BEHAVIOURAL EQUIVALENCES FOR DYNAMIC WEB DATA

Sergio Maffeis and Philippa Gardner
Department of Computing, Imperial College London, UK.
{maffeis,pg}@doc.ic.ac.uk

Abstract We study behavioural equivalences for dynamic web data in **Xdπ**, a model for reasoning about behaviour found in (for example) dynamic web page programming, applet interaction, and web-service orchestration. **Xdπ** is based on an idealised model of semistructured data, and an extension of the **π-calculus** with locations and operations for interacting with data. The equivalences are non-standard due to the integration of data and processes, and the presence of locations.

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. A process definition can be included in a document as an atomic piece of data, 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 [1], distributed query patterns [19], hyperlinks, forms, and scripting.

The **Xdπ-calculus** [7] provides a formal description of such systems. It is based on a network of locations (peers) containing a (semi-structured) data model, and **π-like** processes [17, 20, 10] for modelling process interaction, process migration, and interaction with data. The data model consists of unordered labelled trees, with embedded

processes for querying and updating 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.

A behavioural understanding of dynamic web data can serve as a starting point for the use of formal techniques. Moreover, the combination of web services and scripted processes provides the data engineer with many alternative patterns for exchanging information on the web [2, 19], and equational reasoning becomes useful to show, for example, that some complex data-exchange protocol conforms to its specification.

We study behavioural equivalences in Core **X***dπ*, which is a slight adaptation of **X***dπ*, where both the data and the process component of a network are explicitly located, and therefore easier to analyse independently. We identify two main notions of contextual equivalence for *open* networks, based on the observation of the data structure at each location, or of the capabilities of process to access data. We derive the corresponding process equivalences so that when two equivalent pieces of code are put in the same position in a network, the resulting networks cannot be distinguished by an observer. Process equivalences appear to be sensitive to the set of locations composing the network. This feature, together with having scripted processes as values, requires non trivial techniques for defining a labelled-bisimulation-based proof method. We address interested readers to the full paper [15] for all the technical details.

Related Work. Our model is related to the Active XML approach to data integration developed independently by Abiteboul et al. [2]. Several distributed query languages, such as [19, 14, 4], extend traditional query languages with facilities for distribution awareness. Our approach is closest to the `ubQL` query language of [19], partly motivated by ideas from the **π-calculus** [18]. Process calculi have also been used for example to study security properties of web services [8], and to program XML-based Home Area Networks devices [3].

In [7] we have defined a first notion of barbed equivalence, and we have sketched a proof method based on higher-order bisimulation. In this paper we study in detail behavioural equivalences, improving and extending significantly the previous results. Core **X***dπ* uses ideas from [5], and the contextual equivalences are based on the reduction-closed framework of [12]. Our labelled transition system and bisimulation exploit a translation technique from higher-order to first-order actions proposed in [13], and based on [21]. Ours is the first attempt to study behavioural equivalences of web-based (higher-order) data-sharing applications, and is characterised by its emphasis on dynamic data.

2 Core X*dπ*

In **X***dπ*, a peer-to-peer network is represented as a set of locations (we regard *location* and *peer* as synonyms), each containing a data-tree and some processes. In order to reason modularly on data and processes, we instead model a network in Core **X***dπ* as a pair (D, P), where D is a set of located trees, each one representing the data component of a location, and P is a multiset of located-process, representing both the services provided by each peer and the agents in execution on behalf of other peers.

$$
\begin{array}{lll}
\text{(TREES)} & U \equiv U' \Longrightarrow \mathtt{a}[U] \equiv \mathtt{a}[U'] & \\
\text{(VALUES)} & v' \equiv w' \wedge \tilde{v} \equiv \tilde{w} \Longrightarrow v', \tilde{v} \equiv w', \tilde{w} & P \equiv Q \Longrightarrow \Box P \equiv \Box Q \\
\text{(PROCESSES)} & (\nu c)(\nu c')P \equiv (\nu c')(\nu c)P & (\nu c)0 \equiv 0 \\
 & c \notin fn(P) \Longrightarrow P \mid (\nu c)Q \equiv (\nu c)(P \mid Q) & \\
 & V \equiv V' \wedge P \equiv Q \Longrightarrow l{\cdot}\mathsf{update}_p(\chi, V).P \equiv l{\cdot}\mathsf{update}_p(\chi, V').Q & \\
\text{(STORES)} & \forall l.\, D(l) \equiv B(l) \Longrightarrow D \equiv B & \\
\text{(NETWORKS)} & D \equiv B \wedge P \equiv Q \Longrightarrow (D, P) \equiv (B, Q) &
\end{array}
$$

Table 1. **Structural congruence for Core X$d\pi$ is the least congruence satisfying alpha-conversion, the commutative monoidal laws for $(0, |)$ on trees and processes, and the axioms reported above.**

Trees. Our data model extends the unordered labelled rooted trees of [6], with leaves which can either be scripted processes or pointers to data. We use the following constructs: *edge labels* denoted by a, b, c $\in \mathcal{A}$, *path expressions* denoted by $p, q \in \mathcal{E}$ and used to identify specific subtrees, and *location names* 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

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

Tree 0 denotes a rooted tree with no content. Tree $T_1 \mid T_2$ denotes the composition of T_1 and T_2, which simply joins the roots. A tree of the form a[...] 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. The structural congruence for trees states that trees are unordered, and scripted processes are identified up to the structural congruence for processes (see Table 1).

We regard a path p as a function from trees to sets of nodes (up to structural congruence): $p(T)$ denotes the tree T where the nodes identified by p are selected. For simplicity we do not show node identifiers explicitly, but we underline the selected nodes. We describe paths using a subset of XPath [16], where "a" denotes a step along an edge labelled a, "/" denotes path composition, ".." a step back, "//" any node, and ".", which can appear only in paths inside trees, denotes the path from the root to the current node. For example, in $\underline{\mathtt{a}[\underline{\mathtt{a}[S]} \mid \mathtt{b}[S'] \mid \mathtt{c}[T']]}$ we have underlined the nodes selected by path //a.

