MIKADO Global Computing Project IST-2001-32222

Mobile Calculi Based on Domains

Flat Net Architectures for Global Computing Systems: Types and Behaviour

MIKADO Deliverable D2.1.0

Title :	Flat Net Architectures for Global Computing Systems: Types and Behaviour
Editor :	D. Gorla (Univ. of Florence)
Authors :	D. Gorla (Univ. of Florence)
Classification :	Deliverable D2.1.0, Public
Reference :	RR/WP2/2
Date :	January 2004

Abstract

This paper aims at collecting some of the foundational work on languages for global computing systems relying on a flat net architecture. The main case-study languages dealt through the paper are Dπ, $lsd\pi$ and μKLAIM, but also other two languages (namely Amadio's $\pi_{1\ell}$-calculus and Sewell's NOMADIC PICT) are also analysed. We compare their language design issues, their type theories and their behavioural theories. Moreover, by means of some informal encoding, we shall give an evidence on how strongly related they are. Finally, we shall comment on how the work presented here can influence the development of more complex theories dealing with hierarchical structured languages for global computing.

Contents

1 Introduction

Technological advances of both computers and telecommunication networks, and definition of more efficient communication protocols are leading to a ever-increasing integration of computing systems and to diffusion of so called *global computers*. Global computers share all the features of traditional distributed systems, like components distribution, concurrency, absence of a global state, asynchrony of changes of local states, and also have all the features of *open* systems, where new entities (usually called agents or processes) can dynamically enter and exit the system: heterogeneity, autonomous components, dynamic change of system configuration, mobility of programs and data. Global computer applications distinguish themselves from traditional distributed applications not only in terms of *scalability* (huge number of users and nodes), *connectivity* (both availability and bandwidth), *heterogeneity* (operating systems and application software) and *autonomy* (of administration domains having strong control of their resources), but particularly in terms of the ability of dealing with *dynamic* and *unpredictable* changes of their network environment (e.g. availability of network connectivity, lack of resources, node failures, network reconfigurations and so on).

A key research challenge is to identify what abstractions are more appropriate for programming global computers and to supply foundational and effective tools to support development and certification (establishing and proving correctness) of global computing applications. At a foundational level, several models and languages, presented as process calculi or strongly based on them, have been developed that have improved the formal understanding of the complex mechanisms underlying network awareness and code mobility. We mention the Ambient calculus [6], the Dπ–calculus [20], the DJoin calculus [10], KLAIM [8], the Kell calculus [35], and Nomadic Pict [36]. Their programming models encompass abstractions to represent the execution contexts of the net where applications roam and run, and mechanisms for coordinating and monitoring the use of resources, and for supporting the specification and the implementation of security policies.

This paper aims at collecting some of the work on languages for global computing systems relying on a flat net architecture. In these languages, processes (the computational entities) are allocated in different nodes of a net, where a node is a named place hosting computations and offering resources for such computations. A net is just a collection of nodes, i.e. no structure nor hierarchy between nodes is assumed. Moreover, it is usually assumed that each node is reachable from any other node of the net that owns its name.

These assumptions simplify the foundational work done for these languages. Indeed, the absence of predefined location patterns to execute an action in a net greatly simplifies the operational semantics and, thus, the underlying behavioural theories. This is evident, e.g., when comparing the bisimulation congruences developed for Ambient-like languages [26, 7, 27] with the corresponding theories developed for μKLAIM [15] and Dπ [17]. Also the typing systems developed for Ambient-like languages are more complicate then those developed, e.g., for Dπ: the latter ones extend π–calculus types [31] with the migration and channel creation capabilities.

Languages exploiting a hierarchical net architecture (e.g. Mobile Ambients [6], DJoin [10] and Kell [35]) are usually very well-suited for expressing logical relations between distributed entities. For example, developing a firewall application can be quite hard in flat languages, while is very simple in Ambient or in DJoin. Similarly, by exploiting a hierarchical language, it is very natural to express notions of resources availability and movement (see, e.g., the Calculus for Mobile Resources [13]). Finally, the notions of routing strategies and failures of subnets can be simply implemented by using the DJoin calculus. On the contrary, flat languages put the focus only on the distribution and mobility aspects; any logical notion has to be somehow encoded or explicitly introduced.

Thus, one may wonder whether studying global computing languages relying on a flat net architecture is interesting. The purpose of this paper is to give a positive answer to this question. The main thesis is that, because of the simple net model they assume, these languages can be used as a starting point for developing sophisticated theories. Indeed, the skeletal syntax and semantics that characterise these languages allows to focus the attention only on the theories under development, without the interference of annoying external details. Then, once the theories are set for the simpler setting, one can extend them to more complicated languages. Moreover, even if skeletal in their structure, these languages contain all the features of a global computing language: they have clear notions of domain and mobility, they support several forms of communication, they can be the basis for studying failures, connections/disconnections, distributed security policies, and so on.

The paper is organised as follows. We shall firstly analyse the interconnections between three case-study languages, namely Dπ [20], μKLAIM[1] [14] and $lsd\pi$ [33], from a design language point of view and we shall point out how strongly related they are. Then, we shall give insights on the foundational work done in these settings, pointing out several interesting and orthogonal theoretical contributions developed through the years. This part will also help in showing the commonalities and the differences between the languages. Moreover, it will point out the possible contributions that each language can give in the development of sophisticated theories for the new MIKADO calculus.

2 Language Design Issues

In this section, we shall describe the languages studied by firstly describing some of their main design issues. By also exploiting simple informal encodings and programming examples, we shall focus our attention on the communication and on the mobility primitives, since these are the main ingredient of any language for programming global computing applications. Then we shall give the formal syntax of the languages and some hints on their semantics. All the examples and the technicalities discussed below are just a selection of key ideas; we leave the papers cited in our exposition for more details. Moreover, the encodings are only informal and no property is claimed: an interesting future work could be the formalization of the ideas just sketched here.

2.1 Distribution Model and Mobility Primitives

The languages studied are related by the fact that the net is described as a flat collection of located processes. Processes are executed in the nodes where they are located and can change their allocation during their computation (this is the main effect of migration). Nodes are uniquely referred via their names. Both μKLAIM and Dπ/$lsd\pi$ structural equivalences (that relate nets that intuitively represent the same net up-to syntactic rearrangements) rely on the rule

$$l[\![P]\!] \mid l[\![Q]\!] \;\equiv\; l[\![P|Q]\!] \qquad\qquad \text{(or } l :: P \parallel l :: Q \;\equiv\; l :: P|Q \text{ in } \mu\text{KLAIM)}$$

This rule states that two processes located at the same node can be considered as a unique parallel process located at that node. In Dπ/$lsd\pi$ a located process is written $l[\![P]\!]$, while in μKLAIM it is written $l :: P$; they both denote a node named l hosting a process P. The operator '$|$' denotes the parallel composition of Dπ nodes/processes (and similarly for $lsd\pi$) and of μKLAIM processes, while '$\parallel$' denotes the parallel composition of μKLAIM nodes. This structural rule differentiate the case-study languages from several Ambient-like calculi, where two ambients with the same name a are two different resources that can be nondeterministically selected to engage a reduction with a process performing an action over a.

A node is a place for hosting resources and running processes. In Dπ/$lsd\pi$, a node is considered as the basic support for communication (since only co-located processes can communicate), while in μKLAIM a node is the unit for specifying access control policies (see Section 3.3). Node names are first class citizens

[1]μKLAIM is a process calculus at the core of KLAIM [8] and permits, w.r.t. KLAIM, a simpler and cleaner theory because of its minimal syntax (without higher-order communication, with only one kind of node addresses, without allocation environments – that map logical addresses to physical ones – and without parameterised process definitions). However, it retains all the basic features of the original language and so we think that studying μKLAIM is a good compromise between reality and simplicity.

both in Dπ and in μKLAIM: they can be dynamically created and passed through the computation. In *lsd*π, on the contrary, localities cannot be passed via communication (this is necessary to correctly manage the lexical scope discipline). A fresh node plays the role of a private resource since only the processes knowing its locality name can use the resources located at the node.

Nets are unstructured collections of nodes. Each node of a net can be reached by any process in the net, assuming that the process knows the locality associated to the node. Hence, when performing a remote operation (i.e. a migration, or also a communication primitive in μKLAIM), no explicit routing nor fixed positions in a certain hierarchy of the involved nodes are required. This is different (and simpler) from most of hierarchically designed languages, like, e.g., Mobile Ambients [6] and its variants, DJoin [10] and Kell [35].

Migration is *weak* in the three languages: indeed, the migrating processes are blocked until they arrive in the target node of their migration. Syntactically, the primitives for migration in Dπ and μKLAIM are, resp., $\mathbf{go}\,l.P$ and $\mathbf{eval}(P)@l.Q$. The **go** primitive moves the continuation process P in the specified locality l, while the **eval** primitive spawns for execution process P in l, while leaving the continuation process Q in the original locality. In *lsd*π, a corresponding primitive is not present; when referring to a remote channel like $c@h$, the continuation process is automatically moved in the locality h hosting the channel. Hence, in *lsd*π migration is *subjective* (since the migration is self-inflicted by the moving process by its action prefixing), while it is *objective* in Dπ and μKLAIM (since the spawned code is the the continuation of the **go** prefix or the argument of an **eval** action).

The mobility mechanisms described so far are quite similar. Indeed, the Dπ process $\mathbf{go}\,l.\_$ can be easily implemented in μKLAIM as the process $\mathbf{eval}(\_)@l.\mathbf{nil}$, while it is implemented in *lsd*π as

$$(\nu a@l)(a@l?().\_ \mid a@l!\langle\rangle) \qquad\qquad \text{for } a@l \text{ fresh for } \_$$

On the other hand, the μKLAIM migration primitive $\mathbf{eval}(\_{}_1)@l.\_{}_2$ is implemented in Dπ (and accordingly in *lsd*π) as the process

$$\mathbf{go}\,l.(\_{}_1 \mid \mathbf{go}\,k.\_{}_2)$$

where we assume that the starting locality is k.

2.2 Communication

There are two main paradigms for interprocess communication developed in the literature for concurrent systems: the *channel-based* paradigm [28, 29] and the *tuple space-based* paradigm [11]. In the first one, the communicating processes specify a channel name when synchronising; in the π–calculus [29] also some data are exchanged while synchronising. Hence, the communication is essentially synchronous and nominal, even if later versions of the π–calculus [21, 4, 2] rely on an asynchronous communication model (where the sending process is not blocked waiting to interact with a receiving process). The tuple spaces paradigm is based on the notion of *tuple* that is a sequence of information fields. Differently from the first paradigm, the tuple space mechanism is intrinsically asynchronous and anonymous. Indeed, the sender produces a datum in a tuple space (without synchronising with any receiving process) and the receiver looks for a tuple in the tuple space matching some specified pattern, called *template*, that is a sequence of both information fields (called actuals) and parameters (called formals): hence the datum is retrieved by only using pattern matching[2]. The tuple space paradigm was generalised in [12] to allow the use of several tuple spaces, thus increasing the flexibility of the resulting language.

These two paradigms have been extended to include explicitly named localities. Dπ and *lsd*π extend, resp., the π–calculus and the asynchronous π–calculus models by explicitly allocating processes and channels in a net and adding a primitive for programming code mobility. μKLAIM (and thus μKLAIM) extends the LINDA [11] coordination language in a similar way.

[2]A tuple matches a template whenever they have the same number of fields and corresponding fields match – a formal field matches any actual of the same type and two actual fields match only if identical.

