Behavioural Types for a Calculus of Concurrent Objects*

António Ravara
amar@math.ist.utl.pt
Departamento de Matemática
Instituto Superior Técnico
Universidade Técnica de Lisboa

Vasco T. Vasconcelos
vv@di.fc.ul.pt
Departamento de Informática
Faculdade de Ciências
Universidade de Lisboa

June 97

Abstract

We present a new type system for TyCO, a name-passing calculus of concurrent objects. The system captures dynamic aspects of objects' behaviours, namely non-uniform service availability of objects. The notion of processes without errors is loosened, demanding only weak fairness in the treatment of messages.

1 Motivation

The role of name-passing process algebras in the study of the foundations of concurrent programming had been stressed by several authors (*c.f.* [KY95, PT95]). The advantages of the object-oriented paradigm are also well know, and there are several proposals of name-passing calculus with such characteristics, as for example [Pie95]. In all of them one can find a static type system enjoying the subject-reduction property (invariance of a type under reduction) and having as corollary the absence of run-time errors for well-typed processes.

Types in object-oriented languages are "partial specifications of object's behaviour", and they give information about the object's interface; in name-passing process algebras types are used as a discipline for communication, and they are assigned to names; in an object-oriented calculus names are possible locations of objects and the type of a name captures aspects of the object's semantics.

In a setting where (active) concurrent objects are characterized by *non-uniform service availability* [Nie95], a static "types-as-interfaces" approach is not suitable to capture dynamic aspects of objects' behaviours, namely the dependency of the objects' interface from its internal state. We propose types as graphs (representing objects as state-transition systems), and demand weak fairness in the treatment of messages, in the sense that a request may be respond not immediately but sometime in the future. The paper presents a type system that is able to type objects with a non-uniform service availability, while preserving the subject-reduction property.

*This is a full version of a paper to appear in Euro-Par'97.

A typical process not typable by "traditional" type systems [VH93, VT93, KY95, LW95] is a one-place buffer that only allows read operations when it is full, and write operations when it is empty.

$$\begin{array}{rcl} \mathrm{Empty}(b) & = & b \triangleright [\, \mathit{write} : (u)\, \mathrm{Full}(b\, u)\,] \\ \mathrm{Full}(b\, u) & = & b \triangleright [\, \mathit{read} : (r)\; r \triangleleft \mathit{val} : [u] \mid \mathrm{Empty}(b)\,] \end{array}$$

The type systems mentioned above assign interface-like types to names. Therefore, name b should have a single interface, containing both methods' labels *write* and *read*, and thus the example presented can not be typed. Nevertheless, the behaviour of the process (alternating between *write* and *read* operations) is very clear. Furthermore, a process containing the redex $\mathrm{Empty}(b) \mid b \triangleleft \mathit{read}\!:\![r]$ should not be considered an error, for the presence of a message $b \triangleleft \mathit{write} : [u]$ makes the reception of the *read* message possible.

The development of a type system able to type processes like the one above is the main motivation of this work.

2 The calculus of objects

TyCO (TYped Concurrent Objects) [VT93, Vas94] is an object-oriented name-passing calculus with asynchronous communication between concurrent objects via labelled messages carrying names. The calculus is developed along the trends of well-known models of concurrency, such as the π-calculus [Mil91, MPW92], the ν-calculus [HT91], and the actor model of computation [HBS73, Agh86].

2.1 Syntax

Consider *names* $u, v, x, y \in \mathcal{N}$, *labels* $\mathsf{a,b,c} \in \mathcal{L}$, and *processes* $P, Q \in \mathcal{P}$. Let $\tilde{v}$ stand for a sequence of names, and $\tilde{x}$ for a sequence of pairwise distinct names.

Definition 2.1 The set $\mathcal{P}$ of *processes* is given by the following grammar.

$$P \quad ::= \quad x \triangleright M \quad \mid \quad x \triangleleft m \quad \mid \quad P \mid Q \quad \mid \quad \nu x\, P \quad \mid \quad !x \triangleright M \quad \mid \quad \mathbf{0}$$

where $M \stackrel{\mathrm{def}}{=} \sum_{i\in \mathrm{I}} \mathsf{a}_i : (\tilde{x}_i)\, P_i$ for I a finite index set, and $m \stackrel{\mathrm{def}}{=} \mathsf{a} : [\tilde{v}]$.

The basic processes are objects $x \triangleright M$, located at some name x and composed of a finite collection M of labelled methods (with pairwise distinct labels), and asynchronous labelled messages $x \triangleleft \mathsf{a}\!:\![\tilde{v}]$, targeted at some object's location x and selecting its method a with actual parameters $\tilde{v}$. Each method $\mathsf{a} : (\tilde{x})\, P$ is labelled by a distinct label a, has formal parameters $\tilde{x}$ and body P. The other constructors are the concurrent composition of processes, the restriction of the scope of a name to a process, the replication of objects, and the terminated process.

For typing purposes, we impose one important restriction on processes: the formal parameters $\tilde{x}$ in a method $\mathsf{a}\!:\!(\tilde{x})\, P$ are not allowed to be locations of objects in the body P. Such a restriction is present in most object-oriented languages, where the creation of a new name and a new object are tightly coupled.

Notation 2.2 We abbreviate a method $a : ()\,\mathbf{0}$ to a, and a process $\nu x_1 \cdots \nu x_n\, P$ to $\nu \tilde{x}\, P$.

Definition 2.3 An occurrence of a name x in a process P is *bound* if it is in a part of P with the form $a : (\tilde{w} x \tilde{y})\, Q$ or $\nu x\, Q$; otherwise the occurrence of x is *free*. The set $fn(P)$ of *free names* in a process P is defined accordingly, and so is *alpha-conversion*, denoted by $\equiv_\alpha$. The process $P[\tilde{v}/\tilde{x}]$ denotes the simultaneous substitution of the free occurrences of $\tilde{x}$ in P by $\tilde{v}$, defined only when $\tilde{x}$ and $\tilde{v}$ have the same length.