Located Processes. Our processes are based on asynchronous **π_2-processes** [5] extended with an operation for manipulating the tree structure (update) and one for selecting a script for execution (run). Generic variables are x, y, z, channel names or channel variables are a, b, c, the meaning will be clear from the context, and values are

$$u, v, w ::= T \mid c \mid l \mid p \mid \Box P$$

We use the notation $\tilde{z}$ for vectors of variables, and $\tilde{v}$ for vectors of values and variables. Identifiers U, V range over scripted processes, pointers and trees. *Patterns* χ, ξ have the form $\chi ::= X \mid @x{:}y \mid \Box X$, where X denotes a tree or process variable. The set of processes, denoted by $\mathcal{P}$, is given by

$$P, Q, R ::= \quad 0 \mid P \mid P \mid \overline{l \cdot b}\langle \tilde{v} \rangle \mid l \cdot b(\tilde{z}).P \mid !l \cdot b(\tilde{z}).P \mid (\nu c)P$$
$$\mid l \cdot \mathsf{update}_p(\chi, V).P \mid l \cdot \mathsf{run}_p$$

The processes in the first line of the grammar are constructs arising from the **π_2-calculus**: the *output* process $\overline{l \cdot b}\langle \tilde{v} \rangle$ denotes a vector of values $\tilde{v}$ waiting to be sent via channel b at location l, the *input* process $l \cdot b(\tilde{z}).P$ is waiting to receive values from an output process via channel b at l, and the standard *nil, composition, restriction* and *replicated input.* Channel names $\mathcal{C}$ 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 can be restricted. We assume the usual notions of *free* and *bound* names (fn, bn) for session channels. Scripted processes cannot have free session names. We assume a simple sorting discipline on channels.

Command $l \cdot \mathsf{run}_p$ activates the scripted processes selected by the path expression p in the tree at l. Command $l \cdot \mathsf{update}_p(\chi, V).P$ is used to interact with the data tree at l. In an update, V may contain variables and must have the same sort as χ. The variables free in χ are bound in V and P. The update command finds all the values V_i given by the path p, and pattern-matches these values with χ to obtain the substitution σ_i when it exists. For each successful pattern-matching, it replaces the V_i with $V\sigma_i$ and evolves to $P\sigma_i$. Below we give some basic commands derived from update:

$$l \cdot \mathsf{copy}_p(X).P \triangleq l \cdot \mathsf{update}_p(X, X).P \qquad \text{copy the tree at } p \text{ and use it in } P$$
$$l \cdot \mathsf{cut}_p(X).P \triangleq l \cdot \mathsf{update}_p(X, 0).P \qquad \text{cut the tree at } p \text{ and use it in } P$$
$$l \cdot \mathsf{paste}_p\langle T \rangle.P \triangleq l \cdot \mathsf{update}_p(X, X \mid T).P \quad \left\{ \begin{array}{l} \text{where } X \text{ is not free in } T \text{ or } P, \\ \text{paste tree } T \text{ at } p \text{ and evolve to } P \end{array} \right.$$

Networks and Stores. A network is represented by a pair *(D, P)* where the first component (the *store*) is a finite partial function from location names to trees, and the second component is a process. Interaction between processes and data is always local, as will be shown by rules (UPDATE) and (RUN) in Table 2, and consequently we regard the store as *distributed.* We write *dom*(*D*) to denote the domain of store *D.* We write $D_1 \uplus D_2$ for the union of stores D_1 and D_2 with disjoint domains. The network (*D, P*) is well-formed if *D* and *P* contain no free variables, and all the scripted processes have no free session names.

Our reduction semantics on networks will be closed with respect to *network contexts* $(C_{\mathcal{S}}, C_{\mathcal{P}})$, where *store contexts* $C_{\mathcal{S}}$ are defined by $C_{\mathcal{S}} ::= - \mid C_{\mathcal{S}} \uplus D$ and *process contexts* $C_{\mathcal{P}}$ are defined by $C_{\mathcal{P}} ::= - \mid C_{\mathcal{P}} \mid P \mid (\nu c)\, C_{\mathcal{P}}$. Given a network (D, P) and a context $C = (C_{\mathcal{S}}, C_{\mathcal{P}})$, we write $C\{(D, P)\}$ for their composition: for example, if $C_{\mathcal{S}} = - \uplus B$, $C_{\mathcal{P}} = (\nu c)-$ then $C\{(D, P)\} = (D \uplus B, (\nu c)P)$. A composition involving stores is defined only for stores with disjoint domains. We will omit the subscripts from contexts when no ambiguity can arise.

Reduction Semantics. The reduction relation $\rightarrow$, relying on an updating function $\rightsquigarrow$, describes processes interaction, the interaction between processes and data, and (implicitly) the movement of processes across locations (Table 2).

$$\text{(Com)} \quad (\{l \mapsto T\}, \overline{l{\cdot}c}\langle\tilde{v}\rangle \mid l{\cdot}c(\tilde{x}).P) \rightarrow (\{l \mapsto T\}, P\{\tilde{v}/\tilde{x}\})$$

$$\text{(!Com)} \quad (\{l \mapsto T\}, \overline{l{\cdot}c}\langle\tilde{v}\rangle \mid !l{\cdot}c(\tilde{x}).P) \rightarrow (\{l \mapsto T\}, !l{\cdot}c(\tilde{x}).P \mid P\{\tilde{v}/\tilde{x}\})$$

$$\text{(Update)} \quad \frac{p(T) \rightsquigarrow_{p,l,\chi,V} T', \{\sigma_1, \cdots, \sigma_n\}}{(\{l \mapsto T\}, l{\cdot}\mathsf{update}_p(\chi, V).P) \rightarrow (\{l \mapsto T'\}, P\sigma_1 \mid \cdots \mid P\sigma_n)}$$

$$\text{(Run)} \quad \frac{p(T) \rightsquigarrow_{p,l,\Box X,\Box X} T, \{\{\Box P_1/\Box X\}, \cdots, \{\Box P_n/\Box X\}\}}{(\{l \mapsto T\}, l{\cdot}\mathsf{run}_p) \rightarrow (\{l \mapsto T\}, P_1 \mid \cdots \mid P_n)}$$

(Reduction) The reduction relation on processes is the smallest relation closed with respect to reduction contexts, structural congruence and the axioms above.

$$\text{(Zero)} \quad 0 \rightsquigarrow_\Theta 0, \emptyset \qquad \text{(Link)} \quad \mathtt{a}[@m{:}q] \rightsquigarrow_\Theta \mathtt{a}[@m{:}q], \emptyset$$

$$\text{(Script)} \quad \mathtt{a}[\Box Q] \rightsquigarrow_\Theta \mathtt{a}[\Box Q], \emptyset \qquad \text{(Node)} \quad \frac{T \rightsquigarrow_\Theta T', \Sigma}{\mathtt{a}[T] \rightsquigarrow_\Theta \mathtt{a}[T'], \Sigma}$$

$$\text{(Par)} \quad \frac{T \rightsquigarrow_\Theta T', \Sigma_1 \qquad S \rightsquigarrow_\Theta S', \Sigma_2}{T \mid S \rightsquigarrow_\Theta T' \mid S', \Sigma_1 \oplus \Sigma_2}$$

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

(Updating Function) Above, Σ is a multiset of substitutions, and $\oplus$ is multiset union, and $\Theta = p, l, \chi, V$ are the parameters of an update or run command.

