Modelling Dynamic Web Data*

Philippa Gardner Sergio Maffeis

Imperial College London
{pg,maffeis}@doc.ic.ac.uk

Abstract

We introduce X$d\pi$, a peer-to-peer model for reasoning about the dynamic behaviour of web data. It is based on an idealised model of semi-structured data, and an extension of the π-calculus with process mobility and with an operation for interacting with data. Our model can be used to reason about behaviour found in, for example, dynamic web page programming, applet interaction, and service orchestration. We study behavioural equivalences for X$d\pi$, motivated by examples.

1 Introduction

Web data, such as XML, plays a fundamental rôle in the exchange of information between globally distributed applications. Applications naturally fall into some sort of mediator approach: systems are divided into peers, with mechanisms based on XML for interaction between peers. The development of analysis techniques, languages and tools for web data is by no means straightforward. In particular, although web services allow for interaction between processes and data, direct interaction between processes is not well-supported.

Peer-to-peer data management systems are decentralised distributed systems where each component offers the same set of basic functionalities and acts both as a producer and as a consumer of information. We model systems where each peer consists of an XML data repository and a working space where processes are allowed to run. Our processes can be regarded as agents with a simple set of functionalities; they communicate with each other, query and update the local repository, and migrate to other peers to continue execution. Process definitions can be included in documents[1], and can be selected for execution by other processes. These functionalities are enough to express most of the dynamic behaviour found in web data, such as web services, distributed (and replicated) documents [2], distributed query patterns [27], hyperlinks, forms, and scripting.

In this paper we introduce the X$d\pi$-calculus, which provides a formal semantics for the systems described above. It is based on a network of locations (peers) containing a (semi-structured) data model, and π-like processes [23, 28, 19] for modeling process interaction, process migration, and interaction with data. The data model consists of unordered labelled trees, with embedded processes for querying and updating such data, and explicit pointers for referring to other parts of the network: for example, a document with a hyperlink referring to another site and a light-weight trusted process for retrieving information associated with the link. The idea of embedding processes (scripts) in web data is not new: examples include Javascript, SmartTags and calls to web services. However, web applications do not in general provide direct communication between active processes, and process coordination therefore requires specialised orchestration tools. In contrast, distributed process interaction (communication and co-ordination) is central to our model, and is inspired by the current research on distributed process calculi.

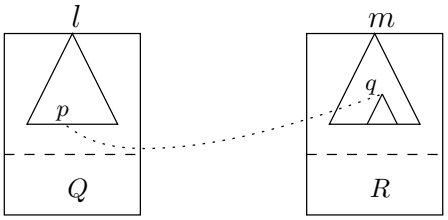
We study behavioural equivalences for X$d\pi$. In particular, we define when two processes are equivalent in such a way that when the processes are put in the same position in a network, the resulting networks are equivalent. We do this in several stages. First, we define what it means for two X$d\pi$-networks to be equivalent. Second, we indicate how to translate X$d\pi$ into a simpler calculus (Xπ_2), where the location structure has been pushed inside the data and processes. This translation technique, first proposed in [9], enables us to separate reasoning about processes from reasoning about data and networks. Finally, we define process equivalence and study examples. In particular, we sketch a labelled-bisimulation-based proof method for process equivalence. Full details on the translation and equivalences can be found in [15].

A Simple Example. We use the *hyperlink example* as a simple example to illustrate our ideas. Consider two locations (machines identified by their IP

* This paper is a revised version of [14]

[1] We regard process definitions in documents as an atomic piece of data, and we do not consider queries which modify such definitions.

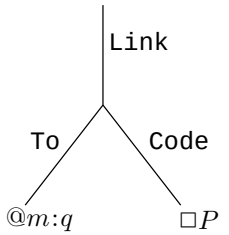
addresses) l and m. Location l contains a hyperlink at p referring to data identified by path q at location m:



In the working space of location l, the process Q activates the process embedded in the hyperlink, which then fires a request to m to copy the tree identified by path q and write the result to p at l.

The hyperlink at l, written in both XML notation (LHS) and ambient notation used in this paper (RHS), is:

```
<Link>
    <To>
     m:q
    </To>
    <Code>
     P
    </Code>
</Link>
```



$$\texttt{Link}[\,\texttt{To}[\,@m{:}q\,]\,|\,\texttt{Code}[\,\Box P\,]\,]$$

This hyperlink consists of two parts: an external pointer $@m{:}q$, and a scripted process $\Box P$ which provides the mechanism to fetch the subtree q from m. Process P has the form

$$P = \mathsf{read}_{p/\texttt{Link/To}}(@x{:}y).\overline{load}\langle x, y, p\rangle$$

The read command reads the pointer reference from position $p/\texttt{Link/To}$ in the tree, substitutes the values m and q for the parameters x and y in the continuation and evolves to the output process $\overline{load}\langle m, q, p\rangle$, which records the target location m, the target path q and the position p where the result tree will go. This output process calls a corresponding input process inside Q, using π-calculus interaction. The specification of Q has the form:

$$Q_s \;= !load(x, y, z).\mathsf{go}\, x.\, \mathsf{copy}_y(X).\mathsf{go}\, l.\mathsf{paste}_z\langle X\rangle$$

The replication ! denotes that the input process can be used as many times as requested. The interaction between the $\overline{load}$ and $load$ replaces the parameters x, y, z with the values m, q, p in the continuation. The process then goes to m, copies the tree at q, comes back to l, and pastes the tree to p.

The process Q_s is just a specification. We refine this specification by having a process Q (acting as a service call), which sends a request to location m for the tree at q, and a process R (the service definition) which grants the request. Processes Q and R are defined by

$$Q = !load(x, y, z).(\nu c)(\mathsf{go}\, x.\overline{get}\langle y, l, c\rangle \,|\, c(X).\mathsf{paste}_z\langle X\rangle)$$

$$R = !get(y, x, w).\mathsf{copy}_y(X).\mathsf{go}\, x.\overline{w}\langle X\rangle$$

Once process Q receives parameters from $\overline{load}$, it splits into two parts: the process that sends the output message $\overline{get}\langle q, l, c\rangle$ to m, with information about the particular target path q, the return location l and a private channel name c (created using the π-calculus restriction operator ν), and the process $c(X).\mathsf{paste}_p\langle X\rangle$ waiting to paste the result delivered via the unique channel c. Process R receives the parameters from $\overline{get}$ and returns the tree to the unique channel c at l. Using our definition of process equivalence, we show that Q does indeed have the same intended behaviour as its specification Q_s, and that the processes are interchangeable independently from the context where they are installed.

Related Work. Our work is related to the Active XML approach to data integration developed independently by Abiteboul et al. [4]. They introduce an architecture which is a peer-to-peer system where each peer contains a data model very similar to ours (but where only service calls can be scripted in documents), and a working space where only web service definitions are allowed. Moreover Active XML service definitions cannot be moved around. In this respect our approach is more flexible: for example, we can define an auditing process for assessing a university course—it goes to a government site, selects the assessment criteria appropriate for the particular course under consideration, then moves this information (web service definition) to the university to make the assessment.

Several distributed query languages, such as [25, 21, 8], extend traditional query languages with facilities for distribution awareness. Our approach is closest to the one of Sahuguet and Tannen [27], who introduce the ubQL query language based on streams for exchanging large amounts of distributed data, partly motivated by ideas from the π-calculus. There has been much study of data models for the web in the XML, database and process algebra communities. Our ideas have evolved from those found in [3] and [10]. Our process-algebra techniques have most in common with [20, 9, 18]. Process calculi have also been used for example to study security properties of web services [17], reason about mobile resources [16], and in [26] to sketch a distributed query language. Bierman and Sewell [7] have recently extended a small functional language for XML with π-calculus-based concurrency in order to program Home Area Networks devices.

