DiTyCO: and Experiment in Code Mobility from the Realm Process Calculi

Luís Lopes Fernando Silva
Álvaro Figueira Vasco Vasconcelos

Technical Report Series: DCC-99-3

Departamento de Ciência de Computadores – Faculdade de Ciências
&
Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto
Rua do Campo Alegre, 823 4150 Porto, Portugal
Tel: +351+2+6001672 – Fax: +351+2+6003654
http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

DiTyCO: an Experiment in Code Mobility from the Realm of Process Calculi*

Luís Lopes, Fernando Silva, Álvaro Figueira
DCC-FC & LIACC, Universidade do Porto
e-mail: {lblopes,fds,arf}@ncc.up.pt

Vasco T. Vasconcelos
DI-FCUL, Universidade de Lisboa
e-mail: vv@di.fc.ul.pt

May 1999

Abstract

We propose a simple formal model of distribution for mobile objects in the context of the TyCO process calculus. Code mobility is induced by lexical scoping on names and template process definitions. Objects and messages migrate towards the site where their prefix names are lexically bound. Template process definitions, on the other hand, are downloaded from the site where they are defined, and are instantiated locally upon arrival. Based on this model we describe the run-time support for distribution and code mobility implemented in DiTyCO, an extension of the TyCO programming language.
Keywords: Process-Calculus, Concurrency, Code-Mobility, Implementation.

1 Introduction

Milner, Parrow, and Walker's π-calculus [14] has provided a formal framework for most of the research on communication-based concurrent systems. Several forms and extensions of the asynchronous π-calculus [11] have since been proposed to provide for more direct programming styles, and to improve efficiency and expressiveness [3, 8, 21]. The π-calculus has also been used as a basis to reason about distributed computations. Introducing distribution, code mobility, and failure detection and recovery into π-computations is a fast growing research field, with immediate applications in mobile computing, *web* languages, cryptography, to name a few [1, 5, 9, 20].

We proposed a simple model of distribution for mobile processes [20]. The following major constraints guided its design.

1. The model should be a simple extension of the calculi we have today;
2. it should be independent of the base calculus chosen;
3. must meet realistic expectations of current distributed systems;
4. must be efficiently implementable in current hardware.

*This article was presented at the *Mobile Object Systems (MOS'99)* workshop at ECOOP'99

Here we take a step forward and apply this model to the TyCO programming language [19] to obtain an extension with support for distribution and code mobility. We call the extended language DiTyCO. The model builds on the notion of *site* (or location) where conventional (name-passing, in this case) computation happens. We also have site identifiers, distinct from the usual names. Our processes are *network aware*: names can be local or remote; the distinction is explicit in the syntax. Local names are those of the base calculus; remote names are pairs site-name, called *site-names*.

Sites are composed of *located processes*, i.e., processes paired with site identifiers. They denote the execution of the process at the site, similarly to most proposals to date [2, 9, 10, 17, 16]. Processes at sites can be put to run in parallel. Furthermore, since name-passing calculi are capable of extruding the scope of a (local) name, our networks are equipped with a site-name restriction operation.

In summary, networks are sites equipped with a composition and a restriction operator. This yields a *flat organization of sites*, that reflects the architectures of current implementations of high performance networks [6, 7]. Our target hardware architectures are low-cost, off-the-shelf, clusters of PCs interconnected with a high speed network. We feel that, the site organization should map the low level hardware architecture as closely as possible to allow an efficient implementation of the model. From a formal point of view our site organization is quite close to Distributed-π [16] and contrasts with the tree structure of Mobile Join [8] and the nested structure of Ambients [5].

The model has two logical levels: processes and networks (cf. [2, 10, 16]). Local computations happen at sites, as prescribed by the semantics of the base calculus. Remote computations occur between prefixed processes at different sites. In TyCO [21] such prefixed processes are either remote message invocation or the migration of objects. The migration of prefixed processes is deterministic, point to point, and asynchronous; synchronization only happens locally, at reduction time.

We adhere to the *lexical scoping in a distributed context* of Obliq [4]. The free names of any "piece of code" transmitted over the network are bound to the original location. Network transmission implies the translation of the free names in the code in order to reflect the new site where the code is to be executed.

An important design decision related to points 3 and 4 above is the incapacity of the model to create remote names and the inability to spawn processes at remote sites, thus providing for site protection against arbitrary uploads. Processes may be spawned remotely only through the collaboration of what we call friend processes [20]. This contrasts with other proposals such as Sewell et al. [17], Amadio [2] and Hennessy and Riely's [10, 16].

Finally, as a first approach, our site identifiers are not first class objects: they cannot be sent in messages. Also we deliberately eschew the following issues: checking whether a site is alive; killing a site [2, 9, 16]; checking whether two remote names reside at the same site [17]; comparing site identifiers, and; dynamically constructing a located name given a name and a site identifier.

The implementation of the model has three layers: network, nodes and sites, although only two – network and sites, are logical. Sites form the basic sequential units of computation. Nodes have a one-to-one correspondence with physical IP nodes and may have an arbitrary number of sites computing either concurrently or in parallel. This intermediate level does not exist in the formal model. It makes the architecture more flexible by allowing multiple sites at a given IP node. Finally, the *network* is composed of multiple DiTyCO nodes connected in a static IP topology. Message passing and code mobility occurs at the level of sites, and at this level the communication topology is dynamic.

DiTyCO is an extension of the TyCO [19] object-based concurrent programming language with support for distributed computations and code mobility. At the programming level it grows from TyCO by introducing two simple constructs for making names visible to the network (**export**) and for making network names available to local processes (**import**). The DiTyCO source code is compiled into byte-code for an extended TyCO virtual machine [13]. Each site is implemented as a thread that runs an extended TyCO byte-code emulator with private machine registers and memory areas. A site is created to run a DiTyCO program and is freed when the program ends.

The outline of the paper is as follows. The next section introduces the formal network model, its syntax and semantics; section 3 introduces the programming model implied by the network model and presents an example using as base language DiTyCO. Section 4 describes the issues involved in compiling DiTyCO to add support for code mobility. Next, in section 5 we describe the software architecture and we end with a discussion on pending implementation issues which will be the focus of future research.