Table 2. **Reduction axioms and updating function for Core X$d\pi$.**

First we describe the reduction relation. Rules (Com) and (!Com) are basically the standard communication rules for the **π-calculus,** except that processes only communicate if they are at the same location l, and l is in the store. Rule (Update) provides interaction between processes and data. Given the command $l{\cdot}\mathsf{update}_p(\chi, V).P$ and the tree T at l in the store, the updating function $\rightsquigarrow$ takes $p(T)$ as an argument, matches each identifier U_i in $p(T)$ with the pattern χ to obtain the substitution σ_i, replaces each U_i with $V\sigma_i$ in T, and returns the continuation process $P_i\sigma_i$. Rule (Run) is a special case of update, where the tree is not modified, and the scripted processes $\Box P_i$ identified by $p(T)$ are activated in parallel to yield the continuation P_i.

We now describe the updating function $\rightsquigarrow$, which is parameterised by p, l, χ, V, the arguments of an update or run command. The first five rules define simply a traversal of the tree collecting the set of substitutions Σ, whereas rule (Up) is responsible for the actual update. It applies to the identified nodes (underlined), matching U with χ, to obtain substitution σ (in our case patterns are simple, and pattern-matching is trivial, but the approach can be extended to more complicated patterns). When σ exists, the process continues recursively updating $V\sigma$, until some subtree V' with a set of substitutions denoted by Σ is returned. At this point U is replaced with V', and Σ is returned, together with $\sigma\{l/\circlearrowleft, p/.\}$ (where any references to the current location $\circlearrowleft$ and position "." are substituted by the actual values l and p).

For example, consider $T = \mathtt{c}[\mathtt{a}[T_1] \mid \mathtt{a}[T_2] \mid \mathtt{b}[S]]$ and $T' = \mathtt{c}[\mathtt{a}[0] \mid \mathtt{a}[0] \mid \mathtt{b}[S]]$, and a $\mathsf{cut}_{\mathtt{c/a}}(X)$ command to remove the subtrees at c/a. We have

$$(\{l \mapsto T\}, l{\cdot}\mathsf{cut}_{\mathtt{c/a}}(X).P) \rightarrow (\{l \mapsto T'\}, P\{T_1/X\} \mid P\{T_2/X\})$$

where the subtrees T_1 and T_2, identified by `c/a`, are removed from the store, and each is passed to a copy of P. As an example of command run and of the substitution of local references, consider $S = \mathtt{a}[\,\mathtt{b}[\,\Box\overline{\circlearrowleft \cdot a}\langle v\rangle\,]\,|\,\mathtt{b}[\,\Box m\cdot\mathsf{run}_{./../c}\,]\,]$. We have

$$(\{l \mapsto S\}, \mathsf{run}_{\mathtt{a/b}}) \rightarrow (\{l \mapsto S\}, \overline{l\cdot a}\langle v\rangle \,|\, m\cdot\mathsf{run}_{\mathtt{a/b/../c}})$$

The store S is unaffected by the run operation, which spawns the two processes identified by `a/b`, where the local path `./../c` is replaced by `a/b/../c`, and $\circlearrowleft$ is replaced by l. Note that $m\cdot\mathsf{run}_{\mathtt{a/b/../c}}$ is located at m, which is not in the domain of the store. There is no reduction rule for such a process, which represents mobile code "lost" due to network partitioning, or to an invalid network address. In fact, in our model it is not possible for a process to create a new location. Processes represent either scripts or web services, none of which could realistically create new peers, hence the domain of a network is invariant under reduction. Nonetheless we consider open systems, since we admit network composition. Our approach differs from the one of e.g. [9], where process migration can have the effect of creating a new location. We will see in Section 3 how our choice requires new techniques for studying behavioural equivalences.

We conclude the section with an example on web services, see [7] for other motivating examples (other web services, XLink, e-forms). Consider, at location m, a web service `get` for downloading data which, given a path expression p, returns a stream of messages containing the subtrees denoted by p at m. The service is described by process $m\cdot\mathbf{get} = !m\cdot get(x, y, z).m\cdot\mathsf{copy}_x(Y).\overline{y\cdot z}\langle Y\rangle$, where channel *get* inputs a path x, a location y, and a channel z, and returns its results at y on z. The corresponding service invocation from l is

$$l\cdot\mathtt{call}(m, get, p) = (\nu c)(\overline{m\cdot get}\langle p, l, c\rangle \,|\, !l\cdot c(Y).R),$$

where R is some code handling each result. We will see in Section 4 that invoking $m\cdot\mathbf{get}$ with $l\cdot\mathtt{call}(m, get, p)$ is equivalent to running (from l) the specification $l\cdot\mathbf{spec} = m\cdot\mathsf{copy}_x(Y).R$.

3 Contextual Equivalences for Core X$d\pi$

In this section we study equivalences for networks and processes. In particular, we define when two processes are equivalent in such a way that when they are put in the same position in the network, the resulting networks are equivalent. In Section 4, we introduce a proof method for showing process equivalence.

Network Equivalences. We base our network equivalences on the reduction-closed framework of Honda and Yoshida [12]. The equivalences depend on the choice of observables, and we have studied several cases.

In the setting of dynamic web data, a natural criterion to decide when two systems are equivalent is to compare the structure of the data tree at each location without looking directly at processes, which can be seen as working in the background, and hence not directly observable. The analysis of processes is implicit in the reduction closure property. Below we will define *tree congruence* as the equivalence induced by tree observations. In the full paper [15], we show that tree congruence coincides with two other reduction congruences induced by different observables: one records whether a located tree is empty, the other records located output capabilities.

Another natural choice for observables, motivated by security concerns, is to consider the capabilities of a processes to access data. This notion of equivalence, defined later on as *barbed congruence,* proves to be more restrictive than tree congruence.

We begin with standard generic definitions, based on some observation relation $N \downarrow_\beta$ which states that network N exhibits the observable β. We then study specific observation relations.

DEFINITION 1 *The* weak observation relation *induced by* $\downarrow_\beta$, *denoted by* $\Downarrow_\beta$, *is defined by* $N \Downarrow_\beta \triangleq \exists N'. N \rightarrow N' \wedge N' \downarrow_\beta$. *The* reduction congruence induced by $\downarrow_\beta$, *denoted by* $\simeq$, *is the largest symmetric relation* $\dot{\simeq}$ *on networks such that* $N \dot{\simeq} M$ *implies*

- *N and M have the same observables:* $N \downarrow_\beta \Rightarrow M \Downarrow_\beta$;
- $\dot{\simeq}$ *is reduction-closed:* $N \rightarrow N' \Rightarrow (\exists M'. M \rightarrow^* M' \wedge N' \dot{\simeq} M')$;
- $\dot{\simeq}$ *is closed under network contexts:* $\forall C. C[N] \dot{\simeq} C[M]$.