2.2 Operational semantics

Definition 2.4 *Structural congruence* is the smallest congruence relation over processes generated by the following rules.

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

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

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

Definition 2.5 The result $M \bullet m$ of applying a communication m to a collection of methods M is the process $P[\tilde{v}/\tilde{x}]$ if m is of the form $a : [\tilde{v}]$ and $a : (\tilde{x}) P$ is a method in M and the substitution is defined.

Definition 2.6 *One-step reduction* $\rightarrow$ is the smallest relation over processes generated by the following rules.

$$\begin{array}{llll} \textsc{Com} & x \triangleright M \mid x \triangleleft m \rightarrow M \bullet m & \textsc{Rep} & !x \triangleright M \mid x \triangleleft m \rightarrow !x \triangleright M \mid M \bullet m \\ \textsc{Par} & P \mid R \rightarrow Q \mid R \ \text{ if } P \rightarrow Q & \textsc{Res} & \nu x\, P \rightarrow \nu x\, Q \ \text{ if } P \rightarrow Q \end{array}$$

$$\textsc{Str} \qquad P' \rightarrow Q' \ \text{ if } P' \equiv P,\ \ P \rightarrow Q,\ \ Q \equiv Q'$$

Reduction $\twoheadrightarrow$ is the relation $\equiv \cup \rightarrow^+$, where $\rightarrow^+$ denotes the transitive closure of $\rightarrow$.

2.3 Error-processes

In order to deal with objects' non-uniform service availability we need to loosen the notion of process with error. The new concept of error requires two further notions.

Definition 2.7 A *context* $\mathbf{C}$ is the concurrent composition of messages and a constant $[]$ (called the *hole*).

Filling the hole of a context $\mathbf{C}$ with a process P results in the process $\mathbf{C}[P]$.

Definition 2.8 1. A process P has a (replicated) *x-redex* if $P \equiv \nu \tilde{u}\, (!x \triangleright M \mid x \triangleleft m \mid Q)$.
2. A process P has a *bad x-redex* if P has an x-redex and $M \bullet m$ is not defined.

Definition 2.9 A process P is an *error*, notation $P \in \textsc{Err}$, if

1. $\exists_{\mathbf{C}}\ \mathbf{C}[P] \twoheadrightarrow Q$, and Q has a bad x-redex, for some x not in $\mathbf{C}$, and
2. $\forall_{\mathbf{C}'}\ \mathbf{C}'[Q] \twoheadrightarrow R$, and R has a bad x-redex.

Errors are processes with bad x-redexes that persist throughout reduction. A occasional bad x-redex is not enough to make the process an error. So, we give messages a chance to find their target, and therefore, we say that this calculus have weak fairness in the treatment of messages.

Example 2.10 *Error and non-error processes*

1. $S \stackrel{\text{def}}{=} \text{Empty}(x) \,|\, x \lhd \mathit{read}\!:\![u] \notin \textsc{Err}$ since, although the process S is a bad x-redex, we have $\mathbf{C}[S] \to \text{Full}(x\ v) \,|\, x \lhd \mathit{read}\!:\![u]$ for $\mathbf{C} \stackrel{\text{def}}{=} x \lhd \mathit{write}\!:\![v] \mid []$ containing no bad x-redexes;

2. $P \stackrel{\text{def}}{=} y \rhd [\, b : x \lhd \mathrm{a} \,] \,|\, !x \rhd [\, c \,] \in \textsc{Err}$ since $\mathbf{C}[P] \to\ x \lhd \mathrm{a} \,|\, !x \rhd [\, c \,]$ with $\mathbf{C} \stackrel{\text{def}}{=} y \lhd b \,|\, []$ and no context can undo the bad x-redex;

3. $Q \stackrel{\text{def}}{=} z \rhd [\, e : (y)\, y \lhd \mathrm{a} \,] \,|\, w \rhd [\, l : !u \rhd [\, c \,] \,] \notin \textsc{Err}$, although we have $\mathbf{C}[Q] \twoheadrightarrow R \stackrel{\text{def}}{=} u \lhd \mathrm{a} \,|\, !u \rhd [\, c \,]$ for $\mathbf{C} \stackrel{\text{def}}{=} z \lhd e\!:\![u] \,|\, [] \,|\, w \lhd l$, and R is a persistent bad redex; but note that u is in $\mathbf{C}$, violating the condition in the definition, meaning that the environment is the generator of the error.

3 The type system

Processes are implicitly typed: although no type information is present in processes, it can be inferred by a type system that assigns type to names and sets of name-type pairs (called typings) to processes.

3.1 Types

A type is a *graph* whose nodes (states) can be interpreted as an object's interface and the arcs (transitions) as the invoked methods. The type of an object represents its possible life-cycles.

Definition 3.1 The set $\mathcal{T}$ of *types* is inductively defined as follows.

1. $\mathcal{D} \subseteq \mathcal{T}$, for $\mathcal{D}$ an initial algebra of some fixed data types;
2. $(V, I, A) \subseteq \mathcal{T}$, where V is a nonempty set of nodes, $I \subseteq V$ is a nonempty set of initial nodes, and $A \subseteq V \times (\mathcal{L} \times \mathcal{T}^*) \times V$ is a set of arcs labelled by $\mathcal{L} \times \mathcal{T}^*$. Graphs are directed and contain no isolated nodes. We further require that a graph with more than one initial node is the disjoint union of connected components, one for each initial node.

