Namespace logic: A logic for a reflective higher-order calculus

L.G. Meredith[1] and Matthias Radestock[2]

[1] CTO, Djinnisys Corporation
505 N72nd St, Seattle, WA 98103, USA,
`lgreg.meredith@gmail.com`
[2] CTO, LShift, Ltd.
6 Rufus St, London N1 6PE, UK
`matthias@lshift.net`

Abstract. In [19] it was observed that a theory like the π-calculus, dependent on a theory of names, can be closed, through a mechanism of quoting, so that (quoted) processes provide the necessary notion of names. Here we expand on this theme by examining a construction for a Hennessy-Milner logic corresponding to an asynchronous message-passing calculus built on a notion of quoting.

Like standard Hennessy-Milner logics, the logic exhibits formulae corresponding to sets of processes, but a new class of formulae, corresponding to sets of names, also emerges. This feature provides for a number of interesting possible applications from security to data manipulation. Specifically, we illustrate formulae for controlling process response on ranges of names reminiscent of a (static) constraint on port access in a firewall configuration. Likewise, we exhibit formulae in a names-as-data paradigm corresponding to validation for fragment of `XML Schema`.

1 Introduction

Starting from the practical end of things, whether we consider MAC addresses, IP addresses, domain names or URL's it is clear that distributed computing is practiced, today, using names. Moreover, it is essential to the programs that administer as well as to the ones that compute over this distributed computing infrastructure that these names have structure. Thus, when we look to theory, especially a theory, like the π-calculus, of computing based on interaction over named channels, to help us with this practice some story must be told about how the structure of these names contributes to interaction and computation over (channels named by) them.

Starting from the theoretical end, nowhere in the tools available to the computer scientist is there a countably infinite set of *atomic* entities that might function as names. 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.

Thus the fact that the π-calculus ([20]) is not a closed theory, but rather a theory dependent upon some theory of names is both enabling and limiting. This openness of the theory has been exploited, for example, in π-calculus implementations, like the execution engine in Microsoft's Biztalk [17], where an ancillary binding language providing a means of specifying a 'theory' of names; e.g., names may be tcp/ip ports or urls or object references, etc. Reasoning foundationally, however, 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 – especially of *distributed* 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?

1.1 Overview and contributions

In [19] we presented a theory of an asynchronous message-passing calculus built on a notion of quoting in which names have the structure of quoted processes, and may be thought of as representing the code of some process, i.e. a reification of the syntactic structure of some process (up to some equivalence). Name-passing, as such, becomes a way of passing the code of a process as a message, and 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. [3] As is standard with higher-order calculi, replication and/or recursion is no longer required as a primitive operation. Somewhat more interestingly, the introduction of a process constructor to dynamically convert a process into its code is essential to obtain computational completeness, and simultaneously supplants the function of the ν operator. [4]

In this paper we take the idea a little further via an investigation of a Hennessy-Milner logic for this calculus. The logic is a form of spatial logic ([6], [7]) with operators detecting structural as well as behavioral content of process. Further, like many other logics for message-passing calculi it describes formulae denoting sets of such processes in a more or less standard manner, but the additional reflective structure on names also gives rise to a new class of formulae. These formulae denote sets of *names*, referred to in the sequel as namespaces and causing us to dub the logic *namespace logic*.

These new formulae suggest approaches to various application domains, e.g. reasoning about security, or the structure of the data passed between processes, that differ somewhat from the current treatment of these domains using message-passing calculi. For example, the analytic framework was not designed with security in mind, and as such has no additional security-specific features like nonce construction or unpacking, as is found in Gordon's spi-calculus ([3]), and yet has very simple formulae to express such properties as that a process will only ever receive requests from a given range of ports. Moreover, these properties are expressed as *formulae*, not as process specifications, thus observance is measured by satisfaction not protocol equivalence. Further, while closer in spirit – ala the proposition-as-types paradigm – to type-based approaches like Gordon and Jeffrey's approach to typing correspondence assertions [14] or Abadi's various type systems for security ([2] [1]), it is a logic and not a type system with the attendant advantages and disadvantages. For example, a very broad range of properties may be expressed, but the system is only semi-decidable. Likewise, neither the calculus nor the logic were designed with any particular data analysis in mind, and yet we find relatively simple treatment of the semantics of validation for a fragment of `XML` schema.