We now define tree congruence. Comparing trees up to structural congruence would be overly restrictive, since scripted processes can be semantically equivalent without being structurally congruent. We consider a weaker notion of equivalence on trees which does not look at scripts or pointers. These can be analysed indirectly by suitable contexts.

DEFINITION 2 *We define* observation congruence $(\equiv^t)$, *as the structural congruence of Table 1 with the additional axioms* $\Box P \equiv^t \Box Q$ *and* $@l{:}p \equiv^t @m{:}q$.

As an example of observation congruence, consider $T = \mathbf{a}[\mathbf{b}[\Box P] \mid \mathbf{b}[T'] \mid \mathbf{c}[@l{:}p]]$ and $S = \mathbf{a}[\mathbf{c}[@m{:}q] \mid \mathbf{b}[S'] \mid \mathbf{b}[\Box Q]]$, with $T' \equiv^t S'$. We have $T \equiv^t S$.

DEFINITION 3 *A* tree observable *has the form* $l \cdot T$, *where* l *is a location name and T is a tree. We define the observation relation* $N \downarrow_{l \cdot T}$ *on networks and tree observables by* $N \downarrow_{l \cdot T} \triangleq \exists C, S. N = C\{(\{l \mapsto S\}, 0)\} \wedge S \equiv^t T$: *that is, N contains a location* l *with an S tree-congruent to T.* Tree congruence $(\simeq^t)$ *is the reduction congruence induced by tree observables.*

For example, consider the network $\mathbf{alt}(T, S) = (\{l \mapsto S\}, (\nu c)(\overline{l \cdot c}\langle T \rangle \mid \mathbf{swap}))$, and the process $\mathbf{swap} = !l \cdot c(X).l \cdot \mathbf{update}_/(Y, X).l \cdot \mathbf{update}_/(Z, Y).\overline{l \cdot c}\langle Z \rangle$, which records in Y the tree at l, replaces it by X, and then does the inverse action. We have that $\mathbf{alt}(T, S) \simeq^t \mathbf{alt}(S, T)$ for any T and S, since each process can mimic the other and swap the trees, even if the two networks start with different stores. As an example of non-equivalence, and of how scripts are analysed by contexts, consider the network $\mathbf{net}(P) = (\{l \mapsto \mathbf{a}[\Box P]\}, 0)$ and processes $P_1 = \circlearrowleft \cdot \mathbf{cut}_/(X)$ and $P_2 = \circlearrowleft \cdot \mathbf{paste}_/(\mathbf{b}[0])$. We have $\mathbf{net}(P_1) \not\simeq^t \mathbf{net}(P_2)$, since $\mathbf{test} = (-, - \mid l \cdot \mathbf{run}_a)$ distinguishes $\mathbf{net}(P_1)$ from $\mathbf{net}(P_2)$: $\mathbf{test}\{\mathbf{net}(P_1)\} \Downarrow_{l \cdot 0}$ but $\mathbf{test}\{\mathbf{net}(P_2)\} \not\Downarrow_{l \cdot 0}$.

We now consider a different equivalence notion based on the observation of *barbs* revealing where a process can potentially read or write in a located tree.

DEFINITION 4 *A* barb *has the form* $l \cdot p$, *where* l *is a location name and* p *is a path expression. We define the observation relation* $N \downarrow_{l \cdot p}$ *on networks and barbs*

by $N \downarrow_{l \cdot p} \triangleq \exists C, T, \chi, U, P.\ N \equiv C\{(\{l \mapsto T\}, l \cdot \mathbf{update}_p(\chi, U).P)\}$*: that is, N contains a location* l *with an* $\mathbf{update}_p$ *command.* Barbed congruence $(\simeq^b)$ *is the reduction congruence induced by barbs.*

For example, if $\mathbf{xch}(T_1, T_2) = (\nu c)(\overline{l \cdot c}\langle\rangle \mid !l \cdot c().l \cdot \mathbf{update}_p(X, T_1).l \cdot \mathbf{update}_p(X, T_2).\overline{l \cdot c}\langle\rangle)$ then we have $(\{l \mapsto S\}, \mathbf{xch}(T_1, T_2)) \simeq^b (\{l \mapsto S\}, \mathbf{xch}(T_2, T_1))$, for all T_1, T_2 and S. In fact, the processes have the same barbs, and if S contains a subtree at p, they can simulate each other.

Notice that a barb $l \cdot p$ merely records the location and the path at which some update command could take place, giving no information on *how* the data could be modified, and ignoring run commands. Again, this information can be observed indirectly using some context.

THEOREM 5 *Barbed congruence strictly implies tree congruence:* $\simeq^b \subsetneq \simeq^t$.

The inclusion is strict: for all D, $(D, 0) \simeq^t (D, l \cdot \mathbf{copy}_p(x))$, since the stores are equal and $l \cdot \mathbf{copy}_p(x)$ has no effect, but $(D, 0) \not\simeq^b (D, l \cdot \mathbf{copy}_p(x))$ since $l \cdot \mathbf{copy}_p(x) \downarrow_{l \cdot p}$. This correspond to the intuition that barbed congruence is more operational than tree congruence. Structural congruence for networks is included in $\simeq^b$, and therefore in $\simeq^t$.

Process Equivalences. We now analyse process behaviour, which is influenced by the locations present in the network *(network connectivity).* Consider replacing the definition of a service at location l, which uses only local data, with an equivalent one depending on data from another location m. If we can assume that m is always connected, then the behaviour of the services is the same. On the other hand, if location m should fail, the behaviour of the new one is affected. With network equivalences, the "reliable" locations are those in the domain of the store. With process equivalences, it is necessary to state explicitly the minimum set of reliable locations. For example, consider $\mathbf{oldS} = l \cdot \mathbf{cut}_\prime(X)$ and $m \cdot \mathbf{newS} = (\nu c)(\overline{m \cdot c}\langle\rangle \mid m \cdot c(x).l \cdot \mathbf{cut}_x(X))$. The two processes are equivalent if m is reliable, otherwise they are not: in the context $(\{l \mapsto T\}, -)$ the first process can delete T, but the second one cannot move. As a consequence, in order for two processes to be equivalent, they must be equivalent in all possible network contexts, starting from a given domain.

DEFINITION 6 *Given a network equivalence* $\simeq$ *and a set of location names* Λ*, we define the induced* domain process equivalence *by* $\sim_\Lambda = \{(P, Q) \mid \forall D.\ \Lambda \subseteq dom(D) \Longrightarrow (D, P) \simeq (D, Q)\}$. Domain tree equivalence, $(\sim^t_\Lambda)$*, is the domain process equivalence induced by* $\simeq^t$*, and* domain barbed equivalence $(\sim^b_\Lambda)$*, is the one induced by* $\simeq^b$.

