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

June 2000

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. This notion of error-free process detects messages that are never accepted, either because the requested method does not exist at all, or because the object is blocked and cannot change its state to accept the request. We formalise non-uniform concurrent objects in TyCO, a name-passing object calculus, and ensure program safety via a type system. Types-as-interfaces is an approach unsuitable to model non-uniform service availability, hence we define types as terms of a process algebra that describes dynamic aspects of the behaviour of objects. The type system assigns terms of this algebra to the names in processes, ensuring that typable processes do not attain errors.

1 Introduction

Herein we study non-uniform concurrent objects in TyCO [30, 33], an asynchronous name-passing calculus, object based, along the lines of the π-calculus [20, 21]. 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 [23]. 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 particular 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 more flexible style of programming and at a type discipline that copes with non-uniform concurrent objects. Our proposal is the following: the interface of an object describes only those methods that are enabled; when a client asks for a disabled method the message is not 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 a communication 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, since 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.

*Full version of the extended abstract published in the proceedings of Concur'00.

Traditional type systems assign rigid interface-like types to the names of objects [17, 20, 33]. 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 clearly that the buffer alternates between *write* and *read* operations.

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

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

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

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.

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. A more thorough discussion on non-uniform objects in TyCO, with more complex examples, can be found elsewhere [26].

We have been working on a theory of types able to accommodate this style of programming [27, 28]. 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. In this work we incorporate and develop the idea of types as graphs [28]. Our first proposal of a looser notion of communication error is there, but it did not incorporate the detection of some deadlocks. 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 [27]. 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 that ensures safety properties resulting from the absence of persistent bad redexes, guaranteeing that typable processes will not run into communication errors.

The next section presents the syntax and the operational semantics of TyCO, Section 3 builds the notion of error-free processes, Section 4 presents the syntax, the operational semantics, and the equivalence notion of ABT, and finally, Section 5 presents the type systems and the main results. Sketches of proofs of these results can be found in a technical annex.

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 [7, 14] where the objects behave according to the principles of the actor model of concurrent computation [3, 13] (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, $\vec{a}$ denotes a pair of sequences of names, the first identified by $\vec{a}_{\mathsf{i}}$, and the second by $\vec{a}_{\mathsf{o}}$, $\vec{a} = \vec{a}_{\mathsf{i}}; \vec{a}_{\mathsf{o}}$. Furthermore, assume a countable set of *process variables*, $X, Y, \ldots$, disjoint from the previous sets.

Definition 2.1 (Processes). The following grammar defines the *set of processes*.

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

where $M ::= \{l_i(\tilde{x}_i) = P_i\}_{i \in I}$, and I is a non-empty finite indexing set.

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 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 $(\mathbf{rec}\, X(\vec{x}).a?M)[\vec{v}]$ denotes a *persistent object*. It is a recursive definition together with two sequences of arguments: input names and output names. We have adapted this constructor from the asynchronous π_1-calculus [4]. The process $X[\vec{v}]$ is a recursive call. 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*, $\mathbf{0}$, denotes the terminated process.

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. 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 parameters $\tilde{x}$ are not allowed to be locations of objects in the body P (i.e. we rule out processes like $a?\{l(x) = x?M\}$). 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), as well as in recent versions of the asynchronous π-calculus [4, 6, 18], and in the join-calculus [12]. The authors claim that this restriction has an object-oriented flavour, since a name has a clear scope where all the inputs are defined, and outside that scope other objects can access its methods, but cannot create new objects with that name. This restriction is crucial; we would not know how to type non-uniform objects without it.

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: *write*, with a parameter that is the new value to store; and *read*, also with a parameter that is the return address.

$$(\mathbf{rec}\; \mathit{Cell}(c; v).c?\{\mathit{write}(u) = \mathit{Cell}[c; u]\,,\; \mathit{read}(r) = r!\mathit{val}[v] \mid \mathit{Cell}[c; v]\})[a; w]$$

A *write* request to $\mathit{Cell}[a, w]$ triggers a computation that stores the value b in the Cell, overwriting the previous value w: $\mathit{Cell}[a; w] \mid a!\mathit{write}[b] \longrightarrow \mathit{Cell}[a; b]$. A *read* request does not change the state of the cell, but simply sends to the client the values it stores: $\mathit{Cell}[a; b] \mid a!\mathit{read}[r] \longrightarrow \mathit{Cell}[a; b] \mid r!\mathit{val}[b]$

The original formulation of TyCO used replication to provide for persistent objects. Despite its simplicity, replication is unwieldy. A recent formulation [30] uses a form of process-declaration of which the recursive definition herein is a particular case. Defining replication as an abbreviation of process-declaration is straightforward.

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, and that it binds looser than the operator '$|$'. Finally, $\{\tilde{x}\}$ denotes the set of names of the sequence $\tilde{x}$, $\{\vec{x}\}$ denotes $\{\vec{x}_\mathsf{i}\} \cup \{\vec{x}_\mathsf{o}\}$, and $x\vec{x}$ denotes $x\vec{x}_\mathsf{i}; \vec{x}_\mathsf{o}$.

A Labelled Transition System. Following Milner *et al.* [21], 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 2.3 (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 $(\mathbf{rec}\, X(\vec{x}).a?M)[\vec{v}]$ of P; otherwise the occurrence of X is *free.*
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 $(\mathbf{rec}\, X(\vec{x}).a?M)[\vec{v}]$ where $x \in \{\vec{x}\}$; otherwise the occurrence of x is *free.* Accordingly, we define the set $\mathrm{fn}(P)$ of the free names in P, the set $\mathrm{bn}(P)$ of the bound names in P, and the set $\mathrm{n}(P) \stackrel{\mathrm{def}}{=} \mathrm{fn}(P) \cup \mathrm{bn}(P)$.
3. *Alpha-conversion*, denoted by $\equiv_\alpha$, 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 names in $\tilde{x}$; 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 simultaneous substitution in P of the pair of sequences in $\vec{v}$ for the respective sequences in $\vec{x}$. Finally, the process $P[A/X]$ denotes the simultaneous substitution in P of the part A for the free occurrences of X.

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, like $(\mathbf{rec}\, X(\vec{x}).a?M)[\vec{v}]$, condition $\mathrm{fn}(a?M) \subseteq \{\vec{x}\}$ holds. Notice that $\{\vec{x}_\mathsf{i}\} \cap \{\vec{x}_\mathsf{o}\}$ is not necessarily empty. A very simple static type system ensures these conditions, as well as syntactic correctness.

Definition 2.4 (Structural Congruence). The following rules inductively define the *structural congruence relation.*

$P \equiv Q$, if $P \equiv_\alpha Q$

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

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

$\nu x\, \mathbf{0} \equiv \mathbf{0} \qquad \nu xy\, P \equiv \nu yx\, P \qquad \nu x\, P \mid Q \equiv (\nu x\, P) \mid Q$ 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.

Definition 2.6 (Action Labels). The following grammar defines the *set of action labels* where $\{\tilde{x}\} \subseteq \{\tilde{v}\} \setminus \{a\}$: $\quad m ::= \tau \mid a?l[\tilde{v}] \mid \nu\tilde{x}\, a!l[\tilde{v}]$.