2 The Model

We first overview the model for distribution and mobility [20] that serves as the basis for the DiTyCO implementation. The model has two levels: on the first level we have the *processes* in the base calculus; on the second level we build *networks*. The ideas presented in the previous section can be embodied in any name-passing calculus that satisfies a few mild pre-conditions (e.g. [3, 11, 12, 14, 21]), but the focus of this paper will be on the TyCO process calculus and its language implementation.

TyCO is based on the TyCO calculus [21], a name-passing calculus in the line of the asynchronous π-calculus [11] that incorporates, in place of (unlabeled) messages and receptors ($\overline{a}v$, $a(x).P$), labeled messages and objects composed of methods as the fundamental processes:

$a!l[\tilde{v}]$	message
$a?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$	object

In the syntax above, a is a name, $\tilde{v}, \tilde{x}_1, \ldots, \tilde{x}_n$ are sequences of names, $l, l_1, \ldots, l_n$ are *labels*. Labels constitute a syntactic category distinct from names. Labels $l_1, \ldots, l_n$, and also the names in each $\tilde{x}_i$, are pairwise distinct. A message $a!l_i[\tilde{v}]$ selects the method l_i in an object $a?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$; the result is the process P_i where names in $\tilde{v}$ replace those in $\tilde{x}_i$.

These primitives are further combined by the following standard constructs in concurrent programming.

$P_1 \mid P_2$	concurrent composition
new x P	name hiding
def D **in** P	definition
$X[\tilde{v}]$	instantiation

In the syntax above D stands for a sequence of, possibly mutually recursive, definitions of template processes: $X_1(\tilde{x}_1) = P_1$ **and**...**and** $X_n(\tilde{x}_n) = P_n$. The X_i are variables over process templates and are accordingly named *template variables*. An instantiation $X[\tilde{v}]$ of a template process $X(\tilde{x}) = P$ is another form of reduction that results in the process P where names in $\tilde{v}$ replace those in $\tilde{x}$. Contrary to the conventional practice in name-passing calculi, we let the scope of a **new** extend as far to the right as possible. We single out a label — *val*— to be used in objects with a single method. This allows to abbreviate messages and objects. The **let** constructor is quite useful in getting back results; the syntax is taken from Pict [15].

$a![\tilde{v}]$	abbreviates	$a!\mathit{val}[\tilde{v}]$
$a?(\tilde{x}) = P$	abbreviates	$a?\{\mathit{val}(\tilde{x}) = P\}$
let $x = a!l[\tilde{v}]$ **in** P	abbreviates	(**new** r $a!l[\tilde{v}r] \mid r?(x) = P$)

We now proceed to build the second level of the model on top of TyCO. We introduce a new class of identifiers, *sites*, distinct from names or any other class of identifiers in the base calculus. *Located names* are site-name pairs. We let s range over sites. A name a located at site s is denoted by $s{\cdot}a$. Similarly, *located template variables* are site-template variable pairs. A template variable X located at site s is denoted by $s{\cdot}X$. Names and template variables are collectively known as *identifiers* and ranged over by w. We then allow located identifiers to occur in any position in the base calculus where (non-binding occurrences of) identifiers can. The calculus thus obtained constitutes the *first* *level* of the model. Since site identifiers are introduced anew, there must be no provision in the base calculus for binding located identifiers. As such, at this level, a located identifier behaves as any other constant in the base calculus.

The *second level* is composed of site-process pairs called *located processes*, composed via conventional parallel, restriction, and definition operators. The set of networks is given by the following grammar.

$$N \quad ::= \quad s : P \quad | \quad N \parallel N \quad | \quad \textsf{new}\ s{\cdot}aN \quad | \quad \textsf{def}\ s{\cdot}D\ \textsf{in}\ N \quad | \quad \mathbf{0}$$

The bindings in networks are as expected: a located name $s{\cdot}a$ occurs *free* in a network if $s{\cdot}a$ is not in the scope of a **new** $s{\cdot}aN$; otherwise $s{\cdot}a$ occurs *bound*. The set of free located names in a network N, notation $\mathrm{fn}(N)$, is defined accordingly. Similarly, a located template variable $s{\cdot}X$ occurs free in a network N if $s{\cdot}X$ is not in the scope of a **def** $s{\cdot}D$ **in** N, i.e., X is not one of the X_i in D. The set $\mathrm{ft}(N)$ of free located template variables in networks is defined accordingly.

Structural congruence allows us to abstract from the static structure of networks; it is defined as the least relation closed over composition and restriction, that satisfies the monoid laws for parallel composition, as well as the following rules[1].

$$
\begin{array}{rl}
(\textsc{Nil}) & s : \mathbf{0} \equiv \mathbf{0} \\
(\textsc{Split}) & s : P_1 \parallel s : P_2 \equiv s : (P_1 \mid P_2) \\
(\textsc{New}) & s : \textsf{new}\ aP \equiv \textsf{new}\ s{\cdot}a(s : P) \\
(\textsc{GcNew}) & \textsf{new}\ s{\cdot}a\ \mathbf{0} \equiv \mathbf{0} \\
(\textsc{Def}) & s : \textsf{def}\ D\ \textsf{in}\ P \equiv \textsf{def}\ s{\cdot}D\ \textsf{in}\ s : P \\
(\textsc{GcDef}) & \textsf{def}\ s{\cdot}D\ \textsf{in}\ \mathbf{0} \equiv \mathbf{0} \\
(\textsc{Extr}) & N_1 \parallel \textsf{new}\ s{\cdot}aN_2 \equiv \textsf{new}\ s{\cdot}a(N_1 \parallel N_2) \quad \text{if } s{\cdot}a \notin \mathrm{fn}(N_1) \\
(\textsc{ExtrD}) & N_1 \parallel \textsf{def}\ s{\cdot}D\ \textsf{in}\ N_2 \equiv \textsf{def}\ s{\cdot}D\ \textsf{in}\ (N_1 \parallel N_2) \quad \text{if } \mathrm{bt}(D) \cap \mathrm{ft}(N_1) = \emptyset \\
(\textsc{Swap}) & \textsf{new}\ r{\cdot}a(\textsf{def}\ s{\cdot}D\ \textsf{in}\ N) \equiv \textsf{def}\ s{\cdot}D\ \textsf{in}\ \textsf{new}\ r{\cdot}aN \quad \text{if } r{\cdot}a \notin \mathrm{fn}(s{\cdot}D) \\
(\textsc{Merge}) & \textsf{def}\ s{\cdot}D\ \textsf{in}\ \textsf{def}\ r{\cdot}D'\ \textsf{in}\ N \equiv \textsf{def}\ s{\cdot}D\ \textsf{and}\ r{\cdot}D'\ \textsf{in}\ N \quad \text{if } \mathrm{ft}(s{\cdot}D) \cap \mathrm{ft}(r{\cdot}D') = \emptyset
\end{array}
$$