Moreover, another programming feature differentiates the languages proposed: in Dπ and *lsd*π the communication is *local*, in that the sending and the receiving processes must be allocated in the same location (i.e., in the location hosting the communication channel used), while μKLAIM communication can take place between *remote* processes. Actually, these two programming choices are just a syntactic difference, since the run-time implementation of a remote communication is very similar in the languages mentioned. Indeed, when two remote Dπ/*lsd*π processes have to communicate, a migration (hence a synchronisation between the nodes hosting the involved entities) must occur. In μKLAIM a similar synchronisation occurs to produce/retrieve a datum remotely but is hidden in the primitives; thus, μKLAIM communication is somehow 'more higher-level' than Dπ/*lsd*π ones.

We claim that the different programming choices discussed so far can be encoded one in the other. In what follows we shall only give informal encodings; we leave as a future work the development of them into fully abstract (or at least operational correspondent) encodings. We start with the easier task: proving that the μKLAIM choice of having remote communications is just a way to abstract away from some implementing details. To this aim, we shall now show an implementation of a remote communication by using migration and local communication; the converse is trivial (indeed, a remote communication is local if the target locality and the executing locality do coincide). For example, let us consider a net where two remote processes P and Q, located resp. in l and k, wants to communicate by using another locality h as a medium (in Dπ/*lsd*π this means that P and Q will use some channel c located in h, while in μKLAIM this means that the processes will exploit h's tuple space). We call P_D/Q_D, P_l/Q_l and P_K/Q_K the implementations of P and Q in Dπ, *lsd*π and μKLAIM respectively. Their definition is the following

$$\begin{array}{lcllcllcl}
P_D & \triangleq & \mathbf{go}\, h.c!\langle V\rangle \mid P'_D & Q_D & \triangleq & \mathbf{go}\, h.c?(X).\mathbf{go}\, k.Q'_D & Q''_D & \triangleq & Q'_D[^V/_X] \\
P_l & \triangleq & c@h!\langle\tilde{v}\rangle \mid P'_l & Q_l & \triangleq & (\nu a)(a!\langle\rangle \mid c@h?(\tilde{x}).a?().Q'_l) & Q''_l & \triangleq & Q'_l[^{\tilde{v}}/_{\tilde{x}}] \\
P_K & \triangleq & \mathbf{out}(V)@h \mid P'_K & Q_K & \triangleq & \mathbf{in}(X)@h.Q'_K & Q''_K & \triangleq & Q'_K[^V/_X]
\end{array}$$

where $\_{}_1[^{\_2}/_{\_3}]$ denotes the (capture avoiding and pattern respecting) substitution of $\_{}_2$ for $\_{}_3$ in $\_{}_1$. In Dπ, $c!\langle V\rangle$ and $c?(X)$ denote the output of value V and the input of a value matching pattern X along channel c (and similarly for *lsd*π). In μKLAIM, the primitives $\mathbf{out}(V)@\cdot$ and $\mathbf{in}(X)@\cdot$ place a tuple containing the values V and removes a tuple matching pattern X in the (possibly remote) tuple space of $\cdot$. The semantics of Dπ/*lsd*π require that h is a locality name and hosts a channel c; this is checked by the type systems, that are indeed a fundamental part of the languages. In μKLAIM the semantics itself verifies the existence of locality h before firing the **out**, and the existence of a tuple matching template X in the tuple space of h before firing the **in** action. Hence, once verified these conditions, the execution of the nets implementing the protocol mentioned are

$$\begin{array}{lll}
\mathbf{D\pi:} & l[\![P_D]\!] \mid k[\![Q_D]\!] \mid h[\![R_D]\!] & \to l[\![P'_D]\!] \mid k[\![Q_D]\!] \mid h[\![R_D|c!\langle V\rangle]\!] \\
& & \to l[\![P'_D]\!] \mid h[\![R_D \mid c!\langle V\rangle \mid c?(X).\mathbf{go}\, k.Q'_D]\!] \\
& & \to l[\![P'_D]\!] \mid h[\![R_D \mid \mathbf{go}\, k.Q''_D]\!] \\
& & \to l[\![P'_D]\!] \mid k[\![Q''_D]\!] \mid h[\![R_D]\!] \\
\\
lsd\pi\mathbf{:} & l[\![P_l]\!] \mid k[\![Q_l]\!] \mid h[\![R_l]\!] & \to l[\![P'_l]\!] \mid k[\![Q_l]\!] \mid h[\![R_l|c!\langle\tilde{v}\rangle]\!] \\
& & \to l[\![P'_l]\!] \mid (\nu a@k)(k[\![a!\langle\rangle]\!] \mid h[\![R_l \mid c!\langle\tilde{v}\rangle \mid c?(\tilde{x}).a@k?().Q'_l]\!]) \\
& & \to l[\![P'_l]\!] \mid (\nu a@k)(k[\![a!\langle\rangle]\!] \mid h[\![R_l \mid a@k?().Q''_l]\!]) \\
& & \to l[\![P'_l]\!] \mid k[\![(\nu a)(a!\langle\rangle \mid a?().Q''_l)]\!] \mid h[\![R_l]\!] \\
& & \to l[\![P'_l]\!] \mid k[\![Q''_l]\!] \mid h[\![R_l]\!] \\
\\
\mu\text{KLAIM}\mathbf{:} & l :: P_K \parallel k :: Q_K \parallel h :: R_K & \succ\!\!\!\longrightarrow l :: P'_K \parallel k :: Q_K \parallel h :: R_K|\langle V\rangle \\
& & \succ\!\!\!\longrightarrow l :: P'_K \parallel k :: Q''_K \parallel h :: R_K
\end{array}$$

The Dπ and the *lsd*π versions differ in very few details, while retaining the idea that each remote action has to be properly implemented via migrations (or implicit migrations in *lsd*π). Indeed, *lsd*π is another way to give the semantics to Dπ; as we shall see when presenting formally the two calculi, the main difference is that, while Dπ is defined by straightforwardly extending the π–calculus with distribution and mobility, *lsd*π exploits non-standard notions of free/bound names and scoping of names; in this way, it removes a lot of well-formedness checks from the Dπ type systems and charges them into the syntax.

On the other hand, μKLAIM and Dπ are quite different in their design issues; however, a straightforward correspondence between the languages is possible. E.g., in the protocol just shown, the first Dπ reduction corresponds to the first μKLAIM reduction in that they both generate an output barb in h and require a synchronisation between the involved nodes, while the last three Dπ reductions are summarised by the last μKLAIM reduction. Indeed, the two **go** actions (the second and fourth Dπ reduction) are mimicked by the implicit synchronisation between Q_K and h, while the Dπ communication (third reduction) is the execution of the **in** action.

We are left with proving that the channel-based and the tuple-based paradigms are, again, just a syntactic sugar to express in different ways the same allowed communications. We start with the easy task: encoding Dπ channels in μKLAIM tuple spaces. The idea is simple: whenever a value V is transmitted on a channel a located at l, the tuple space of l will contain a tuple $\langle a, V\rangle$, i.e. a tuple whose first field is the name of the channel and the following fields are the components of value V. Thus, the Dπ network

$$l[\![\, a!\langle v_1,\ldots,v_n\rangle \,]\!] \;\mid\; l[\![\, a?(x_1,\ldots,x_n).P_D \,]\!]$$

will be translated in the μKLAIM net

$$l :: \mathbf{out}(a,v_1,\ldots,v_n)@l \;\mid\; \mathbf{in}(a,!x_1,\ldots,!x_n)@l.P_K$$

where P_D and P_K are, resp., the Dπ and the μKLAIM implementations of a continuation process P. The execution of the previously shown nets is

$$\begin{array}{rcl} l[\![\, a!\langle v_1,\ldots,v_n\rangle \,]\!] \mid l[\![\, a?(x_1,\ldots,x_n).P_D \,]\!] & \longrightarrow & l[\![\, P_D[^{v_1,\ldots,v_n}/_{x_1,\ldots,x_n}] \,]\!] \\ l :: \mathbf{out}(a,v_1,\ldots,v_n)@l \mid \mathbf{in}(a,!x_1,\ldots,!x_n)@l.P_K & \succ\!\!\longrightarrow & l :: \langle a,v_1,\ldots,v_n\rangle \mid \mathbf{in}(a,!x_1,\ldots,!x_n)@l.P_K \\ & \succ\!\!\longrightarrow & l :: P_K[^{v_1,\ldots,v_n}/_{x_1,\ldots,x_n}] \end{array}$$

The fact that the μKLAIM version requires two reductions is only related to the asynchrony of the calculus (while Dπ is built up on a synchronous communication infrastructure). Since Dπ only considers well-typed networks, we are ensured that the μKLAIM pattern matching will never fail and thus both the μKLAIM reductions will be performed.

Let us now encode the tuple-based paradigm by using Dπ channels. For the sake of simplicity, we shall encode the CKLAIM calculus (i.e. a version of μKLAIM where tuples are only one field long, can contain only locality names and where there is no **read** primitive – see μKLAIM syntax below for more details): this is not a restriction, since μKLAIM can be encoded (in a semantically equivalent way w.r.t. μKLAIM barbed bisimulation, see Definition 4.1) into this simpler calculus [15]. We can assume that each Dπ location has one channel for each type of transmissible values; e.g. the channel in locality l that can carry integer values is written $c^l_{\mathbf{int}}$, while the channel in k that can carry locations of type K is written $c^k_{\mathtt{loc}\{K\}}$. By exploiting this feature, $\mathbf{out}(v)@l.\_$, $\mathbf{in}(!x)@l.\_$ and $\mathbf{in}(v)@l.\_$ located in node k can be respectively encoded as

$$\mathbf{go}\, l.c^l_{type(v)}!\langle v\rangle.\mathbf{go}\, k.\_ \qquad\qquad \mathbf{go}\, l.c^l_{type(x)}?(x).\mathbf{go}\, k.\_$$

$$P \;\triangleq\; \mathbf{go}\, l.c^l_{type(v)}?(x).\mathbf{if}\ v = x\ \mathbf{then\ go}\, k.\_\ \ \mathbf{else}\ c^l_{type(v)}!\langle x\rangle.\mathbf{go}\, k.P$$

where we used process definition to model Dπ recursive behaviours (in this case, this can be easily modelled by exploiting replication). Obviously, in μKLAIM the value of a base value v can be always statically determined and also the value of a variable x can be statically inferred by examining how the continuation process uses the variable.

N	$::=$	$l :: P$	(single node)	P	$::=$	$\mathbf{nil}$	(null process)
	$\mid$	$N_1 \parallel N_2$	(net composition)		$\mid$	$\langle et \rangle$	(datum)
a	$::=$		(process actions)		$\mid$	$a.P$	(action prefixing)
		$\mathbf{in}(T)@\ell$			$\mid$	$P_1 \mid P_2$	(parallel composition)
	$\mid$	$\mathbf{read}(T)@\ell$			$\mid$	A	(process invocation)
	$\mid$	$\mathbf{out}(t)@\ell$		T	$::=$	$t \mid\ !x \mid\ !u \mid\ T_1, T_2$	(templates)
	$\mid$	$\mathbf{eval}(P)@\ell$		t	$::=$	$e \mid \ell \mid t_1, t_2$	(tuples)
	$\mid$	$\mathbf{newloc}(u)$		et	$::=$	$V \mid l \mid et_1, et_2$	(evaluated tuples)
				e	$::=$	$V \mid x \mid \ldots$	(expressions)

Table 1: μKLAIM Syntax

2.3 Syntax and Semantics of the Case-Study Languages

To conclude the presentation of our case-study languages, we shall give their formal syntax and the key rules regulating their semantics. The interested reader is referred to the cited full papers for more details.