The *silent action* τ denotes internal communication in the process; the *input action* $a?l[\tilde{v}]$ represents the reception 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.*

Out	$a!l[\tilde{v}] \xrightarrow{a!l[\tilde{v}]} \mathbf{0}$	
In	$a?M \xrightarrow{a?l[\tilde{v}]} M \bullet l[\tilde{v}]$	
Rin	$(\textsf{rec}\, X(x\vec{x}).x?M)[a\vec{v}] \xrightarrow{a?l[\tilde{v}]} M[\textsf{rec}\, X(x\vec{x}).x?M/X][a\vec{v}/x\vec{x}] \bullet l[\tilde{v}]$	
Com	$a?M \mid a!l[\tilde{v}] \xrightarrow{\tau} M \bullet l[\tilde{v}]$	
Rcom	$(\textsf{rec}\, X(x\vec{x}).x?M)[a\vec{v}] \mid a!l[\tilde{v}] \xrightarrow{\tau} M[\textsf{rec}\, X(x\vec{x}).x?M/X][a\vec{v}/x\vec{x}] \bullet l[\tilde{v}]$	

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

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

Table 1: The asynchronous transition relation.

Definition 2.7 (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* of an action label m.

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 [15], and by Amadio *et al.* for the asynchronous π-calculus [5]. The asynchronous transition relation is the smallest relation on processes generated by the rules of Table 1. The relation expresses the behaviour of a process in terms of its possible exchanges of messages with the environment, this characteristic emphasising its asynchronous nature. Notice that rules In, Rin, Com, and Rcom assume that message application is defined. Otherwise, the transition does not take place.

Some simple lemmas, relating transitions with the form of processes, will be useful to prove properties of the type system. Transition induction carries the proofs.

Lemma 2.1. Let $P \xrightarrow{m} Q$. Then

1. $\mathrm{bn}(Q) \subseteq \mathrm{bn}(P)$, and
2. (a) if $m = a?l[\tilde{v}]$ then $\mathrm{fn}(Q) \subseteq \mathrm{fn}(P) \cup \{\tilde{v}\}$;
   (b) if $m = \nu\tilde{x}\, a!l[\tilde{v}]$ then $\mathrm{fn}(Q) \subseteq \mathrm{fn}(P) \cup \mathrm{bn}(m)$;
   (c) if $m = \tau$ then $\mathrm{fn}(Q) \subseteq \mathrm{fn}(P)$.

Lemma 2.2. Let $P \xrightarrow{m}$ denote $\exists_Q\, P \xrightarrow{m} Q$.

1. $P \xrightarrow{\nu\tilde{x}\, a!l[\tilde{v}]}$, if, and only if, $P \equiv \nu\tilde{x}\, a!l[\tilde{v}] \mid P'$;
2. $P \xrightarrow{a?l_j[\tilde{v}]}$, for some $j \in I$, if, and only if,
   either $P \equiv \nu\tilde{u}\, a?\{l_i(\tilde{x}_i) = P_i\}_{i\in I} \mid P'$ or $P \equiv \nu\tilde{u}\, (\textsf{rec}\, X(x\vec{x}).x?\{l_i(\tilde{x}_i) = P_i\}_{i\in I})[a\vec{v}] \mid P'$;
3. $P \xrightarrow{\tau}$, if, and only if, for some $j \in I$,
   either $P \equiv \nu\tilde{u}\, a?\{l_i(\tilde{x}_i) = P_i\}_{i\in I} \mid a!l_j[\tilde{v}] \mid P'$, or
   $P \equiv \nu\tilde{u}\, (\textsf{rec}\, X(x\vec{x}).x?\{l_i(\tilde{x}_i) = P_i\}_{i\in I})[a\vec{v}] \mid a!l_j[\tilde{v}] \mid P'$.

Behavioural equivalences (barbed congruence or asynchronous bisimulation) are defined in the standard way, following either Honda and Yoshida [15], or Amadio *et al.* [5]. Adapting to this setting the results reported in these works is straightforward.

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).

Notation. For the rest of this section, consider that m denotes only input actions. Let $\Longrightarrow$ denote $\xrightarrow{\tau}{}^*$, $\stackrel{m}{\Longrightarrow}$ denote $\Longrightarrow\xrightarrow{m}\Longrightarrow$, and if $\widetilde{m}$ is $m_1 \cdots m_n$ then $\stackrel{\widetilde{m}}{\Longrightarrow}$ denotes $\stackrel{m_1}{\Longrightarrow} \cdots \stackrel{m_n}{\Longrightarrow}$.

Redexes. 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. Finally, the notion of redex generation from a sequence of messages captures some causality between an environment and a redex emerging from a process in that environment.

In the following definitions, consider that, when we refer to objects, we do not distinguish the ephemeral from the persistent. For the sake of simplicity, for both definitions of redex we omit their respective versions with recursive objects.

Definition 3.1 (Redexes).

1. A process P *has a redex* R if R is a subterm of P not under a prefix.
2. 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 $Q \stackrel{m}{\Longrightarrow}$.
3. A $al\tilde{v}$-redex is *persistent* in P if it is present in all the derivatives of P.
4. A non-empty sequence $\widetilde{m}$ *makes an $al\tilde{v}$-redex emerge from P by substituting $x_0\tilde{x}$*, if
   (a) P has a subterm $x_0!l[\tilde{x}]$ within a part of the form $l(\tilde{w}) = Q$ where $\{x_0\tilde{x}\} \cap \{\tilde{w}\} \neq \emptyset$, and
   (b) $P \stackrel{\widetilde{m}}{\Longrightarrow} \nu\tilde{z}\, Q[a\tilde{v}/x_0\tilde{x}] \mid R \Longrightarrow P'$ and P' has an $al\tilde{v}$-redex.
5. A sequence $\widetilde{m}$ *generates* an *$al\tilde{v}$-redex* in P, if there is $x_0\tilde{x}$ such that
   (a) $\widetilde{m}$ makes an $al\tilde{v}$-redex emerge from P by substituting $x_0\tilde{x}$, and
   (b) there is $\tilde{n}$ that does not make an $al\tilde{v}$-redex emerge from P by substituting $x_0\tilde{x}$.

Notion of Error-Free Processes. Error-free processes do not 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 components of the redex. Thus, messages that may be accepted at some time in the future do not cause errors. In short, errors are *persistent bad redexes*.

Definition 3.2 (Error-Free Processes). The process P is *error-free* if, for all $al\tilde{v}$,

$$\forall_{Q,\widetilde{m}} P \stackrel{\widetilde{m}}{\Longrightarrow} Q \text{ and } Q \text{ has a bad } al\tilde{v}\text{-redex not generated by } \widetilde{m} \text{ implies}$$
$$\exists_{R,\tilde{n}} Q \stackrel{\tilde{n}}{\Longrightarrow} R \text{ and } R \text{ does not have a bad } al\tilde{v}\text{-redex.}$$

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*: the object does not have the method requested by the message, and it will never have it (cf. the first example below);
2. *blocking*: 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 (cf. the third example below).

Example 3.3. Consider the one-place buffer defined in the introduction[1].