Notation 3.2 We use α, β, γ to denote types. For a given graph α, V_α denotes its set of nodes, I_α denotes its set of initial nodes, and A_α denotes its set of arcs; the label of an arc is denoted by t. Graphs are consider equal up to isomorphism on nodes.

We need also a notion of terminal nodes of a graph to define complete paths. A node is terminal if there are no arcs starting from it or otherwise the arcs end at an initial node.

Definition 3.3 The set T_α of *terminal nodes* of a graph α is the set

$$\{v \in V_\alpha \setminus I_\alpha \mid \nexists_{u \in V_\alpha}\, (v, t, u) \in A_\alpha, u \neq v\} \cup \{v \in V_\alpha \setminus I_\alpha \mid \exists_{u \in I_\alpha}\, (v, t, u) \in A_\alpha\}.$$

The union of graphs is a sum of behaviours. A graph that is the disjoint union of connected components represents a set of possible behaviours of an object, each behaviour represented by a connected subgraph.

Definition 3.4 The *union* $\alpha \uplus \beta$ of types α and β is the type γ such that

1. if α, β are graphs, then

$$\gamma \stackrel{\text{def}}{=} \begin{cases} (V_\alpha \cup V_\beta, I_\alpha \cup I_\beta, A_\alpha \cup A_\beta), \text{ if } V_\alpha \cap V_\beta = \emptyset \\ (V_\alpha \cup V_\beta, I_\alpha \setminus (V_\beta \setminus I_\beta) \cup I_\beta \setminus (V_\alpha \setminus I_\alpha), A_\alpha \cup A_\beta), \text{ otherwise;} \end{cases}$$

2. if $\alpha, \beta \in \mathcal{D}$ and $\alpha = \beta$, then $\gamma \stackrel{\text{def}}{=} \alpha$;
3. the union is undefined otherwise.

A graph that is the product of two graphs represents the joint behaviour (parallel composition or interleaving) of two objects located at a same name.

Definition 3.5 The *interleaving* $\alpha \parallel \beta$ of graphs α and β is the graph γ such that

1. $V_\gamma \stackrel{\text{def}}{=} V_\alpha \times V_\beta$, and $I_\gamma \stackrel{\text{def}}{=} I_\alpha \times I_\beta$, and
2. $A_\gamma \stackrel{\text{def}}{=} \{(uv, t, u'v) \mid \forall_{(u,t,u') \in A_\alpha} \exists_{v \in V_\beta}\} \cup \{(uv, t, uv') \mid \forall_{(v,t,v') \in A_\beta} \exists_{u \in V_\alpha}\}$.

Types abstract from objects' concrete behaviour: two different objects with equivalent behaviours have the same type. The equivalence relation is a pair of binary relations over types, one over the nodes of the graph and the other over the types labelling the graph's arcs.

Definition 3.6 *Bisimilarity on types.*

1. A symmetric binary relation $\mathcal{R} \subseteq V_\alpha \times V_\beta$ is a *bisimulation on graphs* (over a binary relation $\mathcal{C}$ on types) if $\forall_{u \in V_\alpha} \forall_{v \in V_\beta}$ such that $u\mathcal{R}v$, if $(u, \mathrm{a} : \tilde{\alpha}, u') \in A_\alpha$ then $\exists_{(v,\mathrm{a}:\tilde{\beta},v') \in A_\beta}$ with $u'\mathcal{R}v'$ and $\tilde{\alpha}\mathcal{C}\tilde{\beta}$[1].
2. Two nodes $u \in V_\alpha$, $v \in V_\beta$ are *bisimilar over* $\mathcal{C}$, denoted by $u \sim_{\mathcal{C}} v$, if there is a bisimulation $\mathcal{R}$ over $\mathcal{C}$ such that $u\mathcal{R}v$.

Compatibility of types.

1. A symmetric binary relation $\mathcal{C} \subseteq \mathcal{T} \times \mathcal{T}$ is a *type compatibility*, if $\alpha\mathcal{C}\beta$ implies $\forall_{u \in I_\alpha} \exists_{v \in I_\beta}\, u \sim_{\mathcal{C}} v$[2] when α, β are graphs, or $\alpha = \beta$ otherwise.
2. Two types α and β are *compatible*, denoted by $\alpha \sim \beta$, if there exists a type compatibility relation $\mathcal{C}$ such that $\alpha\mathcal{C}\beta$.

The compatibility relation $\sim$ is the largest type compatibility; thus, two graph types are compatible if all their nodes are bisimilar over the compatibility relation.

One can easily observe that $\mathcal{T}/\sim$ specifies a class of process *behaviours*. Moreover, $\sim$ is a congruence with respect to $\parallel$ and $\uplus$.

Lemma 3.7 Suppose α, β and γ are graphs, and $\alpha \sim \beta$; then $(\alpha \parallel \gamma) \sim (\beta \parallel \gamma)$ and $(\alpha \uplus \gamma) \sim (\beta \uplus \gamma)$.

---

[1]Let $\alpha_1 \cdots \alpha_k \mathcal{C} \beta_1 \cdots \beta_k \stackrel{\text{def}}{=} \alpha_1 \mathcal{C} \beta_1 \wedge \cdots \wedge \alpha_k \mathcal{C} \beta_k$.

[2]This condition is enough to guaranty the bisimilarity of the graphs, since graphs do not have unreachable nodes.

Proof 1. It is trivial to show the conditions of the definition, since the type constructor is simply a disjoint union.

2. The proof is standard; let $\mathcal{S} = \{((\alpha \parallel \gamma), (\beta \parallel \gamma)) \mid \alpha \sim \beta\}$; using the definition of $\parallel$ it is straightforward to show that $\mathcal{S}$ is a bisimulation on graphs. ∎

Paths in graphs play an important role in our system. We compare types to know if a message can be accepted by an object.

