LPAR}\quad \frac{\alpha \xrightarrow{\pi} \alpha'}{\beta \parallel \alpha \xrightarrow{\pi} \beta \parallel \alpha'} \qquad \text{REC}\quad \frac{\alpha[\mu t.\alpha/t] \xrightarrow{\pi} \alpha'}{\mu t.\alpha \xrightarrow{\pi} \alpha'}$$

The *parallel composition* ('$\parallel$') of types is the type of concurrent composition of several objects located at the same name (interpreted as different parts of the same object, each in a different state), or the type of the concurrent composition of several messages targeted to the same name. Finally, the term $\mu t.\alpha$ (for $\alpha \not\equiv t$) denotes a recursive type. It allows the definition of infinite types, enabling us to characterise the behaviour of persistent objects. An occurrence of the variable t in the body α of the type $\mu t.\alpha$ is *bound*; the occurrence of t in the type α is *free*. Moreover, $\mathrm{fv}(\alpha)$ is the set of variables that occur free in α. The type $\alpha[\beta/t]$ denotes the substitution in α of the free occurrences of t by β. Assume the variable convention and that types are equal up-to α-conversion.

Operational semantics. We define a structural operational semantics for the Algebra of Behavioural Types via a labelled transition relation on the types.

DEFINITION 4.2. The set of *labels* is given by the grammar $\quad \pi \quad ::= \quad \upsilon \quad | \quad l(\widetilde{\alpha})$

The label υ denotes a silent transition that releases a blocked object, A label $l(\widetilde{\alpha})$ denotes a transition corresponding to the invocation of method l with actual parameters of types $\widetilde{\alpha}$. The labelled transition relation is inductively defined by the rules of the Table 2. The axiom ACT is in fact an axiom-scheme that captures two cases:

1. the basic transition is the invocation of a method with name l and parameters of types $\widetilde{\alpha}$, yielding the type of the body of that method;
2. the unblocking transition is the occurrence of a silent transition υ to release a blocked type.

Furthermore, parallel composition preserves the transition relation, and the transitions of the recursive type $\mu t.\alpha$ are exactly those of the type $\alpha[\mu t.\alpha/t]$.

Notation. Assume the following abbreviations.

- Let $\mathbf{0}$ denote the empty type (sum with empty indexing set); we omit the sum symbol if the indexing set is singular, and we use the plus operator ('+') to denote binary sums of types.
- The type $l(\widetilde{\alpha})$ denotes $l(\widetilde{\alpha}).\mathbf{0}$, and l denotes $l()$; we write $\alpha \equiv \beta$ when the types α and β are *syntactically identical*.
- The *interface* of a type α, denoted $\mathrm{int}(\alpha)$, is the set of its observable labels, e.g., in a labelled sum $\alpha \equiv \sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$ it is the set $\{l_i(\widetilde{\alpha}_i)\}_{i\in I}$, the interface of a blocked type is empty, and the interface of a parallel composition is the union of the interfaces of the components.
- Let $\Longrightarrow$ denote $\xrightarrow{\upsilon}{}^*$.

Type equivalence. Elsewhere we define a notion of equivalence that we call *label-strong bisimulation, lsb* [25]. It is a *higher-order strong bisimulation on labels* and a *weak bisimulation on unblockings*. We require that if α and β are bisimilar then:

1. if α offers a particular method then also β offers that method, and the parameters of the methods are pairwise bisimilar;
2. if α offers an unblocking transition then β can offer zero or more unblocking transitions.

The rigorous definition of the type equivalence notion follows.

DEFINITION 4.3 (BISIMILARITY ON TYPES).

1. A symmetric binary relation $R \subseteq T \times T$ is a *label-strong bisimulation* if whenever $\alpha R \beta$:
   (a) $\alpha \xrightarrow{l(\tilde{\alpha})} \alpha'$ implies $\exists_{\beta', \widetilde{\beta}} (\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ and $\alpha' \tilde{\alpha} R \beta' \widetilde{\beta})$; [2]
   (b) $\alpha \xrightarrow{v} \alpha'$ implies $\exists_{\beta'} (\beta \Longrightarrow \beta'$ and $\alpha' R \beta')$;
2. Two types α and β are *label-strong bisimilar*, or simply *bisimilar*, and we write $\alpha {\approx} \beta$, if there is a label-strong bisimulation R such that $\alpha R \beta$.