1. Process $\mathsf{Empty}[b] \mid b!\mathit{think}$ is an error, since the object b will never have the method *think*.
2. Processes $R \stackrel{\text{def}}{=} \mathsf{Empty}[b] \mid b!\mathit{read}[r]$ and $S \stackrel{\text{def}}{=} \mathsf{Empty}[b] \mid b!\mathit{write}[u] \mid b!\mathit{write}[v]$ are not errors, since the bad redexes can disappear, if the environment provides the right messages.
3. 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 [8]). However, $\nu b\, R \mid a!l[b]$ is not an error, since the name b can be extruded, so the bad redex is not necessarily persistent.
4. 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 different messages (it alternates between label l_1 and l_2). 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 message participating in the bad am-redex are always different.
5. We do not want to reject processes that compute erroneously due only to the incorrect behaviour of the environment.
   (a) $Q \stackrel{\text{def}}{=} \mathsf{Empty}[b] \mid c?\{l(x) = x!\mathit{think}\}$ is not an error, although there are interactions leading to errors, as $Q \xrightarrow{c?l[b]} \mathsf{Empty}[b] \mid b!\mathit{think}$.
   (b) $P \stackrel{\text{def}}{=} (\mathbf{rec}\, X(x;).x?\{l = X[x;]\})[a;] \mid b?\{l(x) = \nu v\, x!l[v] \mid v?\{l(x) = x!l\} \mid c?\{l(x) = x!l[a]\}$ is not an error, although it has derivatives which are errors, depending on the requests from the environment. So, $P \xrightarrow{b?l[c]}\xrightarrow{\tau}\xrightarrow{\tau}\xrightarrow{\tau} (\mathbf{rec}\, X(x;).x?\{l = X[x;]\})[a;]$, which is error-free, however, $P \xrightarrow{c?l[b]}\xrightarrow{\tau}\xrightarrow{\tau} \nu v\, (\mathbf{rec}\, X(x;).x?\{l = X[x;]\})[a;] \mid v?\{l(x) = x!l\} \mid a!l[v]$, which is an error.

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 [32]. 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 define the Algebra of Behavioural Types, ABT, a process algebra where terms denote types that provide a description of 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. Therefore, the types provide an internal view of the objects that inhabit them, which is synchronous, since it detects the arrival of messages (by performing transitions). The syntax and operational semantics of ABT are similar to a fragment of CCS [19],

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

the class of Basic Parallel Processes (BPP) of Christensen [9] 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. For a more thorough discussion on ABT see the reference [27], where we present an axiomatisation of the equivalence, complete 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 the sets disjoint.

Definition 4.1 (Behavioural Types). *The following grammar defines the set $\mathcal{T}$ of behavioural types.*

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

where I is a finite, possibly empty, indexing set, and each $\widetilde{\alpha}_i$ is a finite, possibly empty, sequence of 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*, denoting that the type α is unavailable. 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* ('$\parallel$') 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. The grammar defines a context-free language which allows the definition of regular infinite types, enabling us to characterise the behaviour of persistent objects.

Definition 4.2 (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.*

Operational Semantics. We define a structural operational semantics for the Algebra of Behavioural Types via a labelled transition relation on the types. Assume the variable convention and that types are equal up-to α-conversion.

Definition 4.3 (Labels). *The following grammar defines the set of labels:*

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

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.

Notice that ACT is in fact an axiom-scheme which captures two cases:

1. the basic transition $l(\widetilde{\alpha})$ — execution of a method — is the invocation of a method with name l and parameters of types $\widetilde{\alpha}$, yielding the type α of the object in the body of the method;
2. a silent transition υ — unblocking — releases 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 $\alpha[\mu t.\alpha/t]$.

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

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

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

Notation. Assume the following abbreviations.

1. Let $\mathbf{0}$ 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.
2. The type $l(\widetilde{\alpha})$ denotes $l(\widetilde{\alpha}).\mathbf{0}$, and l denotes $l()$.
3. The *interface* of a type α, denoted $\mathrm{int}(\alpha)$, is the set of its observable labels: in a labelled sum $\sum_{i\in I} l_i(\widetilde{\alpha}_i).\alpha_i$ 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.
4. Let $\Longrightarrow$ denote $\xrightarrow{v}{}^*$, $\xRightarrow{\pi}$ denote $\Longrightarrow\xrightarrow{\pi}\Longrightarrow$, and if $\tilde{\pi}$ is $\pi_1\cdots\pi_n$, then $\xRightarrow{\tilde{\pi}}$ denotes $\xRightarrow{\pi_1}\cdots\xRightarrow{\pi_n}$.

Type Equivalence. Elsewhere we define a notion of equivalence that we call *label-strong bisimulation, lsb* [27]. 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; and
2. if α offers an unblocking transition, then β offers zero or more unblocking transitions.

Definition 4.4 (Bisimilarity on Types).

1. A symmetric binary relation $R \subseteq \mathcal{T}\times\mathcal{T}$ is a *label-strong 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) $\alpha \xrightarrow{v} \alpha'$ 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. The sum of method types and the sum of blocked types preserve bisimilarity. This result is easy to verify, since both sums are guarded. Furthermore, the operators of 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), contravariant on the parameters' types. Thus, if α simulates α', $\alpha\leq\alpha'$, then we say that α is a *subtype* of α'. The simulation is a *subtyping* relation, and the partially ordered set $(\mathcal{T},\leq)$ is a complete lattice having type $\mathbf{0}$ as top. Moreover, the operators of ABT, as well as *lsb*, preserve *subtyping.*

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

5 Type assignment

To ensure that a process is error-free (Definition 3.2), the interactions within the process should be disciplined in such a way that persistent bad redexes do not appear. The types are partial specifications of the behaviour of names in the process, i.e. they 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 this discipline, preventing 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 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.

Definition 5.1 (Typing Judgements). Consider $\gamma \stackrel{\mathrm{def}}{=} (\alpha, \beta)$ and $\vec{\gamma} \stackrel{\mathrm{def}}{=} (\tilde{\alpha}, \widetilde{\beta})$.

1. *Type assignment to names* are formulae $a{:}\gamma$.
2. *Type assignment to process variables* are formulae $X{:}\vec{\gamma}$.
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 $\Gamma \vdash P$. When such a formula holds we say that P is *typable*.

Notation. Let $\mathrm{dom}(\Gamma)$ be the set of the names and the process variables in Γ. Then:

1. $\Gamma \cdot a{:}\gamma$ denotes $\Gamma \cup \{a{:}\gamma\}$ provided that $a \notin \mathrm{dom}(\Gamma)$, and $\Gamma \setminus a$ denotes Γ without the formulae with the name a.
2. Let $\Gamma(a) = \gamma$ if $a{:}\gamma \in \Gamma$, and $\Gamma(a) = (\mathbf{0}, \mathbf{0})$ otherwise; furthermore, $\Gamma(a)_{\mathsf{i}} = \alpha$ and $\Gamma(a)_{\mathsf{o}} = \beta$.
3. Let $\tilde{a}{:}(\tilde{\alpha}, \widetilde{\beta})$ be a sequence of typing assignments, and consider $\Gamma \cdot \tilde{a}{:}\vec{\gamma}, \Gamma(\tilde{a}), \Gamma(\tilde{a})_{\mathsf{i}}$, and $\Gamma(\tilde{a})_{\mathsf{o}}$ defined as above.
4. Consider bisimulation and subtyping on γ-types pairwise defined; hence, assume $\gamma \leq \gamma'$ 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 $\Gamma(a)_{\mathsf{o}} \approx \Gamma'(a)_{\mathsf{o}}$.

Definition 5.2 (Substitution on Typings). A *typing substitution*, $\Gamma[z/x]$, is defined as follows: $(\Gamma \cdot x{:}\gamma)[z/x] = \Gamma \cup \{z{:}\gamma\}$ if $z \notin \mathrm{dom}(\Gamma)$ or $\Gamma(z) \approx \gamma$, and $\Gamma[z/x] = \Gamma$ if $x \notin \mathrm{dom}(\Gamma)$.

Capturing the Notion of Communication Error. A term of the algebra of types defines the semantics of the usage of a name in a process. Therefore, we refer to two syntactic categories of types, input and output, describing names as object locations or message targets. An input-type characterises the sequences of messages an object can accept, and thus the object life-cycle; an output-type characterises the messages sent to an object. Due to the locality condition, the names that are parameters of methods have an empty input-type and a non-empty output-type.