While the main focus of the paper is the logic, and some suggestive examples, to provide a self-contained presentation, the paper also presents a concrete instance of a minimal reflective asynchronous message-passing calculus and the manner in which its processes and names witness the formulae of the logic. As in [19] where we took the view that the main contribution of the concrete machinery was to provide an instrument to bring to life a set of questions regarding the role of names in calculi of interaction, here we assert that the real contribution manifest by the logic is an instrument to better frame and sharpen those questions. These questions include the calculation of name equality as a computation to be considered within the framework of interaction and the roles of name equality in substitution versus synchronization. These questions don't really come to life, though, without the instruments in hand. So, we turn immediately to the formal presentation.

2 The calculus

This presentation is essentially the same as the one found in [19].

---

[3] Following the tradition started by Smith and des Rivieres, [10] we dubbed this ability to turn running code into data and back again, reflection; and hence, called the calculus the *r*eflective, *h*igher-*o*rder calculus, or rho-calculus, for short, or ρ-calculus for even shorter.

[4] In fact, [19] gives a compositional encoding of the ν operator into the calculus, making essential use of dynamic quote as well as dequote.

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

$$
\begin{array}{lll}
\rho\text{-calculus} & P, Q ::= 0 & \text{null process} \\
& \mid\ x(y)\ .\ P & \text{input} \\
& \mid\ x\langle\!|P|\!\rangle & \text{lift} \\
& \mid\ \ulcorner\neg x\urcorner & \text{drop} \\
& \mid\ P \mid Q & \text{parallel} \\
& x, y ::= \ulcorner P\urcorner & \text{quote}
\end{array}
$$

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, $\ulcorner P\urcorner$.

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

Lift and drop Despite the fact that names are built from (the codes of) processes, we still maintain a careful disinction in kind between process and name; thus, name construction is not process construction. So, if one wants to be able to generate a name from a given process, there must be a process constructor for a term that creates a name from a process. This is the motivation for the production $x\langle\!|P|\!\rangle$, dubbed here the *lift* operator. The intuitive meaning of this term is that the process P will be packaged up as its code, $\ulcorner P\urcorner$, and ultimately made available as an output at the port x.

A more formal motivation for the introduction of this operator will become clear in the sequel. But, it will suffice to say now that $\ulcorner P\urcorner$ is impervious to substitution. In the ρ-calculus, substitution does not affect the process body between quote marks. On the other hand, $x\langle\!|P|\!\rangle$ is susceptible to substitution and as such constitutes a dynamic form of quoting because the process body ultimately quoted will be different depending on the context in which the $x\langle\!|P|\!\rangle$ expression occurs.

Of course, when a name is a quoted process, it is very handy to have a way of evaluating such an entity. Thus, the $\ulcorner\neg x\urcorner$ 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 $\ulcorner\neg x\urcorner$ 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 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 2. It is well-known that replication is not required in a higher-order process algebra [23]. 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. To illustrate this we present below an encoding of $!P$ in this calculus. Intuitively, this will amount to receiving a quoted form of a process, evaluating it, while making the quoted form available again. The reader familiar with the λ-calculus will note the formal similarity between the crucial term in the encoding and the paradoxical combinator [4].

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.

$$x\,[y] \triangleq x\langle\!|^\neg y^\ulcorner\!|\rangle$$

The null process As we will see below, the null process has a more distinguished role in this calculus. It provides 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 [16]; or the empty game is the sole game out of which all games are built in Conway's theory of games and numbers [8]. This analogy to these other theories draws attention, in our opinion, to the foundational issues raised in the introduction regarding the design of calculi of interaction.

2.1 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 $\ulcorner 0 \urcorner$ [5]. 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. It should be pointed out, though, that as soon as we had the null process we also had 0 | 0 and 0 | 0 | 0 and consequently, we had the names $\ulcorner 0\ |\ 0 \urcorner$, and $\ulcorner 0\ |\ 0\ |\ 0 \urcorner$, and .... But, since we ultimately wish to treat these compositions as merely other ways of writing the null process and not distinct from it, should we admit the codes of these processes as distinct from $\ulcorner 0 \urcorner$?

