Typing Non-uniform Concurrent Objects

António Ravara[1] and Vasco T. Vasconcelos[2]

[1] Department of Mathematics, Instituto Superior Técnico, Portugal
[2] Department of Informatics, Faculty of Sciences, University of Lisbon, Portugal

Abstract. Concurrent objects may offer services non-uniformly, constraining the acceptance of messages on the states of objects. We advocate a looser view of communication errors. Safe programmes must guarantee that every message has a chance of being received if it requests a method that may become enabled at some point in the future. We formalise non-uniform concurrent objects in TyCO, a name-passing object calculus, and ensure program safety via a type system. Types are terms of a process algebra that describes dynamic aspects of the behaviour of objects.

1 Introduction

Herein we study non-uniform concurrent objects in TyCO [25, 27], an asynchronous name-passing object calculus along the lines of the π-calculus [15, 16]. Concurrent objects may offer services non-uniformly according to synchronisation constraints, that is, the availability of a service may depend on the state of the object [18]. Objects exhibiting methods that may be enabled or disabled, according to their internal state, are very common in object-oriented programming (think of a stack, a buffer, an FTP server, a bank account, a cash machine). Nevertheless, the typing of concurrent objects poses specific problems, due to the non-uniform availability of their methods.

The typed λ-calculus is a firm ground to study typing for sequential object-oriented languages, with a large body of research and results, namely on record-types for objects. However, a static notion of typing, like types-as-interfaces, is not powerful enough to capture dynamic properties of the behaviour of concurrent objects. Hence, we aim at a type discipline that copes with non-uniform concurrent objects. The interface of an object should describe only those methods that are enabled, and when a client asks for a disabled method the message should not be rejected if the object may evolve to a state where the method becomes available. To achieve this objective, we propose a looser view of communication errors, such that an object-message pair is not an error if the message may be accepted at some time in the future. Therefore, an error-free process guarantees that messages are given a chance of being attended, as a permanently enabled object eventually receives a message target to it. Errors are those processes containing an object that persistently refuses to accept a given message, either because the object is blocked, or because it will never have the right method for the message.

Traditional type systems assign rigid interface-like types to the names of objects [15, 27]. Take the example of a one-place buffer that only allows read operations when it is full, and write operations when it is empty. We like to specify it as follows, showing that the buffer alternates between *write* and *read*.

$$\begin{array}{l}\textbf{def}\ \mathsf{Empty}(b) = b?\{\mathit{write}(u) = \mathsf{Full}[b,u]\}\\ \textbf{and}\ \mathsf{Full}(b,u) = b?\{\mathit{read}(r) = r!\mathit{val}[u] \mid \mathsf{Empty}[b]\}\end{array}$$

The referred type systems reject the example above, since name b alternates its interface. An alternative typable implementation uses the busy-waiting technique to handle non-available operations.

$$\begin{array}{l}\textbf{def}\ \ \mathsf{Buf}(b,v,\mathit{empty}) =\\ \qquad b?\{\mathit{write}(u) = \textbf{if}\ \mathit{empty}\ \textbf{then}\ \mathsf{Buf}[b,u,\mathit{false}]\\ \qquad\qquad\qquad\quad \textbf{else}\ b!\mathit{write}[u] \mid \mathsf{Buf}[b,v,\mathit{false}]\\ \qquad\qquad \mathit{read}(r) = \ \textbf{if}\ \mathit{empty}\ \textbf{then}\ b!\mathit{read}[r] \mid \mathsf{Buf}[b,v,\mathit{true}]\\ \qquad\qquad\qquad\quad \textbf{else}\ r!\mathit{val}[v] \mid \mathsf{Buf}[b,v,\mathit{true}]\}\end{array}$$

In the second implementation, a process containing the redex $\mathsf{Buf}[b,v,\mathit{empty}] \mid b!\mathit{read}[r]$ 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 $\mathsf{Empty}[b] \mid b!\mathit{read}[r]$ 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 deadlocked process like $\nu b\,(\mathsf{Empty}[b] \mid b!\mathit{read}[r])$ should be considered an error. In conclusion, the implementations behave similarly, in the sense that both accept a *read* message when the buffer is full, afterwards becoming empty, and accept a *write* message when the buffer is empty, afterwards becoming full. A more thorough discussion on non-uniform objects in TyCO, with more complex examples, can be found elsewhere [20].

We have been working on a theory of types able to accommodate this style of programming [21, 22]. We adopt a types-as-behaviours approach, such that a type characterises the semantics of a concurrent objects by representing all its possible life-cycles as a state-transition system. Types are terms of a process algebra, fuelled by an higher-order labelled transition system, providing an internal view of the objects that inhabit them. It constitutes a synchronous view, since a transition corresponds to the reception of a message by an object. 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 is based on a bisimulation that is incomparable with other notions in the literature, and for which we define an axiomatic system, complete for image-finite types [21]. Therefore, types are partial specifications of the behaviour of objects, able to cope with non-uniform service availability. They are also suitable for specifying communication protocols, like pop3, since a type can represent sequences of requests to a server, and also deals with choice.

Equipped with a more flexible notion of error and with these richer types, we develop a type system which guarantees that typable processes will not run into communication errors.

2 The Calculus of Objects

TyCO (Typed Concurrent Objects) is a name-passing calculus featuring asynchronous communication between concurrent objects via labelled messages carrying names. The calculus is an object-based extension of the asynchronous π-calculus [5, 11] where the objects behave according to the principles of the actor model of concurrent computation [1] (with the exception of the uniqueness of actors' names and the fairness assumption).

Syntax. Consider *names* a, b, v, x, y, and *labels* $l, m, n, \ldots$, 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, $\boldsymbol{a}$ denotes a pair of sequences of names, the first identified by $\tilde{a}_i$, and the second by $\tilde{a}_o$, $\boldsymbol{a} = \tilde{a}_i; \tilde{a}_o$. Furthermore, assume a countable set of process variables, $X, Y, \ldots$, disjoint from the previous sets.