We define co-inductively two new higher-order relations on types, contravariant on the parameters, an *agreement* relation, and a *compatibility* relation. The agreement relation ensures the absence of blockings: if a type denotes a graph, or a rational tree (cf. [28]), 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 — a safety approach to detect bad redexes. 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 ensures the absence of messages-never-understood. It is essentially a relaxing of the previous relation by allowing the environment to "help", providing some transitions: an output-type is compatible with an input-type if the former has a path that coincides with a *projection* of a path of the latter.

Definition 5.3 (Agreement on Types). Fix Γ and P such that $\Gamma \vdash P$.

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. $\exists_{l,\alpha',\beta',\widetilde{\beta}_1,\widetilde{\beta}_2}\ \alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$ and $\beta \xrightarrow{l(\widetilde{\beta}_2)} \beta'$ and $\widetilde{\alpha}_2\ S\ \widetilde{\beta}_1$ for all $a!l[\tilde{v}]$ subterms of P such that $\Gamma(a) \approx (\alpha,\beta)$ and $\Gamma(\tilde{v}) \approx (\widetilde{\alpha}_2,\widetilde{\beta}_2)$, and $\alpha' S \beta'$,
   ii. $\forall_{l,\alpha',\beta',\widetilde{\beta}_1\widetilde{\beta}_2}\ (\alpha \xrightarrow{l(\widetilde{\beta}_1)} \alpha'$ and $\beta \xrightarrow{l(\widetilde{\beta}_2)} \beta'$ and $\widetilde{\alpha}_2\ S\ \widetilde{\beta}_1$ for all $a!l[\tilde{v}]$ subterms of P such that $\Gamma(a) \approx (\alpha,\beta)$ and $\Gamma(\tilde{v}) \approx (\widetilde{\alpha}_2,\widetilde{\beta}_2))$ implies $\alpha' S \beta'$, and
   iii. if $\mathrm{int}(\alpha)=\emptyset$ or $\mathrm{int}(\beta)=\emptyset$ or $\mathrm{int}(\alpha)=\mathrm{int}(\beta)$, then
   $\alpha \xrightarrow{v} \alpha'$ implies $\exists_{\beta'} \beta \Longrightarrow \beta'$ and $\alpha'\ S\ \beta'$, and
   $\beta \xrightarrow{v} \beta'$ 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 and 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 in Definition 5.3 '$\alpha \equiv \mathbf{0}$' with 'α finite' in condition 1.c, 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. Some side conditions of the rules deserve further explanation: 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, in the rule Res, we do not discard the bound variable from the typing. This is because we may need its type-pair to verify an agreement relation.

Notice that, for the sake of simplicity, assume that processes do not have free process variables.

Example 5.4 (Typing Objects).

1. One-place boolean buffer (cf. the example in the introduction):
$$\{b{:}(\mu t.\mathit{write}(\mathit{bool}).\mathit{read}(\mathit{val}).t, \mathbf{0})\} \vdash \mathsf{Empty}[b].$$
   The type of the object clearly specifies its behaviour.
2. Internal non-determinism: consider an object with internal non-determinism, like
$$a?\{l = \nu c\ c!m_1 \mid c!m_2 \mid c?\{m_1 = a?l_1, m_2 = a?l_2\}\}.$$
   The input-type of a expresses the choice: $\Gamma(a)_\mathsf{i} = l.(v.l_1 + v.l_2)$.
3. Persistent objects (cf. [10]): consider the following three objects:
   (a) $(\textsf{rec}\,X(a;).a?\{\mathit{rep} = X[a;] \mid a?\{\mathit{mess}\}\})[a;]$,
   (b) $(\textsf{rec}\,X(b;).b?\{\mathit{quest}(x) = X[b;] \mid x!\mathit{rep}\})[b;]$, and
   (c) $(\textsf{rec}\,X(c;).c?\{\mathit{rep} = X[c;], \mathit{mess} = X[c]\})[c;]$.

   The parallel composition of all the objects together with the messages $b!\mathit{quest}[a]$ and $b!\mathit{quest}[c]$ is an error-free process P. It is easy to check that:
   (a) $\Gamma(a)_\mathsf{i} = \mu t.\mathit{rep}.(t \| \mathit{mess}) \bowtie_\Gamma^P \mathit{rep} = \Gamma(a)_\mathsf{o}$,

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

Table 3: Typing Checking System.

   (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 $\Gamma \stackrel{\text{def}}{=} \{b{:}(\mu t.write(bool).read(val).t, read(val))\}$ and $P \stackrel{\text{def}}{=} \nu b\, \mathsf{Empty}[b] \mid b!read[r]$. Observe that P is an error and $\Gamma(b)_{\mathsf{i}} \not\bowtie^P_\Gamma \Gamma(b)_{\mathsf{o}}$, but $\Gamma \vdash P$.
   (b) Let $\Gamma \stackrel{\text{def}}{=} \{a{:}(m, \upsilon.n), b{:}(l, \mathbf{0})\}$ and $P \stackrel{\text{def}}{=} \nu ab\, b?\{l = a!n\} | a?\{m\}$. Notice that $\Gamma(a)_{\mathsf{i}} \not\bowtie^P_\Gamma \Gamma(a)_{\mathsf{o}}$, whereas P is not an error, and $\Gamma \vdash P$.
   (c) Let $\Gamma \stackrel{\text{def}}{=} \{a{:}(n.m, \upsilon.n \parallel m), b{:}(l, l)\}$ and $P \stackrel{\text{def}}{=} \nu a\, b?\{l = a!n\} \mid a?\{n = a?\{m\}\} \mid a!m \mid b!l$; although P is neither an error nor a deadlock, and $\Gamma \vdash P$, 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 $a?m \mid a?l$; possible input-types of a are $m \parallel l$, $l \parallel m$, $m.l + l.m$, 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 5.5 (Minimal Typing).
1. A typing Γ is *smaller* than a typing Δ, $\Delta \leq \Gamma$, if
   (a) $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Gamma)$, and
   (b) $\Delta(a) \leq \Gamma(a)$, for all $a \in \mathrm{dom}(\Delta)$;
2. The typing Γ is a *minimal typing* for a process P, if
   (a) $\Gamma \vdash P$, and
   (b) $\forall_\Delta\, \Delta \vdash P$ implies $\Delta \leq \approx \Gamma$.

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

Proposition 5.1 (Basic Properties of Minimal Typings).

1. (Existence) A typable process has a minimal typing.
2. (Uniqueness) A minimal typing of a process is unique up to type equivalence.

Proposition 5.2 (Recursive Output Types). Take a minimal Γ for P. If, for some name a, the type $\Gamma(a)_{\mathsf{o}}$ has a subterm of the form $\mu t.\beta$, then β is a blocked sum.

Lemma 5.3 (Basic Properties of Typable Processes). Consider that $\Gamma \vdash P$;

1. (Substitution) If $x \in \mathrm{fn}(P)$ and $\Gamma[z/x]$ is defined, then $\Gamma[z/x] \vdash P[z/x]$.
2. (Congruence) If $P \equiv Q$, then $\Gamma \vdash Q$.
3. (Typability of subterms) If Q is a subterm of P then $\Delta \vdash Q$, for some Δ.

The main purpose of the type system is the detection of potential run-time errors. Being an error is a property of processes which, in this case, cannot be modularly checked. Thus, we cannot introduce side conditions in the rules to check that for each name in a typing the input type is compatible or agrees with the output type. Therefore, after type checking a process, we still have to compute its minimal typing, and to check whether the types of the free names are compatible, and whether the types of the bound names agree.