The μKLAIM Process Calculus. We now give a more formal presentation of μKLAIM following [14]. The syntax of μKLAIM is reported in Table 1. As a matter of notation, we use l to range over *localities*, u over *locality variables*, ℓ over localities and locality variables, V over basic values and x over value variables.

The exact syntax of *expressions*, e, is deliberately not specified; we just assume that expressions contain, at least, basic values and variables. *Tuples*, t, are sequences of *actual fields*. These contain expressions, localities or locality variables. *Templates* T are patterns used to select tuples in a tuple space. They are sequences of actual and *formal fields*; the latter ones are used to bind variables to values.

Processes, the μKLAIM active computational units, can perform five different basic operations to retrieve/place tuples from/into a tuple space, send processes for execution on (possibly remote) nodes, and create new nodes. Notice that **newloc** is the only action not indexed with an address because it always acts locally; all the other actions explicitly indicate the (possibly remote) locality where they will take place. Processes can be either data or are built up from the special process **nil**, that does not perform any action, and from the basic operations by using action prefixing, parallel composition and process definition. Some well-formedness checks are in order to avoid processes like $a.\langle t \rangle$.

Nets are finite collections of nodes where processes and tuple spaces can be allocated. A *node* is a couple $l :: P$, where locality l is the address of the node and P is the (parallel) process located at l. The tuple space located at l is part of P because data are represented as processes.

μKLAIM semantics is given by exploiting a *structural congruence* $\equiv$, identifying only nets whose equality is immediately obvious from their syntactical structure, and a reduction relation $\succ\!\!\longrightarrow$. Net reductions are defined over configurations of the form $L \vdash N$, where L is a finite subset of localities containing all the localities named in N and it is needed to ensure global freshness of new addresses (rule (NEW)). For the sake of readability, when a reduction does not generate any fresh addresses we write $N \succ\!\!\longrightarrow N'$ instead of $L \vdash N \succ\!\!\longrightarrow L \vdash N'$. The rules for the basic actions are

$$(\textsc{Out}) \qquad \frac{et = \mathcal{E}[\![\, t \,]\!]}{l :: \mathbf{out}(t)@l'.P \parallel l' :: P' \succ\!\!\longrightarrow l :: P \parallel l' :: P' | \langle et \rangle}$$

$$(\textsc{Eval}) \qquad l :: \mathbf{eval}(Q)@l'.P \parallel l' :: P' \succ\!\!\longrightarrow l :: P \parallel l' :: P' | Q$$

$$(\textsc{New}) \qquad \frac{l' \notin L}{L \vdash l :: \mathbf{newloc}(u).P \succ\!\!\longrightarrow L \cup \{l'\} \vdash l :: P[{}^{l'}\!/u] \parallel l' :: \mathbf{nil}}$$

N ::=		(networks)	P ::=		(threads)
	$\mathbf{0}$	(empty)		**stop**	(termination)
$\mid$	$l[\![P]\!]$	(agent)	$\mid$	$\mathbf{go}\, l.P$	(movement)
$\mid$	$(\nu_l\, e : E)N$	(restriction)	$\mid$	$u!\langle U\rangle.P$	(output)
$\mid$	$N_1 \mid N_2$	(composition)	$\mid$	$u?(X : T).P$	(input)
			$\mid$	$(\nu e : E)P$	(restriction)
U ::=		(values)	$\mid$	$P_1 \mid P_2$	(composition)
	bv	(base value)	$\mid$	$*P$	(replication)
$\mid$	u	(identifier)	$\mid$	$\mathbf{if}\ U_1 = U_2\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2$	(matching)
$\mid$	$U@u$	(located value)	X ::=		(patterns)
$\mid$	$U_1,\ldots,U_n$	(tuple)		x	(variable)
			$\mid$	$X@x$	(located pattern)
			$\mid$	$X_1,\ldots,X_n$	(tuple)

Table 2: Dπ Syntax

$$(\textsc{In}) \qquad \frac{match(\mathcal{E}[\![\,T\,]\!], et) = \sigma}{l :: \mathbf{in}(T)@l'.P \parallel l' :: \langle et\rangle \ \succ\!\!\longrightarrow\ l ::\ P\sigma\ \parallel l' :: \mathbf{nil}}$$

$$(\textsc{Read}) \qquad \frac{match(\mathcal{E}[\![\,T\,]\!], et) = \sigma}{l :: \mathbf{read}(T)@l'.P \parallel l' :: \langle et\rangle \ \succ\!\!\longrightarrow\ l ::\ P\sigma\ \parallel l' :: \langle et\rangle}$$

Function $\mathcal{E}[\![\cdot]\!]$ evaluates (if possible) all the expressions occurring in $\cdot$ before producing a tuple in a tuple space (rule (Out)) or before using a template to retrieve a tuple (rules (In) and (Read)). Function $match(T,t)$ checks whether the tuple t matches against the template T (this holds whenever t and T have the same number of fields and corresponding fields match – a formal field matches any value of the same type and two actual fields match only if identical): in this case it returns the substitution of the actuals in t for the corresponding formals in T. The reduction relation is extended to complex nets in the expected way. Notice that each rule different from (New) can be executed only if the target locality is present in the net; in Dπ and *lsd*π this explicit control is hidden in the type system.

The Dπ-calculus. We now give the syntax of Dπ by following [20], as illustrated by Table 2. In defining the syntax, we presuppose the existence of a set of variables, ranged over by $x,y,z,\ldots$, and a set of names, ranged over by e. Names can be both locality names, ranged over by $h,k,l,\ldots$, and channel names, ranged over by $a,b,c,\ldots$. Names and variables are generally called identifiers and are ranged over by $u,v,w,\ldots$. Channels are located in nodes. When a channel is communicated within its home node, just its name is passed; but when a channel is sent/received outside its home node, a *located* value/pattern must be used. For example, the located value $a@l$ describes a (non-local) channel name a located in a (remote) locality l. Patterns deconstruct these values accordingly. This is necessary since Dπ communication is local; indeed, if the receiver of a channel $a@l$ wants to use a, it must firstly migrate in l. Hence, it is necessary to have information both on the channel name (to use it) and on its home location (to reach it).[3]

Both variables and names are typed by using atomic types ranged over by E, which may be either *channel types* A (expressing what kind of values the channel can transmit), *location types* K (expressing what kind of resources, i.e. channels and privileges, the node owns), or some predefined *basic types* BT. Variables may additionally be assigned one of the *compound types*, ranged over by T (that are obtained

[3]This is not necessary in *lsd*π since the compound value $a@l$ can be used without being decomposed; indeed, the operational semantics will automatically spawn at l the process naming $a@l$.

from the atomic types by using sequentialisation – i.e. $T_1,\ldots,T_n$ – and localisation – i.e. $A_1,\ldots,A_n@K$). Typing information is essential in the syntax of Dπ terms because a type system will be used to prevent from reducing ill-formed nets (see Section 3.1).

The main syntactic category is that of a system, consisting of a set of agents running independently in parallel. An agent is a located thread, where a thread is simply a term of the thread language (very similar to the synchronous polyadic π–calculus). Systems are combined using the static combinators of the π–calculus, namely parallel composition and restriction. The main difference between Dπ threads and π–calculus processes is the presence in the formers of the **go** primitive that allows a thread to move from one location to another.

Name restriction in a thread $(\nu e : E)P$ creates a new private name e of type E. If E is a channel type then e is considered a channel co-located with the thread, whereas if it is a location type then it is considered a new location. This name creation is handled in exactly the same way as name creation in the π–calculus, and similarly for the standard notion of free and bound occurrences of variables and names in systems and threads. Standard π–calculus notions of alpha-equivalence and substitution are also assumed; substitutions rely on a straightforward notion of pattern matching. No special provision is necessary for located values such as $a@l$: in substitutions these values are treated as simple tuples.

Dπ reduction relation, $\rightarrow$, is mainly characterised by the following rules

$$\begin{array}{llcll}
(\text{R-G}{\small\text{O}}) & l[\![\, \mathbf{go}\, k.P \,]\!] & \rightarrow & k[\![\, P \,]\!] & \\
(\text{R-C}{\small\text{OMM}}) & l[\![\, c?(X).P \,]\!] \mid l[\![\, c!\langle U\rangle.Q \,]\!] & \rightarrow & l[\![\, P[^{U}\!/X] \,]\!] \mid l[\![\, Q \,]\!] & \\
(\text{R-E}{\small\text{Q}}_1) & l[\![\, \mathbf{if}\ U = U\ \mathbf{then}\ P\ \mathbf{else}\ Q \,]\!] & \rightarrow & l[\![\, P \,]\!] & \\
(\text{R-E}{\small\text{Q}}_2) & l[\![\, \mathbf{if}\ U = V\ \mathbf{then}\ P\ \mathbf{else}\ Q \,]\!] & \rightarrow & l[\![\, Q \,]\!] & \text{if } U \neq V
\end{array}$$

The notion of substitution is generalised in an obvious way for sequences of values and structured values; however, for $[^{U}\!/X]$ to be well-defined, it must be that the structure of the pattern X matches the structure of the value U. Notice that no check on the existence of the target locality of a migration is explicitly done in the semantics of rule (R-GO). Since Dπ is a typed language, this check can be charged to the type system; hence, in well-typed nets, all migrations will use as target a name that indeed is a locality name in that net.

The *lsd*π-calculus. An important feature of Dπ is that new resources (i.e. channels) are uniquely located. Thus, for example, the Dπ net

$$s[\![\, (\nu a)(\mathbf{go}\, r.a?().P \mid a?().Q) \,]\!]$$

is not well-typed since, after one execution step, it reduces to

$$(\nu_s a)(s[\![\, a?().Q) \,]\!] \mid r[\![\, a?().P) \,]\!])$$

that violates the uniqueness of a's allocation in s; hence, the Dπ type system will reject it by statically discovering the problem. However, because of such requirements on names, writing correct Dπ programs can become complicated and has to exploit several times the use of the type system. To fix the problem, *lsd*π adopts a *lexically scoped name discipline* to avoid confusion between local and remote names: by inspecting the name of the channel and its site of occurrence, the physical location of a channel can be known at all times.

The syntax of *lsd*π, as reported in Table 3 following [33], is very close to Dπ syntax and relies on a set of sites, ranged over by $s,r,t,\ldots$, and a set of simple channel names, ranged over by $a,b,c,x,\ldots$. The two main differences w.r.t. Dπ are the values passed in the communication (in *lsd*π only channels are passed, while in Dπ also localities are trasmissible) and the use of names. The name discipline characterizing *lsd*π impose that located channels $a@s$ belong to the site where they are (*explicitly*) located, s; simple channels a are (*implicitly*) located at the current site, the site where the process in which they occur is located. This implies

Channels:	u	$::=$	a	$\mid$	$a@s$							
Globals:	g	$::=$			$a@s$	$\mid$	s					
Names:	n	$::=$	a	$\mid$	$a@s$	$\mid$	s					
Processes:	P	$::=$	$\mathbf{0}$	$\mid$	$P_1 \mid P_2$	$\mid$	$(\nu n)P$	$\mid$	$u!\langle\tilde{u}\rangle$	$\mid$	$u?(\tilde{x})P$	$\mid \; u?*(\tilde{x})P$
Networks:	N	$::=$	$\mathbf{0}$	$\mid$	$N_1 \mid N_2$	$\mid$	$(\nu g)N$	$\mid$	$s[\![P]\!]$			

Table 3: *lsdπ* Syntax

that, when migrating process P from site r to site s, the free channels of P must be renamed accordingly: thus simple channels become explicitly located at r; channels located at s become simple (dropping '$@s$'); all the others remain as they were. However, lexical scoping together with compound names introduce subtleties in the definition of free and bound names and hence can cause problems when applying alpha-conversion, name instantiation (as a consequence of a communication) or name translation (as a consequence of migrations, as explained above). In particular, non standard notions of free/bound names and substitutions[4] are needed to ensure that, when applying substitutions to processes and nets, the correspondence between the binders therein and the corresponding bound names does not change.

