A module system for a reflective higher-order calculus

L.G. Meredith [1]

CTO, Djinnisys Corporation
505 N72nd St, Seattle, WA 98103

Matthias Radestock [2]

CTO, LShift, Ltd.
6 Rufus St, London N1 6PE

Abstract

Due to the atomic nature of names, encodings in the π-calculus of abstract data types – from natural numbers to queues – take the form of processes. To get a match between programming practice, in which elements of abstract data types are passed as the contents of messages (e.g., `XML`), and theory, one is then forced to adopt a higher-order approach, or to leave the relationship between the theory of data structures and the theory of programs unspelled out. In the former case another problem remains in that the process structure of data is at another level of abstraction than the process structure using these data as data. We illustrate an approach using a reflective, higher-order approach affords the possibility to model data as processes, but abstract away from the process structure of data when treating them as data. An appealing feature of the approach is that it identifies a possible unit of abstraction for reflective higher-order languages that may be used as the basis of a module system.

Key words: concurrency, message-passing, process calculus, reflection

1 Introduction

The π-calculus ([11]) is not a closed theory, but rather a theory dependent upon some theory of names. Taking an operational view, one may think of the π-calculus as a procedure that when handed a theory of names provides a theory of processes that communicate over those names. This openness of the theory has been exploited in π-calculus implementations, like the execution engine in Microsoft's Biztalk [9], where an ancillary binding language providing a means of specifying a 'theory' of names; e.g., names may be tcp/ip

[1] lgreg.meredith@gmail.com
[2] matthias@lshift.net

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

ports or urls or object references, etc. But, foundationally, one might ask if there is a closed theory of processes, i.e. one in which the theory of names arises from and is wholly determined by the theory of processes. Behind this question lurk a whole host of other exciting and potentially enlightening questions regarding the role of names with structure in calculi of interaction and the relationship between the structure of names and the structure of processes.

Speaking provocatively, nowhere in the tools available to the computer scientist is there a countably infinite set of *atomic* entities. All such sets, e.g. the natural numbers, the set of strings of finite length on some alphabet, etc., are *generated* from a finite presentation, and as such the elements of these sets inherit *structure* from the generating procedure. As a theoretician focusing on some aspects of the theory of processes built from such a set, one may temporarily forget that structure, but it is there nonetheless, and comes to the fore the moment one tries to build *executable* models of these calculi.

To illustrate the point, when names have structure, name equality becomes a computation. But, if our theory of interaction is to provide a basis for a theory of computation, then certainly this computation must be accounted for as well. Moreover, the fact that any realization of these name-based, mobile calculi of interaction must come to grips with names that have structure begs the question: would the theoretical account of interaction be more effective, both as a theory in its own right and as a guide for implementation, if it included an account of the relationships between the structure of names and the structure of processes?

Beyond these foundational questions we find a host of practical questions about the relationship between theory and practice. Due to the atomic nature of names, encodings in the π-calculus of abstract data types – from natural numbers to queues – take the form of processes. To get a match between programming practice, in which elements of abstract data types are passed as the contents of messages (e.g., `XML`), and theory, one is then forced to adopt a higher-order approach, or to leave the *relationship* between the theory of data structures and the theory of programs unspelled out. In the former case another problem remains in that the process structure of data is at another level of abstraction than the process structure using these data as data. We illustrate an approach using a reflective, higher-order approach affords the possibility to model data as processes, but abstract away from the process structure of data when treating them as data. An appealing feature of the approach is that it identifies a possible unit of abstraction for reflective higher-order languages that may be used as the basis of a module system.

1.1 Overview and contributions

In [**?**] presented a theory of an asynchronous message-passing calculus built on a notion of quoting. Names are quoted processes, and as such represent the code of a process, a reification of the syntactic structure of the process (up to some equivalence). Name-passing, then becomes a way of passing the code of a process as a message. In the presence of a dequote operation, turning the code of a process into a running instance, this machinery yields higher-order characteristics without the introduction of process

variables. Following the tradition started by Smith and des Rivieres, [3] they dubbed this ability to turn running code into data and back again, reflection; hence, the name *r*eflective, *h*igher-*o*rder calculus, or ρ-calculus for short.