For example, for any Λ, $\mathbf{xch}(T_1, T_2) \sim^b_\Lambda \mathbf{xch}(T_2, T_1)$. Similarly to the case for network equivalences (Theorem 5), we have $\sim^b_\Lambda \subsetneq \sim^t_\Lambda$, with the same counterexample.

In order to be able to replace a process sub-term by an equivalent one, we extend process equivalences to *open* terms (terms with free variables).

DEFINITION 7 Full process contexts *are defined by*

$$C ::= - \mid C \mid P \mid (\nu c)\ C \mid l \cdot a(\tilde{x}).C \mid !l \cdot a(\tilde{x}).C \mid l \cdot \mathbf{update}_p(\chi, V).C$$

DEFINITION 8 *A substitution* σ *is a* closing substitution *for P iff* $P\sigma$ *is closed. Given an equivalence ~ for closed processes, and two open processes P and Q, we say that* $P \sim Q$ *iff* $P\sigma \sim Q\sigma$ *for all closing substitutions* σ.

THEOREM 9 *For all* Λ, *(i) if* $\Lambda \subset \Lambda'$ *then* $\sim^t_\Lambda \subset \sim^t_{\Lambda'}$ *and* $\sim^b_\Lambda \subset \sim^b_{\Lambda'}$; *(ii)* $\sim^t_\Lambda$ *and* $\sim^b_\Lambda$ *(both on open and closed processes) are congruences over full contexts.*

As an example for the strict inclusion of (i), consider the processes `olds` and $m{\cdot}\mathbf{newS}$ given above. We have `olds` $\sim^b_{l,m}$ $m{\cdot}\mathbf{newS}$ but `olds` $\not\sim^b_l$ $m{\cdot}\mathbf{newS}$. In the full paper [15], we show that process tree and barbed equivalences are in fact the largest congruences compatible with the corresponding network equivalences.

Core $\mathbf{X}d\pi$ is an extension of the asynchronous π-**calculus,** and accordingly the *asynchrony law* – stating that the presence of a communication buffer cannot be observed – holds also in our setting: $!l{\cdot}a(x).\overline{l{\cdot}a}\langle x\rangle \sim^b_\Lambda 0$. On the other hand, the law for *equators* does not hold: **let** $l{\cdot}\mathbf{E}(a,b) = !l{\cdot}a(x).\overline{l{\cdot}b}\langle x\rangle \mid !l{\cdot}b(x).\overline{l{\cdot}a}\langle x\rangle$, then

$$l{\cdot}\mathbf{E}(a,b) \mid \overline{l{\cdot}c}\langle a\rangle \not\sim^b_\Lambda l{\cdot}\mathbf{E}(a,b) \mid \overline{l{\cdot}c}\langle b\rangle,$$

since a context can read b from c at l, and use it at some fresh location m where no equator is defined. In the next section we will show how using *distributed equators* it is possible regard different names interchangeably only on some designated locations.

4 A Proof Method for Process Equivalence

The process equivalence given in Definition 6, is hard to use in practice, because it requires closure under all store and process contexts. In this section we provide a coinductive equivalence which does not quantify over contexts.

The main difficulties involved in defining such an equivalence for Core $\mathbf{X}d\pi$ are caused by having scripted processes among values, and by barbed equivalence being sensitive to the presence of locations. We solve the first problem by translating messages containing scripts into ones where each script is replaced by a uniquely named *trigger* (a placeholder), and placing in parallel some *definitions* associating each trigger with the code of the scripted process. Using this approach it is possible to analyse the interaction between scripts and their contexts. For a discussion of this technique see [13, 21], where it is used on the higher-order π-**calculus.** We solve the second problem using an adaptation of the bisimulation approach to families of relations indexed by sets of locations, which we call *domain-dependent bisimilarity.* Communication is asynchronous, hence we borrow techniques from the asynchronous π-**calculus.**

Labelled Transition System. Let $\mathcal{K}$, ranged over by i, j, k, be the set of *trigger* names, disjoint from the channel names in $\mathcal{C}$. We introduce a construct $\langle k \Leftarrow \Box P\rangle$, called a *definition,* which associates a scripted process to the trigger name k. There is no reduction rule for definitions, which are analysed only in the labelled transition system (lts). Parallel compositions of processes and definitions are called *configurations K, L,* and together with contexts C are given by

$$K, L ::= K \mid K \mid (\nu c)K \mid P \mid \langle k \Leftarrow \Box P\rangle \quad C ::= - \mid K \mid C \mid (\nu c)C$$

where the set of values appearing in processes are extended to contain also triggers where scripts were allowed. We let underlined letters $\underline{u}, \underline{v}$ range over *first-order* values

$$\frac{F(v) = (v'; A; \tilde{k}) \quad F(\tilde{v}) = (\tilde{v}'; A'; \tilde{k}') \quad \tilde{k} \cap (\tilde{k}' \cup fn(v) \cup fn(\tilde{v})) = \emptyset \quad \tilde{k}' \cap (\tilde{k} \cup fn(v) \cup fn(\tilde{v})) = \emptyset}{F(v, \tilde{v}) = (v', \tilde{v}'; A \mid A'; \tilde{k}, \tilde{k}')}$$

$$\frac{F(U) = (U'; A; \tilde{k}) \quad F(\tilde{U}) = (\tilde{U}'; A'; \tilde{k}') \quad \tilde{k} \cap (\tilde{k}' \cup fn(U) \cup fn(\tilde{U})) = \emptyset \quad \tilde{k}' \cap (\tilde{k} \cup fn(U) \cup fn(\tilde{U})) = \emptyset}{F(U, \tilde{U}) = (U', \tilde{U}'; A \mid A'; \tilde{k}, \tilde{k}')}$$

$$F(c) = (c; 0; ())$$

$$F(k) = (k; 0; ())$$

$$F(l) = (l; 0; ())$$

$$F(p) = (p; 0; ())$$

$$F(@l{:}p) = (@l{:}p; 0; ())$$

$$F(0) = (0; 0; ())$$

$$F(\Box P) = (k; \langle k \Leftarrow \Box P \rangle; k)$$

$$\frac{F(T) = (T'; A; \tilde{k})}{F(\mathtt{a}[T]) = (\mathtt{a}[T']; A; \tilde{k})}$$

$$\frac{F(T_1) = (T_1'; A_1; \tilde{k}_1) \quad F(T_2) = (T_2'; A_2; \tilde{k}_2) \quad \tilde{k}_1 \cap (\tilde{k}_2 \cup fn(T_1) \cup fn(T_2)) = \emptyset \quad \tilde{k}_2 \cap (\tilde{k}_1 \cup fn(T_1) \cup fn(T_2)) = \emptyset}{F(T_1 \mid T_2) = (T_1' \mid T_2'; A_1 \mid A_2; \tilde{k}_1, \tilde{k}_2)}$$

Table 3. **The relation F. The important rule is the axiom replacing scripted processes with triggers and generating the corresponding definition. The inductive rules have additional conditions to avoid clashes of trigger names.**

(values not containing scripted processes), and we will omit the underlining when there is no ambiguity.