This question leads to several intriguing and apparently fundamental questions. Firstly, if names have structure, whether this derives from the structure of processes or something else, what is a reasonable notion of equality on names? How much computation, and of what kind, should go into ascertaining equality on names? Additionally, what roles should name equality play in a calculus of processes? In constructing this calculus we became conscious that substitution and synchronization identify two potentially very different roles for name equality to play in name-passing calculi. That these are very different roles is suggested by the fact that they may be carried out by very different mechanisms in a workable and effective theory. We offer one choice, but this is just one design choice among infinitely many. Most likely, the primary value of this proposal is to raise the question. Likewise, we offer a proposal regarding the calculation of name equality that is just one of many and whose real purpose is to make the question vivid. We wish to turn to the core mechanics of the calculus with these questions in mind.

2.2 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

---

[5] pun gratefully accepted ;-)

$$\mathcal{FN}(0) = \emptyset$$
$$\mathcal{FN}(x(y) \, . \, P) = \{x\} \cup (\mathcal{FN}(P) \setminus \{y\})$$
$$\mathcal{FN}(x\langle\!\langle P\rangle\!\rangle) = \{x\} \cup \mathcal{FN}(P)$$
$$\mathcal{FN}(P \mid Q) = \mathcal{FN}(P) \cup \mathcal{FN}(Q)$$
$$\mathcal{FN}(\ulcorner x\urcorner) = \{x\}$$

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 $\mathcal{N}(P)$.

2.3 Structural congruence

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

$$P \mid 0 \equiv P \equiv 0 \mid P$$
$$P \mid Q \quad \equiv \quad Q \mid P$$
$$(P \mid Q) \mid R \quad \equiv \quad P \mid (Q \mid R)$$

2.4 Name equivalence

We now come to one of the first real subtleties of this calculus. Both the calculation of the free names of a process and the determination of structural congruence between processes critically depend on being able to establish whether two names are equal. In the case of the calculation of the free names of an input-guarded process, for example, to remove the bound name we must determine whether it is in the set of free names of the continuation. Likewise, structural congruence includes α-equivalence. But, establishing α-equivalence between the processes $x(z) \, . \, w\langle\!\langle y[z]\rangle\!\rangle$ and $x(v) \, . \, w\langle\!\langle y[v]\rangle\!\rangle$, for instance, requires calculating a substitution, e.g. $x(v) \, . \, w\langle\!\langle y[v]\rangle\!\rangle\{z/v\}$. But this calculation requires, in turn, being able to determine whether two names, in this case the name in the object position of the output, and the name being substituted for, are equal.

As will be seen, the equality on names involves structural equivalence on processes, which in turn involves alpha equivalence, which involves name equivalence. This is a subtle mutual recursion, but one that turns out to be well-founded. Before presenting the technical details, the reader may note that the grammar above enforces a strict alternation between quotes and process constructors. Each question about a process that involves a question about names may in turn involve a question about processes, but the names in the processes the next level down, as it were, are under fewer quotes. To put it another way, each 'recursive call' to name equivalence will involve one less level of quoting, ultimately bottoming out in the quoted zero process.

Let us assume that we have an account of (syntactic) substitution and α-equivalence upon which we can rely to formulate a notion of name equivalence, and then bootstrap our notions of substitution and α-equivalence from that. We take name equivalence, written $\equiv_N$, to be the smallest equivalence relation generated by the following rules.

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

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

2.5   Syntactic substitution

Now we build the substitution used by α-equivalence. We use $Proc$ for the set of processes, $\ulcorner Proc \urcorner$ for the set of names, and $\{\boldsymbol{y}/\boldsymbol{x}\}$ 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.

$$(0)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} = 0$$

$$(R \mid S)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} = (R)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} \mid (S)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\}$$

$$(x(y) \, . \, R)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} = (x)\{\ulcorner Q \urcorner / \ulcorner P \urcorner\}(z) \, . \, ((R\{z/y\})\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\})$$

$$(x \langle\!| R |\!\rangle)\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} = (x)\{\ulcorner Q \urcorner / \ulcorner P \urcorner\} \langle\!| R\{\widehat{\ulcorner Q \urcorner / \ulcorner P \urcorner}\} |\!\rangle$$

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

where

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

and z is chosen distinct from $\ulcorner P \urcorner$, $\ulcorner Q \urcorner$, the free names in Q, and all the names in R. Our α-equivalence will be built in the standard way from this substitution.