Definition 5.6 (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 $a \in \mathrm{dom}(\Gamma) \cap \mathrm{fn}(P)$, $\Gamma(a)_{\mathsf{i}} \prec_{\Gamma}^{P} \Gamma(a)_{\mathsf{o}}$;
2. for all $a \in \mathrm{dom}(\Gamma) \cap \mathrm{bn}(P)$, $\Gamma(a)_{\mathsf{i}} \bowtie_{\Gamma}^{P} \Gamma(a)_{\mathsf{o}}$.

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

Theorem 5.4 (Operational Correspondence). Let $\Gamma \vdash P$ and $P \xrightarrow{m} Q$ 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, $\Gamma(a)_{\mathsf{i}} \xrightarrow{l(\widetilde{\beta})} \approx \Delta(a)_{\mathsf{i}}$ where $\Gamma(\tilde{v})_{\mathsf{o}} \approx \widetilde{\beta}$;
3. $m \equiv \tau$, if, and only if, for some $a \in \mathrm{dom}(\Gamma) \cap \mathrm{n}(P)$, $\Gamma(a)_{\mathsf{i}} \xrightarrow{l(\widetilde{\beta}_1)} \Delta(a)_{\mathsf{i}}$ and $\Gamma(a)_{\mathsf{o}} \xrightarrow{l(\widetilde{\beta}_2)} \Delta(a)_{\mathsf{o}}$.

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 the 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 (in other words, the "expectations") of there targets. Recall that the calculus is local, hence no context may create objects located at extruded names.

Theorem 5.5 (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 $\Gamma(a)_{\mathsf{i}} \xrightarrow{l(\widetilde{\beta})} \alpha$ and $\alpha\Gamma(\tilde{v})_{\mathsf{i}} \prec_{\Gamma}^{Q} \Gamma(a)_{\mathsf{o}}\widetilde{\beta}$, then Q is well-typed.

Theorem 5.4 allows to construct the types for Q in the above theorem.

Corollary 5.6. If P is well-typed then P is error-free.

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. The agreement and compatibility relations do not have causality information, thus there are deadlocks which are not errors and are detected (a nice side-effect — cf. Example 5.4.4.b), but there are interesting processes which are rejected (cf. Example 5.4.4.c).

6 Related work

Our 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 [11], 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' [23] 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 [25] 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 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 [33] 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, as usual in type systems, it rejects more processes than those conforming to the notion of error, that is, leading to persistent bad redexes.

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 [16, 29] 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 [34]. 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 it is indeed possible. The problem of encoding π-calculus in TyCO is under investigation [24].

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 [8]. 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. Well-typed processes behave correctly, this being preserved under reduction. Colaço *et al.* [10] 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 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, therefore without dynamic information, and the type system requires complex operations on a lattice of types. The type inference algorithm is a set-constraints algorithm, a well-known technique widely used in functional languages [2]. Nevertheless, the type system does not look as powerful as ours, namely it does not detect deadlocks. Najm and Nimour [22] 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 parametric equations, equipped with a transition relation, and representing infinite state systems. They define an equivalence relation, a compatibility relation, and a subtyping relation on types, based on the simulation and on the bisimulation relations.

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 PRAXIS/P/EEI/120598/98 DiCoMo, as well as by the ESPRIT IV Working Group 22704 ASPIRE.

References

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

[2] Alexander Aiken and Edward L. Wimmers. Type Inclusion Constraints and Type Inference. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 31–41. ACM Press, 1993.

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

[4] Roberto Amadio. An asynchronous model of locality, failure, and process mobility. In *Coordination Languages and Models*, proceedings of COORDINATION'97, volume LNCS 1282. Springer-Verlag, 1997. Extended version as Rapport de Recherche RR-3109, INRIA Sophia-Antipolis, 1997.

[5] 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*, volume LNCS 1119, pages 163–178. Springer-Verlag, 1996.

[6] 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*, volume LNCS 1119, pages 163–178. Springer-Verlag, 1996.

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

[8] Gérard Boudol. Typing the use of resources in a concurrent calculus. In *Advances in Computing Science*, proceedings of ASIAN'97, volume LNCS 1345, pages 239–253. Springer-Verlag, 1997.

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

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

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

[12] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *Proceedings of the International Conference on Concurrency Theory*, volume LNCS 1119, pages 406–421. Springer-Verlag, 1996.

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

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

[15] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science*, 151:437–486, 1995. An extended abstract appeared in *Proceedings of FST/TCS'93*, volume LNCS 761, pages 373–387. Springer-Verlag, 1993.

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

[17] 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.

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

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

[20] 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.

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

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

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

[24] Uwe Nestmann and António Ravara. What's TyCO, after all? Short abstract. To be presented at Express'2000.

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

[26] 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. URL: http://www.tec.informatik.uni-rostock.de/IuK/congr/oosds99/program.html.

[27] 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.

[28] António Ravara and Vasco T. Vasconcelos. Behavioural types for a calculus of concurrent objects. In *Proceedings of the 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.

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

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

[31] 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.

[32] Vasco T. Vasconcelos and António Ravara. Communication errors in the π-calculus are undecidable. *Information Processing Letters*, 71(5–6):229–233, 1999.

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

[34] Nobuko Yoshida. Graph types for monadic mobile processes. In *Proceedings of the 16th FST/TCS Conference*, volume LNCS 1180, pages 371-386. Springer-Verlag, 1996.

A Proof of the Main Results

This appendix presents sketches of the proofs of the main results of the paper. We first show some standard results regarding typings, then we demonstrate the crucial lemmas of minimal typings, and finally, we prove the soundness of our type system.

A.1 On Typings

We adapt to our setting some standard auxiliary results that will be needed to prove Subject-Reduction and its corollary.

Definition A.1 (Restriction of Typings). The restriction of a typing Γ to the set of names N, denoted $\Gamma \restriction N$, is the typing $\{x{:}\Gamma(x) \mid \; x \in N \cap \mathrm{dom}(\Gamma)\} \cup \{X{:}\Gamma(X) \mid \; X \in \mathrm{dom}(\Gamma)\}$.

Lemma A.1 (Finite Additivity).
Let A and B be sets of names. Then:

1. $\Gamma \restriction (A \cup B) = \Gamma \restriction A \cup \Gamma \restriction B$; and
2. $(\Gamma \cup \Delta) \restriction A = \Gamma \restriction A \cup \Delta \restriction A$.

Proof. Simply by applying the above definition. □

Lemma A.2 (Basic Properties of Typings).

1. (Free names) If $\Gamma \vdash P$ then $\mathrm{fn}(P) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{fv}(P) \subseteq \mathrm{dom}(\Gamma)$.
2. (Narrowing) If $\Gamma \vdash P$ then $\Gamma \restriction \mathrm{n}(P) \vdash P$.
3. (Weakening) If $\Gamma \restriction \mathrm{n}(P) \vdash P$ then $\Gamma \vdash P$.

Proof. In all the three items, an induction on the derivation of $\Gamma \vdash P$ carries the proof.