Definition 3.8 *On paths of a graph.*

1. A *path* μ_α in a graph α is a chain of arcs in α of the form $(u_0, t_1, u_1), \ldots, (u_{n-1}, t_n, u_n)$ with $n \geq 1$. We write $\mu_\alpha^{u,v}$ to denote the path starting at node u and ending at node v.

2. A path $\mu_\alpha^{u,v}$ is *complete* if $u \in I_\alpha$, and $v \in T_\alpha$ or $v \in I_\alpha$ if $\exists_{w \in T_\alpha} (w, t, v) \in A_\alpha$.

3. For a path μ_α, a sequence of some of its arcs preserving the original ordering is called a *projection* of μ_α.

To characterize how graphs evolve with the reduction of processes we need the notion of subgraphs. This notion captures a transition relation on graphs.

Definition 3.9 A graph α is a *subgraph* of a graph β, denoted by $\alpha \leq \beta$, if $\alpha = \beta$ or $\exists_{u \in I_\beta} (u, t, v) \in A_\beta$ with $v \in I_\alpha$ and each path of β starting in v is also a path of α.

The transitive closure of $\leq$ is denoted by $\leq^+$.

Lemma 3.10 If α, β, γ are graphs and $\alpha \leq^+ \beta$, then $(\alpha \parallel \gamma) \leq^+ (\beta \parallel \gamma)$.

Proof Let $u \in I_{(\beta \parallel \gamma)}$, $(u, t, v) \in A_{(\beta \parallel \gamma)}$, and suppose $t \in A_\beta$; then, by hypothesis, $v \in I_\alpha$ and each path of β starting in v is also a path of α; since $t \in A_\beta$ then $v \in I_\gamma$ and by definition of $\parallel$, $v \in I_{\alpha \parallel \gamma}$, and the condition on the paths follows. ∎

Note that we do not have $\alpha \leq^+ \beta \Rightarrow (\alpha \uplus \gamma) \leq^+ (\beta \uplus \gamma)$, since no path starting in some element of I_α is a path of γ. But it is easy to show that $\alpha \leq^+ \beta \Rightarrow \alpha \leq^+ (\beta \uplus \gamma)$.

Types for replicated objects are obtained from a finite graph by means of a fix point operation.

Prop/Definition 3.11 *Replication of a graph.*

1. $\alpha_0 \stackrel{\text{def}}{=} (V_\alpha \setminus T_\alpha, I_\alpha, A_{\alpha_0})$, where
$$A_{\alpha_0} \stackrel{\text{def}}{=} \{(u, t, v) \mid (u, t, w) \in A_\alpha, \text{ and } v = w \text{ if } w \notin T_\alpha \text{ or } v = u \text{ otherwise}\}.$$
2. $\mathcal{F}(\alpha_0) = \biguplus_{v \in V_{\alpha_0} \setminus I_{\alpha_0}} \alpha_0 \sigma_v \uplus \alpha_0$, and $\mathcal{F}(\alpha_{i+1}) = \biguplus_{v \in V_{\alpha_i} \setminus V_{\alpha_{i-1}}} \alpha_0 \sigma_v \uplus \alpha_i$ with $i \geq 1$, where, for each $v \in V_{\alpha_i} \setminus V_{\alpha_{i-1}}$, σ_v is the substitution
$$\sigma_v \stackrel{\text{def}}{=} \begin{cases} I_{\alpha_0} \mapsto \{v\} \\ V_\alpha \setminus I_\alpha \mapsto \{w \mid \text{for each } u \in V_\alpha \setminus I_\alpha,\ w \text{ is fresh}\}. \end{cases}$$
3. The replication $\mathit{repl}(\alpha)$ of a finite graph α is the graph $\mathit{fix}(\mathcal{F}(\alpha_0))$.
4. One can easily see that $\mathcal{F}$ is a continuous function since it is increasing by definition, and it is monotonous since if $\alpha_0 \subseteq \beta_0$ then $\mathcal{F}(\alpha_0) \subseteq \biguplus_{v \in V_{\alpha_0} \setminus I_{\alpha_0}} \alpha_0 \sigma_v \uplus \beta_0 \subseteq \mathcal{F}(\beta_0)$ and, similarly, $\mathcal{F}(\alpha_i) \subseteq \mathcal{F}(\beta_i)$ for some $i \geq 1$.

3.2 Typings

The *name-usage-type* triple $x^*\!:\!\alpha$, with $* \in \{!, \downarrow, \uparrow\}$, is a formula denoting the assignment of type α to name x, location of an object ($\downarrow$), location of a replicated object ($!$), or destination of a message ($\uparrow$).

Definition 3.12 A *typing* Γ is a finite set of name-usage-type triples that has at most two occurrences of the same name, one as an object (replicated or not) and a second as a message.

Example 3.13

1. The message $x \lhd \mathrm{a} : \tilde{u}$ has a typing $\{x^{\uparrow} : (\{u_0, u_1\}, \{u_0\}, \{(u_0, \mathrm{a} : \tilde{\alpha}, u_1)\}), \tilde{u}\!:\!\tilde{\alpha}\}$. Graphically $\{x^{\uparrow}\!: \begin{tikzcd} \odot \arrow[d, "a:\tilde{\alpha}"] \\ \bullet \end{tikzcd}, \tilde{u}\!:\!\tilde{\alpha}\}$, where $\odot$ denotes an initial node.