But, given these mutual recursions, the question is whether the calculation of $\equiv_N$ (respectively, $\equiv$, $\equiv_\alpha$) terminates. To answer this question it suffices to formalize our intuitions regarding level of quotes, or quote depth, $\#(x)$, of a name x as follows.

$$\#(\ulcorner P \urcorner) = 1 + \#(P)$$

$$\#(P) = \begin{cases} max\{\#(x) : x \in \mathcal{N}(P)\} & \mathcal{N}(P) \neq \emptyset \\ 0 & otherwise \end{cases}$$

The grammar ensures that $\#(\ulcorner P \urcorner)$ is bounded. Then the termination of $\equiv_N$ (respectively, $\equiv$, $\equiv_\alpha$) is an easy induction on quote depth.

2.6   Dynamic quote: an example

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

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

$$w[\ulcorner y[z] \urcorner] \widehat{\{u/z\}} = w[\ulcorner y[z] \urcorner]$$

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. $x(z) \, . \, w \langle\!| y[z] |\!\rangle$, 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.

2.7 Semantic substitution

The substitution used in α-equivalence is really only a device to formally recognize that binding occurrences do not depend on the specific names. It is not the engine of computation. The proposal here is that while synchronization is the driver of that engine, the real engine of computation is a semantic notion of substitution that recognizes that a dropped name is a request to run a process. Which process? Why the one whose code has been bound to the name being dropped. Formally, this amounts to a notion of substitution that differs from syntactic substitution in its application to a dropped name.

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

In the remainder of the paper we will refer to semantic and syntactic substitutions simply as substitutions and rely on context to distinguish which is meant. Similarly, we will abuse notation and write $\{y/x\}$ for $\widehat{\{y/x\}}$.

Finally equipped with these standard features we can present the dynamics of the calculus.

2.8 Operational Semantics

The reduction rules for ρ-calculus are

$$\frac{x_0 \equiv_N x_1}{x_0 \langle\!| Q |\!\rangle \mid x_1(y) . P \to P\{\ulcorner Q \urcorner / y\}} \tag{COMM}$$

In addition, we have the following context rules:

$$\frac{P \to P'}{P \mid Q \to P' \mid Q} \tag{PAR}$$

$$\frac{P \equiv P' \qquad P' \to Q' \qquad Q' \equiv Q}{P \to Q} \tag{EQUIV}$$

The context rules are entirely standard and we do not say much about them, here. The communication rule does what was promised, namely make it possible for agents to synchronize and communicate processes packaged as names. For example, using the comm rule and name equivalence we can now justify our syntactic sugar for output.

$$\begin{aligned} & x[z] \mid x(y) . P \\ = {} & x\langle\!| \ulcorner z \urcorner |\!\rangle \mid x(y) . P \\ \to {} & P\{\ulcorner \ulcorner z \urcorner \urcorner / y\} \\ \equiv {} & P\{z/y\} \end{aligned}$$

But, it also provides a scheme that identifies the role of name equality in synchronization. There are other relationships between names with structure that could also mediate synchronization. Consider, for example, a calculus identical to the one presented above, but with an alternative rule governing communication.

$$\frac{\forall R.[P_{channel} \mid Q_{channel} \to^* R] \Rightarrow R \to^* 0}{\ulcorner Q_{channel} \urcorner \langle\!| Q |\!\rangle \mid \ulcorner P_{channel} \urcorner (y) . P \to P\{\ulcorner Q \urcorner / y\}} \tag{COMM-ANNIHILATION}$$

Intuitively, it says that the codes of a pair of processes, $P_{channel}$, $Q_{channel}$, stand in channel/co-channel relation just when the composition of the processes always eventually reduces to 0, that is, when the processes annihilate one another. This rule is well-founded, for observe that because $0 \equiv 0 \mid 0$, $0 \mid 0 \rightarrow^* 0$. Thus, $\ulcorner 0 \urcorner$ serves as its own co-channel. Analogous to our generation of names from 0, with one such channel/co-channel pair, we can find many such pairs. What we wish to point out about this rule is that we can see precisely an account of the calculation of the channel/co-channel relationship as deriving from the theory of interaction. We do not know if the computation of name equality has a similar presentation, driving home the potential difference of those two roles in calculi of interaction.