Rule Nil garbage collects terminated located processes, whereas rules GcNew and GcDef garbage collect unused names and definitions, respectively. When used from left to right, the rule Split gathers processes under the same location, allowing reduction to happen; the right to left usage is for isolating prefixed processes to be transported over the network (see rule Move in the reduction relation below). The remaining rules allow the scope of a name (rule New) or a template definition (rule Def), local to a process, to extrude and encompass a network with several located processes (rules Extr and ExtrD), to aggregate template definitions (rule Merge) and to interchange **new**s and **def**s (rule Swap).

Non-located identifiers in processes are implicitly located at the site the process occurs at: an identifier w occurring in a network $r : P$ is implicitly located at site r. When sending identifiers over the network, the implicit locations of identifiers need to be preserved, if we are to abide by the lexical scoping convention. As such, an identifier w moving from site r to any other site must become $r{\cdot}w$. Similarly a located identifier $s{\cdot}w$ arriving at site s may drop its explicit location. The remaining identifiers need no translation.

A *translation of identifiers* from site r to site s is a total function σ_{rs} defined as follows[2].

$$\sigma_{rs}(w) \stackrel{\text{def}}{=} r{\cdot}w \qquad \sigma_{rs}(s{\cdot}w) \stackrel{\text{def}}{=} w \qquad \sigma_{rs}(w) \stackrel{\text{def}}{=} w$$

[1]Rules Nil, Split, New and Extr are taken from Hennessy-Riely [10]; rules Nil, Split, and Extr are present in Sewell et al. [17] as well; the remaining rules are adapted from [18]

[2]The last rule should be applied last.

For the purpose of defining the reduction relation over networks we introduce a new syntactic category C defined as:

$$
\begin{array}{lll}
C & ::= & !l[\tilde{v}] & \text{message body} \\
& & ?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\} & \text{object body}
\end{array}
$$

Processes prefixed at located names play a crucial role in the model, by moving towards the location of the located name: a process $s{\cdot}aC$ is meant to move to site s. If aC is a message then $s{\cdot}aC$ has the form $s{\cdot}a!l[\tilde{v}]$ and denotes a remote message send; if, on the other hand, aC is an object then $s{\cdot}aC$ has the form $s{\cdot}a?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$ and denotes an object migration operation. We thus see that conceptually there is not much difference between a remote message send and a process migration; in section 5 we show that from an implementation point of view the difference is not abysmal either.

Given the above definition, the reduction relation for networks is given by the following rule that enables reduction to occur locally:

$$
(\textsc{Local}) \qquad \frac{P \to Q}{s : P \to s : Q}
$$

plus the familiar rules for parallel composition, restriction, definitions and structural congruence which we omit, together with the two axioms below. The first axiom supports the mobility of prefixed processes such as messages and objects.

$$
(\textsc{Move}) \qquad r : s{\cdot}aC \to s : a(C\sigma_{rs})
$$

If prefix $s{\cdot}aC$ is located at site r, then, in order to keep the lexical scope of names, the free names in C must translated according to $C\sigma_{rs}$. So, when sending $s{\cdot}aC$ from r to s we actually transmit $(s{\cdot}aC)\sigma_{rs} = aC\sigma_{rs}$. This is the essence of the axiom Move. Rule Local then allows processes in sites to evolve locally.

As an example let us try a remote procedure call in TyCO. The client at site s invokes the procedure p at site r with a local argument v, waits for the reply and continues with P. The procedure accepts a request and answers a local name u (somewhere in the body Q of the procedure).

$$
\begin{array}{lr}
s : \textbf{new } a(r{\cdot}p![va] \mid a?(y) = P) \;\|\; r : p?(xr) = Q \equiv & (\textsc{New},\textsc{Extr}) \\
\textbf{new } s{\cdot}a(s : r{\cdot}p![va] \;\|\; s : a?(y) = P \;\|\; r : p?(xr) = Q) \to & (\textsc{Move}) \\
\textbf{new } s{\cdot}a(r : p![s{\cdot}v\ s{\cdot}a] \;\|\; s : a?(y) = P \;\|\; r : p?(xr) = Q) \equiv & (\textsc{Split}) \\
\textbf{new } s{\cdot}a(s : a?(y) = P \;\|\; r : (p![s{\cdot}v\ s{\cdot}a] \mid p?(xr) = Q)) \to & (\textsc{Local}) \\
\textbf{new } s{\cdot}a(s : a?(y) = P \;\|\; r : Q[s{\cdot}v\ s{\cdot}a/xr]) \to^* & \\
\textbf{new } s{\cdot}a(s : a?(y) = P \;\|\; r : s{\cdot}a![u]) \to\to & (\textsc{Move},\textsc{Split},\textsc{Local}) \\
\textbf{new } s{\cdot}a(s : P[r{\cdot}u/y]) \equiv s : \textbf{new } aP[r{\cdot}u/y] & (\textsc{New})
\end{array}
$$

We thus see that a remote communication involves two reduction steps: one to get the message/object to the target site and the other to consume the message/object at the target (cf. [9]); the former is an asynchronous operation, the latter requires a rendez-vous. This reflects actual implementations.