The grammar in Table 1 defines the set of processes. *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 parameters, and has an arbitrary process P for body. An asynchronous message has a name a as target, and carries a *labelled value* $l[\tilde{v}]$ that selects the method l with arguments $\tilde{v}$. The location of an object or the target of a message is the *subject* of the process; arguments of messages are its *objects*. The process $(\mathbf{rec}\, X(\boldsymbol{x}).a?M)[\boldsymbol{v}]$ denotes a *persistent object*, as it is done in the asynchronous π_1-calculus [2]. It is a recursive definition together with two sequences of arguments: input and output. The original formulation of TyCO used replication to provide for persistent objects. Despite its simplicity, replication is unwieldy. A recent formulation [25] uses 'process-declaration', of which the recursive definition herein is a particular case.

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 within P); *inaction*, denoted $\mathbf{0}$, is the terminated process. The process $X[\boldsymbol{v}]$ is a recursive call.

In contrast with the actor model, and with most object-oriented languages, in TyCO there can be several objects sharing the same location and locations without associated objects. A *distributed object* is a parallel composition of several objects sharing the same location, possibly with different sets of methods, describing different copies of the same object, each in a different state. The type system ensures that *only the output capability of names may be transmitted*, e.g. in a method $l(\tilde{x}) = P$ the parameters $\tilde{x}$ are not allowed to be locations of objects in the body P. 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), as well as in recent versions of the asynchronous π-calculus [2, 4, 13], and in the join-calculus [10]. This restriction is crucial; we would not know how to type non-uniform objects without it.

Notation. In $l(\tilde{x}) = P$, we omit the parentheses when $\tilde{x}$ is the empty sequence, writing $l = P$. Furthermore, we abbreviate to l a method $l = \mathbf{0}$ or a labelled value $l[]$, and abbreviate to $\nu\tilde{x}\,P$ a process $\nu x_1 \cdots \nu x_n\,P$. Also, we consider that the operator 'ν' extends as far to the right as possible. Finally, let $\{\tilde{x}\}$ denote the set of names of the sequence $\tilde{x}$, and $\{\boldsymbol{x}\}$ denote $\{\tilde{x}_i\} \cup \{\tilde{x}_o\}$. Henceforth, we assume the standard convention on names and process variables. Furthermore, we consider that process variables occur only bound in processes, and that in a recursive definition, $(\mathbf{rec}\,X(\boldsymbol{x}).a?M)[\boldsymbol{v}]$, condition $\mathrm{fn}(a?M) \subseteq \{\boldsymbol{x}\}$ holds. Notice that $\{\tilde{x}_i\} \cap \{\tilde{x}_o\}$ is not necessarily empty.

Syntax:

$$P ::= a?M \;\mid\; a!l[\tilde{v}] \;\mid\; P \mid Q \;\mid\; \nu x\,P \;\mid\; (\mathbf{rec}\,X(\boldsymbol{x}).a?M)[\boldsymbol{v}] \;\mid\; X[\boldsymbol{v}] \;\mid\; \mathbf{0}$$
where $M ::= \{l_i(\tilde{x}_i) = P_i\}_{i \in I}$, and I is a non-empty finite indexing set.

Structural Congruence:

$$P \equiv Q, \text{ if } P \equiv_\alpha Q; \qquad a?M \equiv a?M', \text{ if } M \text{ is a permutation of } M';$$
$$P \mid \mathbf{0} \equiv P, \quad P \mid Q \equiv Q \mid P, \text{ and } (P \mid Q) \mid R \equiv P \mid (Q \mid R);$$
$$\nu x\,\mathbf{0} \equiv \mathbf{0}, \quad \nu xy\,P \equiv \nu yx\,P, \text{ and } \nu x\,P \mid Q \equiv \nu x\,(P \mid Q) \text{ if } x \notin \mathrm{fn}(Q).$$

Action Labels: $m ::= \tau \;\mid\; a?l[\tilde{v}] \;\mid\; \nu\tilde{x}\,a!l[\tilde{v}]\,,$ where $\{\tilde{x}\} \subseteq \{\tilde{v}\} \setminus \{a\}\,.$

Message Application: $M \bullet l[\tilde{v}] \stackrel{\text{def}}{=} P[\tilde{v}/\tilde{x}]$, if $l(\tilde{x}) = P$ is a method in M.

Asynchronous Transition Relation:

$$\textsc{Out}\ a!l[\tilde{v}] \xrightarrow{a!l[\tilde{v}]} \mathbf{0} \qquad \textsc{In}\ a?M \xrightarrow{a?l[\tilde{v}]} M \bullet l[\tilde{v}] \qquad \textsc{Com}\ a?M \mid a!l[\tilde{v}] \xrightarrow{\tau} M \bullet l[\tilde{v}]$$
$$\textsc{Rin}\quad (\mathbf{rec}\,X(x\boldsymbol{x}).x?M)[a\boldsymbol{v}] \xrightarrow{a?l[\tilde{v}]]} M[\mathbf{rec}\,X(x\boldsymbol{x}).x?M/X][a\boldsymbol{v}/x\boldsymbol{x}] \bullet l[\tilde{v}]$$
$$\textsc{Rec}\quad (\mathbf{rec}\,X(x\boldsymbol{x}).x?M)[a\boldsymbol{v}] \mid a!l[\tilde{v}] \xrightarrow{\tau} M[\mathbf{rec}\,X(x\boldsymbol{x}).x?M/X][a\boldsymbol{v}/x\boldsymbol{x}] \bullet l[\tilde{v}]$$
$$\textsc{Par}\ \frac{P \xrightarrow{m} Q}{P \mid R \xrightarrow{m} Q \mid R}\ (\mathrm{bn}(m) \cap \mathrm{fn}(R) = \emptyset) \qquad \textsc{Open}\ \frac{P \xrightarrow{\nu\tilde{x}\,a!l[\tilde{v}]} Q}{\nu x\,P \xrightarrow{\nu x\tilde{x}\,a!l[\tilde{v}]} Q}\ (a \notin \{x\tilde{x}\})$$
$$\textsc{Res}\ \frac{P \xrightarrow{m} Q}{\nu x\,P \xrightarrow{m} \nu x\,Q}\ (x \notin \mathrm{fn}(m) \cup \mathrm{bn}(m)) \qquad \textsc{Struct}\ \frac{P \equiv P' \quad P' \xrightarrow{m} Q' \quad Q' \equiv Q}{P \xrightarrow{m} Q}$$

Table 1. Typed Concurrent Objects.

A Labelled Transition System. Following Milner *et al.* [16], we define the operational semantics of TyCO via two binary relations on processes, a static one — structural congruence — and a dynamic one — a labelled transition relation in an early style — as this formulation expresses more naturally the behavioural aspects of the calculus that we seek.