Structural congruence is extended to configurations in the obvious way. A configuration K is well-formed if its processes are well-formed, there is at most one $\langle k \Leftarrow \Box P \rangle$ for each k, and processes in definitions do not contain triggers. When an output or update transition takes place in the lts, we use a relation F to incorporate the triggers. F relates the potentially higher-order values $\tilde{v}$ with the triple $(\underline{\tilde{u}}; A; \tilde{k})$ consisting of the first order values $\underline{\tilde{u}}$, obtained by replacing each scripted process $\Box P$ in $\tilde{v}$ with a unique trigger k, the configuration A consisting of a parallel composition of definitions $\langle k \Leftarrow \Box P \rangle$, and the unique triggers $\tilde{k}$. The actual relation F is defined as a homomorphism on all terms, with $F(\Box P) = (k; \langle k \Leftarrow \Box P \rangle; k)$ for scripts (see Table 3).

Transition labels α_l are indexed with the location at which actions take place, and are defined by

$$\alpha_l ::= (\tilde{c}, \tilde{k})\overline{l{\cdot}c}\langle \underline{\tilde{v}} \rangle \mid l{\cdot}c(\underline{\tilde{v}}) \mid l{\cdot}\tau \mid (\tilde{k})l{\cdot}\mathsf{update}_p(\underline{\tilde{U}}, (\chi)\underline{\tilde{V}}) \mid l{\cdot}\mathsf{run}_p \mid l{\cdot}k(p)$$

Labels for input and output are standard, first-order labels. Label $l{\cdot}\tau$ denotes communication at l. The label for update contains a vector $\tilde{U}$ corresponding to the potential results of pattern-matching χ with values at path p in some tree (the range of Σ in the updating function) and treats $(\chi)\tilde{V}$ as an abstraction on the pattern variables (which are therefore subject to alpha-conversion). The vector $(\tilde{k})$ is used by the side conditions of the lts to enforce freshness of triggers, and binds the triggers $\tilde{k}$. Label run_p just records a run at p, and label $l{\cdot}k(p)$ signals that the script defined by k is selected for execution, with parameters l and p. Structural congruence extends to actions in the obvious way.

(COM) $\overline{l{\cdot}c}\langle\tilde{v}\rangle \mid l{\cdot}c(\tilde{x}).P \xrightarrow{l\cdot\tau} P\{\tilde{v}/\tilde{x}\}$

(COM!) $\overline{l{\cdot}c}\langle\tilde{v}\rangle \mid !l{\cdot}c(\tilde{x}).P \xrightarrow{l\cdot\tau} !l{\cdot}c(\tilde{x}).P \mid P\{\tilde{v}/\tilde{x}\}$

(UPDATE) $l{\cdot}\mathsf{update}_p(\chi, V).P \xrightarrow{(\tilde{k})l\cdot\mathsf{update}_p(\underline{\tilde{U}},(\chi)\underline{\tilde{V}'})} P\{\underline{U_1}/\chi\} \mid \cdots \mid P\{\underline{U_n}/\chi\} \mid A$
for any $\underline{\tilde{U}} = \{\underline{U_1}, \ldots, \underline{U_n}\}$ with $\tilde{k}$ fresh, and $F(V_1, \ldots, V_n) = (\underline{\tilde{V}'}; A; \tilde{k})$, where each $V_i = V$

(OUT) $\overline{l{\cdot}c}\langle\tilde{v}\rangle \xrightarrow{(\tilde{k})\overline{l{\cdot}c}\langle\underline{\tilde{u}}\rangle} A$ where $F(\tilde{v}) = (\underline{\tilde{u}}; A; \tilde{k})$ (IN) $0 \xrightarrow{l\cdot c(\underline{\tilde{v}})} \overline{l{\cdot}c}\langle\underline{\tilde{v}}\rangle$

(TRIGGER) $\langle k \Leftarrow \Box P\rangle \xrightarrow{l\cdot k(p)} \langle k \Leftarrow \Box P\rangle \mid P\{l/\circlearrowleft\}\{p/.\}$ (RUN) $l{\cdot}\mathsf{run}_p \xrightarrow{l\cdot\mathsf{run}_p} 0$

(RES) $\dfrac{K \xrightarrow{\alpha_l} K'}{(\nu c)K \xrightarrow{\alpha_l} (\nu c)K'}\ c \notin n(\alpha_l)$ (PAR) $\dfrac{K \xrightarrow{\alpha_l} K'}{K \mid L \xrightarrow{\alpha_l} K' \mid L}\ bn(\alpha_l) \cap fn(L) = \emptyset$

(STRUCT) $\dfrac{K \equiv L \xrightarrow{\alpha_l} L' \equiv K'}{K \xrightarrow{\alpha_l} K'}$ (OPEN) $\dfrac{K \xrightarrow{(\tilde{d},\tilde{k})\overline{l{\cdot}c}\langle\tilde{v}\rangle} K'}{(\nu b)K \xrightarrow{(b,\tilde{d},\tilde{k})\overline{l{\cdot}c}\langle\tilde{v}\rangle} K'}\ b \neq c,\ b \in fn(\tilde{v}) \setminus \tilde{d}$

Table 4. **Labelled Transition System for Core X*dπ*. The structural and communication rules are standard. The (OUT) rule uses relation F to replace scripts with triggers and produces a parallel composition of the associated definitions. The (IN) rule only allows first-order values, and the (UPDATE) rule can be regarded as a combination of input, output and communication.**