Another kind of remote interaction, which we call *download*, occurs when a site r finds an instantiation of the form $s{\cdot}X[\tilde{v}]$. A template variable X prefixed with a site name s indicates that X was originally defined at site s. An instantiation $s{\cdot}X[\tilde{v}]$ at a site r means that the definition of X is to be downloaded from the network. Once it arrives at r the Local rule enables the instantiation to occur.

$$
(\textsc{Download}) \qquad \textbf{def } s{\cdot}D \textbf{ in } (r : s{\cdot}X_i[\tilde{v}] \;\|\; N) \to \textbf{def } s{\cdot}D \textbf{ in } (r : \textbf{def } X_i(\tilde{x}_i) = P_i\sigma_{sr} \textbf{ in } X_i[\tilde{v}] \;\|\; N)
$$

Note that when moving P_i to r we must keep the lexical bindings for the free variables of P_i, so we actually download $P_i\sigma_{sr}$. Also, DOWNLOAD does not cache the downloaded template, and so other instantiations of $s{\cdot}X$ will again download P_i. The lexical scope of the template variables prevents a complete solution since processes with references to $s{\cdot}X$ may be downloaded at any time during a computation. However, this can easily be circumvented from an implementation point of view by using a simple caching mechanism.

Below we present an example of this kind of interaction. We move a piece of code that contains a template variable X. The code for X is downloaded thereafter.

$$
\begin{array}{lr}
r : \textbf{def}\ X(x) = P\ \textbf{in}\ s{\cdot}a?() = X[b]\ \parallel\ s : a![] \equiv & (\textsc{Def},\textsc{ExtrD}) \\
\textbf{def}\ r{\cdot}X(x) = P\ \textbf{in}\ (r : s{\cdot}a?() = X[b]\ \parallel\ s : a![]) \to & (\textsc{Move}) \\
\textbf{def}\ \ldots\ \textbf{in}\ (s : a?() = r{\cdot}X[r{\cdot}b]\ \parallel\ s : a![]) \to & (\textsc{Split},\textsc{Local}) \\
\textbf{def}\ \ldots\ \textbf{in}\ s : r{\cdot}X[r{\cdot}b] \to & (\textsc{Download}) \\
\textbf{def}\ \ldots\ \textbf{in}\ s : \textbf{def}\ X(x) = P\sigma_{rs}\ \textbf{in}\ X[r{\cdot}b] \to & (\textsc{Local}) \\
\textbf{def}\ \ldots\ \textbf{in}\ s : P\sigma_{rs}[r{\cdot}b/x] &
\end{array}
$$

Note that with DOWNLOAD the calculus closely mimics the way class definitions are downloaded in languages such as Java.

3 Programming

The programming model associated with the framework described in the previous section is rather simple requiring just two new declarations.

export identifier **in** process
import identifier **from** site **in** process

export is used to raise a local name or template definition to the network level of the model, making it visible to other sites. **import** lowers one of these names or template definitions to the context of a site so that it may be used locally. There is no need to change the syntax of the base language whatsoever. In particular we never write located identifiers explicitly. The translation into the base calculus extended with located identifiers is straightforward.

$$
\begin{aligned}
[\![\textbf{export}\ w\ \textbf{in}\ P]\!] &\stackrel{\text{def}}{=} [\![P]\!] \\
[\![\textbf{import}\ w\ \textbf{from}\ s\ \textbf{in}\ P]\!] &\stackrel{\text{def}}{=} [\![P[s.w/w]]\!]
\end{aligned}
$$

We thus see that the **export** declaration is really unnecessary. As programs are to be closed, we could take the view that every free identifier in a program is to be exported. From a programming point of view we, however, feel that the dual **import/export** declarations impose a more disciplined programming style, avoiding, for example, the automatic exporting of names that the programmer forgot to protect with a **new**. The remainder of this section is devoted to a programming example in the extended language – DiTyCO, to attest the simplicity and flexibility of the model.

The example illustrates code transmission over the network. The idea is from Fournet et al. [9], but we have taken advantage of objects in TyCO to allow for the downloading of different applets.

An applet server provides for the downloading of k different applets through the k methods of an object. The server locates applet P_j at name p provided with the invocation of the method applet_j. Here is the code to be run at site sumatra.

def AppletServer (self) =
  self ? {
    applet$_1$(p) = p?(x)=P$_1$ | AppletServer[self],
    ...
    applet$_k$(p) = p?(x)=P$_k$ | AppletServer[self]}
in export appletserver
in AppletServer[appletserver]

Each client creates a fresh name where the applet server is supposed to locate the applet, then invokes the server with this name, and, in parallel, triggers the applet.

import appletserver **from** sumatra
in new p appletserver!*applet*$_j$[p] | p![v]

Let us try to understand how the server and the client interact. We start by translating the **import/export** clauses to obtain

sumatra: **def** ... **in** AppletServer[appletserver] ||
client: **new** p sumatra·appletserver!*applet*$_j$[p] | p![v]

Then, the message sumatra·appletserver!*applet*$_j$[p] moves to the server (yielding the message appletserver!*applet*$_j$[client·p]) with one MOVE reduction step, one local reduction at the server invokes the *applet*$_j$ method, and one final MOVE step migrates the applet client·p?(x)=P$_j$ back to the client, yielding the process:[3]

sumatra: **def** ... **in** AppletServer[appletserver] ||
client: **new** p p?(x)=P$_j\sigma_{\text{sumatra client}}$ | p![v]

Notice how the structural congruence rules NEW and EXTR are used (from left to right) to allow name p at client to encompass both sites, and then (from right to left) to bring p local to the client again. Notice also that the applet body gets translated to reflect its new site: if P refers to a name a local to the applet server, then P$_j\sigma_{\text{sumatra client}}$ refers to the remote name sumatra·a.

It should be obvious that a client does not need to have the applet explicitly downloaded to its site; a message appletserver!*applet*$_j$[site·p] will migrate the code for the applet to site.

4 Compilation Issues

The source code of DiTyCO programs is compiled into an intermediate representation by the language compiler. This intermediate syntax has a simple layout, is compact and maps almost one-to-one with the byte-code representation. The syntax of the intermediate representation reflects exactly the way the corresponding byte-code is structured. The nested structure of the source program is preserved in the final byte-code. This enables us to efficiently extract byte-code blocks that have to be moved between sites at run-time.

As we have seen remote communication originating in a site r occurs when we find messages, $s{\cdot}a\,!\,l[\tilde{v}]$, or objects, $s{\cdot}a?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$, located at a name $s{\cdot}a$ that has been imported from the network layer, more precisely, from another site s. Another possibility for remote interaction occurs when we instantiate templates, such as $s{\cdot}X[\tilde{v}]$, which are defined at remote sites.