2. The object $x \rhd [\,\mathrm{a}\,]$ has a typing $\{x^{\downarrow} : \begin{tikzcd} \odot \arrow[d, "a"] \\ \bullet \end{tikzcd}\}$. The input-name $x^{\downarrow}$ has the type

   $\alpha_1 = \begin{tikzcd} & \odot \arrow[dl, "a"'] \arrow[dr, "b"] & \\ \bullet & & \bullet \end{tikzcd}$ in $x \rhd [\,\mathrm{a}\ +\ b\,]$, and $\alpha_2 = \begin{tikzcd} \odot \arrow[d, "a"'] \\ \bullet \arrow[d, "b"'] \\ \bullet \end{tikzcd}$ in $x \rhd [\,\mathrm{a} : x \rhd [\,b\,]\,]$.

3. The input-name $x^{\downarrow}$ has the type $\beta = \begin{tikzcd} & \odot \arrow[d, "a"] & \\ & \bullet \arrow[dl, "b"'] \arrow[dr, "c"] & \\ \bullet & & \bullet \end{tikzcd}$ in $x \rhd [\,\mathrm{a} : x \rhd [\,b + c\,]]$; the type

   $\gamma = \begin{tikzcd} & \odot \arrow[dl, "a"'] \arrow[dr, "a"] & \\ \bullet \arrow[d, "b"'] & & \bullet \arrow[d, "c"] \\ \bullet & & \bullet \end{tikzcd}$ in $x \rhd [\,\mathrm{a} : \texttt{if}\ y\ \texttt{then}\ x \rhd [\,b\,]\ \texttt{else}\ x \rhd [\,c\,]\,]$ [3], and the type

   $\delta = \begin{tikzcd} & \odot \arrow[dl, "a"'] \arrow[dr, "a"] & \\ \bullet \arrow[d, "b"'] & & \bullet \arrow[d, "b"] \\ \bullet & & \bullet \end{tikzcd}$, in $x \rhd [\mathrm{a} : \texttt{if}\ g\ \texttt{then}\ x \rhd [b]\ \texttt{else}\ x \rhd [b]]$.

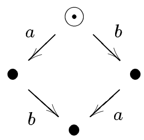
   Notice that $\beta \not\sim \gamma$ but $\alpha_2 \sim \delta$.

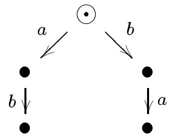
4. The input-name $x^{\downarrow}$ has the type $\begin{tikzcd} \odot \arrow[d, "c"'] & \odot \arrow[d, "d"] \\ \bullet & \bullet \end{tikzcd}$ in $y \rhd [\,\mathrm{a} : x \rhd [\,c\,] + b : x \rhd [\,d\,]]$; the graph is the union of two connected components, each one representing a possible behaviour of the name x.

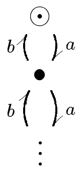
5. Until now we have only dealt with messages and objects. The parallel composition of processes poses new problems.

---

[3] $\texttt{if}\ y\ \texttt{then}\ P\ \texttt{else}\ Q \stackrel{\text{def}}{=} \nu z\,(y \lhd \mathit{val} : z, z \rhd [\,\mathit{true} : P\ +\ \mathit{false} : Q\,])$

The input-name $x^{\downarrow}$ has the type  in $x \rhd [a] | x \rhd [b]$, and the type

 in $x \rhd [a : x \rhd [b] \ + \ b : x \rhd [a]]$. The first graph results from the interleaving of the graphs of the two objects. Intuitively, each path represents a possible sequence of actions. The types are bisimilar.

6. Replication has the consequence that graph types are no longer acyclic nor finite. The rule uses the greatest fix point of a function that, in each terminal node of a graph "hangs" that graph; for example, in $!x \rhd [a : x \rhd [b]]$, the input-name $x^!$ have the type .

Notation 3.14 Let $dom(\Gamma)$ be the set of the name-usage pairs in each triple of Γ; then $\Gamma \cdot x^* : \alpha$ denotes the union of Γ and $\{x^* : \alpha\}$, provided $x^* \notin dom(\Gamma)$. For $x^* : \alpha \in \Gamma$ let $\Gamma(x^*) \stackrel{\text{def}}{=} \alpha$; let $\Gamma \setminus x^*$ denote the typing Γ without the occurrences of formulas with x^*. Let $\Gamma \upharpoonright P$ denote the restriction of Γ to the free names in P, and let $\Gamma[z/x]$ denote the result of replacing in Γ occurrences of x by the fresh name z.

If the parallel composition of two processes has a bad x-redex then the processes should not be composed. In this case we say that their respective typings are not compatible. But if the graphs of all output-names in a typing have some complete path that match with a projection of some path of the respective replicated input-name in the other typing, then there are no persistent bad x-redexes and the typings are compatible.

Definition 3.15 Two typings Γ and Δ are *compatible*, denoted by $\Gamma \asymp \Delta$, if $\Gamma(x^!)$ has a projection equal to some complete path of $\Delta(x^{\uparrow})$, and vice-versa.

The notions of union and interleaving extend to compatible typings.

Definition 3.16 The *union* $\Gamma \uplus \Delta$ of two compatible typings is the typing:

1. $\Gamma \cup \Delta$, if $dom(\Gamma) \cap dom(\Delta) = \emptyset$;
2. $(\Gamma \cdot x^* : \gamma) \uplus (\Delta \setminus x^*)$, if $\Gamma(x^*) = \alpha$, $\Delta(x^*) = \beta$, and $\gamma \stackrel{\text{def}}{=} \begin{cases} \alpha, & \text{if } \alpha \sim \beta \\ \alpha \uplus \beta, & \text{otherwise;} \end{cases}$
3. $\{x^* : \alpha, x^{\uparrow} : \beta\} \cup ((\Gamma \setminus x^*) \uplus (\Delta \setminus x^{\uparrow}))$, if $\Gamma(x^*) = \alpha$, $\Delta(x^{\uparrow}) = \beta$, and $* \in \{!, \downarrow\}$;
4. $\{x^! : \alpha\} \cup ((\Gamma \setminus x^!) \uplus (\Delta \setminus x^{\downarrow}))$, if $\Gamma(x^!) = \alpha$, $\Delta(x^{\downarrow}) = \beta$, and $\alpha = repl(\gamma)$, for some γ such that $\gamma \sim \beta$;
5. $\{x^! : \alpha, x^{\downarrow} : \beta\} \cup ((\Gamma \setminus x^!) \uplus (\Delta \setminus x^{\downarrow}))$, if $\Gamma(x^!) = \alpha$ and $\Delta(x^{\downarrow}) = \beta$.