Here we present a variation of that calculus that may be parameterized in an abstract data type (ADT). In fact, it accommodates extending the ADT with rewrite rules suggesting that certain kinds of rewrite systems might be an appropriate unit of abstraction for process calculi. The introduction these abstractions introduces context of greater complexity, forcing the development of a more graphical approach to the specification of the dynamics. The technique takes its inspiration from Milner's bigraphs [7].

2 Abstract data types and rewrite systems

We say that an abstract data type is a structure $\mathcal{A} = (A_0, A_1, ..., A_n)$, where A_i is the set of function symbols of arity i, A_0 denoting the set of constants.

Given a set, V, of *variables*, with $V \cap \bigcup_i A_i = \emptyset$, we build, recursively, the set, A_w, of terms as follows.

$$\frac{a \in A_0}{a \in A_w} \qquad (\textsc{const})$$

$$\frac{a_0, ..., a_n \in A_w \cup V,\ a \in A_{n+1}}{a(a_0, ..., a_n) \in A_w} \qquad (\textsc{expr})$$

We say that a term is *ground* if contains no variables. We let A_G denote the set of ground terms.

A substitution, σ, is a partial map from variables to terms. We write substitutions with finite domains $\{t_0/v_0, \ldots, t_n/v_n\}$. Substitutions lift, uniquely, to maps from terms to terms by $\sigma(f(t_0, \cdots, t_n)) = f(\sigma(t_0), \cdots, \sigma(t_n))$. We abuse notation and use the same symbol, polymorphically, for both maps. As is standard, we write application of substitutions to terms with the substitution on the right; e.g. $\sigma(t)$ is written $t\sigma$.

We take a set of rewrite rule to be a set of the form $React \subseteq A_w \times A_w$.

3 The calculus

3.0.1 Notation

We let P, Q, R range over processes and x, y, z range over names.

Given an ADT, $\mathcal{A}$, we build the terms of the calculus as follows.

ρ-calculus++	P, Q	$::= 0$	null process
		$\mid\ c \in A_0$	constant
		$\mid\ a(\!\lvert \vec{P} \rvert\!)$	term
		$\mid\ x(\vec{y})\ .\ P$	input
		$\mid\ x\langle\!\lvert \vec{P} \rvert\!\rangle$	lift
		$\mid\ \urcorner x \ulcorner$	drop
		$\mid\ P \mid Q$	parallel
	a	$::= x$	...
		$\mid\ f \in A_i, 1 \leq i \leq n$	function
	x, y	$::= \ulcorner P \urcorner$	quote

3.0.2 Quote

Working in a bottom-up fashion, we begin with names. The technical detail corresponding to the π-calculus' parametricity in a theory of *names* shows up in standard presentations in the grammar describing terms of the language: there is no production for names; names are taken to be terminals in the grammar. Our first point of departure from a more standard presentation of an asynchronous mobile process calculus is here. The grammar for the terms of the language will include a production for names in the grammar. A name is a *quoted* process, $\ulcorner P \urcorner$.

3.0.3 Parallel

This constructor is the usual parallel composition, denoting concurrent execution of the composed processes.

3.0.4 Lift and drop

Following [**?**] we maintain a careful disinction in kind between process and name; thus, name construction is not process construction. So, there must be a process constructor for a term that creates a name from a process. This is the motivation for the production $x\langle\!\lvert \vec{P} \rvert\!\rangle$, called the *lift* operator. Unlike the calculus in [**?**], however, both the lift operator and the input operator are polyadic. The intuitive meaning of this term is that the processes $\vec{P}$ will be packaged up as their codes, $\ulcorner \vec{P} \urcorner$, and ultimately made available as output at the port x.

Of course, when a name is a quoted process, it is very handy to have a way of evaluating such an entity. Thus, the $\urcorner x \ulcorner$ operator, pronounced *drop* x, (eventually) extracts the process from a name. We say 'eventually' because this extraction only happens when a

quoted process is substituted into this expression. A consequence of this behavior is that $\ulcorner x \urcorner$ is inert except under and input prefix. One way of saying this is that if you want to get something done, sometimes you need to drop a name, but it should be the name of an agent you know.