The structural congruence is now charged with the burden of properly handling compound names. In particular, some non standard rules for modifying the scope of bound names and some rules for converting compound channels into simple ones are in order. Some of them are listed below, as a simplified variant of those reported in [33].

$$\begin{array}{llcll}
(\text{S-S}{\scriptstyle\text{COP}}) & ((\nu a)P) \mid Q & \equiv & (\nu a)(P|Q) & \text{if } \{a, a@\_\} \cap fn(Q) = \emptyset \\
(\text{S}{\scriptstyle\text{N}}\text{-S}{\scriptstyle\text{CO}}\text{S}_1) & (\nu g)s[\![P]\!] & \equiv & s[\![(\nu g)P]\!] & \text{if } g \notin \{s, \_@s\} \\
(\text{S}{\scriptstyle\text{N}}\text{-S}{\scriptstyle\text{CO}}\text{S}_2) & (\nu a@s)s[\![P]\!] & \equiv & s[\![(\nu a@s)P]\!] & \text{if } a \notin fn(P) \\
(\text{S}{\scriptstyle\text{N}}\text{-S}{\scriptstyle\text{CO}}\text{S}_3) & (\nu a@s)s[\![P]\!] & \equiv & s[\![(\nu a)P]\!] & \text{if } a@s \notin fn(P) \\
(\text{S}{\scriptstyle\text{N}}\text{-M}{\scriptstyle\text{IG}}\text{O}) & s[\![a@s!\langle\tilde{u}\rangle]\!] & \equiv & s[\![a!\langle\tilde{u}\rangle]\!] & \\
(\text{S}{\scriptstyle\text{N}}\text{-M}{\scriptstyle\text{IG}}\text{I}) & s[\![a@s?(\tilde{x}).P]\!] & \equiv & s[\![a?(\tilde{u}).P]\!] &
\end{array}$$

The reduction semantics then relies on the rules

$$\begin{array}{llcll}
(\text{R}{\scriptstyle\text{P}}\text{-C}{\scriptstyle\text{OMM}}) & a!\langle\tilde{u}\rangle \mid a?(\tilde{x}).P & \to & P[^{\tilde{u}}/_{\tilde{x}}] & \\
(\text{R}{\scriptstyle\text{N}}\text{-M}{\scriptstyle\text{IG}}\text{O}) & r[\![a@s!\langle\tilde{u}\rangle]\!] & \to & s[\![(a@s!\langle\tilde{u}\rangle)\sigma_r]\!] & \text{if } r \neq s \\
(\text{R}{\scriptstyle\text{N}}\text{-M}{\scriptstyle\text{IG}}\text{I}) & r[\![a@s?(\tilde{u}).P]\!] & \to & s[\![(a@s?(\tilde{u}).P)\sigma_r]\!] & \text{if } r \neq s
\end{array}$$

where $P\sigma_r \triangleq P[a_1@r/a_1, \ldots, a_n@r/a_n]$ and $\{a_1, \ldots, a_n\}$ are the free simple channel names in P.

3 Type Systems

In this section we shall give an overview of the different works on types done for the case-study languages. However, while types are a crucial element for defining a correct semantics of Dπ and *lsdπ* nets (even if in this second case their relevance is reduced), μKLAIM types are only used for the purpose of controlling nets

[4] Their definition is quite elaborate and is left for the interested reader [33]. To give an idea of the scoping problems in *lsdπ*, notice that $(\nu a@s)\_$ binds all the occurrences of a in $\_$ (either simple or composed with *any* locality), while $(\nu s)\_$ only binds s. These problems are present in Dπ too, but there they are solved by complicating the typing procedure.

behaviour. Indeed, as we mentioned in Section 2.3, Dπ/*lsd*π semantics strongly rely on preliminary type controls that forbids ill-formed nets (where, e.g., a process attempts to migrate over a not existing node) from being executed.

However, a part from syntax related checks, all the type systems for the languages studied in this paper are unified by the purpose of *controlling resource access*. Types should mainly monitor the use of *communication media* and *migrations*. The main differences among the various type systems is the way in which the typing information is exploited (only at compile time or, partially, also at run time) and the way in which it is stored in the system (centralised in some omniscient authority, split in several disjoint parts and assigned to different locations, partially split and partially shared between locations). In particular, we would like to remark some crucial general points.

- A totally static type checked language is surely interesting from a theoretical point of view, but is quite unusable in practise. Indeed, the net is usually too large to allow a preliminary type checking of all its nodes or, even worst, not all nodes want to be type checked (e.g. malicious nodes hosting viruses or misbehaving processes). Hence, a certain amount of run time overhead is necessary if we want to save the expressive power of the languages (indeed, one can easily imagine very strict syntactic rules that, even if protecting a net from misbehaviours, also reject legal nets). However, global type checking can be used as a tool to ensure partial type checking systems work correctly.

- The presence of a unique typing context (an omniscient authority) allows for a greater number of static checks but it is quite unrealistic especially for WANs, where different administrators are responsible for the assignment of different policies. Thus, for the sake of realism, the typing information must be somehow split between the domains of the calculi (i.e. the nodes of the net). Again, maintaining some shared information simplifies and makes more efficient the type checking, but it is not always a possible assumption.

Because of their different programming features, the types for the languages considered have different semantics. In particular

- the type of a μKLAIM node controls the activity of processes running in the node in terms of the legality of their actions (downloading/consuming tuples, producing tuples, activating processes and creating new nodes) that can be performed *over* other nodes of the net;

- Dπ location types control process migrations, the use of channels and their creation that can take place *in* the node itself;

- similarly, *lsd*π types control the correct usage of channels.

Moreover, μKLAIM and Dπ (and *lsd*π) also adopt different typing choices: μKLAIM opts for a totally partitioned and distributed type context[5], while Dπ [20] accepts the presence of an omniscient authority. Thus, the original type system for Dπ is only exploited at compile time: once a net has been successfully type checked, it can be executed without any further runtime check. This is not the case in μKLAIM where two dynamic checks are unavoidable:

1. when spawning a process Q from node l over node k, a preliminary type check of Q against k's policy must be performed. Indeed, since l and k can be under the control of different net administrators, no compile time information about k's policy is available in l. Thus, the static inference will always accept actions like $\mathbf{eval}(Q)@k$, once ensured that l can spawn code over k (but this information is locally available in l). Notice two things:

[5]This is the main difference between the type systems developed for μKLAIM [14, 16] and KLAIM [9]: in KLAIM, a global typing knowledge was assumed and hence the run time overhead was minimised.

- in [9], where a global type environment was available, this run time check was not present, also because of a more complex types structure;
- in [19, 34], where more realistic and elaborated scenarios are taken into account, this run time check appears also in Dπ semantics.

2. when retrieving a tuple from a tuple space via a **in**/**read** action, it must be verified that the tuple accessed matches against the template specified and that the continuation process is still well-typed once the formals of the template occurring in it are replaced with the corresponding actuals of the accessed tuple. This kind of checks are avoided in Dπ: indeed, the type environment assigns a type to the channel used in the communication and the static inference verifies that both the pattern and the value involved in the communication respect this type. In this way, it is ensured the typing compliance of the pattern specified against the value received in the communication, and also that the continuation process behaves properly w.r.t. the values received in the communication. A similar solution cannot be used in μKLAIM because it would somehow require the assignment of a type to a tuple space (the communication medium corresponding to Dπ channels); but this seems an intrinsic violation of the nature of tuple spaces that are repositories of tuple *of different kind*.

We now give some details on the type theory developed for the case-study languages, by also sketching some refined variants of the basic theory. The principles mentioned above are instantiated to the particular features of the languages, by controlling resource creation, process migration and resource access via *behavioural types*. A behavioural type is like a standard type in an imperative or object oriented language, but it describes the behaviour of the typed entities by mentioning the resources an entity can offer/use and what kind of operations are legal over these resources. Like standard types, behavioural types rely on a notion of subtyping that allows for an entity of a certain type to occur wherever an entity of a supertype can occur.

3.1 Types for Dπ

The foundational work of Dπ types is [20]. Types are used to express the capabilities a node can offer to the hosted processes, namely which kind of operations a process running in the node can execute. To this aim, the basic typing entities in Dπ are *capabilities*, as defined by the following productions:

$$\begin{array}{lrcl} \text{Locality Capabilities:} & \kappa & ::= & \mathbf{move} \;\big|\; \mathbf{newc} \;\big|\; a : A \\ \text{Channel Capabilities:} & \alpha & ::= & \mathtt{r}\langle T\rangle \;\big|\; \mathtt{w}\langle T\rangle \end{array}$$

Intuitively, if a locality offers a **move** capability, then it accepts incoming agents; if it offers a **newc** capability, then it allows the creation of new local channels; if it offers a $a : A$ capability, then it offers a channel named a whose type is A. Similarly, if a channel is declared with a $\mathtt{r}\langle T\rangle$ capability, then it can be used to read values whose type is at most T, while the $\mathtt{w}\langle T\rangle$ capability enables the emission over that channel of values whose type is at least T.

Types are built over capabilities. Locality types are collections of locality capabilities; channel types are collections of channel capabilities; transmission types can be either base types, location types and channel types, or can be built up by using sequentialisation and localisation. They are formally defined as the terms generated by the grammar

$$\begin{array}{lrcl} \text{Locality Types:} & K & ::= & \mathtt{loc}\{\widetilde{\kappa}\} \\ \text{Channel Types:} & A & ::= & \mathtt{res}\{\widetilde{\alpha}\} \\ \text{Transmission Types:} & T & ::= & BT \;\big|\; K \;\big|\; A \;\big|\; T_1, \ldots, T_n \;\big|\; A_1, \ldots, A_n@K \end{array}$$

together with some consistency constraints. The latter ones essentially requires that

- locality types are allowed at most one capability for each channel name

- channel types are constrained to have at most one read capability and one write capability; moreover, it is also required that the type of the values sendable on the channel is at least the type of the values receivable from the channel (this requirement is necessary to handle subtyping without breaking subject reduction).

Dπ types come equipped with a subtyping relation formalising the ordering notions mentioned above. The subtyping relation, <:, relies on the following rules

$$\frac{A <: A'}{a : A <: a : A'} \qquad \frac{T <: T'}{\mathtt{r}\langle T\rangle <: \mathtt{r}\langle T'\rangle} \qquad \frac{T' <: T}{\mathtt{w}\langle T\rangle <: \mathtt{w}\langle T'\rangle}$$

$$\frac{\forall\, \kappa' \in K'\ \exists\, \kappa \in K\ :\ \kappa <: \kappa'}{K <: K'} \qquad \frac{\forall\, \alpha' \in A'\ \exists\, \alpha \in A\ :\ \alpha <: \alpha'}{A <: A'}$$

where the relation is homomorphically extended to sequences (and thus to localised types $A_1,\ldots,A_n@K$ that are considered as the sequence $A_1,\ldots,A_n,K$). Intuitively, the subtype relation can be thought of as the reverse subset inclusion between sets of capabilities. Thus, $K <: K'$ if each capability in K' can be matched by a lower or equal (w.r.t. <:) capability in K.