Definition 1 (Free and Bound Variables and Substitution).

1. *An occurrence of a process variable X in a process P is bound, if it occurs in the part M of a subterm $(\mathbf{rec}\, X(\boldsymbol{x}).a?M)[\boldsymbol{v}]$ of P; otherwise the occurrence of X is free. We define accordingly the set* $\mathrm{fv}(P)$ *of the free variables in P.*
2. *An occurrence of a name x in a process P is bound if it occurs in the subterm Q of the the part $l(\tilde{w}x\tilde{y}) = Q$, or in the subterm $\nu x\, Q$, or in the subterm $(\mathbf{rec}\, X(\boldsymbol{x}).a?M)[\boldsymbol{v}]$ where $x \in \{\boldsymbol{x}\}$; otherwise the occurrence of x is free. Accordingly, we define the set* $\mathrm{fn}(P)$ *of the free names in P.*
3. *Alpha-conversion, $\equiv_\alpha$, affects both bound names and bound process variables.*
4. *The process $P[\tilde{v}/\tilde{x}]$ denotes the simultaneous substitution in P of the names in $\tilde{v}$ for the free occurrences of the respective namess in $\tilde{x}$; it is defined only when $\tilde{x}$ and $\tilde{v}$ are of the same length. Similarly, the process $P[\boldsymbol{v}/\boldsymbol{x}]$ denotes the simultaneous substitution in P of the pair of sequences in $\boldsymbol{v}$ for the respective sequences in $\boldsymbol{x}$. Finally, the process $P[A/X]$ denotes the simultaneous substitution in P of the part A for the free occurrences of X.*

Table 1 defines the action labels. The *silent action* τ denotes internal communication in the process; the *input action* $a?l[\tilde{v}]$ represents the reception on the name a of an l-labelled message carrying names $\tilde{v}$ as arguments; the *output action* $\nu\tilde{x}\, a!l[\tilde{v}]$ represents the emission to a of an l-labelled message carrying names $\tilde{v}$ as arguments, 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 (Free and Bound Names in Labels). *An occurrence of a name x in an action label m is bound, if it occurs in the part $a!l[\tilde{v}]$ of $\nu\tilde{y}x\tilde{z}\, a!l[\tilde{v}]$; otherwise the occurrence of x is free. Accordingly, we define the sets* $\mathrm{fn}(m)$ *and* $\mathrm{bn}(m)$ *of the free names and of the bound names in an action label m.*

We define the operational semantics of TyCO via an *asynchronous transition relation* that is inspired by the labelled relation defined Amadio *et al.* for the asynchronous π-calculus [3]. The asynchronous transition relation is the smallest relation on processes generated by the respective rules in Table 1, assuming the structural congruence relation inductively defined by the respective rules of Table 1. Notice that rules In, Rin, Com, and Rec assume that message application is defined. Otherwise, the transition does not take place.

3 Error-Free 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: (1) the message requests a method that does not exist in the target object; (2) the message requests a method available in the object, but with a wrong number of arguments. To deal with non-uniform service availability of concurrent objects, this static notion of error is unsuitable. We propose a looser understanding of what is a process with a communication error where the situations mentioned

above are no longer considered errors, if the request may be accepted by the object at some time in the future (after changing its state).

The new notion of 'process without communication errors' needs the auxiliary notion of *redex*. A redex is a object-message pair sharing the same subject. If message application — the contractum of the redex — is defined, then the redex may reduce, and we call it a *good redex*; otherwise, it is a *bad* one. Since an object's location is not unique, there may be several redexes for the same message. To stress the identity of the object and that of the labelled value involved in the redex, we define $al\tilde{v}$-redexes. For the rest of this section, consider that m denotes only input actions. Let $\Longrightarrow$ denote $\xrightarrow{\tau}{}^{*}$, $\overset{m}{\Longrightarrow}$ denote $\Longrightarrow\xrightarrow{m}\Longrightarrow$, and $\overset{\widetilde{m}}{\Longrightarrow}$ denote sequences of $\overset{m}{\Longrightarrow}$. In the following definitions, when we refer to objects, we do not distinguish the ephemeral from the persistent.

Definition 3 (Redexes).

1. *The parallel composition of a distributed object $\Pi_{i\in I}\, a?M_i$ and a message $a!l[\tilde{v}]$ is an $al\tilde{v}$-redex. A process of the form $\nu\tilde{x}\,\Pi_{i\in I}\, a?M_i \mid a!l[\tilde{v}] \mid Q$ has an $al\tilde{v}$-redex, if there is no input action m with subject a such that $Q \overset{m}{\Longrightarrow}$.*
2. *A $al\tilde{v}$-redex is persistent in P if it is present in all the derivatives of P.*
3. *A non-empty $\widetilde{m}$ makes an $al\tilde{v}$-redex emerge from P by substituting $x_0\tilde{x}$, if*
   (a) *P has a subterm $x_0!l[\tilde{x}]$ in a part of the form $l(\tilde{w}) = Q$ where $\{x_0, \tilde{x}\} \cap \{\tilde{w}\} = \emptyset$, and*
   (b) *$P \overset{\widetilde{m}}{\Longrightarrow} \nu\tilde{z}\, Q[a\tilde{v}/x_0\tilde{x}] \mid Q'$, which has an $al\tilde{v}$-redex.*
4. *A sequence $\widetilde{m}$ generates an $al\tilde{v}$-redex in P, if*
   (a) *there is $x_0\tilde{x}$ such that $\widetilde{m}$ makes an $al\tilde{v}$-redex emerge from P by substituting $x_0\tilde{x}$, and*
   (b) *there is $\tilde{n} \neq \widetilde{m}$ such that $\tilde{n}$ makes a $bl\tilde{u}$-redex emerge from P by substituting $x_0\tilde{x}$, and $b\tilde{u} \neq a\tilde{v}$.*

An occasional bad redex is not enough to make the process an error, if further computation can consume (at least) one of the components of the redex. In short, errors are *persistent bad redexes*.