Remark 3.1 The lift operator turns out to play a role analogous to $(\nu\ x)P$. As mentioned in the introduction, it is essential to the computational completeness of the calculus, playing a key role in the implementation of replication. It also provides an essential ingredient in the compositional encoding of the ν operator.

Remark 3.2 It is well-known that replication is not required in a higher-order process algebra [14]. While our algebra is *not* higher-order in the traditional sense (there are not formal process variables of a different type from names) it has all the features of a higher-order process algebra. Thus, it turns out that there is no need for a term for recursion.

3.0.5 Input and output

The input constructor is standard for an asynchronous name-passing calculus. Input blocks its continuation from execution until it receives a communication. Lift is a form of output which – because the calculus is asynchronous – is allowed no continuation. It also affords a convenient syntactic sugar, which we define here.

$$x[y] \triangleq x\langle\!|\ulcorner y \urcorner|\!\rangle$$

3.0.6 Terms and the null process

In [**?**] the null process played a distinguished role. It provided the sole atom out of which all other processes (and the names they use) arise much in the same way that the number 0 is the sole number out of which the natural numbers are constructed; or the empty set is the sole set out of which all sets are built in ZF-set theory [8]; or the empty game is the sole game out of which all games are built in Conway's theory of games and numbers [2].

One of the theoretical motivations underlying the development of this calculus was the simple question: what would it mean to add additional atoms? Several questions immediately follow on from this one:

- what are laws governing interactions amongst these new atoms and what are those governing interaction between the new atoms and ordinary processes?
- what sort of name equality must obtain?

Considering examples, like the naturals, led to the observation that the naturals are in fact processes in Milner's original treatment – unavoidably so. Using Milner's encoding as our guide in the reflective setting, constants of our ADT, $\mathcal{A}$, show up as processes. They are atoms, just like 0. More general terms find representation in the form $f(\!|\vec{P}|\!)$, with (dropped) names providing the set of variables.

3.1 The name game

Before presenting some of the more standard features of a mobile process calculus, the calculation of free names, structural equivalence, etc., we wish to consider some examples of processes and names. In particular, if processes are built out of names, and names are built out of processes, is it ever possible to get off the ground? Fortunately, there is one process the construction of which involves no names, the null process, 0. Since we have at least one process, we can construct at least one name, namely $\ulcorner 0 \urcorner$ [3]. Armed with one name we can now construct at least two new processes that are evidently syntactically different from the 0, these are $\ulcorner 0 \urcorner[\ulcorner 0 \urcorner]$ and $\ulcorner 0 \urcorner(\ulcorner 0 \urcorner) \,.\, 0$. As we might expect, the intuitive operational interpretation of these processes is also distinct from the null process. Intuitively, we expect that the first outputs the name $\ulcorner 0 \urcorner$ on the channel $\ulcorner 0 \urcorner$, much like the ordinary π-calculus process $x[x]$ outputs the name x on the channel x, and the second inputs on the channel $\ulcorner 0 \urcorner$, much like the ordinary π-calculus process $x(x) \,.\, 0$ inputs on the channel x.

Of course, now that we have two more processes, we have two more names, $\ulcorner\ulcorner 0 \urcorner[\ulcorner 0 \urcorner]\urcorner$ and $\ulcorner\ulcorner 0 \urcorner(\ulcorner 0 \urcorner) \,.\, 0 \urcorner$. Having three names at our disposal we can construct a whole new supply of processes that generate a fresh supply of names, and we're off and running.

3.2 Free and bound names

The syntax has been chosen so that a binding occurrence of a name is sandwiched between round braces, $(\cdot)$. Thus, the calculation of the free names of a process, P, denoted $\mathcal{FN}(P)$ is given recursively by

$$\begin{aligned}
\mathcal{FN}(0) &= \emptyset \\
\mathcal{FN}(c) &= \emptyset \\
\mathcal{FN}(f(\!|P_0, ..., P_n|\!)) &= \bigcup_i \mathcal{FN}(P_i) \\
\mathcal{FN}(x(\!|P_0, ..., P_n|\!)) &= \{x\} \bigcup_i \mathcal{FN}(P_i) \\
\mathcal{FN}(x(y) \,.\, P) &= \{x\} \cup (\mathcal{FN}(P) \setminus \{y\}) \\
\mathcal{FN}(x\langle\!|P|\!\rangle) &= \{x\} \cup \mathcal{FN}(P) \\
\mathcal{FN}(P \mid Q) &= \mathcal{FN}(P) \cup \mathcal{FN}(Q) \\
\mathcal{FN}(\ulcorner x \urcorner) &= \{x\}
\end{aligned}$$

An occurrence of x in a process P is *bound* if it is not free. The set of names occurring in a process (bound or free) is denoted by $\mathcal{N}(P)$.