The usual properties of bisimilarities hold, namely $\approx$ is an equivalence relation and a fixed point, in the sense that $\alpha \approx \beta$ holds if, and only if,

1. $\alpha \xrightarrow{l(\tilde{\alpha})} \alpha'$ implies $\exists_{\beta', \widetilde{\beta}} (\beta \xrightarrow{l(\widetilde{\beta})} \beta'$ and $\alpha' \tilde{\alpha} \approx \beta' \widetilde{\beta})$;
2. $\alpha \xrightarrow{v} \alpha'$ implies $\exists_{\beta'} (\beta \Longrightarrow \beta'$ and $\alpha' \approx \beta')$.

The sum of method types and the sum of blocked types preserve bisimilarity. This result is easy to verify, since the sum is guarded. Furthermore, the operators of parallel composition and recursion also preserve bisimilarity. Briefly, *lsb* is a higher-order congruence relation.

The congruence is a symmetric simulation; the simulation is a *subtyping* relation, $\leq$, i.e., it is a quasi-order (reflexive, transitive, but not anti-symmetric), and any two types which simulate each other are equivalent (bisimilar). Thus, if α' simulates α then $\alpha' \leq \alpha$.

5 Type assignment

To ensure that a process will not run into communication errors the interactions within the process should be disciplined in such a way that persistent bad redexes do not appear. This discipline is imposed by a type system that assigns types to the names of a process. The types are partial specifications of the behaviour of names in the process: they specify the interactions that can happen on a name and the types of the names carried by it. Hence, the types provide the necessary information to prevent communication errors.

Typings. We need to distinguish the usage of names within a process—either as locations of objects or as targets of messages—since it is the simultaneous presence of both in a process that may result in communication errors. Recall that the objects are non-uniform, and that we implement distributed objects by parallel composition of several objects with the same subject.

DEFINITION 5.1 (TYPING JUDGEMENTS).

- *Type assignment* to names are formulae $a{:}(\alpha, \beta)$, where α is the type of the name a as subject of objects and β is the type of a as subject of messages.
- *Typings* Γ, Δ, are finite sets of type assignment to names, where no name occurs twice.
- *Type assignment* to process variables are formulae $X{:}(\tilde{\alpha}, \widetilde{\beta})$, where $\tilde{\alpha}$ are the types of the input parameters of X and $\widetilde{\beta}$ are the types of the output parameters of X.
- A *Base* B is a finite set of type assignment to process variables, where no variable occurs twice.
- *Judgements* are formulae $B, \Gamma \vdash P$. If $B = \emptyset$ then we write $\Gamma \vdash P$.

Notation. Let $\mathrm{dom}(\Gamma)$ be the set of the names in Γ, and let γ denote the type-pair (α, β). Then:

- $\Gamma{\cdot}a{:}\gamma$ denotes $\Gamma \cup \{a{:}\gamma\}$, provided $a \notin \mathrm{dom}(\Gamma)$, $\Gamma(a){=}\gamma$ if $a{:}\gamma {\in} \Gamma$, and $\Gamma(a){=}(\mathbf{0}, \mathbf{0})$ otherwise; furthermore, $\Gamma(a)_1{=}\alpha$ and $\Gamma(a)_2{=}\beta$;
- let $\vec{\gamma} \stackrel{\text{def}}{=} (\tilde{\alpha}, \widetilde{\beta})$, and define similarly $\Gamma{\cdot}\tilde{a}{:}\vec{\gamma}, \Gamma(\tilde{a}), \Gamma(\tilde{a})_1$, and $\Gamma(\tilde{a})_2$;
- consider bisimulation on γ-types pairwise defined;
- for basis, define similarly $\mathrm{dom}(B)$, and $B{\cdot}X{:}\vec{\gamma}$.