We mention, as a brief aside, that there is no reason why 0 is special in the scheme above. We posit a family of calculi, indexed by a set of processes $\{S_\alpha\}$, and differing only in their communication rule each of which conforms to the scheme below.

$$\frac{\forall R.[P_{channel} \mid Q_{channel} \rightarrow^* R] \Rightarrow R \rightarrow^* R' \equiv S_\alpha}{\ulcorner Q_{channel} \urcorner \langle\!| Q |\!\rangle \mid \ulcorner P_{channel} \urcorner (y) \cdot P \rightarrow P\{\ulcorner Q \urcorner / y\}} \quad \text{(Comm-annihilation-S)}$$

We explore this family of calculi in a forthcoming paper. For the rest of this paper, however, we restrict our attention to the calculus with the less exotic communication rule, using $\rightarrow$ for reduction according to that system and $\Rightarrow$ for $\rightarrow^*$.

3 Replication

As mentioned before, it is known that replication (and hence recursion) can be implemented in a higher-order process algebra [23]. As our first example of calculation with the machinery thus far presented we give the construction explicitly in the ρ-calculus.

$$D(x) \triangleq x(y) \cdot (x[y] \mid \ulcorner y \urcorner)$$
$$!P(x) \triangleq x \langle\!| D(x) \mid P |\!\rangle \mid D(x)$$

$$
\begin{aligned}
!P(x) \\
= \quad & x\langle\!|(x(y) \cdot (x[y] \mid \ulcorner y \urcorner)) \mid P|\!\rangle \mid x(y) \cdot (x[y] \mid \ulcorner y \urcorner) \\
\rightarrow \quad & (x[y] \mid \ulcorner y \urcorner)\{\ulcorner(x(y) \cdot (\ulcorner y \urcorner \mid x[y])) \mid P \urcorner / y\} \\
= \; & x[\ulcorner(x(y) \cdot (x[y] \mid \ulcorner y \urcorner)) \mid P \urcorner] \mid (x(y) \cdot (x[y] \mid \ulcorner y \urcorner)) \mid P \\
\rightarrow \quad & \cdots \\
\rightarrow^* \quad & P \mid P \mid \ldots
\end{aligned}
$$

Of course, this encoding, as an implementation, runs away, unfolding $!P$ eagerly. As it is instructive to construct a lazier – and more implementable – replication operator, restricted to input-guarded processes we recommend this exercise to the reader interested in gaining further inside into the mechanics of the calculus.

4 Bisimulation

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

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

$$\frac{P \downarrow_{\mathcal{N}} x \ \textit{or} \ Q \downarrow_{\mathcal{N}} x}{P \, \textit{/} \, Q \downarrow_{\mathcal{N}} x} \qquad\qquad \textit{(}\textsc{Par-barb}\textit{)}$$

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 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:*

1. *If $P \to P'$ then $Q \Rightarrow Q'$ and $P' \, \mathcal{S} \, _{\mathcal{N}} Q'$.*
2. *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 Logic

Namespace logic resides in the subfamily of Hennessy-Milner logics discovered by Caires and Cardelli and known as spatial logics [7]. Thus, as is seen below, in addition to the action modalities, we also find formulae for *separation*, corresponding, at the logical level, to the structural content of the parallel operator at the level of the calculus. Likewise, we have quantification over names.

In this connection, however, we find an interesting difference between spatial logics investigated heretofore and this one. As in the calculus, we find no need for an operator corresponding to the ν construction. However, revelation in spatial logic, is a structural notion [7]. It detects the *declaration* of a new name. No such information is available in the reflective calculus or in namespace logic. The calculus and the logic can arrange that names are used in a manner consistent with their being declared as new in the π-calculus, but it cannot detect the declaration itself. Seen from this perspective, revelation is a somewhat remarkable observation, as it seems to be about detecting the programmer's intent.

reflective logic	$\phi, \psi ::= true$	verity
	$\mid \ 0$	nullity
	$\mid \ \neg\phi$	negation
	$\mid \ \phi \& \psi$	conjunction
	$\mid \ \phi \mid \psi$	separation
	$\mid \ \ulcorner b \urcorner$	descent
	$\mid \ a \langle\!\langle \phi \rangle\!\rangle$	elevation
	$\mid \ \langle a?b \rangle \phi$	activity
	$\mid \ \mathsf{rec} \ X \, . \, \phi$	greatest fix point
	$\mid \ \forall n : \psi \, . \, \phi$	quantification
	$a ::= \ulcorner \phi \urcorner$	indication
	$\mid \ b$	...
	$b ::= \ulcorner P \urcorner$	nomination
	$\mid \ n$	...