3.3 Structural congruence

The *structural congruence* of processes, noted $\equiv$, is the least congruence, containing α-equivalence, $\equiv_\alpha$, that satisfies the following laws:

[3] pun gratefully accepted ;-)

$$
\begin{aligned}
P \mid 0 &\equiv P \equiv 0 \mid P \\
P \mid Q &\equiv Q \mid P \\
(P \mid Q) \mid R &\equiv P \mid (Q \mid R)
\end{aligned}
$$

3.4 Name equivalence

As is noted in [**?**] name equivalence introduces one of the first real subtleties of reflective calculi. Fortunately, the subtleties may be safely ignored.

$$\frac{}{\ulcorner\urcorner x \ulcorner\urcorner \equiv_N x} \qquad \text{(Quote-drop)}$$

$$\frac{P \equiv Q}{\ulcorner P \urcorner \equiv_N \ulcorner Q \urcorner} \qquad \text{(Struct-equiv)}$$

3.5 Semantic substitution

We use $Proc$ for the set of processes, $\ulcorner Proc \urcorner$ for the set of names, and $\{\vec{y}/\vec{x}\}$ to denote partial maps, $s : \ulcorner Proc \urcorner \to \ulcorner Proc \urcorner$. A map, s lifts, uniquely, to a map on process terms, $\widehat{s} : Proc \to Proc$ by the following equations.

$$
\begin{aligned}
(0)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} &= 0 \\
(R \mid S)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} &= (R)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} \mid (S)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} \\
(x(\vec{y}) \,.\, R)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} &= (x)\{\ulcorner Q \urcorner / \ulcorner P \urcorner\}(\vec{z}) \,.\, ((R\{\widehat{\vec{z}/\vec{y}}\})\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\}) \\
(\vec{x}\langle\!|R|\!\rangle)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} &= (x)\{\ulcorner Q \urcorner / \ulcorner P \urcorner\}\langle\!|R\{\widehat{\ulcorner \vec{Q} \urcorner / \ulcorner P \urcorner}\}|\!\rangle \\
(\ulcorner x \urcorner)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} &= \begin{cases} Q & x \equiv_N \ulcorner P \urcorner \\ \ulcorner x \urcorner & otherwise \end{cases} \\
(f(|\vec{R}|))\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} &= f(|R\{\widehat{\ulcorner \vec{Q} \urcorner / \ulcorner P \urcorner}\}|) \\
(x(|\vec{R}|))\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} &= \begin{cases} \ulcorner Q \urcorner(|R\{\widehat{\ulcorner \vec{Q} \urcorner / \ulcorner P \urcorner}\}|) & x \equiv_N \ulcorner P \urcorner \\ x(|R\{\widehat{\ulcorner \vec{Q} \urcorner / \ulcorner P \urcorner}\}|) & otherwise \end{cases}
\end{aligned}
$$

where

$$(x)\{\ulcorner Q \urcorner / \ulcorner P \urcorner\} = \begin{cases} \ulcorner Q \urcorner & x \equiv_N \ulcorner P \urcorner \\ x & otherwise \end{cases}$$

and $\vec{z}$ are chosen distinct from $\ulcorner P \urcorner$, $\ulcorner Q \urcorner$, the free names in Q, and all the names in R. In the remainder of the paper we will refer to semantic substitutions simply as substitutions. Similarly, we will abuse notation and write $\{y/x\}$ for $\widehat{\{y/x\}}$.