We explain now the rules for the lts; the formal definition is given in Table 4. Labelled transitions are defined for well-formed configurations. We have standard contextual and communication rules in the asynchronous style of [11], with the side conditions adapted to avoid clashes of trigger names. The rule for input and output are

$$(\text{IN})\quad 0 \xrightarrow{l\cdot c(\underline{\tilde{v}})} \overline{l{\cdot}c}\langle\underline{\tilde{v}}\rangle \qquad (\text{OUT})\quad \overline{l{\cdot}c}\langle\tilde{v}\rangle \xrightarrow{(\tilde{k})\overline{l{\cdot}c}\langle\underline{\tilde{u}}\rangle} A$$

where $F(\tilde{v}) = (\underline{\tilde{u}}; A; \tilde{k})$. Any scripted process in v is replaced by a trigger in $\underline{\tilde{u}}$, and A is the parallel composition of all the definitions associated with $\tilde{k}$. In an input transition, values must necessarily be first-order. The rule for updates is

$$(\text{UPDATE})\quad l{\cdot}\mathsf{update}_p(\chi, V).P \xrightarrow{(\tilde{k})l\cdot\mathsf{update}_p(\underline{\tilde{U}},(\chi)\underline{\tilde{V}'})} R \mid A$$

for any first-order vector $\underline{\tilde{U}} = \{\underline{U_1}, \cdots, \underline{U_n}\}$, $F(V_1, \ldots, V_n) = (\underline{\tilde{V}'}; A; \tilde{k})$ where each $V_i = V$, $\tilde{k}$ is fresh, and $R = P\{\underline{U_1}/\chi\} \mid \cdots \mid P\{\underline{U_n}/\chi\}$. These conditions are determined by viewing $\underline{\tilde{U}}$ as (first-order) parameters received in input, and $\underline{\tilde{V}}$ as parameters of a subsequent output. We conclude with the rules for running a script and analysing its definition:

$$(\text{RUN})\ l{\cdot}\mathsf{run}_p \xrightarrow{l\cdot\mathsf{run}_p} 0 \quad (\text{TRIGGER})\ \langle k \Leftarrow \Box P\rangle \xrightarrow{l\cdot k(p)} \langle k \Leftarrow \Box P\rangle \mid P\{l/\circlearrowleft\}\{p/.\}$$

The first rule simply records the location and path from which we run a script; the second one effectively executes a copy of a script, initialised with l and p.

Domain Bisimilarity. We introduce our bisimulation equivalence. The intuition is that when two processes are running in a domain Λ, we need to check that, if a process makes an action α_l with $l \in \Lambda$, then the other one can mimic it, possibly relying on the

existence of other locations in Λ. If $l \notin \Lambda$ we need not worry about matching actions. But since the domain can be extended by composing networks, we need to make sure that actions not in Λ are also matched, this time in a different relation parameterised by $\Lambda \cup \{l\}$.

We use the notation $K \xrightarrow{\tau}_\Lambda K'$ if $K \xrightarrow{l\cdot\tau} K'$ for some $l \in \Lambda$, and $\xrightarrow{\alpha_l}\!\!\!\!\rightarrow_\Lambda \triangleq \xrightarrow{\tau^*}_\Lambda \circ \xrightarrow{\alpha_l} \circ \xrightarrow{\tau^*}_\Lambda$ if $l \in \Lambda, \alpha_l \neq l\cdot\tau$, and $\xrightarrow{\tau}\!\!\!\!\rightarrow_\Lambda \triangleq \xrightarrow{\tau^*}_\Lambda$. The function $bn(-)$ extends to triggers in the obvious way. We say that an action α_l is relevant to a configuration K, abbreviated by $rel(\alpha_l, K)$, if $bn(\alpha_l) \cap fn(K) = \emptyset$.

DEFINITION 10 *A family of symmetric relations on configurations (indexed with sets of locations)* $\dot{\approx} = \{\dot{\approx}_\Lambda | \Lambda \subseteq \mathcal{L}\}$ *is a* domain bisimulation *if* $K \dot{\approx}_\Lambda L$ *and* $K \xrightarrow{\alpha_l} K'$ *implies:*

1. *if* $l \in \Lambda$ *with* $rel(\alpha_l, L)$ *then* $L \xrightarrow{\alpha'_l}\!\!\!\!\rightarrow_\Lambda L'$ *where* $\alpha'_l \equiv \alpha_l$ *and* $K' \dot{\approx}_\Lambda L'$;
2. *if* $l \notin \Lambda$ *then* $K \dot{\approx}_{\Lambda\cup\{l\}} L$.

Domain bisimilarity $(\approx)$ *is the pointwise largest domain bisimulation. Two open processes P, Q are* Λ*-bisimilar iff for all closing substitutions* σ, $P\sigma \approx_\Lambda Q\sigma$.

In the long version [15], we show that domain bisimilarity is defined as the largest fixpoint of a monotonic operator on families of relations. Showing that $K \approx_\Lambda L$ consists of exhibiting a domain bisimulation $\dot{\approx} = \{\dot{\approx}_\Delta | \Lambda \subseteq \Delta \subseteq \mathcal{L}\}$ such that $K \dot{\approx}_\Lambda L$. It is less burdensome than it may seem: the family is monotonic, and therefore starting from the pairs in $\dot{\approx}_\Lambda$, we can build each $\dot{\approx}_{\Delta\cup\{l\}}$ from $\dot{\approx}_\Delta$ adding only the pairs where the first component makes a move at l.

THEOREM 11 *For all* Λ, *(i) if* $\Lambda \subset \Lambda'$ *then* $\approx_\Lambda \subset \approx_{\Lambda'}$; *(ii)* $\approx_\Lambda$ *is a congruence on configurations, and the restriction of* $\approx_\Lambda$ *to processes is a congruence on processes.*

This theorem corresponds to Theorem 9, but point (ii) here is much harder to prove since the definition of $\approx_\Lambda$ does not require closure under contexts. The congruence property of $\approx_\Lambda$ plays a fundamental role in the theorem below, justifying the use of domain bisimilarity as a proof method for our process equivalences.

THEOREM 12 *Process bisimilarity is a sound approximation of process barbed congruence: for all* Λ, *if* $P \approx_\Lambda Q$ *then* $P \sim^b_\Lambda Q$.

The converse implication does not hold, as can be seen from $\mathtt{xch}(T, S) \sim^b_\Lambda \mathtt{xch}(S, T)$ and point (1) below. We leave to future work the study of complete characterisations of the contextual equivalences, which we believe could be based on a notion of weak bisimulation able to abstract away (partly) from update actions.

Examples. We start with an example of the proof method. We call the process $\mathtt{dE}(l\cdot a, m\cdot b) = !l\cdot a(\tilde{x}).\overline{m\cdot b}\langle\tilde{x}\rangle \mid !m\cdot b(\tilde{x}).\overline{l\cdot a}\langle\tilde{x}\rangle$ a *distributed equator.* It has the effect of making the use of channel a at l undistinguishable from the use of channel m at b, a key property to define optimisations for web services. Let $E_1 = \mathtt{dE}(l\cdot a, m\cdot b) \mid \overline{l\cdot a}\langle\tilde{v}\rangle$ and $E_2 = \mathtt{dE}(l\cdot a, m\cdot b) \mid \overline{m\cdot b}\langle\tilde{v}\rangle$. We show that $E_1 \approx_{\{l,m\}} E_2$. We need to give a domain bisimulation $\mathcal{R} = \{\mathcal{R}_\Delta\}_{\{l,m\}\subseteq\Delta}$ such that $\mathcal{R}_{\{l,m\}}$ contains the two processes. In

this case, it suffices to take the family where $\mathcal{R}_\Delta = \{(E_1, E_2), (E_2, E_1)\} \cup I$ for all Δ, where I is the identity relation. In fact, if $E_1 \xrightarrow{\alpha_l} E_1'$ then $E_2 \xrightarrow{m\cdot\tau} \xrightarrow{\alpha_l} E_1'$, and similarly for m. The case for α_n with $n \notin \{l, m\}$ is analogous.