[2] We write $\alpha' \tilde{\alpha} R \beta' \widetilde{\beta}$ to denote $\alpha' R \beta', \alpha_1 R \beta_1, \ldots,$ and $\alpha_n R \beta_n$.

Capturing the notion of communication error. Our purpose is the definition of a type system correct with respect to the notion of process with a communication error (Definition 3.3).

A term of the type algebra defines the semantics of the usage of a name in a process. Therefore we refer to two syntactic categories of types, input and output, as a name can be an object location or a message target. An input-type characterises the sequences of messages an object can accept, thus the object life-cycle; an output-type characterises the messages sent to an object. Due to the locality condition, objects of messages and parameters of methods have output-types.

We define co-inductively two new higher-order contravariant relations on types, an *agreement* relation, and a *compatibility* relation. The agreement relation ensures the absence of local deadlocks: if a type denotes a graph (cf. [26]), an output-type agrees with an input-type if they share at least one path, and moreover, all paths which have a common prefix are equal (up-to agreement on the parameters, contravariantly). Unblocking occurs only when at least one of the types is blocked, or when both types have the same interface—a safety approach to detect bad redexes. Moreover, if one of the types is empty then it is in the relation with any type (a message or an object by themselves are not errors). The compatibility relation ensures the absence of messages-never-understood. It is essentially a relaxing of the previous relation by allowing the environment to "help", providing some steps: an output-type is compatible with an input-type if the first has a path that coincides with a *projection* of a path of the former.

Definition 5.2 (Agreement on types). Consider that $\Gamma \vdash P$ and let $a{:}(\alpha,\beta) \in \Gamma$.

1. A binary relation $S \subseteq T \times T$ is an *agreement* on Γ and P, if whenever $\alpha S \beta$
   (a) $\beta{\approx}\mathbf{0}$ or $\alpha{\approx}\mathbf{0}$, or else
   (b) i. $\exists_{l,\alpha',\beta',\widetilde{\beta}_1,\widetilde{\beta}_2}(\alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha', \beta \xrightarrow{l(\widetilde{\beta}_2)} \beta'$, for all $a!l[\tilde{v}]$ subterms of P such that $\Gamma(\tilde{v})_2 = \widetilde{\beta}_2$ we have $\Gamma(\tilde{v})_1 S \widetilde{\beta}_1$, and $\alpha' S \beta'))$;
   ii. $\forall_{l,\alpha',\beta',\widetilde{\beta}_1\widetilde{\beta}_2}(\alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha',\ \beta \xrightarrow{l(\widetilde{\beta}_2)} \beta'$, and for all $a!l[\tilde{v}]$ subterms of P such that $\Gamma(\tilde{v})_2 = \widetilde{\beta}_2$ we have $\Gamma(\tilde{v})_1 S \widetilde{\beta}_1)$ implies $\alpha' S \beta'$;
   iii. if $\mathrm{int}(\alpha){=}\emptyset$ or $\mathrm{int}(\beta){=}\emptyset$ or $\mathrm{int}(\alpha){=}\,\mathrm{int}(\beta)$ then
   $(\alpha \Longrightarrow \alpha'$ implies $\exists_{\beta'}(\beta \Longrightarrow \beta'$ and $\alpha' S \beta'))$, and
   $(\beta \Longrightarrow \beta'$ implies $\exists_{\alpha'}(\alpha \Longrightarrow \alpha'$ and $\alpha' S \beta'))$.
2. A type α *agrees* with a type β on a typing Γ and a process P, written $\alpha \bowtie_\Gamma^P \beta$, if there is an agreement relation S on Γ, P such that $\alpha S \beta$.

The relation $\bowtie_\Gamma^P$ is the largest type agreement relation. We define similarly the *type compatibility relation* $\prec_\Gamma^P$, substituting '$\alpha{\equiv}\mathbf{0}$' by 'α finite' in condition 1(a), and '$\alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$' by '$\alpha \xrightarrow{\widetilde{\pi}} \alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$, for some $\widetilde{\pi}$', in condition 1(b).