Definition 4 (Error-Free process). *The process P is error-free if, for all $al\tilde{v}$,*

$$\forall_{Q,\widetilde{m}}\, P \overset{\widetilde{m}}{\Longrightarrow} Q \text{ and } Q \text{ has a bad } al\tilde{v}\text{-redex not generated by } \widetilde{m} \text{ implies}$$

$$\exists_{R,\tilde{n}}\, Q \overset{\tilde{n}}{\Longrightarrow} R \text{ and } R \text{ does not have a bad } al\tilde{v}\text{-redex.}$$

The condition on Q ensures that the message participating in a bad $al\tilde{v}$-redex neither comes from, nor is generated by, the environment, since an environment can always interact incorrectly to produce errors. Otherwise, any process with free input-names would be an error.

In conclusion, communication errors are processes containing a message that can never be accepted by a persistent object. We can distinguish two cases: (1) *message-never-understood*, when the object does not have the method requested by the message, and it will never have it; (2) *blocking*, when the object has the method requested by the message, but not in its present interface, and it can never reach a state where it could accept the message.

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

1. Process $\mathit{Empty}[b] \mid b!\mathit{think}$ is an error, since the object b will never have the method *think*. Processes $R \stackrel{\text{def}}{=} \mathit{Empty}[b] \mid b!\mathit{read}[r]$ and $S \stackrel{\text{def}}{=} \mathit{Empty}[b] \mid b!\mathit{write}[u] \mid b!\mathit{write}[v]$ are not errors, since the bad redexes can disappear, if the environment provides the right messages. Processes $\nu b\, R$ and $\nu b\, S$ are erroneous, since in both processes the object b is blocked (as the scope of the object's name is restricted), hence the bad redexes become persistent (cf. an example by Boudol [6]). However, $\nu b\, R \,|\, a!l[b]$ is not an error, since the name b can be extruded, so the bad redex is not necessarily persistent.
2. Process $(\mathbf{rec}\, X(x;y).x?\{l_1 = y!l_1 \mid x?\{l_2 = y!l_2 \mid X[x;y]\}\})[a;a] \mid a!l_1 \mid a!l_2$ is not an error, although there is always a bad redex, but with messages containing different labels. The bad redex is recurring, but not persistent. Process $(\mathbf{rec}\, X(xy;xy).x?\{l = X[xy;xy] \mid x!l \mid y!m \mid y!n \mid y?n\})[ab;ab]$ is an error, even though the object and the messages participating in the bad *am*-redex are always different.
3. We do not want to reject processes that compute erroneously due only to the incorrect behaviour of the environment.
   (a) Process $Q \stackrel{\text{def}}{=} \mathit{Empty}[b] \,|\, c?\{l(x) = x!\mathit{think}\}$ is not an error, although there are interactions with it leading to errors (take $Q \xrightarrow{c?l[b]} \mathit{Empty}[b] \,|\, b!\mathit{think}$).
   (b) Process $(\mathbf{rec}\, X(x;).x?\{l = X[x;]\})[a;] \,|\, b?\{l(x) = \nu v\, x!l[v] \,|\, v?\{l(x) = x!l\} \,|\, c?\{l(x) = x!l[a]\}$ is not an error, although it has derivatives which are errors, depending on the requests coming from the environment.

It is easy to conclude that this notion is undecidable. However, any notion of run-time error of a (Turing complete) language is undecidable, since it can be reduced to the halting problem [26]. A common solution, which should be proved correct, but obviously cannot be complete, is the use of a type system to ensure that well-typed processes do not attain run-time errors.

4 The Algebra of Behavioural Types

We propose a process algebra, the Algebra of Behavioural Types, ABT, where terms denote types that describe dynamic aspects of the behaviour of objects. A type denotes a higher-order labelled transition system where states represent the possible interfaces of an object; state transitions model the dynamic changes of the interfaces by executing a method, thus capturing some causality information. The syntax and operational semantics of ABT are similar to a fragment of CCS [14], the class of Basic Parallel Processes (BPP) of Christensen [7] where communication is not present (parallel composition is merge). However, we need to give a different interpretation to some features of the process algebra, and thus decided to develop ABT. A thorough discussion is presented elsewhere [21].

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

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 the sets disjoint.

The grammar in Table 2 defines the set of behavioural types. A term of the form $l(\widetilde{\alpha}).\alpha$ is a *method type*. The label l in the prefix stands for the name of a method possessing parameters of type $\widetilde{\alpha}$; the type α under the prefix prescribes the behaviour of the object after the execution of the method l with parameters of type $\widetilde{\alpha}$. A term of the form $\upsilon.\alpha$ is a *blocked type*, the type of an unavailable method. The sum '$\sum$' is a non-deterministic type composition operator that: (1) gathers together several method types 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 several blocked types in the *blocked sum* $\sum_{i\in I}\upsilon.\alpha_i$; after being released, the object behaves according to one of the types α_i. The *parallel composition* ('$\|$') is the type of the concurrent composition of several objects located at the same name (interpreted as different copies 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 $\alpha{\neq}t$) denotes a recursive type, enabling us to characterise the behaviour of persistent objects. Assume the variable convention and that types are equal up to α-conversion.

Definition 5 (Free and Bound Variables). *An occurrence of the variable t in a part α of the type $\mu t.\alpha$ is bound; the occurrence of t in the type α is free. The type $\alpha[\beta/t]$ denotes the substitution in α of β for the free occurrences of t.*

Syntax:

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

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

Action Labels: $\pi ::= \upsilon \;\mid\; l(\widetilde{\alpha})$.

Labelled transition relation:

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

$$\textsc{Rpar}\quad \frac{\alpha \xrightarrow{\pi} \alpha'}{\alpha \parallel \beta \xrightarrow{\pi} \alpha' \parallel \beta} \qquad \textsc{Lpar}\quad \frac{\alpha \xrightarrow{\pi} \alpha'}{\beta \parallel \alpha \xrightarrow{\pi} \beta \parallel \alpha'} \qquad \textsc{Rec}\quad \frac{\alpha[\mu t.\alpha/t] \xrightarrow{\pi} \alpha'}{\mu t.\alpha \xrightarrow{\pi} \alpha'}$$

Table 2. The Algebra of Behavioural Types.

Operational Semantics. A higher-order labelled transition system defines the operational semantics of ABT. Label υ denotes a silent transition that releases a blocked object; label $l(\widetilde{\alpha})$ denotes the invocation of method l with arguments of types $\widetilde{\alpha}$. The rules of Table 2 inductively define the labelled transition relation.