In the case of a message $s{\cdot}a\,!\,l[\tilde{v}]$, the only information that needs to be sent across sites is the label l of the method to be invoked and the arguments $\tilde{v}$ with the substitution σ_{rs} applied. In this

[3]Incidentally, three is the number of reduction steps that Mobile Join [8] takes to perform the same operation.

case there is no byte-code being moved. In the case of an object $s{\cdot}a?\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$, the information that needs to be sent from site r to site s consists of the free variables of $\{l_1(\tilde{x}_1) = P_1, \ldots, l_n(\tilde{x}_n) = P_n\}$ with σ_{rs} applied plus the complete byte-code for the object's method collection. This fact imposes some discipline in the way the DiTyCO source code is compiled into the intermediate language representation and later the byte-code. Finally, in the case of a template definition $X(\tilde{x}) = P$ the byte-code for the template body P with σ_{rs} applied must be sent over the network.

An important observation is that if the nested structure of processes in the source code is preserved in the byte-code then an efficient way of extracting pieces of byte-code at run-time can be devised. Consider the AppletServer example from section 3. As we have seen the applet P_j is selected and downloaded from a server by sending a message with label *applet*$_j$ and a name local to the client requesting the applet. When the code reaches the server, it is *placed* in the argument p. The lexical scope of p forces the code P_j to be downloaded to the client site. Notice that process P_j can have an arbitrary nested structure.

If we preserve the nested structure of the original source code we can easily extract the piece of byte-code corresponding to P_j. We simply need to identify its starting point and extract all the code at that particular nesting level and deeper. This process can easily be visualized in the compilation of the AppletServer example in the intermediate assembly of DiTyCO.

```
main = {
   def AppletServer = {                          (5)
      objf       1
      cput       p0
      trobj      p0 = {
         { applet_1,...,applet_k }
         applet_1 = {                            (1)
            objf       0
            trobj      p0 = {                    (2)
               { val }
               val = { % code for P_1 }
            }                                    (3)
            instof     1,AppletServer
            cput       f0
         }
         ... ... ...
         applet_k = {
            objf       0
            trobj      p0 = {
               { val }
               val = { % code for P_k }
            }
            instof     1,AppletServer
            cput       f0
         }
      }
   }                                             (6)
   export     c0
   instof     1,AppletServer
   cput       c0
}
```

In response to a client invocation of the form:

import appletserver **from** sumatra
in new p appletserver!$applet_j$[p] | p![v] (4)

The appletserver would run the code starting at (1) with p0 bound to p. At (2) the server places the object with method table { `val` } and methods `val = { % code for P_1 }` at the name p which is lexically bound to the client process. The server must then extract the block starting at (2) and ending at (3), a simple operation given that the nested structure of the original source process was preserved in the byte-code. Then it sends the byte-code back to the client, together with any existing free names. Upon arrival to the client site the downloaded code and the bindings are placed at the name p. The client runs the byte-code by sending a local message to p as illustrated in (4). We might also be interested in starting an applet-server at a given client. This could be achieved with the following code:

import AppletServer **from** sumatra
in new p AppletServer[p]

creating a server at the local name p. In this case, since we imported the definition of template AppletServer from site sumatra the complete code for the template must be downloaded from sumatra to the local site. At sumatra the selection of the byte-code to be sent is greatly simplified by its simple and nested layout. We just pack all the code inside the scope of the AppletServer definition, starting at (5) and ending at (6). The nested intermediate assembly simplifies considerably the compile-time task of computing object boundaries and the book-keeping information required by the emulator at run-time.

5 The Implementation of DiTyCO

Our implementation of DiTyCO has three levels: sites, nodes and networks. Sites form the basic sequential units of computation. Nodes have a one-to-one correspondence with physical IP nodes and may have an arbitrary number of sites computing either concurrently or in parallel. This intermediate level does not exist in the formal model. It makes the architecture more flexible by allowing multiple sites at a given IP node. Finally, the network is composed of multiple DiTyCO nodes connected in a static IP topology. Message passing and code mobility occurs at the level of sites, and at this level the communication topology is dynamic.

A simple and basic organization for the implementation of DiTyCO would be a direct correspondence to the formal model, as represented in figure 1(a). It involves a one-to-one mapping of sites into IP addresses. Sites access the network directly, be it to make local names visible to the network, or to use these public names locally to perform computations.

Our implementation extends this basic organization into another slightly more complex but also more flexible structure. This is realized within the DiTyCO-node layer. Nodes map one-to-one into IP nodes, support the concurrent execution of multiple sites and handle all network requests made by the local sites. This structure is illustrated in figure 1 (b).

5.1 The network layer

This layer provides the means for names to be made public and accessible by any other site that knows about them. Explicitly exported names and sites are registered at a *Site Name Server* (SNS). The server maintains two tables, one for sites and another for exported names. Each tuple of the site table includes the lexeme that identifies the site in the source programs (the key attribute), a site identifier (a natural number) and the IP address where the site is located. The key for the table of exported names is compound and uses the lexemes for the name and the site it belongs to. Besides the key, each tuple also contains a unique variable identifier (a natural number). The network address

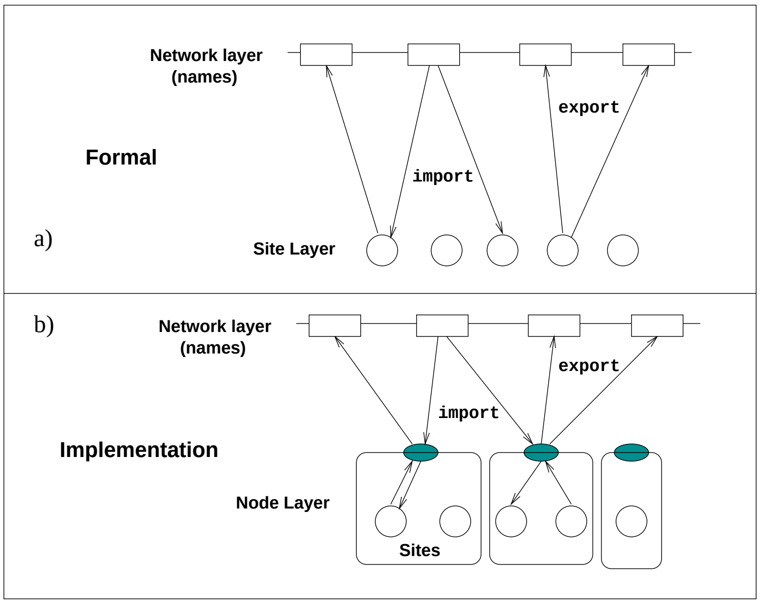

Figure 1: Model and implementation layers.