3.6 Dynamic quote: an example

Anticipating something of what's to come, consider applying the substitution, $\widehat{\{u/z\}}$, to the following pair of processes, $w\langle\!|y\texttt{[}z\texttt{]}|\!\rangle$ and $w\texttt{[}\ulcorner y\texttt{[}z\texttt{]}\urcorner\texttt{]}$.

$$w\langle\!|y\texttt{[}z\texttt{]}|\!\rangle\widehat{\{u/z\}} = w\langle\!|y\texttt{[}u\texttt{]}|\!\rangle$$
$$w\texttt{[}\ulcorner y\texttt{[}z\texttt{]}\urcorner\texttt{]}\widehat{\{u/z\}} = w\texttt{[}\ulcorner y\texttt{[}z\texttt{]}\urcorner\texttt{]}$$

Because the body of the process between quotes is impervious to substitution, we get radically different answers. In fact, by examining the first process in an input context, e.g. $x\texttt{(}z\texttt{)} \,.\, w\langle\!|y\texttt{[}z\texttt{]}|\!\rangle$, we see that the process under the lift operator may be shaped by prefixed inputs binding a name inside it. In this sense, the lift operator will be seen as a way to dynamically construct processes before reifying them as names.

3.7 Contexts

The introduction of terms results in communication possibilities that could not arise in the calculus of [**?**]. Here is a motivating example encoding the sequential composition, $f(\!|a_0, \ldots, a_m|\!); g(\!|b_0, \ldots, b_n|\!)$.

$$\begin{aligned} & x_0\langle\!|a_0, \ldots, a_m|\!\rangle \\ & \texttt{|}\, x_0\texttt{(}v_0, \ldots, v_m\texttt{)} \,.\, f(\!|(\ulcorner v_0 \urcorner \texttt{|}\, z\texttt{[]}), \ldots, (\ulcorner v_m \urcorner \texttt{|}\, z\texttt{[]})|\!) \\ & \texttt{|}\, z\texttt{()} \,.\, \ldots \,.\, z\texttt{()} \,.\, g(\!|b_0, \ldots, b_n|\!) \end{aligned}$$

This example requires communication from the lift on z embedded in the lift on f to the awaiting input on z in parallel with the lift on f. It is more difficult to capture this example with the standard form of the rule for communication. This is where another of Milner's innovations provides guidance. Specifically, bigraphs provide a more general notion of how to specify the dynamics of systems. We do not need the full generality of the bigraph machinery, and may specify the dynamics with two-holed contexts; but, bigraphs provided the core insights into developing a simple presentation of the dynamics of this calculus.

First we develop a restricted class of one-holed contexts.

1-holed contexts	$K_{(1)}$	$::= \Box$	hole
		$\mid\ a(\!|\vec{P}, K_{(1)}, \vec{Q}|\!)$	term
		$\mid\ x\langle\!|\vec{P}, K_{(1)}, \vec{Q}|\!\rangle$	lift
		$\mid\ P \texttt{ | } K_{(1)}$	parallel

Then we build the two-holed contexts we need out of these.

$$
\begin{array}{lrll}
\text{2-holed contexts} & K_{(2)} ::= & K_{(1)} \mid K_{(1)} & \text{2 1-holed parallel} \\
& \mid & a(\!|\vec{P}, K_{(1)}, \vec{Q}, K_{(1)}, \vec{R}|\!) & \text{2 1-holed term} \\
& \mid & x\langle\!|\vec{P}, K_{(1)}, \vec{Q}, K_{(1)}, \vec{R}|\!\rangle & \text{2 1-holed lift} \\
& \mid & P \mid K_{(2)} & \text{1 2-holed parallel} \\
& \mid & a(\!|\vec{P}, K_{(2)}, \vec{Q}|\!) & \text{1 2-holed term} \\
& \mid & x\langle\!|\vec{P}, K_{(2)}, \vec{Q}|\!\rangle & \text{1 2-holed lift}
\end{array}
$$

Finally, equipped with this machinery, we can specify the dynamics of the calculus.