Notation. Let **0** denote the sum with the 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 $l()$. The *interface* of a type α, $\mathrm{int}(\alpha)$, is the set of its observable labels: in a labelled sum it is the set $\{l_i(\widetilde{\alpha}_i)\}_{i\in I}$, in a blocked type it is empty, and in a parallel composition it is the union of the interfaces of the components. Let $\Longrightarrow$ denote $\xrightarrow{\upsilon}{}^*$, $\stackrel{\pi}{\Longrightarrow}$ denote $\Longrightarrow\xrightarrow{\pi}\Longrightarrow$, and $\stackrel{\tilde{\pi}}{\Longrightarrow}$ denote sequences of $\stackrel{\pi}{\Longrightarrow}$.

Type Equivalence. Label-strong bisimulation, *lsb*, is a *higher-order strong bisimulation on labels* and a *weak bisimulation on unblockings*. We require that if α and β are *label-strong bisimilar*, then: (1) if α offers a method, also β offers that method, and the parameters of the methods are pairwise bisimilar; and (2) if α offers an unblocking transition, β offers zero or more unblocking transitions.

Definition 6 (Bisimilarity on Types).

1. *A symmetric relation $R \subseteq T \times T$ is a bisimulation, if whenever $\alpha R \beta$,*
   (a) $\alpha \xrightarrow{l(\widetilde{\alpha})} \alpha'$ implies $\exists_{\beta',\widetilde{\beta}}\, \beta \xrightarrow{l(\widetilde{\beta})} \beta'$ and $\alpha' \widetilde{\alpha} R \beta' \widetilde{\beta}$; [2]
   (b) $\alpha \xrightarrow{\upsilon} \alpha'$ implies $\exists_{\beta'}\, \beta \Longrightarrow \beta'$ and $\alpha' R \beta'$.
2. *Two types α and β are label-strong bisimilar, or simply bisimilar, $\alpha \approx \beta$, 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. The sum of method types and the sum of blocked type preserve bisimilarity, as they are guarded sums. Furthermore, parallel composition and recursion also preserve bisimilarity. Briefly, *lsb* is a higher-order congruence relation. However, it is not known if the relation is decidable.

Type equivalence is a symmetric simulation; the simulation, $\leq$, is a partial order (reflexive, transitive, and anti-symmetric — any two types which simulate each other are equivalent). Thus, if α simulates α', then we say that α is a *subtype* of α'. Moreover, the operators of ABT, as well as *lsb*, preserve *subtyping*.

5 Type Assignment

To ensure that a process is error-free (Definition 4), the interactions within the process should be disciplined in such a way that persistent bad redexes do not appear. The types specify the interactions that can happen on a name and the types of the names carried by it. Thus, a type system assigning types to the names of a process imposes the discipline.

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 their simultaneous presence that may result in communication errors. Therefore, the type system assigns pairs of types to names and pairs of sequences of types to process variables.

[2] We write $\alpha_1 \ldots \alpha_n R \beta_1 \ldots \beta_n$ to denote $\alpha_1 R \beta_1, \ldots,$ and $\alpha_n R \beta_n$.

Definition 7 (Typing Judgements). *Consider* $\gamma \stackrel{\text{def}}{=} (\alpha, \beta)$ *and* $\boldsymbol{\gamma} \stackrel{\text{def}}{=} (\widetilde{\alpha}, \widetilde{\beta})$.

1. *Type assignment to names are formulae* $a{:}\gamma$.
2. *Type assignment to process variables are formulae* $X{:}\boldsymbol{\gamma}$.
3. *Typings,* Γ, Δ, *are finite sets of type assignments — to names or to process variables — where no name and no process variable occurs twice.*
4. *Judgements are formulae* $\Gamma \vdash P$. *Process* P *is typable if such a formula holds.*

Notation. Let $\mathrm{dom}(\Gamma)$ be the set of the names and the process variables in Γ. Then, $\Gamma \cdot a{:}\gamma$ denotes $\Gamma \cup \{a{:}\gamma\}$ if $a \notin \mathrm{dom}(\Gamma)$. Let $\Gamma(a) = \gamma$ if $a{:}\gamma \in \Gamma$, and $\Gamma(a) = (\mathbf{0}, \mathbf{0})$ otherwise; furthermore, $\Gamma(a)_{\mathsf{i}} = \alpha$ and $\Gamma(a)_{\mathsf{o}} = \beta$. Take bisimulation and subtyping on γ-types pairwise defined; hence, $\gamma \leq \gamma'$ if $\gamma_{\mathsf{i}} \leq \gamma'_{\mathsf{i}}$ and $\gamma_{\mathsf{o}} \leq \gamma'_{\mathsf{o}}$, and $\Gamma' \approx \Gamma$ if, for all $a \in \mathrm{dom}(\Gamma) \cap \mathrm{dom}(\Gamma')$, $\Gamma(a)_{\mathsf{i}} \approx \Gamma'(a)_{\mathsf{i}}$ and $\Gamma(a)_{\mathsf{o}} \approx \Gamma'(a)_{\mathsf{o}}$.

Capturing the Notion of Communication Error. We refer to two syntactic categories of types, input and output, describing names as object locations and as message targets. An input-type characterises the sequences of messages that an object can accept, and thus the object life-cycle; an output-type characterises the messages sent to an object. We define co-inductively two new higher-order contravariant relations on types, the *agreement* relation and the *compatibility* relation. The former ensures the absence of blockings, the latter the absence of messages-never-understood. If a type denotes a graph, or a rational tree (cf. [22]), an output-type agrees with an input-type if either they share one path, or a path of one of them is a prefix of a path of the other, 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. Furthermore, if one of the types is equivalent to an empty type, then it agrees with all types (i.e. a message or an object, by themselves, are not errors). The compatibility relation is a relaxing of the agreement relation by allowing the environment to "help", providing some transitions. Hence, the compatibility relation considers *projections* of input-types.

Definition 8 (Agreement on Types). *Fix* Γ *and* P *such that* $\Gamma \vdash P$.