Nets are typed with respect to a *type environment* Γ, whose purpose is to provide a type for all of the free identifiers in the net. Indeed, since the type system is static, it must be defined over open terms; hence type environments must provide types for variables and not only for names. Moreover, it is *global* in that it is assumed to be unique for all the net to be typed; it collects together all the information about the resources in the net and their allocation. Formally, Γ is a finite mapping from identifiers to open location types, where the latter ones have the form $\mathtt{loc}\{\widetilde{u : T}\}$. The subtyping relation is extended pointwise to environments; thus we let $\Gamma <: \Gamma'$ if and only if, for any $w \in dom(\Gamma')$, it holds that $\Gamma(w) <: \Gamma'(w)$.

There are three kind of judgements for Dπ terms, one for values, $\Gamma \vdash_w U : T$, one for threads, $\Gamma \vdash_w P$, and one for networks, $\Gamma \vdash N$. Intuitively, $\Gamma \vdash N$ states that the network N is well-typed w.r.t. Γ. This judgement relies on $\Gamma \vdash_w P$, stating that the thread P is well-typed to run at location w, which in turn uses judgements of the form $\Gamma \vdash_w U : T$ indicating that U is a well-formed value at w and has at least the capabilities specified by T. The keys rules for inferring these judgements are

Values:

$$\frac{\Gamma(u) <: K}{\Gamma \vdash_w u : K} \qquad \frac{\Gamma(w) <: \mathtt{loc}\{u : T\}}{\Gamma \vdash_w u : T} \qquad \frac{\Gamma \vdash_w u : K \qquad \Gamma \vdash_u U : T}{\Gamma \vdash_w U@u : T@K}$$

Threads:

$$\frac{\Gamma \vdash_w u : \mathtt{res}\{\mathtt{w}\langle T\rangle\}\ ,\ U : T\ ,\ P}{\Gamma \vdash_w u!\langle U\rangle.P} \qquad \frac{\Gamma \vdash_w u : \mathtt{res}\{\mathtt{r}\langle T\rangle\} \qquad \Gamma,{}_w X : T \vdash_w P}{\Gamma \vdash_w u?(X : T).P}$$

$$\frac{\Gamma \vdash_w u : \mathtt{loc}\{\mathbf{move}\} \qquad \Gamma \vdash_u P}{\Gamma \vdash_w \mathbf{go}\, u.P} \qquad \frac{\Gamma \vdash_w w : \mathtt{loc}\{\mathbf{newc}\} \qquad \Gamma,{}_w a : A \vdash_w P}{\Gamma \vdash_w (\nu a : A)P}$$

Networks:

$$\frac{\Gamma \vdash_w w : \mathtt{loc}\{\} \qquad \Gamma \vdash_w P}{\Gamma \vdash w[\![P]\!]}$$

Notation $\Gamma,{}_w u : T$ augments the type of w in Γ with the new capability $u : T$; to be defined, w must already be in the domain of Γ and u must be new to $\Gamma(w)$.

Notice that each migration is controlled to ensure that the target locality is indeed a node address and that it accepts incoming agents; in this case the continuation process is typed in the target locality. Similarly, each channel creation is allowed only if the enabling **newc** privilege is offered by the executing node (on the other hand, node creations are always enabled – the rule is not shown). Finally, the rules for prefixes verifies that the channel specified for communicating is used accordingly to the capabilities offered by the locality executing the action. In particular, for an output action $u!\langle U\rangle$ it is verified that U can be assigned a type that is at most the write capability T of channel u; this means that $type(U) <: T$. Similarly, for an input action $u?(X : T')$ it is verified that T' is at least the read capability T'' of channel u; this means that $T'' <: T'$. By using the consistency constraint mentioned when defining channel types, we have that

$$type(U) <: T <: T'' <: T'$$

This ensures that the capabilities required by a receiving process when performing an input from a certain channel will always be satisfied by any process performing an output over the same channel. This fact turns out to be essential when defining type safety (see later) and allows for a way of passing different privileges over the same resources to different threads.

A crucial property of each type system is *subject reduction*, stating that well-typedness is an invariant of the reduction relation. In Dπ this amounts to prove the following

Theorem 3.1 *If* $\Gamma \vdash N$ *and* $N \to N'$ *then* $\Gamma \vdash N'$.

Well-typedness is a sensible semantics notion, since it ensures that well-typed nets never execute during their computation an action that violates the capabilities owned by the node performing it. This property is referred to as *type safety*. To formally express it, the language must be enriched with permissions; indeed, each thread is tagged with the capabilities it has accumulated during its computation. A *runtime error* occurs if a thread attempts to use a name contrary to the limitations imposed by these explicit capabilities. Thus, an agent in the tagged language is a thread located in some node, together with a closed type environment Δ (i.e. a type environment containing no variables) which represents the capabilities of the agent. The main changes affect the definition of Dπ networks, the structural laws for extracting a restricted name from an agent, the reduction rule for communication and the typing rule for agents. The revised versions are

$$N \quad ::= \quad \mathbf{0} \;\big|\; l[\![P]\!]_\Delta \;\big|\; (\nu_l\, e : E)N \;\big|\; N_1 \parallel N_2$$

$$w[\![(\nu a : A)P]\!]_\Delta \quad \equiv \quad (\nu_w\, a : A)\; w[\![P]\!]_{\Delta,\; {}_w a:A}$$

$$w[\![(\nu v : K)P]\!]_\Delta \quad \equiv \quad (\nu v : K)\; w[\![P]\!]_{\Delta, v:K} \qquad \text{if } v \neq w$$

$$w[\![c!\langle U\rangle.Q]\!]_\Delta \mid w[\![c?(X : T).P]\!]_{\Delta'} \quad \mapsto \quad w[\![Q]\!]_\Delta \mid w[\![P[{}^U\!/X]]\!]_{\Delta' \sqcap \{_w U:T\}}$$

$$\frac{\Delta \vdash_w w : \mathtt{loc}\{\} \qquad \Delta \vdash_w P}{\Gamma \mid\!\vdash w[\![P]\!]_\Delta}\; \Gamma <: \Delta$$

where $\mapsto$ and $\mid\!\vdash$ denotes, respectively, the tagged reduction relation and type judgement, while $\_{}_1 \sqcap \_{}_2$ denotes the greatest (w.r.t. $<:$) common subtype of both $\_{}_1$ and $\_{}_2$. In the new type system, we can infer $\Gamma \mid\!\vdash w[\![P]\!]_\Delta$ only if the agent tags Δ are consistent with the global types specified in Γ and the thread P uses the resources in the system only as it is allowed by its tags.

A runtime error occurs whenever a process attempts to perform an action whose enabling capability is not contained in its tags. Formally, the runtime error predicate, $\uparrow$, is defined in terms of the following rules

$$\begin{array}{rcl} l[\![\, \mathbf{go}\, k.P \,]\!]_\Delta & \uparrow & \text{if } \Delta(k) \not<: \texttt{loc}\{\mathbf{move}\} \\ l[\![\, (\nu a).P \,]\!]_\Delta & \uparrow & \text{if } \Delta(l) \not<: \texttt{loc}\{\mathbf{newc}\} \\ l[\![\, a!\langle U\rangle.P \,]\!]_\Delta & \uparrow & \text{if } \Delta_l(U) \not<: wobj(\Delta(l,a)) \\ l[\![\, a?(X:T).P \,]\!]_\Delta & \uparrow & \text{if } robj(\Delta(l,a)) \not<: T \\ l[\![\, a!\langle U\rangle.P \,]\!]_\Delta \mid l[\![\, a?(X:T).Q \,]\!]_{\Delta'} & \uparrow & \text{if } wobj(\Delta(l,a)) \not<: robj(\Delta'(l,a)) \end{array}$$

where $\not<:$ denotes the negation of $<:$, $robj(\_)$ and $wobj(\_)$ denote the types of the objects that may be read or written on the channel $\_$ and $\Delta_l(U)$ denotes the least type, if any, which the type environment Δ can assign to the value U at l. By letting $\not\uparrow$ to denote the negation of predicate $\uparrow$, the type safety theorem can be then formulated as follows

Theorem 3.2 *If $\Gamma \mid\!\vdash N$ then $N \not\uparrow$.*

In [20] very strong connections between $\rightarrow$ and $\mapsto$, and between $\vdash$ and $\mid\!\vdash$, are proved. Thus, by using these results and Theorems 3.1 and 3.2, and by letting $\rightarrow^*$ to denote the reflexive and transitive closure of relation $\rightarrow$, it can be proved that

Theorem 3.3 (Type Soundness) *If $\Gamma \vdash N$ and $N \rightarrow^* N'$ then $N'' \not\uparrow$, where N'' is the tagged version of N'.*

The type system presented so far, relies on the typing of the whole net to establish type soundness. While this assumption is reasonable in LANs (where the number of host is usually small and a central authority governs the net), it is hardly acceptable in WANs. Thus, in [19] an improved type system deals with anonymous networks, i.e. nets where not all agents are assumed well-typed. To prevent bad agents to come in a node k and misuse its resources, a runtime check of the incoming threads will be done against a location type associated to k, collecting the capabilities available locally in k: if the agent behaves conformly to them, then it is accepted and put in execution, otherwise it is refused. Such location types are collected in a closed type environment Δ (i.e. a type environment without variables), which forms with the referred net N a configuration $\Delta \triangleright N$.

Now, the reduction relation takes the form $\Delta \triangleright N \rightarrow N'$ and its main characteristic is in the rule for code movement:

$$\Delta \triangleright l[\![\, \mathbf{go}\, k.P \,]\!] \rightarrow k[\![\, P \,]\!] \qquad \text{if } \Delta(k) \vdash' P$$

The judgement $\Delta(k) \vdash' P$ verifies whether P respects the local type $\Delta(k)$ which gives the names and types of the resources available at k. Its formal definition is similar to the static judgement presented before; the main difference is that now it take into account only the use of resources local to k. Thus, for example,

$$L \vdash' u_1,\ldots,u_n@u : A_1,\ldots,A_n@K \qquad L \vdash' \mathbf{go}\, l.P \qquad \frac{L \vdash' u : \texttt{res}\{T\} \qquad L, X:T \vdash' P}{L \vdash' u?(X:T).P}$$

where L is a open locality type (i.e. a locality type whose domain can include channel variables) and $L, X : T$ extends L only with non-located variables in X. The purpose of this revised judgement is to take into account only the behaviour of P that directly involves local resources of the node; thus a migration or the use of located values are not taken in consideration (they are always assumed well-typed).

The main novelty of the type system is that not all the net is well-typed (w.r.t. a type environment Γ) but only the ‘good’ part of the net (i.e. the nodes that occur in the domain of Γ). Thus, the new static inference relies on the following rules:

$$\frac{l \in dom(\Gamma) \quad \Gamma(l) \vdash' P}{\Gamma \vdash' l[\![\, P \,]\!]} \qquad \frac{l \notin dom(\Gamma)}{\Gamma \vdash' l[\![\, P \,]\!]} \qquad \frac{\Gamma \vdash' N \quad \forall l \in dom(\Gamma)\,.\,\Gamma(l) <: \Delta(l)}{\Gamma \vdash' \Delta \triangleright N}$$

Notice that only the good sites are typed (w.r.t. their local types $\Gamma(l)$). Moreover, a configuration $\Delta \triangleright N$ is legal only if the locality types in Δ associated to the good sites are consistent with the global type environment Γ (i.e. they cannot assume more capability than those effectively present in Γ, but they can safely offer less capabilities). Type soundness can be finally stated accordingly (where a predicate $\uparrow_l$ is defined like $\uparrow$ but also reveals the node where the error happened).