3.8 Operational Semantics

Taking another page from bigraphs, we assume that our ADT, $\mathcal{A}$, has also come equipped with a set of rewrite rules of its own, $React_{\mathcal{A}}$. To make use of these we must also have been provided with a function, $\mathcal{E} : A_0 \cup V \to A_0 \cup \ulcorner Proc \urcorner$, acting as an identity on constants while mapping variables to names. Such a function may be lifted to a function mapping terms built from $\mathcal{A}$ and V, to terms built from $\mathcal{A}$ and $\ulcorner Proc \urcorner$ in the obvious manner. We abuse notation and use $\mathcal{E}$ in polymorphic fashion to denote both the original and extended function, relying on context to convey which is meant. Then, the reduction rules for ρ-calculus++ are

$$\frac{(f(a_0, \ldots, a_m), g(b_0, \ldots, b_n)) \in React_{\mathcal{A}}}{f(\!|\mathcal{E}(a_0), \ldots, \mathcal{E}(a_m)|\!) \to g(\!|\mathcal{E}(b_0), \ldots, \mathcal{E}(b_n)|\!)} \quad (\textsc{React})$$

$$\frac{x_{src} \equiv_N x_{trgt}}{K_{(2)}[x_{src}\langle\!|Q|\!\rangle, x_{trgt}(y)\ .\ P] \to K_{(2)}[0, P\{\ulcorner Q \urcorner / y\}]} \quad (\textsc{Comm})$$

In addition, we have the following context rules:

$$\frac{P \to P'}{K_{(1)}[P] \to K_{(1)}[P']} \quad (\textsc{Context})$$

$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q} \quad (\textsc{Equiv})$$

The React rule effectively imports the rewrite rules from our structure, $\mathcal{A}$.

The comm rule specifies the contexts in which input guarded processes may synchronize and communicated with lift processes. For example, using the comm rule and name equivalence we can now justify our syntactic sugar for output.

$$\begin{aligned} & x[z] \mid x(y) \,.\, P \\ =\, & x\langle\!\lceil \ulcorner z \urcorner \rceil\!\rangle \mid x(y) \,.\, P \\ \rightarrow \quad & P\{\ulcorner\urcorner z \ulcorner\urcorner / y\} \\ \equiv \quad & P\{z/y\} \end{aligned}$$

The context rules also distinguish the ρ-calculus++ from ρ-calculus. In particular, lift was *not* an evaluation context for ρ-calculus, while it is here.

4 Bisimulation

[Ed. note: the introduction of terms significantly increases the complexity of bisimulation. In particular, a term may now be a barb.]

Having taken the notion of restriction out of the language, we carefully place it back into the notion of observation, and hence into the notion of program equality, i.e. bisimulation. That is, we parameterize the notion of barbed bisimulation by a set of names over which we are allowed to set the barbs. The motivation for this choice is really comparison with other calculi. The set of names of the ρ-calculus++ is *global*. It is impossible, in the grammar of processes, to guard terms from being placed into contexts that can potentially observe communication. So, we provide a place for reasoning about such limitations on the scope of observation in the theory of bisimulation.

Definition 4.1 An *observation relation*, $\downarrow_{\mathcal{N}}$, over a set of names, $\mathcal{N}$, is the smallest relation satisfying the rules below.

$$\frac{y \in \mathcal{N},\ x \equiv_N y}{x[v] \downarrow_{\mathcal{N}} x} \qquad \text{(Out-barb)}$$

$$\frac{P \downarrow_{\mathcal{N}} x \text{ or } Q \downarrow_{\mathcal{N}} x}{P \mid Q \downarrow_{\mathcal{N}} x} \qquad \text{(Par-barb)}$$

We write $P \Downarrow_{\mathcal{N}} x$ if there is Q such that $P \Rightarrow Q$ and $Q \downarrow_{\mathcal{N}} x$.

Notice that $x(y).P$ has no barb. Indeed, in ρ-calculus++ as well as other asynchronous calculi, an observer has no direct means to detect if a message sent has been received or not.

Definition 4.2 An $\mathcal{N}$-*barbed bisimulation* over a set of names, $\mathcal{N}$, is a symmetric binary relation $\mathcal{S}_{\mathcal{N}}$ between agents such that $P\ \mathcal{S}_{\mathcal{N}} Q$ implies:

(i) If $P \rightarrow P'$ then $Q \Rightarrow Q'$ and $P'\ \mathcal{S}_{\mathcal{N}} Q'$.