Definition 3.17 The *interleaving* $\Gamma \parallel \Delta$ of two compatible typings is the typing:

1. $\Gamma \cup \Delta$, if $dom(\Gamma) \cap dom(\Delta) = \emptyset$;
2. $(\Gamma \cdot x^* : \gamma) \parallel (\Delta \setminus x^*)$, if $\Gamma(x^*) = \alpha$, $\Delta(x^*) = \beta$, and $\gamma \stackrel{\text{def}}{=} \begin{cases} \alpha \parallel \alpha, & \text{if } \alpha \sim \beta \\ \alpha \parallel \beta, & \text{otherwise;} \end{cases}$
3. $\{x^* : \alpha, x^\uparrow : \beta\} \cup ((\Gamma \setminus x^*) \parallel (\Delta \setminus x^\uparrow))$, if $\Gamma(x^*) = \alpha$, $\Delta(x^\uparrow) = \beta$, and $* \in \{!, \downarrow\}$;
4. $\{x^! : \alpha\} \cup ((\Gamma \setminus x^!) \parallel (\Delta \setminus x^\downarrow))$, if $\Gamma(x^!) = \alpha$, $\Delta(x^\downarrow) = \beta$, and $\alpha = repl(\gamma)$, for some γ such that $\gamma \sim \beta$;
5. $\{x^! : \alpha, x^\downarrow : \beta\} \cup ((\Gamma \setminus x^!) \parallel (\Delta \setminus x^\downarrow))$, if $\Gamma(x^!) = \alpha$ and $\Delta(x^\downarrow) = \beta$.

In the two definitions above, the rules should always be tried in the order presented.

The notion of subgraph naturally extends to typings.

Definition 3.18 Let $\Delta \leq \Gamma$ if $dom(\Delta) \subseteq dom(\Gamma)$ and $\forall_{x^* \in dom(\Delta) \cap dom(\Gamma)}\ \Delta(x^*) \leq \Gamma(x^*)$.

Lemma 3.19 If $\Delta \leq \Gamma$ and $\Gamma \asymp \Lambda$, then $(\Delta \parallel \Lambda) \leq (\Gamma \parallel \Lambda)$.

Proof Follows from the definitions of $\parallel$ and $\leq$ for typings, using lemma 3.11. ■

3.3 Typing assignment system

We finally present the static typing assignment system. It has an axiom or a rule for each constructor of the calculus, plus a weak rule allowing to add a new output-name to a typing.

Definition 3.20 A *typing assignment* is a formula $\Gamma \vdash P$, for any process P and typing Γ. If such a formula is provable using the axiom and rules presented below we say that P is *well-typed* under typing Γ.

Definition 3.21 (Behavioural type system) The typing assignment system is inductively defined by the following axioms and rules.

$$\text{NIL} \qquad \emptyset \vdash \mathbf{0}$$

$$\text{MSG} \qquad \{\tilde{u}^* : \tilde{\alpha}, x^\uparrow : (\{v_0, v_1\}, \{v_0\}, \{(v_0, \mathrm{a} : \tilde{\alpha}, v_1)\})\} \vdash x \lhd \mathrm{a} : [\tilde{u}] \quad (v_0 \neq v_1)$$

$$\text{OBJ} \qquad \frac{\Gamma_i \cdot \tilde{x}_i^\uparrow : \tilde{\alpha}_i \vdash P_i}{\biguplus_{i \in \mathrm{I}} \Gamma_i \uplus \{x^\downarrow : \alpha\} \vdash\ x \rhd \sum_{i \in \mathrm{I}} \mathrm{a}_i : (\tilde{x}_i)\, P_i} \quad (1)$$

$$\text{REP} \quad \frac{\Gamma \cdot x^\downarrow : \alpha \vdash x \rhd M}{\Gamma \cdot x^! : repl(\alpha) \vdash !x \rhd M} \qquad\qquad \text{RES} \quad \frac{\Gamma \vdash P}{\Gamma \backslash x \vdash \nu x\, P}$$

$$\text{PAR} \quad \frac{\Gamma \vdash P \qquad \Delta \vdash Q}{\Gamma \parallel \Delta \vdash P, Q} \quad (\Gamma \asymp \Delta) \qquad\qquad \text{WEAK} \quad \frac{\Gamma \vdash P}{\Gamma \cdot x^\uparrow : \alpha \vdash P}$$

(1) $(\asymp_{i\in I} \Gamma_i) \asymp \{x^{\downarrow} : \alpha\}$, and α is such that $I_\alpha \stackrel{\text{def}}{=} \{u\}$ for u a fresh node, and

$$\begin{aligned} A_\alpha = \; & \bigcup_{i\in \mathrm{I}}\{(u, \mathrm{a}_i : \tilde{\alpha}_i, w) \mid x^* \notin \mathit{dom}(\Gamma_i) \text{ and } w \text{ is fresh}\} \cup \\ & \bigcup_{i\in \mathrm{I}}\{(u, \mathrm{a}_i : \tilde{\alpha}_i, v) \mid x^* \in \mathit{dom}(\Gamma_i) \text{ and for each } v \in I_{\Gamma_i(x^*)}\} \cup \\ & \bigcup_{i\in \mathrm{I}} A_{\Gamma_i(x^*)} \end{aligned}$$

for $* \in \{!, \downarrow\}$.