Theorem 3.4 *If $\Gamma \vdash' \Delta \triangleright N$ and $\Delta \triangleright N \rightarrow^* N'$ then $N' \not\uparrow_l$ for any $l \in dom(\Gamma)$.*

In [34] the work of [19] is improved in two ways. First of all, the locality knowledge Δ over which type checking incoming agents is split into several *partial views* of the global knowledge Γ; in this new setting, different localities can have different (but not conflicting) knowledge of the net. Then, the burden of type checking all the incoming agents is lightened by adding a notion of *trust* between sites: an incoming thread is type checked only if it comes from untrusted nodes.

To allow partial typing w.r.t. partial views of the nets, the syntax of Dπ networks is extended to include *filters* as follows

$$N ::= \ \ldots \ \big| \ \ k\langle\langle \Delta \rangle\rangle$$

The filter for k, $k\langle\langle \Delta \rangle\rangle$, contains a (closed) type environment Δ which gives k's view of the resources in the network. Each location comes equipped with exactly one filter. Filters of good localities must respect the global knowledge of the system; this is included in the new type judgement $\vdash''$, that is defined like $\vdash$ but relies on the rules

$$\frac{\Gamma(k) = \texttt{lbad}}{\Gamma \vdash'' k[\![P]\!]} \qquad \frac{\Gamma(k) \neq \texttt{lbad} \qquad \Gamma \vdash_k P}{\Gamma \vdash'' k[\![P]\!]} \qquad \frac{\Gamma(k) = \texttt{lbad}}{\Gamma \vdash'' k\langle\langle \Delta \rangle\rangle} \qquad \frac{\Gamma(k) = \Delta(k) \neq \texttt{lbad} \qquad \Gamma <: \Delta}{\Gamma \vdash'' k\langle\langle \Delta \rangle\rangle}$$

where the typing association $l : \texttt{lbad}$ in Γ now defines bad localities (i.e., requiring that $\Gamma(k) = \texttt{lbad}$ is the same as requiring $k \notin dom(\Gamma)$ in the framework of [19]).

The runtime semantics now type checks incoming agents w.r.t. the filter of the receiving locality. However, in order to give more expressive power to the language, the incoming agent is also type checked by taking into account its source locality (called, its *authority*). Indeed, it is allowed for the agent to 'lie' about the resources in its home locality: if the latter was a good locality, then the agent does not lie, otherwise it can freely lie because it misuses resources of a bad locality (that are unconstrained). Thus, the operational semantics for migrations is now defined as

$$k[\![\mathbf{go}\, l.P]\!] \ \big| \ \ l\langle\langle \Delta \rangle\rangle \ \rightarrow \ l[\![P]\!] \ \big| \ \ l\langle\langle \Delta \rangle\rangle \qquad\qquad \text{if } k = l \text{ or } \Delta \mid\!\vdash_l^k P$$

The dynamic typing, $\mid\!\vdash_l^k$, is defined like the standard type inference but now takes into account also the agent authority: an agent can refer to its authority's resources freely. This is formalised by the rules

$$\Delta \mid\!\vdash_w^k k : K \qquad\qquad \Delta \mid\!\vdash_k^k a : A \qquad\qquad \Delta \mid\!\vdash_w^k \mathbf{go}\, k.P$$

Essentially, it is allowed to assign any type to k and to all channels declared local in k; moreover, a thread left from k can come back without any check. Notice that the dynamic type check can fail; but while a failure of typing the incoming thread w.r.t. Γ means that the thread misbehaves, the failure of the runtime check (w.r.t. the knowledge in a filter) allows only the conclusion that not enough information is present in the filter to accept the incoming thread (that may or may not be malicious). Thus, filters should be updated during the computation; for example this can happen as a side effect of a communication. Thus,

$$k[\![c?(X:T).P]\!] \mid k[\![c!\langle U \rangle.Q]\!] \mid k\langle\langle \Delta \rangle\rangle \ \rightarrow \ k[\![P[^U\!/X]]\!] \mid k[\![Q]\!] \mid k\langle\langle \Delta \sqcap \{_k U : T\} \rangle\rangle$$

In this way, new information is gleaned from the communicated value; this turns out to be crucial to accumulate locally the evidence that a locality is good/bad (always respecting the global knowledge Γ).

To avoid massive runtime checks, a notion of trust between sites is introduced. If a node l trusts another node k, then l assumes that no misbehaved entities will ever pass through k; thus, l will never type check threads coming from k. The trust relation is not symmetric but it is transitive, thus generating *subnets of trust* that can grow up by communications between trusted nodes (again, to this aim it is fundamental to have the possibility of dynamically updating filters). A thread entering in a web of trust is type checked by the first node of the subnet the thread touches; then, no further checks within the subnet are performed anymore, unless the thread exits from the subnet (indeed, if it wants to enter again, it has to be again type checked).

To deal with this new richer framework, locality types have to be extended in the following way:

$$K \ ::= \ \ldots \ \big| \ \ \mathtt{lbad} \ \big| \ \ \mathtt{ltrust}\{\widetilde{u:T}\}$$

It is assumed that lbad and ltrust are not related by the subtyping relation $<:$; in this way, it is ensured that, once a site is deemed trusted/bad within a certain filter, then this decision is permanent (it is impossible to convert a locality of type lbad in a locality of type ltrust, and viceversa), unless the node associated to the filter is bad itself (in this case, no constraints are required on its type). The new rule for migration is now improved, since it only type checks threads coming from untrusted nodes:

$$k[\![\, \mathbf{go}\, l.P \,]\!] \ \big| \ \ l\langle\!\langle \Delta \rangle\!\rangle \ \rightarrow \ l[\![\, P \,]\!] \ \big| \ \ l\langle\!\langle \Delta \rangle\!\rangle \qquad\qquad \text{if } \Delta(k) <: \mathtt{ltrust} \text{ or } \Delta \mid\!\vdash_l^k P$$

In this new framework, type soundness still holds and is formulated as

Theorem 3.5 *If $\Gamma \vdash'' N$ and $N \rightarrow^* N'$ then $N' \not\uparrow_l$ for any l such that $\Gamma(l) \neq \mathtt{lbad}$.*

In [17] a revised version of Dπ typing system is given. Indeed, in all the work summarised so far it was totally legal for two different localities to use the same name to refer to different channels (i.e. local channels with totally different types). This seems something strange: indeed, if two departments have a printer invoked by accessing a port named printer, one can easily imagine that, by sending documents on the printer port, the documents sent is printed. However, this could be not the case in Dπ: the types of the ports in two different department can be totally different and allow for different kind of operations. On the other hand, it seems too restricting to forbid different localities to use the same name to refer local channels. What seems reasonable is to require that, whenever two localities uses the same name to refer local channels, the type of the resources referred must be in accordance. To this aim, in [17] the notion of *registered channel types*, $\mathtt{rc}\{A\}$, is introduced. When a channel is registered, then it is imposed that all the localities using that name declare the corresponding channels at compatible types. With this additional requirement, the type system becomes more complicated. For example, type environments are lists of type associations and must be deemed well-formed, by using the judgement $\Gamma \vdash \mathbf{env}$, i.e. they must respect the requirement on registered channel names; two sample rules are

$$\frac{\Gamma \vdash \mathbf{env} \qquad \Gamma \vdash w : \mathtt{loc}\{\} \qquad u \notin \Gamma, A}{\Gamma, {}_w u : A \vdash \mathbf{env}}$$

$$\frac{\Gamma \vdash \mathbf{env} \qquad \Gamma \vdash w : \mathtt{loc}\{\} \qquad \Gamma \vdash u : \mathtt{rc}\{A'\} \qquad A' <: A \qquad u \notin \Gamma(w) \qquad u \notin A}{\Gamma, {}_w u : A \vdash \mathbf{env}}$$

Finally, a finer control on the migration rights is introduced by means of capabilities $\mathbf{move}_k$. Intuitively, if a location l is known at type $\mathtt{loc}\{\mathbf{move}_k, \ldots\}$, then agents resident at k have migration rights to l. Well-formedness of environments and typing threads is extended accordingly by using the following key rules

$$\frac{\Gamma \vdash \mathbf{env} \qquad \Gamma \vdash w : \mathtt{loc}\{\} \qquad \Gamma \vdash u : \mathtt{loc}\{\}}{\Gamma, u : \mathbf{move}_w \vdash \mathbf{env}} \qquad \frac{\Gamma \vdash_w u : \mathtt{loc}\{\mathbf{move}_w\} \qquad \Gamma \vdash_u P}{\Gamma \vdash_w \mathbf{go}\, u.P}$$

To conclude, we mention two other works on Dπ type systems. The first one [3] develops a typing discipline to enforce *receptiveness*, i.e. a property ensuring that each output over a located channel will eventually be consumed by a corresponding input action. The second one [18] extends Dπ with higher-order communications and the its type system extends that in [17] with the use of dependent types.

3.2 Types for $lsd\pi$

The type system presented in [33] is very similar to the basic type system for Dπ. Its aim is to prevent arity mismatch in communications. However, $lsd\pi$ type system is simpler than [20] because several well-formedness conditions are enforced by syntactic constraints (see footnote 4 for some details and [33] for a full discussion).

Similarly to Dπ, the type of a channel describes the kind of values transmissible over that channel. Thus, if a can carry k-tuples of channels (recall that in $lsd\pi$ site names cannot be communicated) whose type is $\gamma_1,\ldots,\gamma_k$, then a's type is $Chan(\gamma_1,\ldots,\gamma_k)$. The type of a site captures the types of the free channels at the site; thus, if $a_1,\ldots,a_n$ are the free channels in a site s and a_i has type γ_i, then s has type $\{a_1:\gamma_1,\ldots,a_n:\gamma_n\}$.

Type judgements take the form of $\Gamma \vdash N$ where Γ is a finite mapping from site names into site types. It also includes a special site $\underline{h}$ (h stands for *here*) which designates the current site when typing the occurrence of simple channels in processes. The typing inference relies on the following rules

$$\Gamma \vdash a@s : \Gamma(s)(a) \qquad\qquad \Gamma \vdash a : \Gamma(\underline{h})(a)$$

$$\frac{\Gamma \vdash u : Chan(\widetilde{\gamma})\ ,\ \widetilde{v}:\widetilde{\gamma}}{\Gamma \vdash u!\langle\widetilde{v}\rangle} \qquad \frac{\Gamma \vdash u : Chan(\widetilde{\gamma})\ ,\ \widetilde{x}:\widetilde{\gamma}\ ,\ P}{\Gamma - \{\widetilde{x}@\underline{h}\} \vdash u?(\widetilde{x}).P} \qquad \frac{\Gamma \vdash P}{\Gamma[{}^{s}/_{\underline{h}}] \vdash s[\![P]\!]}$$

Notice that a migration (occurring as a side effect of the reference to a remote name $a@s$) is guaranteed to send code to an existing site (i.e. a site in the domain of Γ), since otherwise the type $\Gamma(s)(a)$ is undefined. Moreover, the 'here' site variable $\underline{h}$ is instantiated when typing a node $s[\![\cdot]\!]$ with the site name s of the node.

More refined types systems for $lsd\pi$ are under development in [25]; we leave their presentation to another paper.