Our proof technique is based on higher-order bisimulation for process languages, a technique studied for example in [30, 12, 29]

Our work is the first attempt to integrate the study of mobile processes and semi-structured data for Web-based data-sharing applications, and is characterised by its emphasis on dynamic data.

2 A Model for Dynamic Web Data

We model a peer-to-peer system as a sets of interconnected locations (*networks*), where the content of each location consists of an abstraction of a XML data repository (the *tree*) and a term representing both the services provided by the peer and the agents in execution on behalf of other peers (the *process*). Processes can query and update the local data, communicate with each other through named *channels* (public or private), and migrate to other peers. Process definitions can be included in documents and can be selected for execution by other processes.

Trees. Our data model extends the unordered labelled rooted trees of [10], with leaves which can either be scripted processes or pointers to data. We use the following constructs: *edge labels* denoted by $\mathsf{a}, \mathsf{b}, \mathsf{c} \in \mathcal{A}$, *path expressions* denoted by $p, q \in \mathcal{E}$ used to identify specific subtrees, and *locations* of the form $\circlearrowleft, l, m \in \mathcal{L}$, where the 'self' location $\circlearrowleft$ refers to the enclosing location. The set of data trees, denoted $\mathcal{T}$, is given by

$$T ::= 0 \mid T \,|\, T \mid \mathsf{a}[\,T\,] \mid \mathsf{a}[\,\Box P\,] \mid \mathsf{a}[\,@l{:}p\,]$$

Tree 0 denotes a rooted tree with no content. Tree $T_1 \,|\, T_2$ denotes the composition of T_1 and T_2, which simply joins the roots. A tree of the form $\mathsf{a}[\,\ldots\,]$ denotes a tree with a single branch labelled a which can have three types of content: a subtree T; a *scripted process* $\Box P$ which is a static process awaiting a command to run; a *pointer* $@l{:}p$ which denotes a pointer to a set of subtrees identified by path expression p in the tree at location l. Processes and path expressions are described below. The structural congruence for trees states that trees are unordered, and scripted processes are identified up to the structural congruence for processes (see Table 7 in the Appendix).

Processes. Our processes are based on π-processes extended with an explicit migration primitive between locations, and an operation for interacting directly with data. The π-processes describe the movement of values between channels. Generic variables are x, y, z, channel names or channel variables are a, b, c and values, ranged over by u, v, w, are

$$u ::= T \mid c \mid l \mid p \mid \Box P \mid x$$

We use the notation $\tilde{z}$ and $\tilde{v}$ to denote vectors of variables and values respectively. Identifiers U, V range over scripted processes, pointers and trees. The set of processes, denoted $\mathcal{P}$, is given by

$$\begin{aligned} P ::= \quad & 0 \mid P \,|\, P \mid (\nu c)P \mid \overline{b}\langle\tilde{v}\rangle \mid b(\tilde{z}).P \mid !b(\tilde{z}).P \\ & \mid \mathsf{go}\, l.P \mid \mathsf{update}_p(\chi, V).P \end{aligned}$$

The processes in the first line of the grammar are constructs arising from the π-calculus: the *nil* process 0, the *composition* of processes $P_1 \,|\, P_2$, the *restriction* $(\nu c)P$ which restricts (binds) channel name c in process P, the *output* process $\overline{b}\langle\tilde{v}\rangle$ which denotes a vector of values $\tilde{v}$ waiting to be sent via channel b, the *input* process $b(\tilde{z}).P$ which is waiting to receive values from an output process via channel b to replace the vector of distinct, bound variables $\tilde{z}$ in P, and *replicated input* $!b(\tilde{z}).P$ which spawns off a copy of $b(\tilde{z}).P$ each time one is requested. We assume a simple sorting discipline, to ensure that the number of values sent along a channel matches the number of variables expected to receive those values. Channel names are partitioned into *public* and *session* channels, denoted $\mathcal{C}_P$ and $\mathcal{C}_S$ respectively. Public channels denote those channels that are intended to have the same meaning at each location, such as *finger*, and cannot be restricted. Session channels are used for process interaction, and are not free in the scripted processes occurring in data.

The *migration* primitive $\mathsf{go}\, l.P$ is common in calculi for describing distributed systems; see for example [18]. It enables a process to go to l and become P. An alternative choice would have been to incorporate the location information inside the other process commands: for example using $\overline{l{\cdot}b}\langle\tilde{v}\rangle$ to denote a process which goes to l and interacts via channel b.

The generic *update* command $\mathsf{update}_p(\chi, U).P$ is used to interact with the data trees. The *pattern* χ has the form

$$\chi ::= X \mid @x{:}y \mid \Box X,$$

where X denotes a tree or process variable. Here U can contain variables and must have the same sort as χ. The variables free in χ are bound in U and P. The update command finds all the values U_i given by the path p, pattern-matches these values with χ to obtain the substitution σ_i when it exists. For each successful pattern-matching, it replaces the U_i with $U\sigma_i$ and starts $P\sigma_i$ in parallel. Simple commands can be derived from this update command, including standard copy_p, cut_p and paste_p commands. We can also derive a run_p command, which prescribes, for all scripted processes $\Box P_i$ found at the end of path p, to run P_i in the workspace.

The structural congruence on processes is similar to that given for the π-calculus, and is given in the Appendix in Table 7. Notice that it depends on the structural congruence for trees, since trees can be passed as values.

Networks. We model networks as a composition of *unique* locations, where each location contains a tree and a set of processes. The set of networks, denoted $\mathcal{N}$, is given by

$$N ::= 0 \mid N \,|\, N \mid l\left[\,T \,\|\, P\,\right] \mid (\nu c)N \mid l\langle P\rangle$$

The *location* $l\left[\,T \,\|\, P\,\right]$ denotes location l containing tree T and process P. It is well-formed when the

tree and process are closed, and the tree contains no free session channels. The network composition $N_1 \mid N_2$ is *partial* in that the location names associated with N_1 and N_2 must be disjoint. Communication between locations is modelled by process migration, which we represent explicitly: the process $l\langle P\rangle$ represents a (higher-order) migration message addressed to l and containing process P, which has left its originating location. In the introduction, we saw that a session channel can be shared between processes at different locations. We must therefore lift the restriction to the network level using $(\nu c)N$. Structural congruence for networks is defined in Table 7 in the Appendix, and is analogous to that given for processes.

Path Expressions. In the examples of this paper, we just use a very simple subset of XPath expressions [22]. In our examples, "/" denotes path composition and ".", which can appear only inside trees, denotes the path to its enclosing node.

Our semantic model is robust with respect to any choice of mechanism which, given some expression p, identifies a set of nodes in a tree T. We let $p(T)$ denote the tree T where the nodes identified by p are selected, and we represent a selected node by underlining its contents. For example the selected subtrees below are S' and T':

$$T = \mathsf{a}[\,\mathsf{a}[\,S\,] \mid \mathsf{b}[\,\underline{S'}\,] \mid \mathsf{c}[\,\underline{T'}\,]\,]$$

A path expression such as //a might select nested subtrees. We give an example:

$$//\mathsf{a}(T) = \underline{\mathsf{a}[\,\underline{\mathsf{a}[\,S\,]} \mid \mathsf{b}[\,S'\,] \mid \mathsf{c}[\,T'\,]\,]}$$

Reduction and Update Semantics. The reduction relation $\searrow$ describes the movement of processes across locations, the interaction between processes and processes, and the interaction between processes and data. Reduction is closed under network composition, restriction and structural congruence, and it relies on the updating function $\rightsquigarrow$ reported in Table 2. The reduction axioms are given in Table 1.