In the following we give a short description of the rules in the behavioural type system.

Nil: the terminated process has an empty typing;

Msg: the graph of a message has a single arc labelled with the message's label to the target, and to the carried names are assign arbitrary types;

Obj: the graph of an object has as many arcs starting from the initial node as the number of initial nodes of its type in each of its methods. Each arc is labelled with the methods's label. In the ending node of each of those arcs we hang the graph of that object as it occurs in the corresponding method;

Rep: the type of a replicated object is the greatest fix-point of the graph of the object;

Par: the parallel composition of two processes has a typing resulting from the interleaving of the components' typings;

Res: bound names are removed from the typing;

Weak: one can add to the typing of a process a type assignment formula with an output-name that does not occur in the typing.

3.4 Some properties of the type system

The main theorem is subject-reduction, with the corollary that guarantees absence of run-time errors in well-typed processes. Another important result is the uniqueness of the type assignment to each free name in a well-typed process.

In order to show that this properties, we need some auxiliary results.

Lemma 3.22 (Typing Lemma) If $\Gamma \vdash P$, then

1. $\mathit{fn}(P) \subseteq \mathit{dom}(\Gamma)$,
2. if $x \notin \mathit{fn}(P)$ then $\Gamma \setminus x^* \vdash P$,
3. $\Gamma \restriction P \vdash P$.

Proof By induction on the length of the deduction of $\Gamma \vdash P$. ■

Lemma 3.23 (Typability of Subterms) If $\Gamma \vdash P$ and Q is a subterm of P, then $\exists_\Delta \Delta \vdash Q$.

Proof By induction on the structure of P. ∎

Lemma 3.24 (Substitution Lemma) If $\Gamma \vdash P$ and $\Gamma(\tilde{z}^*) \sim \Gamma(\tilde{x}^*)$ then $\Gamma \backslash \tilde{x}^* \vdash P[\tilde{z}/\tilde{x}]$.

Proof By substitution of z for x in the deduction. The substitution results in a new instance of each axiom or rule, and the side conditions in the OBJ-rule and PAR-rule are preserved: if $\Gamma \asymp \Delta$ then $\Gamma[\tilde{z}/\tilde{x}] \asymp \Delta[\tilde{z}/\tilde{x}]$. ∎

Lemma 3.25 (Congruence Lemma) If $\Gamma \vdash P$ and $P \equiv Q$, then $\Gamma \vdash Q$.

Proof By a simple case analysis on the rules defining $\equiv$. ∎

Theorem 3.26 (Subject-reduction) If $\Gamma \vdash P$ and $P \twoheadrightarrow Q$, then $\exists_\Delta \Delta \vdash Q$ and $\Delta \leq \Gamma$.

Proof By induction on $\twoheadrightarrow$. If the reduction ends with the STR-rule or $P \equiv Q$ then use the congruence lemma. The non-trivial cases are when reduction ends with the PAR-rule and the COM-axiom (REP is similar to COM).

If reduction ends with the RES-rule let $P \equiv \nu x\, P'$ and $Q \equiv \nu x\, Q'$; by typability of subterms $\exists_{\Gamma'} \Gamma' \vdash P'$ and by hypothesis of the RES-rule $P' \twoheadrightarrow Q'$; then by the induction hypothesis $\exists_{\Delta'} \Delta' \vdash Q'$ and $\Delta' \leq \Gamma'$; by lemma 3.24.2 $\Gamma \setminus x^* \vdash P$; if $x \in fn(P)$ then $\Gamma' = \Gamma \cdot x^*$ else $\Gamma' = \Gamma$, and Δ' is defined similarly; since $\Delta' \leq \Gamma'$ then $\Delta \leq \Gamma$.

If reduction ends with the PAR-rule let $P \equiv P' \,|\, R$ and $Q \equiv Q' \,|\, R$; by typability of subterms $\exists_{\Gamma',\Lambda} \Gamma' \vdash P'$ and $\Lambda \vdash R$ with $\Gamma = \Gamma' \parallel \Lambda$, and by induction hypothesis $\exists_{\Delta'} \Delta' \vdash Q'$ and $\Delta' \leq \Gamma'$. The result follows by lemma 3.20.

If reduction ends with the COM-axiom let $P \equiv x \rhd M \,|\, x \lhd m$. By the PAR-rule and by typability of subterms $\Gamma = \Gamma' \parallel \Delta'$ with $x^\downarrow : \alpha \in \Gamma'$ and $x^\uparrow : \beta \in \Delta'$; if $M \bullet m$ is undefined then $\Delta = \Gamma$ else by the induction hypothesis and by the substitution lemma $\exists_\Delta \Delta \vdash M \bullet m$ ($\Delta = \Gamma_i \cdot \tilde{x}_i^\uparrow[\tilde{v}/\tilde{x}] \vdash M \bullet m$, being $\tilde{v}$ the sequence of names in m), and by the OBJ-rule if $x^\downarrow \in dom(\Delta)$ then $\Delta(x^\downarrow) \leq \alpha$, and if $x^\uparrow \in dom(\Delta)$ then $\Delta(x^\uparrow) \leq \beta$; it follows that $\Delta \leq \Gamma$. ∎

Corollary 3.27 If P is well-typed then $P \notin$ ERR.

Proof Suppose P is well-typed and $P \in$ ERR. By definition of ERR, P has $!x \rhd M$ and $x \lhd m$ as subterms, both typable by typability of subterms, with compatible types. Therefore, $M \bullet m$ is defined, and then it is not a persistent bad x-redex; we have reached an absurd, since, by hypothesis, $P \in$ ERR. ∎

The system enjoys the property of *uniqueness* of the types assigned to the free names in a process.