Using domain bisimilarity, we can also prove the following results referring to examples discussed in Section 2 and Section 3:

1. for any Λ, if $S \not\approx^t T$ then $\mathtt{xch}(T,S) \not\approx_\Lambda \mathtt{xch}(S,T)$;
2. $\mathtt{oldS} \approx_\Lambda m\cdot\mathbf{newS}$ iff $m \in \Lambda$;
3. for any Λ, $!l\cdot a(\tilde{x}).\overline{l\cdot a}\langle\tilde{x}\rangle \approx_\Lambda 0$ and $l\cdot\mathbf{E}(a,b) \mid \overline{l\cdot c}\langle a\rangle \not\approx_\Lambda l\cdot\mathbf{E}(a,b) \mid \overline{l\cdot c}\langle b\rangle$;
4. $(\nu\, get)(m\cdot\mathbf{get} \mid l\cdot\mathtt{call}(m, get, p)) \approx_\Lambda (\nu\, get)(m\cdot\mathbf{get} \mid l\cdot\mathbf{spec})$ iff $m \in \Lambda$.

We conclude with an example on replication of web services. Consider the two services $\mathbf{s_1}$ and $\mathbf{s_2}$, meant to be interchangeable, defined as

$$\mathbf{s_1} = !m\cdot b(\tilde{x}, y, z).(\overline{n\cdot a}\langle\tilde{x}, y, z\rangle \oplus_m S) \qquad \mathbf{s_2} = !n\cdot a(\tilde{x}, y, z).(\overline{m\cdot b}\langle\tilde{x}, y, z\rangle \oplus_n S)$$

where $P \oplus_l Q = (\nu c)(\overline{l\cdot c}\langle\rangle \mid l\cdot c().P \mid l\cdot c().Q)$. Both offer the same service S, but an internal choice determines whether the service will be provided locally, or delegated to the other location. It does not matter if we paste in the data a service call to $\mathbf{s_1}$ or one to $\mathbf{s_2}$, as justified by the equation

$$\mathbf{s_1} \mid \mathbf{s_2} \mid l\cdot\mathbf{paste}_p\langle\mathbf{sc}[\,\Box l\cdot\mathtt{call}(m, b, \tilde{v})\,]\rangle \approx_{\{m,n\}} \mathbf{s_1} \mid \mathbf{s_2} \mid l\cdot\mathbf{paste}_p\langle\mathbf{sc}[\,\Box l\cdot\mathtt{call}(n, a, \tilde{v})\,]\rangle$$

5 Conclusions

We have compared alternative notions of behavioural equivalences for Core **Xdπ** networks, and we have derived corresponding notions of process equivalence which are useful to reason about web-related examples. We have defined a sound proof technique for these equivalences based on the notion of *domain bisimilarity*. Our work illustrates that a behavioural understanding of dynamic web data can be grounded on the existing techniques associated with process calculi, although the adaptation is by no means straightforward.

Acknowledgments. We thank Alex Ahern, Martin Berger, Cristiano Calcagno, Jonathan Hayman, Andrew Phillips, Iain Phillips, Maria Grazia Vigliotti, Nobuko Yoshida and Uri Zarfaty for useful comments and suggestions.

References

[1] Serge Abiteboul, Angela Bonifati, Grégory Cobena, Ioana Manolescu, and Tova Milo. Dynamic XML documents with distribution and replication. In *Proceedings of SIGMOD'03*, 2003.

[2] Abiteboul, S. et al. Active XML primer. INRIA, GEMO Report number 275.

[3] G. Bierman and P. Sewell. Iota: a concurrent XML scripting language with application to Home Area Networks. University of Cambridge Technical Report 557, jan 2003.

[4] Reinhard Braumandl, Markus Keidl, Alfons Kemper, Donald Kossmann, Alexander Kreutz, Stefan Seltzsam, and Konrad Stocker. Objectglobe: Ubiquitous query processing on the internet. To appear in the VLDB Journal: Special Issue on E-Services, 2002.

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

[6] Luca Cardelli and Giorgio Ghelli. A query language based on the ambient logic. In *Proceedings of ESOP'01,* volume 2028 of *LNCS,* pages 1–22. Springer, 2001.

[7] Philippa Gardner and Sergio Maffeis. Modeling dynamic Web data. In Georg Lausen and Dan Suciu, editors, *Proc. of DBPL'03.* LNCS, September 2003.

[8] Andrew Gordon and Riccardo Pucella. Validating a web service security abstraction by typing. In *Proceedings ofthe 2002 ACM Workshop on XML Security,* pages 18–29,2002.

[9] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *Proceedings of HLCL '98,* volume 16.3 of *ENTCS,* pages 3–17. Elsevier, 1998.

[10] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In *Proceedings of ECOOP,* volume 512 of *LNCS,* pages 133–147, Berlin, Heidelberg, New York, Tokyo, 1991. Springer-Verlag.

[11] K. Honda and M. Tokoro. On asynchronous communication semantics. *LNCS,* 612:21–51, 1992.

[12] Kohei Honda and Nobuko Yoshida. On reduction-based process semantics. *Theoretical Computer Science,* 151(2):437–486, 1995.

[13] Alan Jeffrey and Julian Rathke. Contextual equivalence for higher-order pi-calculus revisited. Computer Science Report 04/2002, University of Sussex, 2002.

[14] Alfons Kemper and Christian Wiesner. Hyperqueries: Dynamic distributed query processing on the internet. In *Proceedings of VLDB '01,* pages 551–560, 2001.

[15] Sergio Maffeis and Philippa Gardner. Behavioural equivalences for dynamic web data. Draft available as `http://www.doc.ic.ac.uk/~maffeis/corexdpilong.pdf`. Forthcoming Imperial College London Technical Report, 2004.

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

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

[18] Arnaud Sahuguet, Benjamin Pierce, and Val Tannen. Distributed Query Optimization: Can Mobile Agents Help? Unpublished draft.

[19] Arnaud Sahuguet and Val Tannen. Resource Sharing Through Query Process Migration. University of Pennsylvania Technical Report MS-CIS-01-10, 2001.

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

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