The rules for process movement between locations are inspired by [5]. Rule (Exit) always allows a process $\mathsf{go}\,l.P$ to leave its enclosing location. At the moment, rule (Enter) permits the code of P to be installed at l provided that location l exists[2]. In future work, we intend to associate some security check to this operation. Process interaction (rules (Com) and (!Com)) is inspired by π-calculus interaction. If one process wishes to output on a channel, and another wishes to input on the same channel, then they can react together and transmit some values as part of that reaction.

[2]This feature is peculiar to our calculus, as opposed to e.g. [18], where the existence of the location to be entered is not a precondition to migration. Our choice makes the study of equivalences non-trivial.

The generic (Update) rule provides interaction between processes and data. Using path p it selects for update some subtrees in T, denoted by $p(T)$, and then applies the updating function $\rightsquigarrow$ to $p(T)$ in order to obtain the new tree T' and the continuation process P'. Given a subtree selected by p, the function $\rightsquigarrow$ pattern matches the subtree with pattern χ to obtain substitution σ (when it exists), updates the subtree with $V\sigma$, and creates process $P\sigma$. A formal definition of $\rightsquigarrow$, parameterised by p, l, χ, V, P, is given in Table 2. Rule (Up) deserves some explanation. It matches U with χ, to obtain substitution σ; when σ exists, it continues updating $V\sigma$, and when we obtain some subtree V' along with a process R, it replaces U with V' and it returns R in parallel with $P\sigma\{l/\circlearrowleft, p/.\}$, where any reference to the current location and path is substituted with the actual values l and p[3].

Derived Commands. Throughout our examples, we use the derived commands given in Table 3. In particular note that the definition of run is the only case where we allow the instantiation of a process variable.

Example 2.1 *The following reaction illustrates the* cut *command:*

$$l\left[\,\mathsf{c}[\,\mathsf{a}[\,T\,] \mid \mathsf{a}[\,T'\,] \mid \mathsf{b}[\,S\,]\,] \;\|\; \mathsf{cut}_{\mathsf{c}/\mathsf{a}}(X).P\,\right]$$

$$\searrow\, l\left[\,\mathsf{c}[\,\mathsf{a}[\,0\,] \mid \mathsf{a}[\,0\,] \mid \mathsf{b}[\,S\,]\,] \;\|\; P\{T/X\} \mid P\{T'/X\}\,\right]$$

The cut *operation cuts the two subtrees T and T' identified by the path expression* c/a *and spawns one copy of P for each subtree.*

Now we give an example to illustrate run *and the substitution of local references:*

$$S = \mathsf{a}[\,\mathsf{b}[\,\Box\mathsf{go}\,m.\mathsf{go}\;\circlearrowleft.Q\,] \mid \mathsf{b}[\,\Box\mathsf{cut}_{./../\mathsf{c}}(X).P\,]\,]$$

$$l\left[\,S \;\|\; \mathsf{run}_{\mathsf{a}/\mathsf{b}}\,\right] \searrow l\left[\,S \;\|\; \mathsf{go}\,m.\mathsf{go}\,l.Q \mid \mathsf{cut}_{\mathsf{a}/\mathsf{b}/../\mathsf{c}}(X).P\,\right]$$

The data S is not affected by the run *operation, which has the effect of spawning the two processes found by path* a/b*. Note how the local path ./../*c *has been resolved into the completed path* a/b/../c*, and $\circlearrowleft$ has been substituted by l.*

3 Dynamic Web Data at Work

We give some examples of dynamic web data modelled in X$d\pi$.

[3]The ability to select nested nodes introduces a difference between updating the tree in a top-down rather than bottom-up order. In particular the resulting tree is the same, but a different set of processes P is collected. We chose the top-down approach because is bears a closer correspondence with intuition: a copy of P will be created for each update still visible in the final tree outcome. For example, if $Q = \mathsf{update}_{//}(X, 0).P$

$$\text{(top-down)}\; l\left[\,\mathsf{a}[\,\mathsf{b}[\,T\,]\,] \;\|\; Q\,\right] \searrow l\left[\,\mathsf{a}[\,0\,] \;\|\; P\{\mathsf{b}[\,T\,]/X\}\,\right]$$

$$\text{(bottom-up)}\; l\left[\,\mathsf{a}[\,\mathsf{b}[\,T\,]\,] \;\|\; Q\,\right] \searrow l\left[\,\mathsf{a}[\,0\,] \;\|\; P\{\mathsf{b}[\,0\,]/X\} \mid P\{T/X\}\,\right]$$

because first $\mathsf{a}[\,\mathsf{b}[\,T\,]\,]$ becomes $\mathsf{a}[\,\mathsf{b}[\,0\,]\,]$ giving $P\{T/X\}$, and then $\mathsf{a}[\,\mathsf{b}[\,0\,]\,]$ becomes $\mathsf{a}[\,0\,]$, giving $P\{\mathsf{b}[\,0\,]/X\}$.

(Exit)	$m\left[T \parallel Q \mid \mathsf{go}\, l.P\right] \searrow m\left[T \parallel Q\right] \mid l\langle P\rangle$
(Enter)	$l\left[T \parallel Q\right] \mid l\langle P\rangle \searrow l\left[T \parallel Q \mid P\right]$
(Com)	$l\left[T \parallel \overline{c}\langle\tilde{v}\rangle \mid c(\tilde{z}).P \mid Q\right] \searrow l\left[T \parallel P\{\tilde{v}/\tilde{z}\} \mid Q\right]$
(Com!)	$l\left[T \parallel \overline{c}\langle\tilde{v}\rangle \mid !c(\tilde{z}).P \mid Q\right] \searrow l\left[T \parallel !c(\tilde{z}).P \mid P\{\tilde{v}/\tilde{x}\} \mid Q\right]$
(Update)	$l\left[T \parallel \mathsf{update}_p(\chi, V).P \mid Q\right] \searrow l\left[T' \parallel P' \mid Q\right]$ where $p(T) \rightsquigarrow_{(p,l,\chi,V,P)} T', P'$

Table 1: Reduction Semantics

(Zero) $0 \rightsquigarrow_\Theta 0, 0$ (Link) $@l{:}p \rightsquigarrow_\Theta @l{:}p, 0$ (Proc) $\Box Q \rightsquigarrow_\Theta \Box Q, 0$

(Par) $$\frac{T \rightsquigarrow_\Theta T', R \qquad S \rightsquigarrow_\Theta S', R'}{T \mid S \rightsquigarrow_\Theta T' \mid S', R \mid R'}$$ (Node) $$\frac{U \rightsquigarrow_\Theta U', R}{\mathtt{a}[\,U\,] \rightsquigarrow_\Theta \mathtt{a}[\,U'\,], R}$$

(Up) $$\frac{\mathsf{match}(U, \chi) = \sigma \qquad V\sigma \rightsquigarrow_\Theta V', R \qquad \Theta = (p, l, \chi, V, P)}{\mathtt{a}[\underline{\,U\,}] \rightsquigarrow_\Theta \mathtt{a}[\,V'\,], P\sigma\{l/\circlearrowleft, p/.\} \mid R}$$

Table 2: Update Semantics

Web Services. In the introduction, we described the hyperlink example. Here we generalise this example to arbitrary web services. We define a web service c with parameters $\tilde{z}$, body P, and type of result specified by the distinct variables $\tilde{w}$ bound by P:

$$\mathsf{Def}\ c(\tilde{z})\ \mathsf{as}\ P\ \mathsf{out}\ \langle\tilde{w}\rangle \triangleq !c(\tilde{z}, l, x).\, P.\, \mathsf{go}\, l.\, \overline{x}\langle\tilde{w}\rangle$$

where l and x are fixed parameters (not in $P, \tilde{w}$) which are used to specify the return location and channel. For example, process R described in the introduction can be written $\mathsf{Def}\ get(q)\ \mathsf{as}\ \mathsf{copy}_q(X)\ \mathsf{out}\ \langle X\rangle$.