for a name is composed by the name identifier, the site identifier and its IP location. Assume, for example, that we encounter the directive **export** x at a site borneo, at IP address ip0. The tuple ("borneo",loc0,ip0) would be added to the site table, assuming the site was attributed an identifier 0. In the table of exported names the tuple ("x",var2,"borneo") is inserted, again assuming that the name x was assigned the identifier 2.

Only variables declared with the **export** construct are stored in the SNS. Whenever a site, say sumatra, executes an instruction such as **import** x **from** borneo, it first asks the SNS for the network address of x at site borneo. If a matching tuple is found at the SNS it returns the network address of name x from site borneo, providing site sumatra with the necessary information to communicate on that name. At site sumatra, x will now be represented as a tuple (ip0,loc0,var2).

Currently, the SNS is centralized and all sites know its location in advance. The service is implemented in TCP/IP sockets. Each site may be a client of the SNS server to perform, for example, **import/export** operations. Requests from a site to the server are serviced through a dedicated communication daemon at each IP node.

5.2 The node layer

This layer is composed of DiTyCO nodes, a node per IP address. A DiTyCO node is a pool of sites running concurrently or in parallel, depending on the underlying architecture, a dedicated communication daemon, and a user interface daemon. This architecture is illustrated in figure 2.

A DiTyCO node is implemented as a Unix process. Its components: the sites, communication daemon (TyCOd) and the user interface daemon (TyCOi) – are implemented as threads sharing the address space of the node. Sites execute DiTyCO programs and constitute the basic units of computation in the model. They are created whenever a new program is submitted for execution and cease to exist when the program exits. Users submit new programs for execution in a node through an user interface service handled by TyCOi.

TyCOd is responsible for all the communications between sites. Inter-site communications may be local, when sites belong to the same node, or remote when the sites belong to different nodes. Local communications are optimized through shared memory. Remote communications involve three steps:

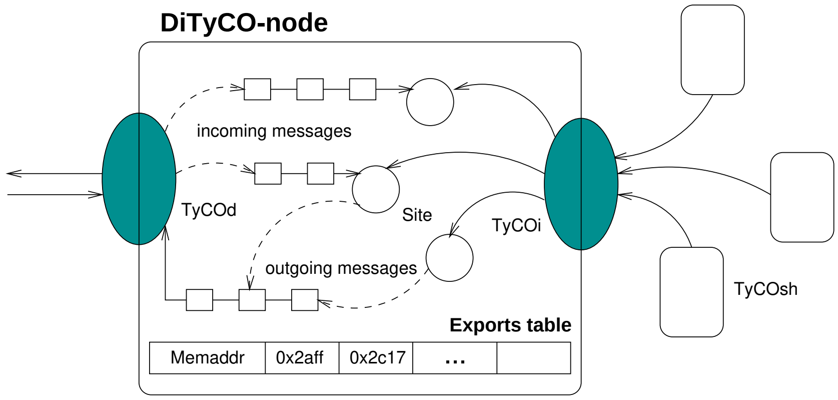

Figure 2: Node architecture.

1. the site that initiates the interaction talks to the local TyCOd;
2. the local TyCOd talks to the TyCOd of the node where the remote site is located, and;
3. the remote TyCOd talks to the target site.

In step 1 a site builds, in a buffer, the message, object or template that is to be sent to the target site. This buffer is then "atomically" placed in a global *ready-to-send* queue that is managed by TyCOd. Step 2 is supported by the network layer and was described in the previous section. Step 3 requires the remote TyCOd to process the incoming TCP/IP message, perform translation of some names from network format (hardware independent representation) into local addresses, and placing the buffer into a site specific queue of incoming messages or objects (this operation must be "atomic").

In addition to pure communication, the TyCOd also performs other duties, such as to check the availability of a name for a site (site names must be unique) and to handle **import/export** instructions. The TyCOd uses site specific tables – *export-tables*, to map names exported by a site with their local addresses in the heap. The key of the table is a unique variable identifier in the form of a natural number. The entries in these tables are added by each of the sites whenever local names are either explicitly exported or extruded in a remote interaction. Only explicitly exported names are added to the SNS table. Synchronization is required whenever a TyCOd has to translate names from network format into local addresses using a site's export-table. This is to ensure that the translation never occurs while the site is undergoing a garbage collection, which would make the translation incorrect and crash the system.

5.3 The site layer

Sites are implemented as threads, each running a DiTyCO byte-code emulator. The basic TyCO emulator [13] is very compact, uses a heap for dynamic data-structures, a run-queue to schedule byte-code blocks, two stacks for keeping local variable bindings and for evaluating builtin expressions. It is implemented in about 3k lines of C code, the binary uses just 29k, and consumes very little system and hardware resources. The DiTyCO emulator extends this architecture with new instructions and two queues for incoming/outgoing byte-code closures that interface with the local communication deamon (TyCOd). Figure 3 illustrates the architecture of the DiTyCO emulator.