Type System. The typing assignment system is inductively defined by the rules of Table 3. There is a type rule for each process constructor, with the exception of the restriction operator that has two rules: one is applied when the restricted name occurs in the process both as location of objects and target of messages (the rule Dfr, which detects possible deadlocks); the other is applied when the restricted name occurs in the process only as location of objects or target of messages (the rule Res). The extra rule ($\approx$) allows the substitution in a typing formula of a type by a bisimilar one. Some side conditions of the rules deserve further explanation:

- rules Var and Rec check that the types of the actual parameters of the process variable are equivalent to the types of the actual parameters;
- rule Msg checks that the output-type of the message's target has a state and the message's label with correct type parameters;
- rule Obj checks that all the parameters of the methods are only used for output, and that the input-type of the object has a state corresponding to the object's interface;

Table 3: Typing Assignment System

$$\textsc{Nil}\quad B,\Gamma\vdash\mathbf{0}\qquad\qquad \textsc{Var}\quad B\cdot X{:}\vec{\gamma},\Gamma\vdash X[\vec{v}]\ \ (1)\qquad\qquad \textsc{Msg}\quad B,\Gamma\vdash a!l[\tilde{v}]\ \ (2)$$

$$\textsc{Obj}\quad \frac{\forall_{i\in\{1,\dots,n\}}\ B,\Gamma\cdot\tilde{x}_i{:}\widetilde{(\mathbf{0},\beta_i)}\vdash P_i}{B,\Gamma\vdash a?\{l_1(\tilde{x}_1)=P_1,\dots,l_n(\tilde{x}_n)=P_n\}}\ \ (3)\qquad\qquad \textsc{Rec}\quad \frac{B\cdot X{:}\vec{\gamma},\Gamma\cdot\vec{x}{:}\vec{\gamma}\vdash a?M}{B,\Gamma\vdash(\mathsf{rec}\,X(\vec{x}).a?M)[\vec{v}]}\ \ (1)$$

$$\textsc{Res}\quad \frac{B,\Gamma\vdash P}{B,\Gamma\vdash\nu x\,P}\qquad \textsc{Par}\quad \frac{B,\Gamma\vdash P\quad B,\Gamma\vdash Q}{B,\Gamma\vdash P\mid Q}\qquad\qquad \approx\quad \frac{B,\Gamma\cdot x{:}\gamma\vdash P}{B,\Gamma\cdot x{:}\delta\vdash P}\ \ (4)$$

1. $\Gamma(\vec{v})\approx\vec{\gamma}$

2. $\exists_{\tilde{\pi},\tilde{\pi}'}\ (\Gamma(a)_2 \stackrel{\hat{\tilde{\pi}}}{\Longrightarrow}\leq l(\widetilde{\beta}), \Gamma(\tilde{v})_2 \stackrel{\hat{\tilde{\pi}}'}{\Longrightarrow}\leq \widetilde{\beta})$

3. $\exists_{\tilde{\pi}}\ \Gamma(a)_1 \stackrel{\hat{\tilde{\pi}}}{\Longrightarrow}\leq \sum_{i\in I} l_i(\widetilde{\beta}_i)$

4. $\gamma\approx\delta$

The type system accepts several type-pairs for each name of a process; not only it accepts bisimilar types but also types from a subtyping lattice, e.g., if $P \stackrel{\text{def}}{=} a?\{l = a?m\} \mid a!m$ then not only $\{a{:}(l.m, m)\} \vdash P$, but also $\{a{:}(l.m, l \parallel m)\} \vdash P$, even though there is no message l with subject a. Moreover, an error process can type-check with a typing "too good" for it: $\{a{:}(l.m, l \parallel m)\} \vdash \nu a\, P$.

Definition 5.3 (Most general typing).

- Let $\Gamma \vdash P$ and $\Gamma' \vdash P$; typing Γ is *more general* than typing Γ', $\Gamma' \leq \Gamma$, if for all $a \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma'), \Gamma'(a)_i \leq \Gamma(a)_i\ (i \in \{1, 2\})$;
- The Typing Γ is the *most general typing* of a process P if
  1. $\Gamma \vdash P$;
  2. $\exists_{\Gamma'} \forall_{\Delta}\ \Delta \vdash P$ implies $\Delta \leq \Gamma' \approx \Gamma$[3].