We specify a service call at l to the service c at m, sending actual parameters $\tilde{v}$ and expecting in return the result specified by distinct bound variables $\tilde{w}$:

$$l{\cdot}\mathsf{Call}\ m{\cdot}c\langle\tilde{v}\rangle\ \mathsf{ret}\ (\tilde{w}).Q \triangleq (\nu b)(\mathsf{go}\, m.\overline{c}\langle\tilde{v}, l, b\rangle \mid b(\tilde{w}).Q)$$

This process establishes a private session channel b, which it passes to the web service as the unique return channel. Returning to the hyperlink example, the process Q running at l can be given by

$$!load(m, q, p).\, l{\cdot}\mathsf{Call}\ m{\cdot}get\langle q\rangle\ \mathsf{ret}\ (X).\mathsf{paste}_p\langle X\rangle$$

Notice that it is easy to model subscription to continuous services in our model, by simply replicating the input on the session channel:

$$l{\cdot}\mathsf{Sub}\ m{\cdot}c\langle\tilde{v}\rangle\ \mathsf{ret}\ (\tilde{w}).P \triangleq (\nu b)(\mathsf{go}\, m.\overline{c}\langle\tilde{v}, l, b\rangle \mid !b(\tilde{w}).P)$$

Note that some web services may take as a parameter or return as a result some data containing another service call (for example, see the *intensional parameters* of [1]). In our system the choice of when to invoke such nested services is completely open, and is left to the service designer.

XLink Base. We look at a refined example of the use of linking, along the lines of XLink. Links specify both of their endpoints, and therefore can be stored in some external repository, for example

$$\mathtt{XLink}[\,\mathtt{To}[\,@n{:}q\,] \mid \mathtt{From}[\,@l{:}p\,] \mid \mathtt{Code}[\,\Box P\,]\,]$$

$$\mathtt{XLinkBase}[\,\mathtt{XLink}[\,\ldots\,] \mid \ldots \mid \mathtt{XLink}[\,\ldots\,]\,]$$

Suppose that we want to download from an XLink server the links associated with node p in the local repository at l.

We can define a function $xload$ which takes a parameter p and requests from the XLink service xls at m all the XLinks originating from $@l{:}p$, in order to paste them under p at location l:

$$!xload(p).l{\cdot}\mathsf{Sub}\ m{\cdot}xls\langle l, p\rangle\ \mathsf{ret}\ (x, y, \Box\chi) \\ .\mathsf{paste}_p\langle\mathtt{Link}[\,\mathtt{To}[\,@x{:}y\,] \mid \mathtt{Code}[\,\Box\chi\,]\,]\rangle$$

Service xls defined below is the XLink server. It takes as parameters the two components l, p making up the `From` endpoint of a link, and returns all the pairs `To`, `Code` defined in the database for $@l{:}p$.

$$\mathsf{Def}\ xls(l, p)\ \mathsf{as}\ P\ \mathsf{out}\ \langle x, y, \Box\chi\rangle$$

$$P = \mathsf{copy}_{p_1}(@x{:}y).\mathsf{copy}_{p_2}(\Box\chi)$$

$$p_1 = \mathtt{XLinkBase/XLink}[\,\mathtt{From}[\,@l{:}p\,]\,]/\mathtt{To}$$

$\mathsf{copy}_p(X).P$	$\triangleq$	$\mathsf{update}_p(X, X).P$	copy the tree at p and use it in P
$\mathsf{read}_p(@x{:}y).P$	$\triangleq$	$\mathsf{update}_p(@x{:}y, @x{:}y).P$	read the pointer at p, use its location and path in P
$\mathsf{cut}_p(X).P$	$\triangleq$	$\mathsf{update}_p(X, 0).P$	cut the tree at p and use it in p
$\mathsf{paste}_p\langle T\rangle.P$	$\triangleq$	$\mathsf{update}_p(X, X \mid T).P$	where X is not free in T or P, paste tree T at p and evolve to P
run_p	$\triangleq$	$\mathsf{update}_p(\Box X, \Box X).X$	run the scripted process at p

Table 3: Derived Commands

$p_2 = \texttt{XLinkBase/XLink[From[}\,@l{:}p\,\texttt{]}\mid\texttt{To[}\,@x{:}y\,\texttt{]]/Code}$

In p_1 we use the XPath syntax $\texttt{XLink[From[}\,@l{:}p\,\texttt{]]/To}$ to identify the node `To` which is a son of node `XLink` and a sibling of $\texttt{From[}\,@l{:}p\,\texttt{]}$; similarly for p_2.

Forms. Forms enhance documents with the ability to input data from a user and then send it to a server for processing. For example, assuming that the server is at location s, that the form is at path p, and that the code to process the form result is called *handler*, we have

$$\begin{array}{ll}\texttt{form[} & \texttt{input[}\,0\,\texttt{]}\\ & \mid \texttt{submit[}\,\Box\mathsf{copy}_{\texttt{./../input}}(X).\mathsf{go}\ s.\overline{handler}\langle X\rangle\,\texttt{]}\\ & \mid \texttt{reset[}\,\Box\mathsf{cut}_{\texttt{./../input}}(X)\,\texttt{]]}\end{array}$$

where $\mathsf{run}_{p/\texttt{form/submit}}$ (or $\mathsf{run}_{p/\texttt{form/reset}}$) is the event generated by clicking on the submit (or reset) button. Some user input T can be provided by a process

$$\mathsf{paste}_{p/\texttt{form/input}}\langle T\rangle$$

and on the server there will be a handler ready to deal with the received data

$$s\left[\, S \parallel !handler(X).P|...\,\right]$$

This example is suggestive of the usefulness of embedding processes rather than just service calls in a document: the code to handle submission may vary from form to form, and for example some input validation could be performed on the client side.

4 Behaviour of Dynamic Web Data

In the hyperlink example of the introduction, we have stated that process Q and its specification Q_s have the same intended behaviour. In this section we provide the formal analysis to justify this claim. We do this in several stages. First, we define what it means for two X$d\pi$ networks to be equivalent. Then, we indicate how to translate X$d\pi$ into another (equivalent) calculus, called Xπ_2, where it is easier to separate reasoning about processes from reasoning about data. Finally, we define process equivalence on Xπ_2 terms.

Network Equivalence. We apply a standard technique for reasoning about processes distributed between locations to our non-standard setting. The network contexts are

$$C ::= - \mid C \mid N \mid (\nu c)\, C$$

We define a *barbed congruence* between networks which is reduction-closed, closed with respect to network contexts, and which satisfies an additional *observation relation* described using *barbs*. In our case, the barbs describe the update commands, the commands which directly affect the data.

Definition 4.1 *A* barb *has the form $l{\cdot}p$, where l is a location name and p is a path. The* observation relation*, denoted by $N \downarrow_{l\cdot p}$, is a binary relation between Xdπ-networks and barbs defined by $N \downarrow_{l\cdot p}$ iff*

$$\exists_{\,C[-],S,\chi,U,P,Q}.\ N \equiv C[l\left[\, S \parallel \mathsf{update}_p(\chi, U).P \mid Q)\,\right]]$$

that is, N contains a location l with an update_p command. The weak observation relation*, denoted $N \Downarrow_{l\cdot\beta}$, is defined by*

$$N \Downarrow_{l\cdot p} \quad \textit{iff} \quad \exists N'.\, N \searrow N' \wedge N' \downarrow_{l\cdot p}$$

Observing a barb corresponds to observe at what points in some data-tree a process has the capability to read or write data.