1. (Free names) Straightforward.
2. (Narrowing) There are three base cases: if the deduction is the axiom NIL the result follows trivially; if it is the axiom MSG, the result follows by item 1 of this lemma, noticing that $\Gamma \restriction (a!l[\tilde{v}])$ still fulfils the axiom's side-condition; the last case — that of the axiom VAR — follows similarly. For the induction step, the interesting cases are when the deduction ends with the rules OBJ or REC.
   (a) Rule OBJ: by induction hypothesis, it holds that $\forall_{i\in I}\, (\Gamma \cdot \tilde{x}_i{:}(\mathbf{0}, \widetilde{\beta_i})) \restriction \mathrm{n}(P_i) \vdash P_i$, hence, using Lemma A.1, $\forall_{i\in I}\, \Gamma \restriction \mathrm{n}(P_i) \cdot \tilde{x}_i{:}(\mathbf{0}, \widetilde{\beta_i}) \restriction \mathrm{n}(P_i) \vdash P_i$, and by weakening we finally get $\forall_{i\in I}\, \Gamma \restriction \mathrm{n}(P_i) \cdot \tilde{x}_i{:}(\mathbf{0}, \widetilde{\beta_i}) \vdash P_i$. The proof concludes by applying the rule OBJ, noticing that the side condition holds by hypothesis.
   (b) Rule REC: Similar to the previous case.
3. (Weakening) Straightforward, by induction on the derivation of $\Gamma \restriction \mathrm{n}(P) \vdash P$. Notice that only the names in $\mathrm{dom}(\Gamma) \cap \mathrm{n}(P)$ play a significant role in the typing rules.

□

Corollary A.3.
If $\Gamma \vdash P$ and $\Gamma \subseteq \Delta$ then $\Delta \vdash P$.

Proof. Since $\Gamma \vdash P$ and $\Gamma \subseteq \Delta$, then $\Gamma \restriction \mathrm{n}(P) = \Delta \restriction \mathrm{n}(P)$, hence, by the narrowing lemma, $\Delta \restriction \mathrm{n}(P) \vdash P$ and the result follows by applying the weakening lemma. □

Lemma A.4 (Basic Properties of Typable Processes).
Consider that $\Gamma \vdash P$.

1. (Substitution) If $x \in \mathrm{fn}(P)$ and $\Gamma[z/x]$ is defined, then $\Gamma[z/x] \vdash P[z/x]$.
2. (Congruence) If $P \equiv Q$ then $\Gamma \vdash Q$.
3. (Typability of subterms) If Q is a subterm of P, then $\Delta \vdash Q$, for some Δ.

Proof.

1. (Substitution) We have to show that for each deduction of $\Gamma \vdash P$ there is a deduction of $\Gamma[z/x] \vdash P[z/x]$. We proceed by induction on the derivation of $\Gamma \vdash P$. Notice that, since $x \in \mathrm{fn}(P)$, by the free names' lemma (A.2. 1), $x \in \mathrm{dom}(\Gamma)$; as $\Gamma[z/x]$ is defined by hypothesis, then one of the following two conditions hold:
   (a) either $z \notin \mathrm{dom}(\Gamma)$, so $z \notin \mathrm{fn}(P)$,
   (b) or $\Gamma \stackrel{\text{def}}{=} \Gamma' \cdot x{:}\gamma$, $\Gamma(z) \approx \gamma$, and $(\Gamma' \cdot x{:}\gamma)[z/x] = \Gamma' \cup \{z{:}\gamma\}$.

   If it is the first condition that holds, the proof is straightforward. We assume now that it is the second condition that holds and proceed with the proof. There are three base cases to consider, depending on the axiom: if the deduction is the axiom Nil the result follows trivially;
   (a) axiom Var: assume $\Gamma \stackrel{\text{def}}{=} \Delta \cdot \{x{:}\gamma\}$; by definition, $\Gamma[z/x] = \Delta \cdot \{z{:}\gamma\}$, and $\Delta(z) \approx \gamma$, since $x \in \{\vec{v}\}$; hence, the result follows;
   (b) axiom Msg: since by hypothesis $\Gamma(z) \approx \gamma$, then the result follows by applying rule $\approx$.

   For the induction step, the only interesting cases are when the deduction ends with rule Obj or rule Rec:
   (a) rule Obj: since $x \in \mathrm{fn}(a?M)$, if x is not a, the result follows using the induction hypothesis; otherwise, as by hypothesis $\Gamma(z) \approx \Gamma(a)$, the rule's side condition holds and so, using the induction hypothesis, the result follows again;
   (b) rule Rec: notice that, as $\mathrm{fn}(a?M) = \{\vec{x}\}$, then $(\Gamma \cdot \vec{x}{:}\vec{\gamma})[z/x] = \Gamma \cdot (\vec{x}{:}\vec{\gamma})[z/x]$; since $\approx$ preserves $\leq$, the result follows using the induction hypothesis.
2. (Congruence) By a case analysis of the rules defining $\equiv$.
   (a) Alpha-conversion requires a proof by induction on the derivation of $\Gamma \vdash P$, using the substitution lemma. The only interesting cases are those involving binding operators. When the derivation finishes with the rule Obj, notice that, by the substitution lemma, $(\Gamma \cdot \tilde{x}{:}\tilde{\gamma})[z/x] \vdash P_i[z/x]$, for all $i \in I$, and the result follows by the rule Obj, using the typing Γ/z.
   (b) The case of rule Rec is treated similarly, and rule that of Res simply demands an application of the substitution lemma.
   (c) For the remaining rules, simple rearrangements on the derivation trees of $\Gamma \vdash P$ carry their proofs.
3. (Typability of subterms) Straightforward, by induction on the structure of P.

□

A.2 On Minimal Typings

As mentioned in the motivation to Definition 5.5, a process being typable by a typing is a weaker property than we need. Candidates for "good" typings are those that are minimal. We characterise them herein, showing that typable processes have unique minimal typings, and that the recursive types of output names are blocked sums — an important fact to ensure the soundness result.

Lemma A.5. If Γ is a minimal typing for P then $\Gamma = \Gamma \restriction \mathrm{n}(P)$.

Proof. The inclusion $\subseteq$ is obvious; the opposite results from narrowing and from Definition 5.5. □

Corollary A.6 (Domain of a Minimal Typing).
Let $\Gamma \vdash P$ and Δ be a minimal typing of P. Then, $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Gamma \restriction \mathrm{n}(P))$.

Proof. Follows by the narrowing lemma and the lemma above. □

Proposition A.7 (Basic Properties of Minimal Typings).

1. (Existence) A typable process has a minimal typing.
2. (Uniqueness) A minimal typing of a process is unique up to type equivalence.

Proof.

1. (Existence) Let $\Gamma \vdash P$. Then, by the narrowing lemma and by the above lemma, a minimal typing Δ of P does not have more names than those in P. We proceed by induction on the derivation of $\Gamma \vdash P$. The bases cases are when the derivation is an axiom. The case of the axiom NIL is obvious: $\emptyset$ is the minimal typing.

   (a) MSG: we show that $\Delta \stackrel{\text{def}}{=} \{a{:}(\mathbf{0}, \upsilon.l(\widetilde{\beta})), \tilde{v}{:}(\mathbf{0}, \widetilde{\beta})\}$ is the minimal typing of $a!l[\tilde{v}]$. Notice that $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Gamma \upharpoonright \mathrm{n}(a!l[\tilde{v}]))$. By Lemma 3.2 of the ABT paper [27], there is a normal form β such that $\Gamma(a) \approx \beta$, and by hypothesis $\Gamma(a)_{\mathsf{o}} \Longrightarrow \xrightarrow{l(\Gamma(\tilde{v})_{\mathsf{o}})}$, hence $\beta \Longrightarrow \xrightarrow{l(\widetilde{\beta})}$ with $\Gamma(\tilde{v})_{\mathsf{o}} \approx \beta$. We do now a case analysis on the form of β:

      i. case β is a labelled sum or a parallel composition, then $\beta \leq l(\widetilde{\beta}) \leq \upsilon.l(\widetilde{\beta})$;

      ii. case β is a blocked sum, then $\beta \leq \approx \upsilon.l(\widetilde{\beta})$.