The basic emulator features a *program* area where the byte-code is kept. The byte-code is composed of sequences of contiguous instructions and method tables (sequences of pointers into the byte-code). Dynamic data-structures such as objects, messages, channels and builtin values are allocated in the *heap*. The basic allocation unit in the heap is the *frame* and consists of one or more contiguous heap words with a descriptor for garbage collection. When a communication or instantiation occurs, a new

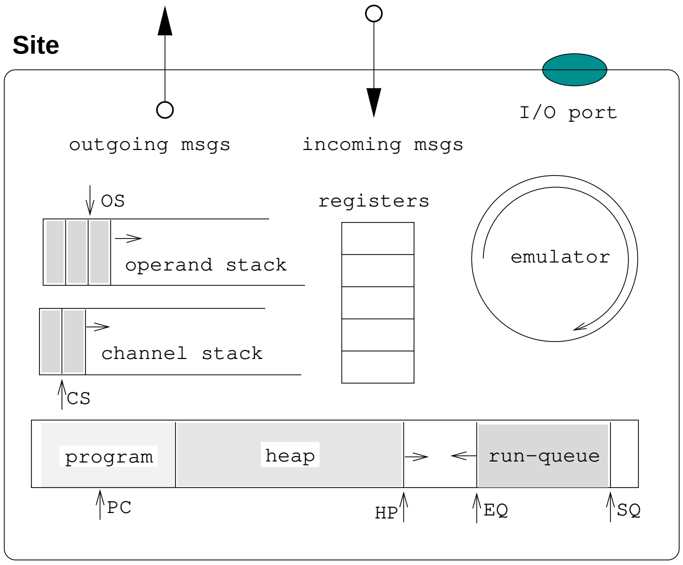

Figure 3: Site Architecture.

virtual machine thread – *vm_thread*, is created. A vm_thread is represented by a frame with three pointers: one for its byte-code, one for a frame with the parameter bindings and one for a frame with the bindings for the free variables. New vm_threads are allocated in the *run-queue* where they wait to be scheduled for execution. The heap and the run-queue are allocated in the bottommost and topmost areas, respectively, of a single memory block. They grow in opposite directions and garbage collection is triggered when a collision is predicted. New variables introduced with `new` statements, are allocated in the *channel stack*. Each position is associated with a new variable and its name binding. Finally, expressions with builtin data-types are evaluated in the *operand stack*. Simple values do not require evaluation and are copied directly in the heap.

6 Interfacing Sites with the Network Layer

The TyCO VM is slightly re-engineered to interface with the network software layer, that supports distribution and code mobility. We now describe the main changes introduced in the virtual machine architecture. First, names may now be either local to a site, that is, allocated in the local heap, or remote, in which case they are allocated in the heap of some other site. This duality between local and network names is reflected in their representation. Local names are represented as pointers to queues in the heap. Network addresses are hardware independent and keep information on which name, from which site, and within which IP address, we are referring to. Two additional instructions – `export` and `import` – are required to make local channels visible to other sites in the network and to fetch the network addresses of exported names, respectively.

The translation from local to network addresses occurs when a local site r sends names to a remote site s. These may be arguments of messages, or free variables of either objects or template definitions. All local names are translated from local pointers to network addresses, prefixed by r. All other names are untouched. The mapping between the local names and their network format is kept in a site specific *export table*.

The translation from network to local addresses is performed when the name bindings arrive at the destination site s. All names lexically bound to site s, those with a network address prefixed by s, are translated to local addresses. The other names remain untouched. In this case also, the translation uses information from the local site's *export table*.

The instructions that deal with communication, `trmsg` and `trobj`, and instantiation `instof`, in the TyCO emulator, must be extended to handle remote interaction. An instruction `trobj x` (take the current object and try to reduce it at name `x`) must first check the internal structure of the address in variable `x`. If it is a pointer to the local heap then the communication takes place within the site. This is the base case and is implemented as originally in the TyCO emulator. If, however, the address is in network format, then `x` is lexically bound to a name in a remote site and the communication must go through the node, and eventually up to the network layer. The site builds a buffer with the byte-code for the object extracted as described in section 4 plus the bindings for the free variables in the object's method collection. Local names in these bindings are translated to network format before being packed in the buffer. For each local name sent over the network as a free variable binding an entry is added to the site's export-table that preserves the mapping between the name and its current address in the heap.

The case for `trmsg x` (take the current message and try to reduce it at name `x`) is the dual, but here we are not required to extract or migrate byte-code. The parameter bindings sent in the message are translated to network format just as above.

For instantiations `instof l` (create an instance of the byte-code at `l`), the emulator must check the inner structure of label `l`. If it is a pointer to the local byte-code, then the template is defined locally and this is the base case. If, however, `l` is in network format then it is a reference to byte-code in a remote site. The emulator sends a request, through the TyCOd, to the remote site for the byte-code of the template. Once the byte-code with its free variable bindings arrives, the `instof` instruction is executed and the instance created.

Messages, objects and template definitions sent by remote sites are placed in the *incoming-queue* by the local TyCOd after some processing to be eventually grabbed by the emulator and used in local computations. Likewise, the emulator places outgoing closures in its *ready-to-send* queue where they are picked by the TyCOd and processed.

7 An Example

Assume we have two sites, borneo and sumatra, each with a specific location and node, respectively ("borneo",loc1,ip1) and ("sumatra",loc4,ip2). The sites interact through a name x which is explicitly exported at the beginning of the program running at sumatra. Assume the following source code running at each node.

borneo: **import** x **from** sumatra **in new** y x![y] | ...

sumatra: **export** x **in** **new** z x?(w)=(w?()=z![]) | ...

The export at site sumatra creates a new local variable x bound to a new name in the heap of that site. Next, an entry is inserted for the address of the variable at an unique index in sumatra's export-table. The index (say 3, for example) will be used to identify the variable hence-forward. Finally sumatra sends a message to TyCOd requesting the network to publish the name x. As a result, tuples ("sumatra",loc4,ip2) and ("x",var3,"sumatra") are inserted, respectively, at the sites and exported names tables (both located at the SNS server).

At site borneo, the emulator processes the import instruction. It first sends a message to the local TyCOd, which in turn queries the SNS requesting the network address of an x variable made available by a site sumatra. The network address (ip2,loc4,var3) is returned to borneo (the import operation is synchronous) and is bound to a local variable x. Still at the same site, the emulator proceeds to the next instruction which creates a new local variable y and binds it to an heap allocated name.

Next, a message process is found. The emulator first checks the destination of the message by looking at the kind of name pointed to by the variable x. In this case, x holds a remote name with address (ip2,loc4,var3). Hence, the message must be sent over the network to the location loc4 at the network address ip2. All arguments of the message local to borneo must be translated into network addresses and, for each argument, a local entry in the node's export-table must be inserted in order to preserve the mapping between variables and heap addresses. In this case y is bound to a local name and so a unique natural number identifier is picked (say 2) and used as an index to y's entry in the export-table. In the message sent to sumatra, y is written in network format, and so we have x![y] re-written as (ip2,loc4,var3)![(ip1,loc1,var2)].