At the end of the deduction of a typing for a process we still have to check if the types of the free names are compatible and the types of the bound names agree. In that case the process is error free.

Definition 5.4 (Well-typed processes). Process P is *well-typed* if, and only if, there is a typing Γ such that

1. Γ is a most general typing of P;
2. for all $a \in \mathrm{dom}(\Gamma) \cap \mathrm{fn}(P)$, $\Gamma(a)_1 \prec^P_\Gamma \Gamma(a)_2$;
3. for all $a \in \mathrm{dom}(\Gamma) \cap \mathrm{bn}(P)$, $\Gamma(a)_1 \bowtie^P_\Gamma \Gamma(a)_2$.

In this case we say Γ is a well-typing for P.

Example 5.5 (Typing Objects (cf. [9])).
$\emptyset, \{a{:}(\mu t.rep.(t \parallel mess), rep), b{:}(\mu t.quest(rep).t, quest(rep) \| quest(rep)), c{:}(\mu t.(rep+mess).t, rep)\} \vdash$
$(\mathsf{rec}\,X(a).a?\{rep = X[a] \parallel a?\{mess\}\})[a] \mid (\mathsf{rec}\,X(b).b?\{quest(x) = X[b] \mid x!rep\})[b] \mid$
$(\mathsf{rec}\,X(c).c?\{rep = X[c], mess = X[c]\})[c] \mid b!quest[a] \mid b!quest[c]$.

Notice that the input-types of a and c are not in a subtyping relation, but since the object b accepts both messages, the input-type of b is compatible with its output type—both objects a and c can receive *rep* messages.

[3] Assume that $\Gamma' \approx \Gamma$ if for all $a \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma'), \Gamma(a)_i \approx \Gamma'(a)_i\ (i \in \{1, 2\})$.

Main Results. We are now in a position to present the properties of the type system. We discuss their proofs in a technical annex.

LEMMA 5.1 (CONGRUENCE LEMMA). If $\Gamma \vdash P$ and $P \equiv Q$, then $\Gamma \vdash Q$.

The following theorem, together with the subsequent corollary, is the main result of this work. The theorem characterises the conditions under which the interactions within a process, and between a process and the environment, preserve the typability of that process: reduction preserves typability; the emission of a message by a process does not affect the existence of a typing for that process; however, the reception of a message by a process can affect the existence of typing for the resulting process, therefore, messages from the environment should respect the semantics ("expectations") of there targets.

THEOREM 5.2 (SUBJECT REDUCTION). Let process P be well-typed.

1. If $P \xrightarrow{\tau} Q$, then Q is well-typed.
2. If $P \xrightarrow{\nu\tilde{x}\, a!l[\tilde{v}]} Q$, then Q is well-typed.
3. If $P \xrightarrow{a?l[\tilde{v}]} Q$ and $\Gamma(a)_1 \prec_{\Gamma}^{P|a!l[\tilde{v}]} (l(\Gamma(\tilde{v})_2) \parallel \Gamma(a)_2)$, then Q is well-typed.

The corollary is a safety result for well-typed processes: it ensures the absence of run-time errors in well-typed processes.

COROLLARY 5.3. If P is well-typed then $P \notin$ ERR.

Notice that the converse of this result is not true. There are several examples of "good" processes (with respect to the notion of error) which are not typable, as the agreement and compatibility relations do not have causality information regarding the order of the names in a process (e.g., $\nu ab\, b?\{l = a!n\} \mid a?\{m\}$). Nevertheless, complete type systems are undecidable [30].

6 Related work

Our proposition of the notion of communication error is, to our knowledge, original. In the context of the lazy λ-calculus [1], Dami proposed a liberal approach to potential errors [10], which is similar to our notion of process-error. He argues that the common notion of erroneous term is over-restrictive: some programs, in spite of having error terms inside them, do not actually generate a run-time error when executed. Since there is a family of programming languages based in the lazy λ-calculus, Dami proposes a lazy approach to errors: a term is considered erroneous if and only if it always generates an error after a finite number of interactions with its context.