   Clearly, $\upsilon.l(\widetilde{\beta})$ repects the side condition of the axiom. Since $\approx$ respects $\leq$, the result follows.

   (b) VAR: the simplest type for each v in $\vec{v}$ is $\mathbf{0}$, the top of the types lattice, and hence, $\Delta(X) = \vec{t} \approx \vec{\mathbf{0}}$.

   The interesting cases of the induction step are the rules OBJ, REC, and PAR. We construct the minimal typings for each case below; one easely verifies that they are under the conditions of the definition.

   (a) OBJ: notice that $\mathrm{dom}(\Delta) \subseteq \mathrm{dom}(\Gamma \upharpoonright \mathrm{n}(P)) \setminus \bigcup_{i \in I} \{\tilde{x}_i\}$. By hypothesis, $\forall_{i \in I}\, \Gamma \cdot \tilde{x}_i{:}(\mathbf{0}, \widetilde{\beta}_i) \vdash P_i$, hence by induction hypothesis, there are minimal typings $\Delta_i \cdot \tilde{x}_i{:}(\mathbf{0}, \widetilde{\beta}'_i) \vdash P_i$, for each $i \in I$. The minimal typing Δ is such that $\Delta(a)_{\mathsf{i}} \approx \sum_{i \in I} l_i(\widetilde{\beta}'_i).\Delta_i(a)_{\mathsf{i}}$ and $\forall_{i \in I} \forall b \in \mathrm{dom}(\Delta_i)$ with $b \neq a$, $\Delta(b) = \sum_{i \in I} \upsilon.\Delta_i(b)$.

   (b) REC: similar to the previous case, assuming $\Delta(\tilde{v}) \approx \mu t.\vec{\gamma}$ and for any other name $a \in \mathrm{dom}(\Delta)$, $\Delta(a) = \mu t.\Delta'(a)$ where $\Delta' \cdot X{:}\vec{\gamma} \cdot x{:}\vec{\gamma}$ is the minimal typing of $a?M$.

   (c) PAR: make the pairwise parallel composition of minimal typings of the components.

2. (Uniqueness) Suppose there is a typing Γ', different from Γ which is also a minimal typing of P. Then, by definition, $\Gamma \leq \approx \Gamma'$, but since, by hypothesis, Γ minimal, $\Gamma' \leq \approx \Gamma$ holds as well. Therefore, we conclude that $\Gamma \approx \Gamma'$.

$\square$

Proposition A.8 (Recursive Output Types).
Take a minimal Γ for P. If, for some name a, the type $\Gamma(a)_{\mathsf{o}}$ has a subterm of the form $\mu t.\beta$ where t is free in β, then β is a blocked sum.

Proof. Recall that we consider that process variables only occur bound in processes. A case analysis of the type system's rules carries the proof, assuming Γ minimal for P, by Proposition 1. Notice that the side conditions of rule REC require that the types of the free output names of $(\mathbf{rec}\, X(\vec{x}).a?M)[\vec{v}]$ have $\mu t. \sum_{i \in I} \upsilon.\beta_i$ as supertype. $\square$

A.3 Preservation of Typedness

We proceed now with the proof of the soundness of the type system. We establish a correspondence between process transitions and type transitions, and characterise persistent bad redexes in terms of the agreement and the compatibility relations.

Proposition A.9 (Preservation of typedness).
Let $\Gamma \vdash P$. If $P \xrightarrow{m} Q$ then $\Delta \vdash Q$, for some Δ.

Proof. Straightforward, by a case analysis on $P \xrightarrow{m} Q$, using Lemmas 2.2, A.2, and A.4. $\square$

Lemma A.10 (Compatibility).
For all $a \in \mathrm{dom}(\Gamma)$, $\Gamma(a)_{\mathsf{i}} \prec_{\Gamma}^{P|a!l[\tilde{v}]} \Gamma(a)_{\mathsf{o}}$ implies $\Gamma(a)_{\mathsf{i}} \prec_{\Gamma}^{P} \Gamma(a)_{\mathsf{o}}$.

Proof. Directly from the definition. $\square$

Theorem A.11 (Operational Correspondence).
Let $\Gamma \vdash P$ and $P \xrightarrow{m} Q$, and let Δ be a typing of 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, $\Gamma(a)_{\mathsf{i}} \xrightarrow{l(\widetilde{\beta})} \approx\Delta(a)_{\mathsf{i}}$ where $\Gamma(\tilde{v})_{\mathsf{o}} \approx \widetilde{\beta}$;
3. $m \equiv \tau$, if, and only if, for some $a \in \mathrm{dom}(\Gamma) \cap \mathrm{n}(P)$, $\Gamma(a)_{\mathsf{i}} \xrightarrow{l(\widetilde{\beta}_1)} \Delta(a)_{\mathsf{i}}$, and $\Gamma(a)_{\mathsf{o}} \xrightarrow{l(\widetilde{\beta}_2)} \Delta(a)_{\mathsf{o}}$.

Proof. Since $\Gamma \vdash P$, by Proposition A.9, also $\Delta \vdash Q$. Assume Γ and Δ as the minimal typings of P and Q respectively, by Proposition 1. For the 'if' direction we sketch the proof of items 1 and 3 (the proof of item 2 is similar to that of item 1).

1. If $\Gamma(a)_{\mathsf{o}} \xrightarrow{l(\widetilde{\beta})}$ and $\Gamma(\tilde{v})_{\mathsf{o}} \approx \widetilde{\beta}$ then, by the axiom Msg and the rule $\approx$, $\Gamma \vdash a!l[\tilde{v}]$; by rules Par and Res, $\Gamma \vdash P \equiv \nu\tilde{x}\, a!l[\tilde{v}] \mid P'$, and the proof concludes using the 'if' direction of the first item of Lemma 2.2.

3 Using the 'if' direction of items 1 and 2, $P \xrightarrow{\nu\tilde{x}\, a!l[\tilde{v}]}$ with $\Gamma(\tilde{v})_{\mathsf{o}} \approx \widetilde{\beta}_2$, and $P \xrightarrow{a?l[\tilde{v}]}$ with $\Gamma(\tilde{v})_{\mathsf{o}} \approx \widetilde{\beta}_1$; since P is well-typed, $\Gamma(a)_{\mathsf{i}} \prec_{\Gamma}^{P} \Gamma(a)_{\mathsf{o}}$, hence $\widetilde{\beta}_2 \prec_{\Gamma}^{P} \widetilde{\beta}_1$, and the proof concludes using the 'only-if' direction of the appropriate items of Lemma 2.2.

The 'only-if' direction has a dual proof, using the 'only-if' direction of Lemma 2.2, together with the necessary rules of the type system, taking into account that Γ is minimal. $\square$

Proposition A.12 (Bad Redexes).
Let $\Gamma \vdash P$. For some $a \in \mathrm{dom}(\Gamma \restriction \mathrm{n}(P))$,

1. $\Gamma(a)_{\mathsf{i}} \not\bowtie_{\Gamma}^{P} \Gamma(a)_{\mathsf{o}}$, if, and only if, $P \Longrightarrow Q$ and Q has a persistent bad redex;
2. $\Gamma(a)_{\mathsf{i}} \not\prec_{\Gamma}^{P} \Gamma(a)_{\mathsf{o}}$, if, and only if, for some sequence $\widetilde{m}$ and process Q, $P \xrightarrow{\widetilde{m}} Q$ and Q has a persistent bad redex, not generated by $\widetilde{m}$.