Definition 4.2 *Barbed congruence ($\simeq_b$) is the largest symmetric relation $\mathcal{R}$ on Xdπ-networks such that $N\,\mathcal{R}\,M$ implies*

- *N and M have the same barbs: $N \downarrow_{l\cdot p} \Rightarrow M \Downarrow_{l\cdot p}$;*
- *R is reduction-closed: $N \searrow N' \Rightarrow (\exists M'.M \searrow^* M' \ \wedge\ N'\,\mathcal{R}\ M')$;*
- *R is closed under network contexts: $\forall C.C[N]\,\mathcal{R}\,C[M]$.*

Example 4.1 *Our first example illustrates that network equivalence does not imply that the initial data trees need to be equivalent:*

$$N \triangleq l\left[\, \mathsf{b}[\,0\,] \parallel !\mathsf{paste}_{\mathsf{b}}\langle \mathsf{a}[\,0\,]\rangle \mid !\mathsf{cut}_{\mathsf{b}}(X)\,\right]$$

$$M \triangleq l\left[\,\mathtt{b}[\,\mathtt{a}[\,0\,]\mid \mathtt{a}[\,0\,]\,]\;\middle\|\; !\mathsf{paste}_{\mathtt{b}}\langle \mathtt{a}[\,0\,]\rangle \mid !\mathsf{cut}_{\mathtt{b}}(X)\,\right]$$

We have $N \simeq_b M$ since each state reachable by one network is also reachable by the other, and vice versa.

An interesting example of non-equivalence is

$$l\left[\,T \;\middle\|\; \mathit{update}_p(X,X).0\,\right] \not\simeq_b l\left[\,T \;\middle\|\; 0\,\right]$$

Despite this particular update (copy) command having no effect on the data and continuation, we currently regard it as observable since it has the capability to modify the data at p, even if it does not use it.

Example 4.2 *Our definition of web service is equivalent to its specification. Consider the simple networks*

$$N = l\left[\,T \;\middle\|\; l{\cdot}\mathsf{Call}\ m{\cdot}c\langle\tilde{v}\rangle\ \mathsf{ret}\ (\tilde{w}).Q \mid R\,\right]$$

$$N_s = l\left[\,T \;\middle\|\; \mathsf{go}\ m.P\{\tilde{v}/\tilde{z}\}.\mathsf{go}\ l.Q \mid R\,\right]$$

$$M = m\left[\,S \;\middle\|\; \mathsf{Def}\ c(\tilde{z})\ \mathsf{as}\ P\ \mathsf{out}\ \langle\tilde{w}\rangle \mid R'\,\right]$$

If c does not appear free in R and R', then

$$(\nu c)(N \mid M) \simeq_b (\nu c)(N_s \mid M)$$

A special case of this example is the hyperlink example discussed in the introduction. The restriction c is used to prevent the context providing any competing service on c. It is clearly not always appropriate however to make a service name private. An alternative approach is to introduce a linear type system, studied for example in [6], to ensure service uniqueness.

Separation of Data and Processes. Our aim is to define when two processes are equivalent in such a way that, when the processes are put in the same position in a network, the resulting networks are equivalent. In the technical report [15], we introduce the Xπ_2-calculus, in which the location structure is pushed locally to the data and processes, in order to be able to talk directly about processes. We translate the X$d\pi$-calculus in the Xπ_2-calculus, and equate X$d\pi$-equivalence with Xπ_2-equivalence.

Here we just summarise the translation and its results using the hyperlink example:

$$N = l\left[\,\mathtt{Link}[\,\mathtt{To}[\,@m{:}q\,]\mid \mathtt{Code}[\,\Box P\,]\,]\;\middle\|\; Q\,\right] \mid m\left[\,S \;\middle\|\; R\,\right]$$

$$Q = !\mathit{load}(m,q,p).(\nu c)(\mathsf{go}\, m.\,\overline{\mathit{get}}\langle q,l,c\rangle \mid c(X).\mathsf{paste}_p\langle X\rangle)$$

$$R = !\mathit{get}(q,l,c).\mathsf{copy}_q(X).\mathsf{go}\, l.\bar{c}\langle X\rangle$$

The translation to Xπ_2 involves pushing the location structure, in this case the l and m, inside the data and processes. We use $(\!|N|\!)$ to denote the translated data and $(\!|S|\!)$ to denote the translation of tree S; also $[\![N]\!]$ for the translated processes and $[\![P]\!]_l$ for the translation of process P which depends on location l. Our hyperlink example becomes

$$(\!|N|\!) = \{l \mapsto \mathtt{Link}[\,\mathtt{To}[\,@m{:}q\,]\mid \mathtt{Code}[\,\Box[\![P]\!]_{\circlearrowleft}\,]\,], m \mapsto (\!|S|\!)\}$$

$$[\![N]\!] = !l{\cdot}\mathit{load}(m,q,p).(\nu c)(l{\cdot}\mathsf{go}\, m.\,\overline{m{\cdot}\mathit{get}}\langle q,l,c\rangle \mid l{\cdot}c(X).\mathsf{paste}_p\langle X\rangle)$$

$$\mid !m{\cdot}\mathit{get}(q,l,c).m{\cdot}\mathsf{copy}_q(X).m{\cdot}\mathsf{go}\, l.\overline{l{\cdot}c}\langle X\rangle$$

The translation of N is denoted by $((\!|N|\!), [\![N]\!])$. There are several points to notice. The data translation $(\!|\_|\!)$ assigns locations to translated trees, which remain the same except that the scripted processes are translated using the self location $\circlearrowleft$: in our example $\Box P$ is translated to $\Box[\![P]\!]_{\circlearrowleft}$. The use of $\circlearrowleft$ is necessary since it is not pre-determined where the scripted process will run. In our hyperlink example, it runs at l. With an HTML form, for example, it is not known where a form with an embedded scripted process will be required. The process translation $[\![\_]\!]$ embeds locations in processes. In our example, it embeds location l in Q and location m in R. After a migration command, for example the $\mathsf{go}\, m.\_$ in Q, the location information changes to m, following the intuition that the continuation will be active at location m. A subtle point of this translation is that in order to preserve the domain of a network during the translation, if a location l contains only the empty process, then a located nil process is produced by the encoding: $[\![0]\!]_l = l{\cdot}0$.

The crucial properties of the encoding are that it preserves the observation relation and is fully abstract with respect to barbed congruence, where the barbed congruence for Xπ_2 is analogous to that for X$d\pi$.

Lemma 4.1 *$N \downarrow_{l\cdot\beta}$ if and only if $((\!|N|\!), [\![N]\!]) \downarrow_{l\cdot\beta}$.*

Theorem 4.1 *$N \simeq_b M$ if and only if $((\!|N|\!), [\![N]\!]) \simeq_b ((\!|M|\!), [\![M]\!])$.*

Process Equivalence. We now have the machinery to define process equivalence. We use the notation (D, P) to denote a network in Xπ_2, where D stands for located trees and P for located processes. A network (D, P) is well formed if and only if $(D, P) = ((\!|N|\!), [\![N]\!])$ for some X$d\pi$ network N.