We let $PForm$ denote the set of formulae generated by the ϕ-production, $QForm$ denote the set of formulae generated by the a-production and $\mathcal{V}$ denote the set of propositional variables used in the rec production.

Inspired by Caires' presentation of spatial logic [5], we give the semantics in terms of sets of processes (and names). We need the notion of a valuation $v : \mathcal{V} \to \wp(Proc)$, and use the notation $v\{\mathcal{S}/X\}$ to mean

$$v\{\mathcal{S}/X\}(Y) = \begin{cases} \mathcal{S} & Y = X \\ v(Y) & otherwise \end{cases}$$

The meaning of formulae is given in terms of two mutually recursive functions,

$$[\![-]\!](-) : PForm \times [\mathcal{V} \to \wp(Proc)] \to \wp(Proc)$$
$$((-))(-) : QForm \times [\mathcal{V} \to \wp(Proc)] \to \wp(\ulcorner Proc \urcorner)$$

taking a formula of the appropriate type and a valuation, and returning a set of processes or a set of names, respectively.

$$[\![true]\!](v) = Proc$$
$$[\![0]\!](v) = \{P : P \equiv 0\}$$
$$[\![\neg\phi]\!](v) = Proc/[\![\phi]\!](v)$$
$$[\![\phi \& \psi]\!](v) = [\![\phi]\!](v) \cap [\![\psi]\!](v)$$
$$[\![\phi \mid \psi]\!](v) = \{P : \exists P_0, P_1.P \equiv P_0 \mid P_1, \ P_0 \in [\![\phi]\!](v), \ P_1 \in [\![\psi]\!](v)\}$$
$$[\![\neg b \ulcorner]\!](v) = \{P : \exists Q, P'.P \equiv Q \mid \neg x \ulcorner, \ x \in ((b))(v)\}$$
$$[\![a \langle\!| \phi |\!\rangle]\!](v) = \{P : \exists Q, P'.P \equiv Q \mid x \langle\!| P' |\!\rangle, \ x \in ((a))(v), \ P' \in [\![\phi]\!](v)\}$$
$$[\![\langle a?b\rangle\phi]\!](v) = \{P : \exists Q, P'.P \equiv Q \mid x(y) \ . \ P', x \in ((a))(v),$$
$$\forall c.\exists z.P'\{z/y\} \in [\![\phi\{c/b\}]\!](v)\}$$
$$[\![rec \ X \ . \ \phi]\!](v) = \cup\{\mathcal{S} \subseteq Proc : \mathcal{S} \subseteq [\![\phi]\!](v\{\mathcal{S}/X\})\}$$
$$[\![\forall n : \psi \ . \ \phi]\!](v) = \cap_{x \in ((\ulcorner\psi\urcorner))(v)}[\![\phi\{x/n\}]\!](v)$$
$$((\ulcorner\phi\urcorner))(v) = \{x : x \equiv_N \ulcorner P \urcorner, P \in [\![\phi]\!](v)\}$$
$$((\ulcorner P \urcorner))(v) = \{x : x \equiv_N \ulcorner P \urcorner\}$$

We say P witnesses ϕ (resp., x witnesses $\ulcorner\phi\urcorner$), written $P \models \phi$ (resp., $x \models \ulcorner\phi\urcorner$) just when $\forall v.P \in [\![\phi]\!](v)$ (resp., $\forall v.x \in [\![\ulcorner\phi\urcorner]\!](v)$).

Theorem 1 (Equivalence). $P \stackrel{.}{\approx} Q \Leftrightarrow \forall\phi.P \models \phi \Leftrightarrow Q \models \phi$.

The proof employs an adaptation of the standard strategy. As noted in the introduction, this theorem means that there is no algorithm guaranteeing that a check for the witness relation will terminate.

Syntactic sugar In the examples below, we freely employ the usual DeMorgan-based syntactic sugar. For example,

$$\phi \Rightarrow \psi \triangleq \neg(\phi \& \neg\psi)$$
$$\phi \lor \psi \triangleq \neg(\neg\phi \& \neg\psi)$$

Also, when quantification ranges over all of $Proc$, as in $\forall n : \ulcorner true \urcorner \ . \ \phi$, we omit the typing for the quantification variable, writing $\forall n \ . \ \phi$.