3.3 Types for μKLAIM

μKLAIM types provide information about the legality of process actions: downloading/consuming tuples, producing tuples, activating processes and creating new nodes. We use $\{r,i,o,e,n\}$ to indicate the set of *capabilities*, where each symbol stands for the operation whose name begins with it; we let π to range over subsets of capabilities. *Types*, ranged over by δ, are functions mapping localities (and locality variables) into subsets of capabilities.

Nodes are now decorated with a type, expressing their security policy; hence, the syntax of μKLAIM nets becomes

$$N \ ::= \ l ::^{\delta} P \quad \big| \quad N_1 \parallel N_2$$

The type of a node is set by a net coordinator and determines the access policy of the node in terms of access rights; type checking will guarantee that only processes whose intentions match the rights granted by the coordinators are allowed to proceed. E.g., the capability e is used to control process mobility; thus, the privilege $[l' \mapsto \{e\}]$ in the type of locality l will enable processes running in l to spawn code over l'.

Apart from occurring in the specification of a node, type related information are introduced in two other syntactic constructs. In action **newloc**$(u:\delta)$, type δ specifies the security policy of the new node. Moreover, bound locality variables, occurring in templates argument of **in**/**read** actions, now take the form of $!\,u:\pi$, where π specifies the access rights corresponding to the operations that the process prefixed by the **in**/**read** action wants to perform at u. In both cases, the type information is not strictly necessary: it increases the flexibility of the **newloc** action (otherwise, some kind of 'default policy' should be assigned to the newly created node) and enables a simpler static type checking.

Informally, for each node of a net, say $l ::^{\delta} P$, the static type checker determines whether the actions that P intends to perform when running at l are enabled by the access policy δ or not; moreover, it verifies that in $a.\_$ the continuation process behaves consistently to the declarations made for locality variables bound by a. This fact is expressed by the type judgement $\delta \vdash_l P$. A net is deemed *well-typed* if for each node $l ::^{\delta} P$ it holds that $\delta \vdash_l P$.

Type related information play a crucial role in the operational semantics, since they enable/disable process migrations and data communications. This fact is expressed by modifying the operational rules for actions **eval** and **in**/**read** as follows. The new reduction rule for **eval** is

$$\frac{\delta' \vdash_{l'} Q}{l ::^{\delta} \mathbf{eval}(Q)@l'.P \parallel l' ::^{\delta'} P' \ \succ\!\!\longrightarrow \ l ::^{\delta} P \parallel l' ::^{\delta'} P'|Q}$$

Notice that the process Q must be dynamically type checked against the policy of node l'; this is necessary since no a-priori knowledge of target node policy can be assumed, and hence no static checking performed in l over the spawned process can be done. The rule for action **in** (the corresponding rule for **read** is omitted) is:

$$\frac{match_{\delta}(\mathcal{T}[\![\, T \,]\!], t) = \sigma}{l ::^{\delta} \mathbf{in}(T)@l'.P \parallel l' :: \langle t \rangle \ \succ\!\!\longrightarrow \ l ::^{\delta} P\sigma \parallel l' :: \mathbf{nil}}$$

The new pattern matching function $match_{\delta}$ is defined like *match* but it also verifies that process $P\sigma$ does not perform illegal actions w.r.t. δ. Because of the static inference, the definition of $match_{\delta}$ simply relies on the following rule:

$$\frac{\pi \subseteq \delta(l')}{match_{\delta}(!\,u : \pi, l') = [l'/u]}$$

Indeed, the static inference verifies that P performs over u at most the operations declared by π; hence, if δ enables the actions identified by π over l', then $P\sigma$ will never violate policy δ due to operations over l'.

Similarly to Dπ, μKLAIM types are *sound*. This means that processes running in well-typed nets do not attempt to execute actions that are not allowed by the capabilities they own (*type safety*), and that this property is preserved along reductions (*subject reduction*). However, differently from Dπ, the definition of μKLAIM runtime errors is straightforward and relies on the rule

$$\frac{cap(a) \notin \delta(tgt(a))}{l ::^{\delta} a.P \uparrow_l}$$

where $tgt(a)$ and $cap(a)$ denote, resp., the target locality and the capability associated to action a.

The theory presented so far ensures that processes running in a net behave accordingly to the policies specified for the sites of the net; however, it is far from being realistic and usable, especially in e-commerce applications, because of its static nature. We now show some simple modifications that enable programming dynamic privileges acquisition; this will allow us to deal with more flexible and sensitive applications of our theory. We conclude by sketching how privilege loss could be added to the picture; the interested reader is referred to [14] for full details and examples.

The main characteristic of the revised theory is the possibility of programming privileges exchange; to this aim, we shall decorate localities in output actions with a *capability specification*, μ, expressing the conveyed privileges. Hence, tuples take now the form

$$t \ ::= \ e \ \big| \ \ \ell : \mu \ \big| \ \ t_1, t_2$$

Formally, μ is a partial function with finite domain from localities (and locality variables) to subsets of capabilities. Intuitively, action $\mathbf{out}(l : [l_1 \mapsto \pi_1, \ldots, l_m \mapsto \pi_m])@l'$ creates a tuple containing locality l that can be accessed only from localities $l_1, \ldots, l_m$; moreover, when the tuple will be retrieved from l_i, l_i's access policy will acquire the privilege $[l \mapsto \pi_i]$. To rule out simple capability forging, we must ensure that the privilege $[l \mapsto \ \pi_1 \cup \ldots \cup \pi_m]$ is really owned by the node executing the **out**. This can be done through a revised tuple evaluation function $\mathcal{T}[\![\cdot]\!]_\delta$, whose most significant definition rule is

$$\frac{\mu = [l_1 \mapsto \pi_1, \ldots, l_m \mapsto \pi_m] \qquad \mu' = [l_1 \mapsto \pi_1 \cap \delta(l), \ldots, l_m \mapsto \pi_m \cap \delta(l)]}{\mathcal{T}[\![\, l : \mu \,]\!]_\delta = l : \mu'}$$

The operational rule for **out** now becomes

$$\frac{\mathcal{T}[\![\, t \,]\!]_\delta = et}{l ::^{\delta} \mathbf{out}(t)@l'.P \parallel l' ::^{\delta'} P' \ \succ\!\!\longrightarrow \ l ::^{\delta} P \parallel l' ::^{\delta'} P' \mid \langle et \rangle}$$

In this new setting, the execution of actions **in** and **read** has two effects: replacing free occurrences of variables with localities/values (like before) and enriching the type of the node performing the action with the privileges granted along with the tuple. By letting the notation $\delta[\delta'']$ to denote the pointwise union of functions δ and δ'', the new rule for **in** (the rule for **read** is similar) becomes:

$$\frac{match_l^{\delta}(\mathcal{T}[\![\, T \,]\!]_\delta, et) = \langle \delta'', \sigma \rangle}{l ::^{\delta} \mathbf{in}(T)@l'.P \parallel l' :: \langle et \rangle \ \succ\!\!\longrightarrow \ l ::^{\delta[\delta'']} \ P\sigma \parallel l' :: \mathbf{nil}}$$

Function $match_l^{\delta}$ differs from $match_\delta$ in two aspects: it returns the substitution σ to be applied to the continuation process together with the privileges passed by (the producer of) the tuple to node l, and it indirectly type checks $P\sigma$ by also considering such privileges. Its definition relies on the rules

$$\frac{\pi \subseteq \delta(l') \cup \mu(l)}{match_l^{\delta}(!\,u : \pi, l' : \mu) = \langle [l' \mapsto \pi], [l'/u] \rangle} \qquad \frac{match_l^{\delta}(T_1, t_1) = \langle \delta_1, \sigma_1 \rangle \quad match_l^{\delta}(T_2, t_2) = \langle \delta_2, \sigma_2 \rangle}{match_l^{\delta}(\ (T_1, T_2)\ ,\ (t_1, t_2)\ ) = \langle \delta_1[\delta_2], \sigma_1 \circ \sigma_2 \rangle}$$

where $\sigma_1 \circ \sigma_2$ denotes composition of substitutions σ_1 and σ_2. Indeed, the static inference ensured that P performs over u at most actions enabled by π. Moreover, function *match* succeeds only if π is enabled by the privileges owned by l over l', joint with the new privileges $\mu(l)$. These two conditions imply that $P\sigma$ type checks.

Since node $l ::^{\delta} P$ can dynamically acquire privileges when P performs **in**/**read** actions, it is possible that statically illegal actions can become permissible at run-time. For this reason, if P intends to perform an action not allowed by δ, the static inference system cannot now reject the process, since the capability necessary to perform the action could in principle be dynamically acquired by l. In such cases, the inference system simply *marks* the action to require its dynamic checking. The static semantics now is built up over the judgement $\delta \vdash_{\mathcal{T}} P \triangleright P'$, where process P' is obtained from P by possibly marking some actions. Intuitively, it means that all the variables in P' are used according to their definition and, when P' is located at l, its unmarked actions are allowed by δ. Once the syntax of processes has been extended to allow processes to contain marked actions, a net can be deemed *executable* if for each node $l ::^{\delta} P$ it holds that $\delta \vdash_{\mathcal{T}} P \triangleright P$ (i.e. if the net already contains all the necessary marks).

As far as the operational semantics is concerned, the rule for **eval** must be modified. In this framework, the process that is actually sent for execution is that resulting (if any) from the type checking of the original incoming process. Thus, such a process contains all necessary marks. Moreover, in order to take into account the execution of marked actions, the following rule must be added

$$\frac{l' = tgt(a) \qquad cap(a) \in \delta(l') \qquad l ::^{\delta} a.P \parallel l' ::^{\delta'} Q \ \succ\!\!\longrightarrow \ N}{l ::^{\delta} \underline{a}.P \ \parallel l' ::^{\delta'} Q \ \succ\!\!\longrightarrow \ N}$$

In substance, this rule says that the marking mechanism acts as an in-lined security monitor by stopping the execution of marked actions whenever the privilege for executing them is missing. Type soundness still holds, but is now formulated in terms of executable nets.

We now comment on possible variations of the type theory. In real situations, a (mobile) process could dynamically acquire some privileges and, from time to time, decide whether it wants to keep them for itself or to share them with other processes running in the same environment, viz. at the same node. Our framework can smoothly accommodate this feature, by associating privileges also to processes and letting them decide whether an acquisition must enrich their hosting node or themselves. Moreover, the privileges could have an expiration date. Timing information can easily be accommodated in the framework presented by simply assigning privileges a validity duration and by updating this information while time passes. Furthermore, 'acquisition of privileges' can be thought of as 'purchase of services/goods'; hence it would be reasonable that a process lose the acquired privilege once it uses the service or passes the good to another process. A simple modification of our framework, for taking into account multiplicities of privileges and their consumption (due, e.g., to execution of the corresponding action or to cession of the privilege to another process), can permit the handling of this new scenario. Finally, the granter of a privilege could decide to revoke the privilege previously granted because of, e.g., a misbehaviour or expiry of the subscription time (in fact, this could be a way of managing expiration dates without assigning privileges a validity duration). To manage privilege revocation we could annotate privileges dynamically acquired with the granter identity and enable processes to use a new 'revoke' operation.