Definition 4.3 *Processes P and Q are* barbed equivalent*, denoted $P \sim_b Q$, if and only if, for all D such that (D, P) is well formed, $(D, P) \simeq_b (D, Q)$.*

Example 4.3 *Recall the web service example in Example 4.2. The processes below are barbed equivalent:*

$$Q_1 = [\![l{\cdot}\mathsf{Call}\ m{\cdot}c\langle\tilde{v}\rangle\ \mathsf{ret}\ (\tilde{w}).Q]\!]_l$$

$$Q_2 = [\![\mathsf{go}\ m.P\{\tilde{v}/\tilde{z}\}.\mathsf{go}\ l.Q]\!]_l$$

$$P_0 = [\![\mathsf{Def}\ c(\tilde{z})\ \mathsf{as}\ P\ \mathsf{out}\ \langle\tilde{w}\rangle]\!]_m$$

$$(\nu c)(Q_1 \mid P_0) \sim_b (\nu c)(Q_2 \mid P_0)$$

Example 4.4 *We give now an example of how it is possible to replicate a web service in such a way that the behaviour of the system is the same as for the non-replicated case. Let internal nondeterminism be represented as* $P \oplus Q \triangleq (\nu a)(\overline{a} \,|\, a.P \,|\, a.Q)$*, where* a *does not occur free in* P, Q*. We define two service calls to two interchangeable services, service* R_1 *on channel* c *and* R_2 *on channel* d*:*

$$Q_1 = [\![l{\cdot}\mathsf{Call}\ m{\cdot}c\langle\tilde{v}\rangle\ \mathsf{ret}\ (\tilde{w}).Q]\!]_l$$

$$Q_2 = [\![l{\cdot}\mathsf{Call}\ n{\cdot}d\langle\tilde{v}\rangle\ \mathsf{ret}\ (\tilde{w}).Q']\!]_l$$

$$P_m = [\![\mathsf{Def}\ c(\tilde{z})\ \mathsf{as}\ P_1\ \mathsf{out}\ \langle\tilde{w}\rangle]\!]_m$$

$$P_1 = [\![\mathsf{go}\ n.\overline{d}\langle\tilde{z}, l, x\rangle \oplus R_1]\!]_m$$

$$P_n = [\![\mathsf{Def}\ d(\tilde{z})\ \mathsf{as}\ P_2\ \mathsf{out}\ \langle\tilde{w}\rangle]\!] n$$

$$P_2 = [\![\mathsf{go}\ m.\overline{c}\langle\tilde{z}, l, x\rangle \oplus R_2]\!]_n$$

We can show that, regardless of which service is invoked, a system built out of these processes behaves in the same way:

$$Q_1 \,|\, P_m \,|\, P_n \sim_b Q_2 \,|\, P_m \,|\, P_n$$

We can also show a result analogous to the single web-service given in Example 4.3. Given the specification process

$$Q_s = [\![\mathsf{go}\ m.R_1\{\tilde{v}/\tilde{z}\}.\mathsf{go}\ l.Q \oplus \mathsf{go}\ n.R_2\{\tilde{v}/\tilde{z}\}.\mathsf{go}\ l.Q']\!]_l$$

we have the equivalence below

$$(\nu c, d)(Q_1 \,|\, P_m \,|\, P_n) \sim_b (\nu c, d)(Q_s \,|\, P_m \,|\, P_n)$$

where the restriction of c *and* d *avoids competing services on the same channel. Now let* $Q'_s = [\![\mathsf{go}\ m.R_1\{\tilde{v}/\tilde{z}\}.\mathsf{go}\ l.Q]\!]_l$ *and let* $Q = Q'$*,* $R_1 = R_2$*, where* R_1 *does not have any barb at* m*. We have*

$$(\nu c, d)(Q_1 \,|\, P_m \,|\, P_n) \ \sim_b \ (\nu c, d)(Q'_s \,|\, P_m \,|\, P_n).$$

This equivalence illustrates that we can replicate a web service without a client's knowledge.

5 A Bisimulation-Based Proof Method

Equivalences in the style of those seen in the previous section, are known to be difficult to use. In particular the condition of closure under contexts involves a universal quantification on processes which complicates the proofs. We give here the idea of a proof method based on a labelled bisimulation where congruence is a derived property. The details can be found in [15]. In particular, we develop a bisimulation equivalence $\approx$ with the property that, given two Xπ_2 processes P, Q, then $P \approx Q$ implies $P \sim_b Q$.

Our bisimulation is based on a labelled transition system for Xπ_2 processes, and we give below a few sample rules which illustrate the main points of the construction. We start with the higher-order features illustrated by the rule for output, given by

$$(\textsc{Out}) \qquad \overline{l{\cdot}c}\langle\tilde{v}\rangle \xrightarrow{\overline{l{\cdot}c}\langle\tilde{v}\rangle} l{\cdot}0$$

Consider the case where $\tilde{v}$ contains a tree with a scripted process $\Box P$ inside. A bisimulation requiring syntactical identity for the action of the simulating process would clearly be too restrictive. For this reason, we resort to higher-order bisimulation: we require the action above to be matched by $\xrightarrow{\tau}{}^{*} \xrightarrow{\overline{l{\cdot}c}\langle\tilde{v'}\rangle} \xrightarrow{\tau}{}^{*}$, where $\tilde{v'}$ contains $\Box Q$, with $P \approx Q$.

Higher order bisimulation for concurrent processes has been studied for example in [30, 12, 29]. A well-known problem with the technique consists in proving the congruence property of bisimulation, which is complicated by having the operators for parallel composition and functional application in the same calculus. In Xπ_2, the only form of functional application is given by the command run for running scripted code. We can get around the congruence problem by showing that our bisimulation, based on the lts with the rule

$$(\textsc{Run}) \qquad l{\cdot}\mathsf{run}_p \xrightarrow{l{\cdot}\mathsf{run}_p} l{\cdot}0$$

is a congruence, without incurring in the problem mentioned above. We then show, by the soundness of the proof method, that this choice is compatible with the semantics of the calculus. The idea is that running a script is just like placing in parallel a new process, and if bisimulation is closed under parallel composition, then it is sound with respect to script execution.

We now illustrate a subtlety of bisimulation related to network composition. Consider the two X$d\pi$ networks $N = l\,[\,T \,\|\, 0\,]$ and $M = l\,[\,T \,\|\, Q\,]$, where $Q = \mathsf{go}\ m.\mathsf{go}\ l.\mathsf{cut}_{/}(X)$. We have that $0 \not\sim_b Q$, since composing N and M with a network containing location m, Q can reduce and produce a barb. The bisimulation relation must therefore be able to cope with the extension of the domain of processes. We adopt the following technique: in order for two processes to be bisimilar, they must have the same domain, and if one makes and action, *possibly extending the domain*, the other one must match the action and become a bisimilar process, and therefore have the same (extended) domain. The extension of the domain is made possible by the lts rule

$$(\textsc{Zero}) \qquad 0 \xrightarrow{\tau} l{\cdot}0$$

The bisimulation relation sketched above is not complete. For example there is a problem inherently connected with the higher-order technique that we have chosen: even requiring bisimilarity for $\Box P$ and $\Box Q$ on the actions can be considered too restrictive. In fact, it could be the case that $P \not\sim_b Q$, but $\Box P$ and $\Box Q$ are only run inside some context C such that $C[P] \sim_b C[Q]$. We leave it to future work to explore this interesting issue.

6 Concluding Remarks

This paper introduces X$d\pi$, a simple calculus for describing the interaction between data and processes across distributed locations. We use a simple data model consisting of unordered labelled trees, with embedded processes and pointers to other parts of the network, and π-processes extended with an explicit migration primitive and an update command for interacting with data. Unlike the π-calculus and its extensions, where typically data are encoded as processes, the X$d\pi$-calculus models data and processes at the same level of abstraction, enabling us to study how properties of data can be affected by process interaction.

Alex Ahern has developed a prototype implementation, adapting the ideas presented here to XML standards. The implementation embeds processes in XML documents and uses XPath as a query language. Communication between peers is provided through SOAP-based web services and the working space of each location is endowed with a process scheduler based on ideas from PICT [24]. We aim to continue this implementation work, perhaps incorporating ideas from other recent work on languages based on the π-calculus [11, 13].

There are many similarities between our model and features of the Active XML [4] implementation, and we are in the process of doing an in-depth comparison between the two projects.