It seems quite natural to have a liberal approach to potential errors in the context of non-uniform concurrent objects. We claim that it allows a more flexible and behaviour-oriented style of programming, and moreover, detects some deadlocks. The type system we propose herein types all processes that "traditional" systems [31] do, except for those that do not conform to the restriction that only the output-capability of names can be transmitted. Obviously, the proposed system types processes considered errors by those systems, like the buffer-cell presented in the introduction, and furthermore, rejects some deadlocked processes (in fact more than those conforming to the notion of error, that is, leading to persistent bad redexes).

The detection of deadlocks using a type system is studied by Kobayashi [15, 27]. He proposes a type system that ensures partial deadlock freedom and partial confluence of programs. The types have information about an ordering on the use of channels, and also about the reliability of the channels used (with respect to non-determinism). Therefore it detects more than our local deadlocks.

The starting point on dealing with objects' non-uniform service availability via type systems is the work of Nierstrasz on regular types for active objects [22]. Puntigam [23] starts from Nierstrasz work, defines a calculus of concurrent objects, and, as we do, defines also a process-algebra of types, but his objectives are different: the system guarantees that all messages that are sent to an object are accepted, and enforces sequencing of messages to achieve the purpose. His work is centred on subtyping and on checking the type assignment.

Concretely on types for mobile processes, we know the works by Boudol, by Colaço, Pantel, and Sallé, and by Najm and Nimour. Boudol's work [7] is on typing for the Blue Calculus (a variant of the π-calculus directly incorporating the λ-calculus). The types are functional, and incorporate Hennessy-Milner logic with recursion (thus, with modalities). Processes inhabit the types, and although this approach captures some causality in the usage of names in a process, we conjecture that the type system accepts a similar family of processes that our does. Colaço's work [9] aims to the detection of "orphan messages" in the calculus of actors (messages that may never be accepted in some actor's execution path, either because the requested service is not in the actor's interface, or due to dynamic changes in a actor's interface). Types are interface-like, with multiplicities, and the type system requires some complex operations on a lattice of types. The type inference algorithm is a set-constraints algorithm. Nevertheless, the type system does not look as powerful as ours, as types do not have dynamic information. Najm and Nimour's work [21] distinguishes between private and public objects' interfaces, and proposes a typing system for the calculus of objects bindings, handling dynamic service offers in private interfaces. They define an equivalence relation, a compatibility relation, and a subtyping relation on types, based on the simulation and on the bisimulation relations, just as we do for our types. The types are sets of guarded equations, regular and deterministic. Deadlock is not treated.

Acknowledgements

Special thanks are due to Gérard Boudol, Jean-Louis Colaço, Silvano Dal-Zilio, and Uwe Nestmann for careful reading of previous versions of this work, as well as for very important suggestions and fruitful discussions.

References

[1] Samson Abramsky. The lazy lambda calculus. *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1989.

[2] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. M.I.T. Press, 1986.

[3] Roberto Amadio. An asynchronous model of locality, failure, and process mobility. In *Coordination Languages and Models*, volume LNCS 1282. Springer-Verlag, 1997. Extended version as Rapport de Recherche RR-3109, INRIA Sophia-Antipolis, 1997.

[4] Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. An extended abstract appeared in *Proceedings of CONCUR'96*, LNCS 1119, pages 163–178.

[5] Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195(2):205–226, 1998. An extended abstract appeared in *Proceedings of CONCUR'96*, LNCS 1119, pages 147–162.

[6] Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche RR-1702, INRIA Sophia-Antipolis, 1992.

[7] Gérard Boudol. Typing the use of resources in a concurrent calculus. In *Advances in Computing Science*, volume LNCS 1345, pages 239–253. Springer-Verlag, 1997.

[8] Søren Christensen. *Decidability and Decomposition in Process Algebras*. PhD thesis ECS-LFCS-93-278, Departament of Computer Science, University of Edinburgh, U. K., 1993.