Proof. We prove the 'only-if' directions. Reasoning backwards one proves the 'if' direction. Notice that the negation of Definition 5.3 makes $\Gamma(a)_{\mathsf{i}}, \Gamma(a)_{\mathsf{o}} \not\approx \mathbf{0}$ and ensures that some transition of $\Gamma(a)_{\mathsf{i}}$ will not be matched by $\Gamma(a)_{\mathsf{o}}$.

1. If no direct transition of $\Gamma(a)_{\mathsf{o}}$ can match some direct transition of $\Gamma(a)_{\mathsf{i}}$ then, by the 'if' direction of Theorem A.11, P has a persistent bad $al\tilde{v}$-redex. Otherwise the case described above holds for some derivatives of $\Gamma(a)_{\mathsf{o}}$ and $\Gamma(a)_{\mathsf{i}}$, and by Theorem A.11. 3, $P \Longrightarrow Q$; now, either Q has a persistent bad $al\tilde{v}$-redexor Q has a persistent bad $bl\tilde{v}$-redex, for some name b argument of a method of a.
2. Since $\Gamma(a)_{\mathsf{i}} \not\approx \mathbf{0}$ then, for some $\widetilde{\pi}$, $\Gamma(a)_{\mathsf{i}} \xrightarrow{\widetilde{\pi}}$. So by Theorem A.11.2, for all $\widetilde{m}$ with subject a such that $P \xrightarrow{\widetilde{m}} Q$, and in Q the situation of the previous item holds; but the bad redex is not generated by $\widetilde{m}$, as the bad redex always appears (cf. Definition 3.1).

$\square$

Theorem A.13 (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 $\Gamma(a)_{\mathsf{i}} \xrightarrow{l(\widetilde{\beta})} \alpha$ and $\alpha\Gamma(\tilde{v})_{\mathsf{i}} \prec_{\Gamma}^{Q} \Gamma(a)_{\mathsf{o}}\widetilde{\beta}$, then Q is well-typed.

Proof. By a case analysis on $P \xrightarrow{m} Q$, using Proposition A.9 and Lemma A.10. $\square$

Corollary A.14 (Soundness of the Type System).
If P is well-typed then it is error-free.

Proof. Assume that for some Q and $\widetilde{m}$, $P \overset{\widetilde{m}}{\Longrightarrow} Q$ where Q has a bad $al\tilde{v}$-redex not generated by $\widetilde{m}$. We show that there is an R and an $\widetilde{n}$ such that $Q \overset{\widetilde{n}}{\Longrightarrow} R$ where R does not have a bad $al\tilde{v}$-redex. The proof is by induction on the sum of the lengths of $\widetilde{m}$ and $\widetilde{n}$.

Base case: Let $\widetilde{m}$ and $\widetilde{n}$ both have length zero; so $P \Longrightarrow Q$, and by successive applications of the Theorem A.13.1, we conclude that Q is well-typed. Thus, by Proposition A.7.1, take Γ as the minimal typing of Q; since we assume that Q has a bad $al\tilde{v}$-redex, it holds that

$$\Gamma(a)_{\mathsf{i}}, \Gamma(a)_{\mathsf{o}} \neq \mathbf{0}, \text{ and } l(\widetilde{\beta}) \in \mathrm{int}(\Gamma(a)_{\mathsf{o}}), l(\widetilde{\beta}) \notin \mathrm{int}(\Gamma(a)_{\mathsf{i}}) \neq \mathbf{0} \text{ where } \tilde{v}{:}(\_, \widetilde{\beta}). \tag{1}$$

To proceed, we consider separately the cases of a being bound or free.

Case $a \in \mathrm{bn}(Q)$**.** $\Gamma(a)_{\mathsf{i}} \bowtie_{\Gamma}^{Q} \Gamma(a)_{\mathsf{o}}$, so there is an $l'(\widetilde{\beta'})$ with $l' \neq l$ or $|\widetilde{\beta'}| \neq |\tilde{v}|$, such that conditions 1b.1(b)i and 1b.1(b)ii of Definition 5.3 hold (notice that condition 1b.1(b)iii holds trivially due to the assertion 1 above). By Theorem A.11. 3, $Q \xrightarrow{v} S$ and S is well-typed again, by Theorem A.13.1. If S has no $al\tilde{v}$-redex, or if it has a good one, the proof is complete. Otherwise, take Δ as the minimal typing of S and notice that, as $a \in \mathrm{bn}(Q)$, also $a \in \mathrm{bn}(S)$, by Lemma 2.1.1, hence $\Delta(a)_{\mathsf{i}} \bowtie_{\Delta}^{S} \Delta(a)_{\mathsf{o}}$. Thus, $\Delta(a)_{\mathsf{o}} \overset{v}{\not\rightarrow}$, so by Proposition A.8, $\Delta(a)_{\mathsf{o}}$ is finite. Therefore, applying this same reasoning a finite number of times leads to the disappearance of the bad $al\tilde{v}$-redex.

Case $a \in \mathrm{fn}(Q)$**.** $\Gamma(a)_{\mathsf{i}} \prec_{\Gamma}^{Q} \Gamma(a)_{\mathsf{o}}$, so there is a $\widetilde{\pi}$ and an $l'(\widetilde{\beta'})$, with $l' \neq l$ or $|\widetilde{\beta'}| \neq |\tilde{v}|$, such that conditions 1b.1(b)i and 1b.1(b)ii of Definition 5.3 hold (again, condition 1b.1(b)iii holds trivially due to the assertion 1 above). Since $\Gamma(a)_{\mathsf{i}} \xrightarrow{\widetilde{\pi}}\xrightarrow{l'(\widetilde{\beta'})}$, by Theorem A.11.2, there is a sequence $\widetilde{n}'$ of input actions with subject a such that $Q \xrightarrow{\widetilde{n}'} S$. However, by hypothesis, $\widetilde{n}'$ has length zero, thus $Q \equiv S$, and the proof follows similarly to the previous case, but noticing that, as $a \in \mathrm{fn}(Q)$, also $a \in \mathrm{fn}(S)$, by Lemma 2.1.2.

Induction step: We still have two cases to consider.

Case $\widetilde{m}$ has length zero, but $\widetilde{n}$ has length greater than zero. Let $P \Longrightarrow Q \xrightarrow{a?l[\tilde{v}]} S$ where Q is well-typed by Theorem A.13.1. Since $a \in \mathrm{fn}(Q)$, $\Gamma(a)_{\mathsf{i}} \prec_{\Gamma}^{Q} \Gamma(a)_{\mathsf{o}}$, for some well-typing Γ, and by Theorem A.11.2, $\Gamma(a)_{\mathsf{i}} \xrightarrow{l(\widetilde{\beta})} \alpha$ where $\tilde{v}{:}(\_, \widetilde{\beta})$; if $\alpha\Gamma(\tilde{v})_{\mathsf{i}} \prec_{\Gamma}^{Q} \Gamma(a)_{\mathsf{o}}\widetilde{\beta}$, by Theorem A.13.3, S is well-typed, and the result follows using the induction hypothesis; otherwise, by Proposition A.12.2, $S \xrightarrow{\widetilde{n}} R$ which has a persistent bad redex, not generated by $\widetilde{n}$. However, the input action $a?l[\tilde{v}]$ is generated by the bad redex from Q, and then we are not under the conditions of Definition 3.2, so the proof is complete.

Case $\widetilde{m}$ and $\widetilde{n}$ both have length greater than zero. The proof is similar to that of the previous case.

$\square$