Developing process equivalences for X$d\pi$ is non-trivial. We have defined a notion of barbed equivalence between processes, based on the update operations that processes can perform on data, and have briefly described a proof method for proving such equivalences between processes. There are other possible definitions of observational equivalence, and a comprehensive study of these choices will be essential in future. The coinductive proof method we have developed is useful for many examples, but it is not complete, and we leave it to future work to study alternative techniques. We also plan to adapt type theories and reasoning techniques studied for distributed process calculi to analyse security properties. This paper has provided a first step towards the adaptation of techniques associated with process calculi to reason about the dynamic evolution of data on the Web.

Acknowledgements. We would like to thank Serge Abiteboul, Tova Milo, Val Tannen, Luca Cardelli, Giorgio Ghelli and Nobuko Yoshida for many stimulating discussions. We thank Alex Ahern and Maria Grazia Vigliotti for comments on a draft of this paper.

References

[1] Abiteboul, S., O. Benjelloun, T. Milo, I. Manolescu and R. Weber, *Active XML: A data-centric perspective on Web services*, Verso Report number 213 (2002).

[2] Abiteboul, S., A. Bonifati, G. Cobena, I. Manolescu and T. Milo, *Dynamic XML documents with distribution and replication*, in: *Proceedings of ACM SIGMOD Conference*, 2003.

[3] Abiteboul, S., P. Buneman and D. Suciu, "Data on the Web: from relations to semistructured data and XML," Morgan Kaufmann, 2000.

[4] Abiteboul, S. et al., *Active XML primer*, INRIA Futurs, GEMO Report number 275 (2003).

[5] Berger, M., "Towards Abstractions for Distributed Systems," Ph.D. thesis, Imperial College London (2002).

[6] Berger, M., K. Honda and N. Yoshida, *Linearity and bisimulation.*, in: *Proceedings of FoSSaCS'02* (2002), pp. 290–301.

[7] Bierman, G. and P. Sewell, *Iota: a concurrent XML scripting language with application to Home Area Networks*, University of Cambridge Technical Report UCAM-CL-TR-557 (2003).

[8] Braumandl, R., M. Keidl, A. Kemper, D. Kossmann, A. Kreutz, S. Seltzsam and K. Stocker, *Objectglobe: Ubiquitous query processing on the internet*, To appear in the VLDB Journal: Special Issue on E-Services (2002).

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

[10] Cardelli, L. and G. Ghelli, *A query language based on the ambient logic*, in: *Proceedings of ESOP'01)*, LNCS **2028** (2001), pp. 1–22.

[11] Conchon, S. and F. L. Fessant, *Jocaml: Mobile agents for Objective-Caml*, in: *Proceedings of ASA'99/MA'99*, Palm Springs, CA, USA, 1999.

[12] Ferreira, W., M. Hennessy and A. Jeffrey, *A theory of weak bisimulation for core cml*, Journal of Functional Programming **8** (1998), pp. 447–491.

[13] Gardner, P., C. Laneve and L. Wischik, *Linear forwarders*, in: *Proceedings of CONCUR 2003*, LNCS **2761** (2003), pp. 415–430.

[14] Gardner, P. and S. Maffeis, *Modeling dynamic Web data*, in: G. Lausen and D. Suciu, editors, *Proc. of DBPL'03*, LNCS **2921**, 2003.

[15] Gardner, P. and S. Maffeis, *Modeling dynamic Web data*, Imperial College London Technical Report (2003).

[16] Godskesen, J., T. Hildebrandt and V. Sassone, *A calculus of mobile resources*, in: *Proceedings of CONCUR'02*, LNCS, 2002.

[17] Gordon, A. and R. Pucella, *Validating a web service security abstraction by typing*, in: *Proceedings of the 2002 ACM Workshop on XML Security*, 2002, pp. 18–29.

[18] Hennessy, M. and J. Riely, *Resource access control in systems of mobile agents*, in: *Proceedings of HLCL '98*, ENTCS **16.3** (1998), pp. 3–17.

[19] Honda, K. and M. Tokoro, *An object calculus for asynchronous communication*, in: *Proceedings of ECOOP*, LNCS **512** (1991), pp. 133–147.

[20] Honda, K. and N. Yoshida, *On reduction-based process semantics*, Theoretical Computer Science **151** (1995), pp. 437–486.

[21] Kemper, A. and C. Wiesner, *Hyperqueries: Dynamic distributed query processing on the internet*, in: *Proceedings of VLDB'01*, 2001, pp. 551–560.

[22] World Wide Web Consortium, *XML Path Language (XPath) Version 1.0*, available at `http://w3.org/TR/xpath`.

[23] Milner, R., J. Parrow and J. Walker, *A calculus of mobile processes, I and II*, Information and Computation **100** (1992), pp. 1–40,41–77.

[24] Pierce, B. C. and D. N. Turner, *Pict: A programming language based on the pi-calculus*, in: *Proof, Language and Interaction: Essays in Honour of Robin Milner* (2000).
URL `citeseer.nj.nec.com/pierce97pict.html`

[25] Sahuguet, A., "ubQL: A Distributed Query Language to Program Distributed Query Systems," Ph.D. thesis, University of Pennsylvania (2002).

[26] Sahuguet, A., B. Pierce and V. Tannen, *Distributed Query Optimization: Can Mobile Agents Help?*, Unpublished draft.

[27] Sahuguet, A. and V. Tannen, *Resource Sharing Through Query Process Migration*, University of Pennsylvania Technical Report MS-CIS-01-10 (2001).

[28] Sangiorgi, D. and D. Walker, "The π-calculus: a Theory of Mobile Processes," Cambridge University Press, 2001.

[29] Sangirogi, D., *Expressing mobility in process algebras: First-order and higher-order paradigms*, PhD thesis, University of Edinburgh (1992).

[30] Thomsen, B., *A theory of higher order communicating systems*, Inf. Comput. **116** (1995), pp. 38–57.

A The Xπ_2-calculus

In Table 4 we give the syntax of Xπ_2. Trees are defined as in X$d\pi$. Networks are represented by pairs where the first component (the *store*) is a function from location names to data trees, and the second component is a parallel composition of located processes. The processes $l{\cdot}0$ and $\overline{l{\cdot}b}\langle\tilde{v}\rangle$ represent respectively a null process and an output process located at l — the input, replicated input, migration and update are similar, and 0 stands for the null location. We use notation $^{l}\!\frown P$ to express that P is a parallel composition of processes located at l. The set $loc(P)$ denotes all l_i such that $P \equiv (\nu\tilde{b})(^{l_1}\!\frown P_1 \mid \cdots \mid {}^{l_n}\!\frown P_n)$. Well-formedness requires that $loc(P) = dom(D)$ for (D, P), that the migration message cannot be prefixed by any other process, that the continuation of an explicitly located process be located at the same location, except for the case of `go` $m.P$, where P must be located at m. Scripted processes must be located at $\circlearrowleft$. Additionally, well-formedness requires the same conditions given for X$d\pi$. In practice we encode X$d\pi$ terms in Xπ_2 terms, which are then well-formed by construction. This guarantees also that starting from a well-formed network (D, P) there is always at least one process (possibly null) located at l, for each $l \in dom(D)$, and there is never a process located at m for $m \notin dom(D)$.

Structural congruence for trees, stores, networks and processes is defined in Table 5. Note how $l{\cdot}0 \mid {}^{l}\!\frown P \equiv {}^{l}\!\frown P$, whereas for example $l{\cdot}0 \not\equiv 0$. In Table 6 below, we give the semantic rules for Xπ_2. The rules correspond closely with those for X$d\pi$.

We give below the definitions of barbed congruence for Xπ_2, which are analogous to the ones for X$d\pi$.