5.1 Examples

Controlling access to namespaces Suppose that $\ulcorner \phi \urcorner$ describes some namespace, i.e. some collection of names. We can insist that a process restrict its next input to names in that namespace by insisting that it witness the formula

$$\langle \ulcorner \phi \urcorner ?b \rangle \; true \; \& \neg \langle \ulcorner \neg \phi \urcorner ?b \rangle true$$

which simply says the the process is currently able to take input from a name in the namespace $\ulcorner \phi \urcorner$ and is not capable of input on any name not in that namespace. In a similar manner, we can limit a server to serving only inputs in $\ulcorner \phi \urcorner$ throughout the lifetime of its behavior [6]

$$\mathsf{rec} \; X \; . \; \langle \ulcorner \phi \urcorner ?b \rangle X \& \neg \langle \ulcorner \neg \phi \urcorner ?b \rangle true$$

This formula is reminiscent of the functionality of a firewall, except that it is a *static* check. A process witnessing this formula will behave as though it were behind a firewall admitting only access to the ports in $\ulcorner \phi \urcorner$ without the need for the additional overhead of the watchdog machinery.

Validating the structure of data Of course, the previous example might make one wonder what a useful namespace looks like. The relevance of this question is further amplified when we observe that processes pass names as messages as well as use them to govern synchronization. The next example, therefore, considers a space of names that might be seen as well-suited to play the role of data, for their structure loosely mimics the structure of the infoset model [9] of XML (sans schema).

$$\phi_{info} = \ulcorner \mathsf{rec} \; X \; . \; (\forall m \; . \; m \langle\!| \forall n \; . \; 0 \vee n \langle\!| X |\!\rangle \vee \mathsf{rec} \; Y \; . \; (\forall n' \; . \; \langle n'?b \rangle (X \vee Y)) \vee (X \mid X) |\!\rangle) \urcorner$$

The formula is essentially a recursive disjunction selecting names that are first of all rooted with an enclosing lift operation – reminiscent of the way an XML document has a single enclosing root; and then are either

- the empty 'document'; or
- an 'element'; or
- a sequence of documents each 'located' at an input action; or
- an unordered group.

Notice that it is possible to parameterize this namespace on names for rooting 'documents' or 'elements'. Currently, these are typed as coming from the whole namespace, $\ulcorner true \urcorner$, but they could come from any subspace.

Moreover, the formula is itself a template for the interpretation of schema specifications [24]. If we boil XSD schema down to its essential type constructors, we have a recursive specification in which a schema is a

- a sequence, or
- a choice, or
- a group, or
- a recursion, in which a type name is bound to a schema definition

---

[6] Of course, this formula also says the server never goes down, either – or at least is always willing to take such input...;-)

of element-tagged schema or schema references, with the recursive specification bottoming out at the simple and builtin types. Abstractly, then essential structures of XSD schema are captured by the grammar

$$
\begin{array}{rlll}
\text{schema} & S ::= & \epsilon & \text{empty document} \\
& | & ESequence & \text{sequence} \\
& | & EChoice & \text{choice} \\
& | & EGroup & \text{group} \\
& | & \mathsf{rec}\ N\,.\,S & \text{recursion} \\
ESequence ::= & \epsilon\ |\ E, ESequence & & \text{sequence of elements} \\
EChoice ::= & \epsilon\ |\ E + EChoice & & \text{choice of elements} \\
EGroup ::= & \epsilon\ |\ E\ |\ EGroup & & \text{group of elements} \\
E ::= & tag(N\ |\ S) & & \text{element}
\end{array}
$$

We use s to range over schema, σ, χ and γ to range over sequences, choices and groups, respectively.

The encoding below, which for clarity makes liberal – but obvious – use of polymorphism and elides the standard machinery for treating recursion variables, illustrates that we can view this grammar as essentially providing a high-level language for carving out namespaces in which the names conform to the schema.

$$
\llbracket \epsilon \rrbracket = \ulcorner 0 \urcorner
$$
$$
\llbracket tag(s), \sigma \rrbracket = \ulcorner \forall n : \llbracket tag \rrbracket\,.\,\langle n?b\rangle(\llbracket s \rrbracket\ |\ \llbracket \sigma \rrbracket) \urcorner
$$
$$
\llbracket tag(s) + \chi \rrbracket = \ulcorner \forall n : \llbracket tag \rrbracket\,.\,(\langle n?b\rangle \llbracket s \rrbracket) \vee \llbracket \chi \rrbracket \urcorner
$$
$$
\llbracket tag(s)\ |\ \gamma \rrbracket = \ulcorner \forall n : \llbracket tag \rrbracket\,.\,(\langle n?b\rangle \llbracket s \rrbracket)\ |\ \llbracket \gamma \rrbracket \urcorner
$$
$$
\llbracket \mathsf{rec}\ N\,.\,s \rrbracket = \ulcorner \mathsf{rec}\ N\,.\,\llbracket s \rrbracket \urcorner
$$