(ii) If $P \downarrow_{\mathcal{N}} x$, then $Q \Downarrow_{\mathcal{N}} x$.

P is $\mathcal{N}$-barbed bisimilar to Q, written $P \dot{\approx}_{\mathcal{N}} Q$, if $P \mathcal{S}_{\mathcal{N}} Q$ for some $\mathcal{N}$-barbed bisimulation $\mathcal{S}_{\mathcal{N}}$.

5 Applications

...

5.1 The naturals

...

5.2 Combinatorial chemistry

...

5.3 `XML`

...

5.4 Ambients

...

5.5 Modules

...

6 Conclusions and future work

We studied an extension of an asynchronous message-passing calculus built out of a notion of quote allowing such a calculus to be parameterized in ADT's and/or rewrite systems. We examine several applications from practical programming considerations to more theoretical concerns.

We note that this work is situated in the larger context of a growing investigation into naming and computation. Milner's studies of action calculi led not only to reflexive action calculi [12], but to Power's and Hermida's work on name-free accounts of action calculi [6] as well as Pavlovic's [13]. Somewhat farther afield, but still related, is Gabbay's theory of freshness [5]. Very close to the mark, Carbone and Maffeis observe a tower of expressiveness resulting from adding very simple structure to names [10]. In some sense, this may be viewed as approaching the phenomena of structured names 'from below'. By making names be processes, this work may be seen as approaching the same phenomena 'from above'. But, both investigations are really the beginnings of a much longer and deeper investigation of the relationship between process structure and name structure.

Beyond foundational questions concerning the theory of interaction, such an investigation may be highly warranted in light of the recent connection between concurrency theory and biology. In particular, despite the interesting results achieved by researchers in this field, there is a fundamental difference between the kind of synchronization observed in the π-calculus and the kind of synchronization observed between molecules at the bio-molecular level. The difference is that interactions in the latter case occur at sites with extension and behavior of their own [4]. An account of these kinds of phenomena may be revealed in a detailed study of the relationship between the structure of names and the structure of processes.

Acknowledgments.

The authors wish to thank Robin Milner for his thoughtful and stimulating remarks regarding earlier work in this direction, and Cosimo Laneve for urging us to consider a version of the calculus without heating rules.

References

[1] Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[2] John Horton Conway. *On Numbers and Games*. Academic Press, 1976.

[3] J. des Rivieres and B. C. Smith. The implementation of procedurally reflective languages. In *ACM Symposium on Lisp and Functional Programming*, pages 331–347, 1984.

[4] Walter Fontana. private conversation. 2004.

[5] M. J. Gabbay. The π-calculus in FM. In Fairouz Kamareddine, editor, *Thirty-five years of Automath*. Kluwer, 2003.

[6] Claudio Hermida and John Power. Fibrational control structures. In *CONCUR*, pages 117–129, 1995.

[7] Ole Høgh Jensen and Robin Milner. Bigraphs and transitions. In *POPL*, pages 38–49, 2003.

[8] Jean-Louis Krivine. The curry-howard correspondence in set theory. In Martin Abadi, editor, *Proceedings of the Fifteenth Annual IEEE Symp. on Logic in Computer Science, LICS 2000*. IEEE Computer Society Press, June 2000.

[9] Microsoft Corporation. Microsoft biztalk server. microsoft.com/biztalk/default.asp.

[10] M.Carbone and S.Maffeis. On the expressive power of polyadic synchronisation in pi-calculus. *Nordic Journal of Computing*, 10(2):70–98, 2003.

[11] Robin Milner. The polyadic π-calculus: A tutorial. *Logic and Algebra of Specification*, Springer-Verlag, 1993.

[12] Robin Milner. Strong normalisation in higher-order action calculi. In *TACS*, pages 1–19, 1997.

[13] Dusko Pavlovic. Categorical logic of names and abstraction in action calculus. *Math. Structures in Comp. Sci.*, 7:619–637, 1997.

[14] David Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes.* Cambridge University Press, 2001.

[15] Davide Sangiorgi. Bisimulation in higher-order process calculi. *Information and Computation*, 131:141–178, 1996.

[16] B. Thomsen. A Theory of Higher Order Communication Systems. *Information and Computation*, 116(1):38–57, 1995.