[9] Jean-Louis Colaço, Mark Pantel, and Patrick Sallé. A set constraint-based analyses of actors. In *2nd Conference on Formal Methods for Open Object-based Distributed Systems*. IFIP, 1997.

[10] Laurent Dami. Labelled reductions, runtime errors, and operational subsumption. In *24th International Colloquium on Automata, Languages and Programming*, volume LNCS 1256, pages 782–793. Springer-Verlag, 1997.

[11] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of CONCUR'96*, LNCS 1119, pages 406–421.

[12] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal, modular actor formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[13] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *5th European Conference on Object-Oriented Programming*, volume LNCS 512, pages 141–162. Springer-Verlag, 1991.

[14] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151:437–486, 1995.

[15] Naoki Kobayashi. A partially deadlock-free typed process calculus. In *12th Annual Symposium on Logic in Computer Science*, pages 128–139. Computer Society Press, 1997. Full version as as Technical Report 97–02, University of Tokyo.

[16] Naoki Kobayashi and Akinori Yonezawa. Towards foundations of concurrent object-oriented programming - types and language design. *Theory and Practice of Object Systems*, 1(4), 1995.

[17] Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In *25th International Colloquium on Automata, Languages and Programming*, volume LNCS 1443, pages 856–967. Springer-Verlag, 1998.

[18] Robin Milner. *Communication and Concurrency.* C. A. R. Hoare Series Editor—Prentice-Hall, 1989.

[19] Robin Milner. The polyadic π-calculus: a tutorial. Technical Report ECS-LFCS 91-180, University of Edinburgh, 1991. Appeared in *Proceedings of the International Summer School on Logic and Algebra of Specification*, Marktoberdorf, August 1991. Reprinted in *Logic and Algebra of Specification*, ed. F. L. Bauer, W. Brauer, and H. Schwichtenberg, Springer-Verlag, 1993.

[20] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–77, 1992.

[21] Elie Najm and Abdelkrim Nimour. A calculus of object bindings. In *2nd IFIP conference on Formal Methods for Open Object-based Distributed Systems*, 1997.

[22] Oscar Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.

[23] Franz Puntigam. Coordination Requirements Expressed in Types for Active Objects. In *11th European Conference on Object-Oriented Programming*, volume LNCS 1241, pages 367–388. Springer-Verlag, 1997.

[24] António Ravara and Luís Lopes. Programming and implementation issues in non-uniform TyCO. Research report DCC-99-1, Department of Computer Science, Faculty of Sciences, University of Porto, 1999. Presented at the PLI'99 Workshop on Object-Oriented Specification Techniques for Distributed Systems and Behaviours. Web page: http://www.tec.informatik.uni-rostock.de/IuK/congr/oosds99/program.html.

[25] António Ravara, Pedro Resende, and Vasco T. Vasconcelos. An algebra of behavioural types. Research report 26/99 DM-IST, Department of Mathematics, Instituto Superior Técnico, 1999.

[26] António Ravara and Vasco T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *3rd International Euro-Par Conference*, volume LNCS 1300, pages 554–561. Springer-Verlag, 1997. Also available as Research report 6/97 DM-IST, Instituto Superior Técnico.

[27] Eijiro Sumii and Naoki Kobayashi. A generalized deadlock-free process calculus. In *3rd International Workshop on High-Level Concurrent Languages*, volume 16 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 1998.

[28] Vasco T. Vasconcelos. Processes, functions, and datatypes. *Theory and Practice of Object Systems*, 5(2):97–110, 1999.

[29] Vasco T. Vasconcelos and Rui Bastos. Core-TyCO, the language definition, version 0.1. DI/FCUL TR 98–3, Department of Computer Science, University of Lisbon, 1998.

[30] Vasco T. Vasconcelos and António Ravara. Communication errors in the π-calculus are undecidable. *Information Processing Letters*, In Print, 1999.

[31] Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *1st International Symposium on Object Technologies for Advanced Software*, volume LNCS 742, pages 460–474. Springer-Verlag, 1993.