Definition A.1 *Reduction contexts for processes are* $C' ::= - \mid C' \mid P \mid (\nu c)C'$*, and reduction contexts for networks are* $C ::= (B \uplus -, C')$*, where* $C[(D, P)]$ *is* $(B \uplus D, C'[P])$*. Composition is defined only for stores with disjoint domains.*

Definition A.2 *Barbs for networks are defined as*

$$(D, P) \downarrow_{l\cdot p} \triangleq \exists C', \chi, U, P'.\ P \equiv C'[l{\cdot}p\chi U.P']$$
$$(D, P) \Downarrow_{l\cdot p} \triangleq (D, P) \rightarrow^* (D', P') \wedge (D', P') \downarrow_{l\cdot p}$$

Definition A.3 *Barbed congruence* ($\cong_b$) *is the largest symmetric relation* $\mathcal{R}$ *on X*π_2 *networks such that* $(D, P)\,\mathcal{R}\,(B, Q)$ *implies:*

- $(D, P) \downarrow_{l\cdot p} \Rightarrow (B, Q) \Downarrow_{l\cdot p}$;
- $(D, P) \rightarrow (D', P') \Rightarrow (\exists B', Q'.(B, Q) \rightarrow^* (B', Q') \wedge (D', P')\,\mathcal{R}\,(B', Q'))$;
- $\forall\ C.C[(D, P)]\,\mathcal{R}\,C[(B, Q)]$.

The encoding from X$d\pi$ to Xπ_2 and the labelled-bisimulation technique are defined formally in [15].

(Trees)	$T ::= 0 \mid T \vert T \mid \mathsf{a}[\,T\,] \mid \mathsf{a}[\,\Box P\,] \mid \mathsf{a}[\,@l{:}p\,]$
(Processes)	$P ::= \quad l{\cdot}0 \mid P \,\vert\, P \mid \overline{l{\cdot}b}\langle\tilde{v}\rangle \mid l{\cdot}b(\tilde{z}).P \mid !l{\cdot}b(\tilde{z}).P \mid (\nu c)P$ $\mid\, l{\cdot}\mathsf{go}\, m.P \mid l{\cdot}\mathsf{update}_p(\chi, V).P \mid l\langle P\rangle \mid 0$
(Stores)	$D ::= \emptyset \mid \{l \mapsto T\} \uplus D$
(Networks)	(D, P)

Table 4: The calculus Xπ_2.

Structural congruence is the least congruence satisfying alpha-conversion, the commutative monoidal laws for $(0, \vert)$ on trees, processes and networks, and the axioms reported below:

(Trees)	$U \equiv U' \Rightarrow \mathsf{a}[\,U\,] \equiv \mathsf{a}[\,U'\,]$
(Values)	$v' \equiv w' \wedge \tilde{v} \equiv \tilde{w} \Rightarrow v', \tilde{v} \equiv w', \tilde{w} \quad P \equiv Q \Rightarrow \Box P \equiv \Box Q$
(Processes)	$(\nu c)0 \equiv 0 \quad l{\cdot}0 \vert^{l\curvearrowright} P \equiv^{l\curvearrowright} P \qquad (\nu c) l{\cdot}0 \equiv l{\cdot}0$
	$c \notin fn(P) \Rightarrow P \vert (\nu c)Q \equiv (\nu c)(P \vert Q) \quad (\nu c)(\nu c')P \equiv (\nu c')(\nu c)P$
	$V \equiv V' \wedge P \equiv Q \Rightarrow l{\cdot}\mathsf{update}_p(\chi, V).P \equiv l{\cdot}\mathsf{update}_p(\chi, V').Q$
(Stores)	$dom(D) = dom(B) \wedge (\forall l \in dom(D).D(l) \equiv B(l)) \Rightarrow D \equiv B$
(Networks)	$D \equiv B \wedge P \equiv Q \Rightarrow (D, P) \equiv (B, Q)$
(Abstractions)	$V \equiv V' \Rightarrow (\chi)V \equiv (\chi)V'$

Table 5: Structural congruence for Xπ_2.

The reduction relation is the least relation generated by the axioms below, closed with respect to structural congruence and contexts.

(Exit)	$(\emptyset, m{\cdot}\mathsf{go}\ l.P) \ \rightarrow (\emptyset, m{\cdot}0 \,\vert\, l\langle P\rangle)$
(Enter)	$(\{l \mapsto T\}, l\langle P\rangle) \ \rightarrow (\{l \mapsto T\}, P)$
(Com)	$(\emptyset, \overline{l{\cdot}c}\langle\tilde{v}\rangle \mid l{\cdot}c(\tilde{x}).P) \rightarrow \ (\emptyset, P\{\tilde{v}/\tilde{x}\})$
(!Com)	$(\emptyset, \overline{l{\cdot}c}\langle\tilde{v}\rangle \mid !l{\cdot}c(\tilde{x}).P) \rightarrow \ (\emptyset, !l{\cdot}c(\tilde{x}).P \vert P\{\tilde{v}/\tilde{x}\})$
(Update)	$\dfrac{p(T) \rightsquigarrow_{p,l,\chi,V} T', \{\sigma_1, \cdots, \sigma_n\}}{(\{l \mapsto T\}, l{\cdot}\mathsf{update}_p(\chi, V).P) \rightarrow \ (\{l \mapsto T'\}, P\sigma_1 \mid \cdots \mid P\sigma_n)}$
(Run)	$\dfrac{p(T) \rightsquigarrow_{p,l,\Box X,\Box X} T, \{\{\Box P_1/\Box X\}, \cdots, \{\Box P_n/\Box X\}\}}{(\{l \mapsto T\}, l{\cdot}\mathsf{run}_p) \rightarrow \ (\{l \mapsto T\}, P_1 \mid \cdots \mid P_n)}$

(The function $\rightsquigarrow$ is analogous to the one defined in Table 1.)

Table 6: Reduction axioms for Xπ_2.

Structural congruence is the least congruence satisfying alpha-conversion, the commutative monoidal laws for $(0, |)$ on trees, processes and networks, and the axioms reported below:

(Trees)

$$U \equiv V \Rightarrow \mathsf{a}[\,U\,] \equiv \mathsf{a}[\,V\,]$$

(Values)

$$P \equiv Q \Rightarrow \Box P \equiv \Box Q \qquad v' \equiv w' \wedge \tilde{v} \equiv \tilde{w} \Rightarrow v', \tilde{v} \equiv w', \tilde{w}$$

(Processes)

$$(\nu c)0 \equiv 0 \qquad c \notin fn(P) \Rightarrow P \,|\, (\nu c)Q \equiv (\nu c)(P \,|\, Q) \qquad (\nu c)(\nu c')P \equiv (\nu c')(\nu c)P$$
$$\tilde{v} \equiv \tilde{w} \Rightarrow \overline{c}\langle\tilde{v}\rangle \equiv \overline{c}\langle\tilde{w}\rangle \qquad V \equiv V' \wedge P \equiv Q \Rightarrow \mathsf{update}_p(\chi, V).P \equiv \mathsf{update}_p(\chi, V').Q$$

(Networks)

$$(\nu c)0 \equiv 0 \qquad c \notin fn(N) \Rightarrow N \,|\, (\nu c)M \equiv (\nu c)(N \,|\, M) \qquad (\nu c)(\nu c')N \equiv (\nu c')(\nu c)N$$
$$P \equiv Q \Rightarrow l\langle P\rangle \Rightarrow l\langle Q\rangle \qquad l\left[\,T \,\|\, (\nu c)P\,\right] \equiv (\nu c)l\left[\,T \,\|\, P\,\right]$$
$$T \equiv S \wedge P \equiv Q \Rightarrow l\left[\,T \,\|\, P\,\right] \equiv l\left[\,S \,\|\, Q\,\right]$$

Table 7: Structural congruence for X$d\pi$.