We emphasize that the example is not meant to be a complete account of XML schema. Rather, it is intended to suggest that with the reflective capabilities the logic gives a fairly intuitive treatment of names as structured data. The simplicity and intuitiveness of the treatment is really brought home, however, when employing the framework analytically. As an example, from a commonsense perspective it should be the case that any XML document that observes a schema automatically also corresponds to an infoset. The reader is encouraged to try her hand at using the framework to establish that if s is a schema, then

$$
x \models \llbracket s \rrbracket \Rightarrow x \models \phi'_{info}
$$

where ϕ'_{info} a suitably modified version of ϕ_{info}.

6   Conclusions and future work

We introduced namespace logic, a spatial-style Hennessy-Milner logic for a reflective asynchronous message-passing calculus built out of a notion of quote. We introduced some examples highlighting potential applications to security and data analysis.

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 [21], but to Power's and Hermida's work on name-free accounts of action calculi [15] as well as Pavlovic's [22]. Somewhat farther afield, but still related, is Gabbay's theory of freshness [12] and the nominal logics [13]. Very close to the mark, Carbone and Maffeis observe a tower of expressiveness resulting from adding very simple structure to names [18]. 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, or applications to security and data analysis 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 [11]. 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.

References

1. Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. In *POPL*, pages 33–44, 2002.
2. Martín Abadi and Bruno Blanchet. Secrecy types for asymmetric communication. *Theor. Comput. Sci.*, 3(298):387–415, 2003.
3. Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *ACM Conference on Computer and Communications Security*, pages 36–47, 1997.
4. Hendrik Pieter Barendregt. *The Lambda Calculus – Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.
5. Luís Caires. Behavioral and spatial observations in a logic for the pi-calculus. In *FoSSaCS*, pages 72–89, 2004.
6. Luís Caires and Luca Cardelli. A spatial logic for concurrency (part i). *Inf. Comput.*, 186(2):194–235, 2003.
7. Luís Caires and Luca Cardelli. A spatial logic for concurrency - ii. *Theor. Comput. Sci.*, 322(3):517–565, 2004.
8. John Horton Conway. *On Numbers and Games*. Academic Press, 1976.
9. John Cowan and Richard Tobin. Xml information set. W3C, 2004.
10. 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.
11. Walter Fontana. private conversation. 2004.
12. M. J. Gabbay. The π-calculus in FM. In Fairouz Kamareddine, editor, *Thirty-five years of Automath*. Kluwer, 2003.
13. Murdoch Gabbay and James Cheney. A sequent calculus for nominal logic. In *LICS*, pages 139–148, 2004.
14. Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. *Theor. Comput. Sci.*, 1-3(300):379–409, 2003.
15. Claudio Hermida and John Power. Fibrational control structures. In *CONCUR*, pages 117–129, 1995.
16. 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.
17. Microsoft Corporation. Microsoft biztalk server. microsoft.com/biztalk/default.asp.
18. M.Carbone and S.Maffeis. On the expressive power of polyadic synchronisation in pi-calculus. *Nordic Journal of Computing*, 10(2):70–98, 2003.
19. L.G. Meredith and Matthias Radestock. A reflective higher-order calculus. In Mirko Viroli, editor, *ETAPS 2005 Satellites*. Springer-Verlag, 2005.
20. Robin Milner. The polyadic π-calculus: A tutorial. *Logic and Algebra of Specification*, Springer-Verlag, 1993.
21. Robin Milner. Strong normalisation in higher-order action calculi. In *TACS*, pages 1–19, 1997.
22. Dusko Pavlovic. Categorical logic of names and abstraction in action calculus. *Math. Structures in Comp. Sci.*, 7:619–637, 1997.
23. David Sangiorgi and David Walker. *The π-Calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
24. Henry S. Thompson, David Beech, Murray Maloney, and Noah Mendelsohn. Xml schema part i: Structures, second edition. W3C, 2004.