1. *A 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.* $\exists_{l,\alpha',\beta',\widetilde{\beta}_1,\widetilde{\beta}_2} \alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha', \beta \xrightarrow{l(\widetilde{\beta}_2)} \beta'$, *and for all* $a!l[\tilde{v}]$ *subterms of* P *such that* $\Gamma(a) \approx (\alpha\beta)$ *and* $\Gamma(\tilde{v}) \approx (\widetilde{\alpha}_2, \widetilde{\beta}_2)$, *we have* $\widetilde{\alpha}_2 \; S \; \widetilde{\beta}_1$ *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* $\Gamma(a) \approx (\alpha\beta)$ *and* $\Gamma(\tilde{v}) \approx (\widetilde{\alpha}_2, \widetilde{\beta}_2)$, *we have* $\widetilde{\alpha}_2 \, S \, \widetilde{\beta}_1$ *implies* $\alpha' S \beta'$,
   iii. if $\mathrm{int}(\alpha) = \emptyset$ *or* $\mathrm{int}(\beta) = \emptyset$ *or* $\mathrm{int}(\alpha) = \mathrm{int}(\beta)$, *then*
   $\alpha \xrightarrow{\upsilon} \alpha'$ *implies* $\exists_{\beta'} \beta \Longrightarrow \beta'$ *and* $\alpha' \; S \; \beta'$, *and*
   $\beta \xrightarrow{\upsilon} \beta'$ *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* $\alpha \bowtie^P_\Gamma \beta$, *if there is an agreement relation* S *on* Γ *and* P *such that* $\alpha \; S \; \beta$.

The relation $\bowtie^P_\Gamma$ is the largest type agreement relation. We define similarly the type compatibility relation $\prec^P_\Gamma$, substituting '$\alpha \equiv \mathbf{0}$' with 'α finite' in condition 1(a), and '$\alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$' with '$\alpha \xrightarrow{\widetilde{\pi}} \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$, for some $\widetilde{\pi}$ without occurrences of silent transitions', in condition 1(b).

Type System. The rules in Table 3 inductively define the typing checking system. There is a type rule for each process constructor, and an extra rule ($\approx$) that allows the substitution of a type with a bisimilar one in a formula. Rules VAR and REC check that the types of the arguments of the process variable are equivalent to the types of its parameters; rule MSG checks that the output-type of the message's target has a transition labelled with the message's label (with the 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 (reachable) state which has all its transitions labelled exactly with those of the object's interface. Notice that RES do not discard the bound variable from the typing. This is because we may need its type-pair to verify an agreement.

NIL	$\Gamma \vdash \mathbf{0}$	VAR	$\Gamma \cdot X{:}\boldsymbol{\gamma} \vdash X[\boldsymbol{v}]$ $(\Gamma(\boldsymbol{v}) \approx \boldsymbol{\gamma})$
MSG	$\Gamma \cdot \tilde{v}{:}(\_, \widetilde{\beta}) \vdash a!l[\tilde{v}]$		$(\Gamma(a)_\mathsf{o} \Longrightarrow \xrightarrow{l(\widetilde{\beta}')}$ and $\widetilde{\beta} \approx \widetilde{\beta}')$
OBJ	$\dfrac{\Gamma \cdot \tilde{x}_i{:}(\mathbf{0}, \widetilde{\beta}_i) \vdash P_i}{\Gamma \vdash a?\{l_i(\tilde{x}_i) = P_i\}_{i\in I}}$		$(\exists_{\widetilde{\pi},\alpha}\, \Gamma(a)_\mathsf{i} \overset{\widetilde{\pi}}{\Longrightarrow} \xrightarrow{l_i(\widetilde{\beta}'_i)}, \forall_{i\in I}$ with $\widetilde{\beta}_i \approx \widetilde{\beta}'_i)$
REC	$\dfrac{\Gamma \cdot X{:}\boldsymbol{\gamma} \cdot \boldsymbol{x}{:}\boldsymbol{\gamma} \vdash a?M}{\Gamma \cdot \boldsymbol{v}{:}\boldsymbol{\gamma}' \vdash (\mathbf{rec}\, X(\boldsymbol{x}).a?M)[\boldsymbol{v}]}$		$(\boldsymbol{\gamma}' \leq \boldsymbol{\gamma}$ and $\forall_{\gamma\in\{\widetilde{\gamma}_o\}} \gamma \approx \mu t. \sum_{i\in I} v.\beta_i)$
RES	$\dfrac{\Gamma \vdash P}{\Gamma \vdash \nu x\, P}$	PAR	$\dfrac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \mid Q} \quad \approx \quad \dfrac{\Gamma \cdot x{:}\gamma \vdash P}{B, \Gamma \cdot x{:}\delta \vdash P}$ $(\gamma \approx \delta)$

Table 3. Typing Checking System.

Example 2 (Typing Objects).

1. One-place boolean buffer: $\{b{:}(\mu t.write(bool).read(val).t, \mathbf{0})\} \vdash Empty[b]$.
2. Internal non-determinism: consider an object with internal non-determinism, $a?\{l = \nu c\, c!m_1 \mid c!m_2 \mid c?\{m_1 = a?l_1, m_2 = a?l_2\}\}$. The input-type of a expresses the choice: $\Gamma(a)_\mathsf{i} = l.(v.l_1 + v.l_2)$.
3. Persistent objects (cf. [8]): consider the following three objects:
   (a) $P_1 \overset{\text{def}}{=} (\mathbf{rec}\, X(a;).a?\{rep = X[a;] \mid a?\{mess\}\})[a;]$,
   (b) $P_2 \overset{\text{def}}{=} (\mathbf{rec}\, X(b;).b?\{quest(x) = X[b;] \mid x!rep\})[b;]$, and
   (c) $P_3 \overset{\text{def}}{=} (\mathbf{rec}\, X(c;).c?\{rep = X[c;], mess = X[c;]\})[c;]$.
   Process $P_1 \mid P_2 \mid P_3 \mid b!quest[a] \mid b!quest[c]$ is error-free. It is easy to check that:

(a) $\Gamma(a)_{\mathsf{i}} = \mu t.rep.(t \| mess) \bowtie^P_\Gamma rep = \Gamma(a)_{\mathsf{o}}$,
(b) $\Gamma(b)_{\mathsf{i}} = \mu t.quest(rep).t \bowtie^P_\Gamma quest(rep) \| quest(rep) = \Gamma(b)_{\mathsf{o}}$, and
(c) $\Gamma(c)_{\mathsf{i}} = \mu t.rep.t + mess.t \bowtie^P_\Gamma rep = \Gamma(c)_{\mathsf{o}}$.