Proposition 3.28 If $\Gamma \vdash P$ and $\Delta \vdash P$ then $\Gamma \restriction P = \Delta \restriction P$.

Proof By a case analysis of the rules defining the type system, noting that, with the exception of the WEAK-rule, each rule defines one and only one typing for a process, up to renaming of nodes. ∎

4 Conclusions and discussion

This paper presents a new type system for TYped Concurrent Objects. The type system captures dynamic aspects of objects' behaviour, namely non-uniform service availability. Types are graphs, and the type compatibility relation is a bisimulation. We propose a new notion of error-process: the persistence of bad x-redexes. Thus, processes without errors have weak fairness in the treatment of messages. The type system enjoys the subject-reduction property.

The presented type system types all processes the previous system [VT93] does, except for those that do not conform to the restriction in section 2. The buffer-cell in section 1 constitutes an example of a process this system types while the does previous not. Nevertheless some "basic" mistakes (like typing *puhs* instead of *push* in process $x \triangleright [\mathit{push} : (w)\, P] \mid x \triangleleft \mathit{puhs} : [v])$, are no longer detected as error-processes.

The starting point for this work are the ideas of Nierstrasz on regular types for active objects. Puntigam also starts from Nierstrasz work, and uses terms of a process algebra (without name-passing) as types [Pun95, Pun96]. His work is centered on subtyping, and not on type assignment systems. There is now a lot of work on types for mobile processes but, up to our knowledge, the only work in the context of mobile processes where types are graph seems to be Yoshida's [Yos96]. Her graphs give information about the deterministic behaviour of a process; our graphs are inspired on Milner's derivation trees [Mil89].

This work is still in a preliminary stage; a lot of work has yet to be done. The points requiring further study include:

1. the proof of the conjecture that the typing assignment is decidable and the typing inference is computable.
2. an algebraic version of the notion of types, more tractable since types will be syntactic. A labelled transition system for processes can easily be defined, leading to a behavioural equivalence on processes.
3. the application of this type system to some object-oriented programming language.
4. the interpretation of types with a temporal logic and the definition of a type-theory.

Acknowledgments

Special thanks are due to A. Sernadas, for his long-term support, to J. -L. Fiadeiro, for his contributions and encouragement, and to K. Honda and N. Yoshida, as well as P. Resende and C. Caleiro, for very stimulating discussions. Comments of the anonymous referees were most valuable.

This work was partially supported by JNICT PRAXIS XXI projects 2/2.1/MAT/46/94 Escola, 2/2.1/MAT/262/94 SitCalc and 2/2.1/TIT/1658/95 LogComp, and by the ESPRIT Working Groups 22704 ASPIRE and 23531 FIREworks.

References

[Agh86] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* M.I.T. Press, 1986.

[HBS73] C. Hewitt, P. Bishop, and R. Steiger. A universal, modular actor formalism for artificial intelligence. In *3rd International Joint Conference on Artificial Intelligence*, pages 235–245, 1973.

[HT91] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *5th European Conference on Object-Oriented Programming*, volume LNCS 512, pages 141–162. Springer-Verlag, 1991.

[KY95] N. Kobayashi and A. Yonezawa. Towards foundations of concurrent object-oriented programming - types and language design. *Theory and Practice of Object Systems*, 1(4), 1995.

[LW95] X. Liu and D. Walker. A polymorphic type system for the polyadic π-calculus. In *6th International Conference on Concurrency Theory*, volume LNCS 962, pages 103–116. Springer-Verlag, 1995.

[Mil89] R. Milner. *Communication and Concurrency.* C. A. R. Hoare Series Editor – Prentice-Hall Int., 1989.

[Mil91] R. Milner. The polyadic π-calculus: a tutorial. Technical report ECS-LFCS 91-180, University of Edinburgh, U. K., 1991.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, i e ii. *Information and Computation*, 100:1–77, 1992. Also available as Technical Report ECS-LFCS 89-85/86, University of Edinburgh.

[Nie95] O. Nierstrasz. Regular types for active objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, pages 99–121. Prentice Hall, 1995.

[Pie95] B. Pierce. Programming in the π-calculus: an experiment in concurrent language design. Tutorial notes for Pict version 3.6k, Computer Laboratory, University of Cambridge, U. K., 1995.

[PT95] B. Pierce and D. Turner. Concurrent objects in a process calculus. In T. Ito and A. Yonezawa, editors, *Theory and Practice in Parallel Programming*, volume LNCS 907, pages 187–215. Springer-Verlag, 1995.

[Pun95] F. Puntigam. Flexible types for a concurrent model. In *1st Workshop on Object-Oriented Programming and Models of Concurrency*, 1995.

[Pun96] F. Puntigam. Types for active objects based on trace semantics. In *1st IFIP Workshop on Formal Methods for Open Object-based Distributed Systems (FMOODS'96),*, 1996.

[Vas94] V. Vasconcelos. *A Process-Calculus Approach to Typed Concurrent Objects.* PhD thesis, Department of Computer Science, Keio University, Japan, 1994.

[VH93] V. Vasconcelos and K. Honda. Principal typing schemes in a polyadic π-calculus. In *4th International Conference on Concurrency Theory*, volume LNCS 715, pages 524–538. Springer-Verlag, 1993.

[VT93] V. Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *1st International Symposium on Object Technologies for Advanced Software*, volume LNCS 742, pages 460–474. Springer-Verlag, 1993.

[Yos96] N. Yoshida. Graph types for monadic mobile processes. In *16th FST/TCS*, volume LNCS 1180, pages 371–386. Springer-Verlag, 1996.