Meanwhile, the emulator at sumatra has created a new local variable y bound to a heap allocated name and has processed the object x?(w)=(w?()=y![]). Since x is a local name and no messages have arrived yet for objects in x (so we assume), the emulator just creates an object frame in the heap and inserts it in the queue for name x.

The message (ip2,loc4,var3)![(ip1,loc1,var2)] coming from borneo arrives at ip2 and is received by the local TyCOd. The TyCOd checks the message for network addresses with prefix (ip2,loc4). This prefix identifies channels that originated at site sumatra. It then uses the node's export table to translate these network addresses into addresses in the heap. The address is selected from the table using the network address suffix as an index (e.g., 3 for (ip2,loc4,var3)). The processed message is then delivered to sumatra, which will eventually pick it up.

The emulator at sumatra upon receiving the message reduces it with the object previously stored at name x. The resulting process is placed in the run-queue and is later executed, generating an object w?() = y![] where the variable w is bound to the network address (ip1,loc1,var2) and y to the address of a local name.

The case now is the dual of the above. sumatra's emulator finds out that w is bound to a remote name. Since now we have an object the byte-code containing the method table and the method bodies is extracted from the program and placed in a buffer to be sent over the network to the location 1 at ip1. The free variables in the methods must also be sent with the byte-code and their addresses translated as described above. The heap address for the local variable y is then inserted (say at index 9) in the export-table of the node and is rewritten in the buffer with its network address: (ip2,loc4,var9). At this point the buffer is sent to TyCOd, which in turn forwards it to the TyCOd at ip1.

This packing, unpacking and book-keeping is repeated for each remote interaction between sites.

8 Future Work

We are currently in the process of implementing the network layer and its interface with the DiTyCO nodes – the communication daemons. Once this is done a first stripped down version of the system will be ready. There is still a lot of programming and research that needs to be done in order to have a robust and efficient system.

We need a distributed type checking mechanism to ensure secure interactions between remote sites. The idea is to have a mixture of static and dynamic type-checking.

On another front we need to introduce fault-tolerance and termination detection in the system. We want to be able to detect site failures, to reconfigure the computation topology and to cleanly terminate computations. There is quite a volume of work that has been done in this subject and we want to profit from it as much as possible.

Last but not least, we need to optimize the implementation so that it performs well with heterogenous network hardware and variable network loads. This implies not only fine-tuning the C code, but also looking for ways to reduce the overhead from the underlying physical network. For example, given that the data that is sent across networks in this model is usually very fine grained, strategies for packing buffers together before sending them may improve performance considerably.

Acknowledgements

The authors would like to thank the anonymous referees whose comments helped to improve the paper. Álvaro Figueira is supported by Ph.D. grant PRAXIS/BD/13971/97. This work was also partially supported by projects Dolphin (contract PRAXIS/2/2.1/TIT/1577/95) and DiCoMo (contract PRAXIS/P/EEI/12059/98).

References

[1] M. Abadi and A. Gordon. A Calculus for Cryptographic Protocols: the Spi-Calculus. In *Computer and Communications Security (CCS'97)*, pages 36–47. The ACM Press, April 1997.

[2] R. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In *International Conference on Coordination Models and Languages (COORDINATION'97)*, volume 1282 of *LNCS*, pages 374–391. Springer-Verlag, 1997.

[3] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[4] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, January 1995.

[5] L. Cardelli and A. Gordon. Mobile Ambients. In *Foundations of Software Science and Computation Structures (FoSSaCS'98)*, volume 1378 of *LNCS*, pages 140–155. Springer-Verlag, 1998.

[6] A. Chien et al. High Performance Virtual Machines (HPVM): Clusters with Supercomputing APIs and Performance. In *Parallel Processing for Scientific Computing (PP'97)*, March 1997.

[7] N. Boden et al. Myrinet: A Gigabit per second Local Area Network. *IEEE-Micro*, 15(1):29–36, February 1995.

[8] C. Fournet and G. Gonthier. The Reflexive Chemical Abstract Machine and the Join-Calculus. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 372–385. ACM, 1996.

[9] C. Fournet, G. Gonthier, and et al. A Calculus of Mobile Agents. In *International Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.

[10] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. Technical report, University of Sussex, February 1998.

[11] K. Honda and M. Tokoro. An Object Calculus for Asynchronous Communication. In *European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *LNCS*, pages 141–162. Springer-Verlag, 1991.

[12] K. Honda, V. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-Based Programming. In *European Symposium on Programming (ESOP'98)*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag, 1998.

[13] L. Lopes, F. Silva, and V. Vasconcelos. A Virtual Machine for the TyCO Process Calculus. In *Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *LNCS*, pages 244–260. Springer-Verlag, September 1999.

[14] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes (parts I and II). *Information and Computation*, 100(1):1–77, 1992.

[15] B. Pierce and D. Turner. Pict: A Programming Language Based on the Pi-Calculus. Technical Report CSCI 476, Computer Science Department, Indiana University, 1997. To appear in *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, MIT Press, 1999.

[16] J. Riely and M. Hennessy. Distributed Processes and Location Failures. In *Automata, Languages and Programming (ICALP'97)*, volume 1256 of *LNCS*, pages 471–481. Springer-Verlag, 1997.

[17] P. Sewell, P. Wojciechowski, and B. Pierce. Location Independence for Mobile Agents. In *Workshop on Internet Programming Languages – International Conference on Computer Languages (ICCL'98)*, 1998.

[18] V. Vasconcelos. Processes, Functions, Datatypes. *Theory and Practice of Object Systems*, 5(2):97–110, 1999.

[19] V. Vasconcelos and R. Bastos. Core-TyCO - The Language Definition. Technical Report TR-98-3, DI / FCUL, March 1998.

[20] V. Vasconcelos, L. Lopes, and F. Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.

[21] V. Vasconcelos and M. Tokoro. A Typing System for a Calculus of Objects. In *International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, volume 742 of *LNCS*, pages 460–474. Springer-Verlag, November 1993.