4. Blocked objects:
   (a) Let $\Gamma \stackrel{\text{def}}{=} \{b{:}(\mu t.write(bool).read(val).t, read(val))\} \vdash \nu b\, Empty[b] | b!read[r]$. The process is an error, and $\Gamma(b)_{\mathsf{i}} \not\bowtie^P_\Gamma \Gamma(b)_{\mathsf{o}}$.
   (b) Let $\Gamma \stackrel{\text{def}}{=} \{a{:}(m, \upsilon.n), b{:}(l, \mathbf{0})\} \vdash \nu ab\, b?\{l = a!n\} \mid a?\{m\}$. The process is not an error, but $\Gamma(a)_{\mathsf{i}} \not\bowtie^P_\Gamma \Gamma(a)_{\mathsf{o}}$.
   (c) Let $\Gamma \stackrel{\text{def}}{=} \{a{:}(n.m, \upsilon.n \parallel m), b{:}(l, l)\} \vdash \nu a\, b?\{l = a!n\} \mid a?\{n\} \mid a!m \mid b!l$. The process is neither an error nor a deadlock, still $\Gamma(a)_{\mathsf{i}} \not\bowtie^P_\Gamma \Gamma(a)_{\mathsf{o}}$.

The type system accepts several type-pairs for each name in a process; not only does it accept bisimilar types, but it also accepts types which are in the subtyping relation. Consider $a?m|a?l$; possible input-types of a are $m \| l$, $l \| m$, $m.l + l.m$, and $l.m + m.l$, all bisimilar. Consider now $a?\{l = a?m\} \mid a!m$; a minimal typing of P is $\{a{:}(l.m, m)\}$, but there are subtypes of $\Gamma(a)$ which also succeed in type-checking with P (like $\{a{:}(l.m, l \parallel m)\}$, even though there is no message l with subject a). Moreover, an error process can type-check with a typing "too generous" for it: $\{a{:}(l.m, l \parallel m)\} \vdash \nu a\, P$, since $\Gamma(a)_{\mathsf{i}} \prec^{\nu a\, P}_\Gamma \Gamma(a)_{\mathsf{o}}$; nevertheless, $\nu a\, P$ is an error.

Definition 9 (Minimal Typing). *Let $\Delta \leq \Gamma$, if $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Gamma)$ and $\Delta(a) \leq \Gamma(a)$, for all $a \in \mathrm{dom}(\Delta)$. We say that Γ is a minimal typing for P, if $\Gamma \vdash P$ and $\forall_\Delta\, \Delta \vdash P$ implies $\Delta \leq \approx\Gamma$.*

Proposition 1 (Basic Properties of Minimal Typings). *A typable process has a minimal typing which is unique up-to type equivalence.*

Proposition 2 (Recursive Output Types). *Take a minimal Γ for P. If, for some name a, $\Gamma(a)_{\mathsf{o}}$ has a subterm of the form $\mu t.\beta$, then β is a blocked sum.*

To be an error is a property of processes which cannot be modularly checked. Thus, we cannot introduce side conditions in the rules to check that input types are compatible or agres with output types. Therefore, after type checking a process, we compute its minimal typing, and check whether the types of the free names are compatible, and whether the types of the bound names agree.

Definition 10 (Well-Typed Processes). *A process P is well-typed, if it is typable and its minimal typing Γ satisfies the following conditions.*

1. *for all $a \in \mathrm{dom}(\Gamma) \cap \mathrm{fn}(P)$, $\Gamma(a)_i \prec^P_\Gamma \Gamma(a)_{\mathsf{o}}$;*
2. *for all $a \in \mathrm{dom}(\Gamma) \cap \mathrm{bn}(P)$, $\Gamma(a)_i \bowtie^P_\Gamma \Gamma(a)_{\mathsf{o}}$.*

Theorem 1 (Operational Correspondence). *Let $\Gamma \vdash P \xrightarrow{m} Q$ with $\Delta \vdash Q$.*

1. *$m \equiv \nu\tilde{x}\, a!l[\tilde{v}]$, if and only if, $\Gamma(a)_{\mathsf{o}} \xrightarrow{l(\widetilde{\beta})} \approx\Delta(a)_{\mathsf{o}}$ where $\Gamma(\tilde{v})_{\mathsf{o}} \approx \widetilde{\beta}$;*
2. *$m \equiv a?l[\tilde{v}]$, if and only if, $\Gamma(a)_i \xrightarrow{l(\widetilde{\beta})} \approx\Delta(a)_i$ where $\Gamma(\tilde{v})_{\mathsf{o}} \approx \widetilde{\beta}$;*
3. *$m \equiv \tau$, if and only if, $\exists a \in \mathrm{n}(P)$, $\Gamma(a)_i \xrightarrow{l(\widetilde{\beta}_1)} \Delta(a)_i$ and $\Gamma(a)_{\mathsf{o}} \xrightarrow{l(\widetilde{\beta}_2)} \Delta(a)_{\mathsf{o}}$.*

The following theorem characterises the conditions under which the interactions within a process, and between a process and the environment, preserve the typability of the process. Theorem 1 allows to construct the typings for Q. Since the calculus is local,no context may create objects located at extruded names.

Theorem 2 (Subject Reduction). *Consider Γ a well-typing for process P.*

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 $\alpha\Gamma(\tilde{v})_i \prec^{Q}_{\Gamma} \Gamma(a)_o\widetilde{\beta}$ with $\Gamma(a)_i \xrightarrow{l(\widetilde{\beta})} \alpha$ then Q is well-typed.*

Corollary 1. *If P is well-typed then P is error-free.*

Notice that the converse of this result is not true. The agreement and compatibility relations do not have causality information, thus there are deadlocks which are not errors and are detected (cf. Example 2.4(b)), but there are interesting processes which are rejected (cf. Example 2.4(c)). The proofs of the above results can be found in the full paper [23].

6 Related work

In the context of the lazy λ-calculus, Dami proposed a liberal approach to potential errors [9], which is similar to our notion of communication-error. He argues that the common notion of erroneous term is over-restrictive, as some programs, in spite of having error terms in them, do not actually attain a run-time error when executed. Since there is a family of programming languages based on 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. Nierstrasz on his work 'Regular types for active objects' [18] discusses on non-uniform service availability of concurrent objects, and proposes notions of behavioural typing and subtyping for concurrent object-oriented programming, notions which take into account dynamic aspects of objects' behaviour. Puntigam [19] starts from Nierstrasz' work, defines a calculus of concurrent objects, a process-algebra of types (with the expressiveness of a non-regular language), and a type system which guarantees that all messages that are sent to an object are accepted, this purpose being achieved by enforcing sequencing of messages. Subtyping is a central issue in his work. It seems quite natural to have a liberal approach to potential errors in non-uniform concurrent objects. It allows a more flexible and behaviour-oriented style of programming and, moreover, detects some deadlocks. In the context of the π-calculus, there is some work on deadlock detection using types. Kobayashi proposes a type system to ensure partial deadlock freedom and partial confluence of programs [12, 24] where types have information about the ordering of the use of channels, and also about the reliability of the channels used (with respect to non-determinism). Yoshida defines graph types for monadic π-processes where the nodes denote basic actions and the edges denote the ordering between the actions [28]. However,

TyCO is not π, and making comparisons with π is not a simple task. Although our types are easily adaptable to π — by associating a fix label, e.g. *val*, to each channel name — the definition of error used herein is difficult to express, if indeed possible. On non-uniform types for mobile processes, we know the works by Boudol, by Colaço, Pantel, and Sallé, and by Najm and Nimour. Boudol proposes a dynamic type system for the Blue Calculus, a variant of the π-calculus directly incorporating the λ-calculus [6]. The types are functional, in the style of Curry simple types, and incorporate Hennessy-Milner logic with recursion — modalities interpreted as resources of names. Processes inhabit the types, and this approach captures some causality in the usage of names in a process, ensuring that messages to a name will meet a corresponding offer. Colaço *et al.* [8] propose a calculus of actors and a type system which aims at the detection of "orphan messages", i.e. messages that may never be accepted, 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 complex operations on a lattice of types. Najm and Nimour [17] propose several versions of a calculus of objects that features dynamically changing interfaces and distinguishes between private and public objects' interfaces. For each version of the calculus they develop a typing system handling dynamic method offers in private interfaces, and guaranteeing some liveness properties. Types are sets of deterministic guarded equations, equipped with a transition relation that induces an equivalence relation, and a subtyping relation, both based on bissimulation.

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. We also thank the anonymous referees for their comments and suggestions. The Danish institute BRICS, the ENSEEIHT at Toulouse, and the Franco-Portugais project "Sémantique des objets concurrents" funded visits of the first author. This work was also partially supported by the Portuguese *Fundação para a Ciência e a Tecnologia*, the PRAXIS XXI Projects 2/2.1/TIT/1658/95 LogComp and P/EEI/120598/98 DiCoMo, as well as by the ESPRIT Group 22704 ASPIRE.

References

1. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. M.I.T. Press, 1986.
2. Roberto Amadio. An asynchronous model of locality, failure, and process mobility. In *Coordination Languages and Models*, LNCS 1282. Springer-Verlag, 1997.
3. Roberto Amadio, Ilaria Castellani, and Davide Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.
4. Michele Boreale. On the expressiveness of internal mobility in name-passing calculi. *Theoretical Computer Science*, 195(2):205–226, 1998.

5. Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche RR-1702, INRIA Sophia-Antipolis, 1992.
6. Gérard Boudol. Typing the use of resources in a concurrent calculus. In *Advances in Computing Science*, LNCS 1345, pages 239–253. Springer-Verlag, 1997.
7. Søren Christensen. *Decidability and Decomposition in Process Algebras*. PhD thesis ECS-LFCS-93-278, Dep. of Computer Science, University of Edinburgh, 1993.
8. Jean-Louis Colaço, Mark Pantel, and Patrick Sallé. A set constraint-based analyses of actors. In *Proc. of FMOODS'97*. IFIP, 1997.
9. Laurent Dami. Labelled reductions, runtime errors, and operational subsumption. In *Proc. of ICALP'97*, LNCS 1256, pages 782–793. Springer-Verlag, 1997.
10. Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proc. of CONCUR'96*, LNCS 1119, pages 406–421. Springer-Verlag, 1996.
11. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *Proc. of ECOOP'91*, LNCS 512, pages 141–162. Springer-Verlag, 1991.
12. Naoki Kobayashi. A partially deadlock-free typed process calculus. In *Proc. of LICS'97*, pages 128–139. Computer Society Press, 1997.
13. Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In *Proc. of ICALP'98*, LNCS 1443, pages 856–967. Springer-Verlag, 1998.
14. Robin Milner. *Communication and Concurrency*. C. A. R. Hoare Series Editor—Prentice-Hall, 1989.
15. Robin Milner. The polyadic π-calculus: a tutorial. In *Logic and Algebra of Specification*. Springer-Verlag, 1993.
16. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100:1–77, 1992.
17. Elie Najm and Abdelkrim Nimour. A calculus of object bindings. In *Proc. of FMOODS'97*. IFIP, 1997.
18. Oscar Nierstrasz. Regular types for active objects. In *Object-Oriented Software Composition*, pages 99–121. Prentice-Hall, 1995.
19. Franz Puntigam. Coordination Requirements Expressed in Types for Active Objects. In *Proc. of ECOOP'97*, LNCS 1241, pages 367–388. Springer-Verlag, 1997.
20. 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.
21. António Ravara, Pedro Resende, and Vasco T. Vasconcelos. An algebra of behavioural types. Research report 26/99 DM-IST, Technical University of Lisbon.
22. António Ravara and Vasco T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *Proc. of Euro-Par'97*, LNCS 1300, pages 554–561. Springer-Verlag, 1997.
23. António Ravara and Vasco T. Vasconcelos. Typing non-uniform concurrent objects. Research report 6/00 DM-IST, Technical University of Lisbon.
24. Eijiro Sumii and Naoki Kobayashi. A generalized deadlock-free process calculus. In *Proc. of HLCL'98*, *Electronic Notes in Theoretical Computer Science*, (16), 1998.
25. Vasco T. Vasconcelos. Processes, functions, and datatypes. *Theory and Practice of Object Systems*, 5(2):97–110, 1999.
26. Vasco T. Vasconcelos and António Ravara. Communication errors in the π-calculus are undecidable. *Information Processing Letters*, 71(5–6):229–233, 1999.
27. Vasco T. Vasconcelos and Mario Tokoro. A typing system for a calculus of objects. In *Proc. of ISOTAS'93*, LNCS 742, pages 460–474. Springer-Verlag, 1993.
28. Nobuko Yoshida. Graph types for monadic mobile processes. In *Proc. of FST/TCS'96*, LNCS 1180, pages 371-386. Springer-Verlag, 1996.