In [16], μKLAIM type system is also refined to incorporate other real systems security features, i.e. granting different privileges to processes coming from different nodes and constraining the operations allowed over different tuples. Thus, for example, if l trusts l', then l security policy could accept processes coming from l' and let them accessing any tuple in its tuple space. If l' is not totally trusted, then l's security policy could grant processes coming from l', e.g., the capabilities for executing **in**/**read** only over tuples that do not contain classified data. To this aim, we let types to be functions from localities (and locality variables) into functions from localities (and locality variables) into sets of capabilities. Intuitively, the association $[l \mapsto l' \mapsto \pi]$ in the policy of node l'' enables processes spawned over l'' by (a process running at) node l to perform over l' the operations enabled by π. Capabilities are still used to specify the allowed process operations, but now they also specify the shape (i.e. number of fields, kind of each field, ...) of **in**/**out**/**read** arguments. For example, the capability $< i\ ,\ \langle \text{“}public\text{”}, - \rangle >$ (where '$-$' is used to denote a generic template field) states that action $\mathbf{in}(T)$ is enabled only if T is made up of two fields and the first one is the string "*public*". Thus, it enables the operations $\mathbf{in}(\text{“}public\text{”}, !x)@...$ and $\mathbf{in}(\text{“}public\text{”}, 3)@...$, while disables operations $\mathbf{in}(\text{“}private\text{”}, !x)@...$ and $\mathbf{in}(!x, !y)@....$

The types for KLAIM proposed in [9] were functions mapping localities (and locality variables) into functions from sets of capabilities to types. A type of the form $[\ell \mapsto \pi \mapsto \delta]$ describes the intention of performing the actions corresponding to π at ℓ; moreover, it imposes constraint δ on the processes that could possibly be spawned at ℓ. Thus, if $[l \mapsto \{e\} \mapsto \delta]$ is in the policy of node l', then processes running at l' can spawn over l code that type checks with δ. This is required in order to enable the static inference to decide whether the spawned process can legally run at l or not (thus, it avoids the dynamic type check when performing **eval** actions). However, to make this possible, it must hold that δ is a subtype of l's type; hence, a global knowledge of node types is required. This can be reasonable for LANs while is hardly implementable in WANs, where usually nodes are under the control of different authorities. The type systems for μKLAIM are more realistic in that the static checker only need local information; however, it is less efficient because it requires a larger amount of dynamic checks.

Moreover, the type theory is complicated by the fact that types can be *recursive*; hence, system of recursive type equations must be solved often. Recursive types are used for typing migrating recursive processes like, e.g., $P \triangleq \mathbf{in}(!x)@l.\mathbf{out}(x)@l'.\mathbf{eval}(P)@l''$. P can be typed by solving the recursive type equation $\delta = [l \mapsto \{i\} \mapsto \bot, l' \mapsto \{o\} \mapsto \bot, l'' \mapsto \{e\} \mapsto \delta]$, where $\bot$ denotes the empty type. However, notice that recursive processes do not necessarily have recursive types: e.g. process $Q \triangleq \mathbf{in}(!x)@l.\mathbf{out}(x)@l'.Q$ has type $[l \mapsto \{i\} \mapsto \bot, l' \mapsto \{o\} \mapsto \bot]$.

4 Behavioural Equivalences

To conclude this paper, we shall now give an outline of the behavioural theories developed for the languages studied. Behavioural theories are usually exploited to abstract away a process/net from its syntactic structure and isolate the essence of its functionality. The theory can be used in several ways, e.g. to prove the soundness of a protocol implementation in the language, to prove some form of correspondence between a process written in a language and its encoding in another language, to establish the theoretical foundation to a optimisation procedure (that, for example, takes a process and produces a more efficient but still functionally equivalent process), and so on.

All our case-study languages come equipped with a behavioural theory based on some form of bisimulation. *lsd*π only provides a bisimulation that turns out to be a congruence, while Dπ's and μKLAIM's bisimulations exactly captures a form of contextually defined equivalence, namely *barbed congruence* [30]. Barbed congruence relies on the notion of *observation* and considers equivalent those nets that cannot be distinguished by any observation in any context during their execution. Having a bisimulation that exactly captures a contextual equivalence is usually considered as evidence for the bisimulation to be the most appropriate notion of behavioural equivalence for the calculus. Indeed, it is both expressive (it captures barbed congruence) and it does not suffer from the context closure present in any contextual congruence.

In the rest of this section, we shall present the semantic theory of the three languages studied. In each subsection, we shall firstly present a contextual relation (whenever present), an operational semantics based on *labelled transition system* (LTS) and a bisimulation built over it. As usual, one can choose whether giving a rather complex LTS but a standard bisimulation or a simple LTS and a more articulated notion of bisimulation. Dπ follows the first approach, while μKLAIM and *lsd*π the second one. For the sake of presentation, we shall omit from this paper all the technical details; the interested reader is referred to the full papers for a complete presentation.

An asynchronous higher-order bisimulation for μKLAIM. The semantics theory developed for μKLAIM in [15] firstly introduces a reduction barbed congruence relation that is considered the touchstone of the behavioural equivalences in μKLAIM, by following the approach in [22]. Then, it introduces a simple LTS and builds up on its top a labelled higher-order asynchronous bisimulation that exactly captures barbed congruence.

For the sake of simplicity, we shall use a slightly different version of the calculus, where the **read** primitive is not present (it can be encoded) and where the operational semantics for the creation of new nodes does not exploit anymore configurations like $L \vdash N$ (see Section 2.3) but relies on the standard π–calculus restriction operator. Thus a net now is described by the following production

$$N ::= \mathbf{0} \;\big|\; l :: P \;\big|\; N_1 \parallel N_2 \;\big|\; (\nu l)N$$

Definition 4.1 (Reduction Barbed Congruence)

- *Predicate $N \downarrow l$ holds true if and only if $N \equiv (\nu\widetilde{l})(N' \parallel l :: \langle t \rangle)$, for some tuple t, and $l \notin \widetilde{l}$. Predicate $N \Downarrow l$ is defined as $\exists N' : N \succ\!\!\longrightarrow^{*} N' \wedge N' \downarrow l$.*

- *Net contexts are defined as $C[\cdot] ::= [\cdot] \;\big|\; N \parallel C[\cdot] \;\big|\; (\nu l)C[\cdot]$*

- Reduction barbed congruence, $\cong$, *is the largest symmetric relation over nets such that, whenever $N_1 \mathfrak{R} N_2$, it holds that:*

1. *if $N_1 \downarrow l$ then $N_2 \Downarrow l$*
2. *if $N_1 \rightarrowtail N_1'$ then $N_2 \rightarrowtail^* N_2'$ and $N_1' \Re N_2'$*
3. *for all net contexts, it holds that $C[N_1] \cong C[N_2]$*

Intuitively, the basic μKLAIM observable is the presence of a datum in a certain tuple space; barbed congruence equates two nets that, once put in a generic context, exhibit the same data in the same localities during all their computations. In order to capture barbed congruence by using a coinductive technique, we give an alternative (equivalent) operational semantics to μKLAIM nets, based on a LTS. The main rules are

$$\frac{l \notin fn(N_2)}{N_1 \parallel (\nu l)N_2 \equiv (\nu l)(N_1 \parallel N_2)} \qquad \frac{match(\mathcal{E}[\![\, T \,]\!], et) = \sigma}{l :: \mathbf{in}(T)@l'.P \xrightarrow{et \,\triangleleft\, l'} l :: P\sigma}$$

$$l :: \mathbf{out}(t)@l'.P \xrightarrow{\langle \mathcal{E}[\![\, t \,]\!] \rangle \,\triangleright\, l'} l :: P \qquad l :: \mathbf{eval}(Q)@l'.P \xrightarrow{Q \,\triangleright\, l'} l :: P$$

$$l :: \mathbf{newloc}(l').P \xrightarrow{\tau} (\nu l')(l :: P \parallel l' :: \mathbf{nil}) \qquad l :: I \xrightarrow{I \,@\, l} l :: \mathbf{nil}$$

$$\frac{N_1 \xrightarrow{et \,\triangleleft\, l'} N_1' \qquad N_2 \xrightarrow{\langle et \rangle \,@\, l'} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2'} \qquad \frac{N_1 \xrightarrow{P \,\triangleright\, l} N_1' \qquad N_2 \xrightarrow{\mathbf{nil} \,@\, l} N_2'}{N_1 \parallel N_2 \xrightarrow{\tau} N_1' \parallel N_2' \parallel l :: P}$$

where I ranges over the inert processes, namely **nil** and $\langle et \rangle$. The rules above essentially states that a **newloc** is turned into a net restriction and that the scope of restricted names can be extended without captures. Moreover, the intentions of sending a datum/process can be concretised (thus originating a τ action) only if there exists in the net the target node of the **out**/**eval**; similarly, the intention of retrieving a datum can be concretised only if the target site contains in its tuple space a datum matching the specified template. Obviously, it is assumed that, whenever a $\mathcal{E}[\![\, \cdot \,]\!]$ occurs, it never fails.

By exploiting the features of this LTS, we can define a labelled bisimulation that exactly captures the barbed congruence given above. For the sake of presentation, we let χ to range over actions τ and $(\nu \widetilde{l})\ I\ @\ l$. As usual, $\Longrightarrow$ denotes $\xrightarrow{\tau}^*$, $\xRightarrow{\mu}$ denotes $\Rightarrow \xrightarrow{\mu} \Rightarrow$ and $\xRightarrow{\hat{\mu}}$ denotes $\Longrightarrow$ if $\mu = \tau$ and $\xRightarrow{\mu}$ otherwise.

Definition 4.2 *A symmetric relation $\Re$ between μ*KLAIM *nets is a* bisimulation *if for each $N_1 \Re N_2$ it holds that:*

1. *if $N_1 \xrightarrow{\chi} N_1'$ then $N_2 \xRightarrow{\hat{\chi}} N_2'$ and $N_1' \Re N_2'$*
2. *if $N_1 \xrightarrow{(\nu \widetilde{l})\ P \,\triangleright\, l} N_1'$ then $N_2 \parallel l :: \mathbf{nil} \Rightarrow N_2'$ and $(\nu \widetilde{l})(N_1' \parallel l :: P) \Re N_2'$*
3. *if $N_1 \xrightarrow{et \,\triangleleft\, l'} N_1'$ then $N_2 \parallel l' :: \langle et \rangle \Rightarrow N_2'$ and $N_1' \parallel l' :: \mathbf{nil} \Re N_2'$*

Bisimilarity, *$\approx$, is the largest bisimulation.*

This bisimulation is somehow inspired by that in [23]. The key idea is that, since sending operations are asynchronous, a sending operation by a net N_1, say $N_1 \overset{(\nu \widetilde{l})\ C \,\triangleright\, l}{\rightarrowtail} N_1'$, can be simulated by a net N_2 in a context where the locality l is provided by performing some internal actions (becoming N_2'). Indeed, since we want our bisimulation to be a congruence, a context that provides the target locality of the sending action should not tell apart N_1 and N_2. Hence, for $N_1 \parallel l :: \mathbf{nil}$ to be simulable by $N_2 \parallel l :: \mathbf{nil}$, it must hold that, upon transitions, $(\nu \widetilde{l})(N_1' \parallel l :: C)$ is simulable by N_2'. Similar considerations holds also for the case of the input actions, but the context is $[\cdot] \parallel l_1 :: \langle l_2 \rangle$.

In [15], this bisimulation has been used to verify the requirements for a μKLAIM implementation of a protocol solving the *'Dining Philosophers'* problem. The basic framework has been then extended to keep into account failures and, by exploiting the modified framework, it has been proved the soundness of a possible solution of the *'k-set agreement'* problem. Finally, by adding a predicate of reachability between μKLAIM nodes in a net, also a routing bisimulation is defined that takes into account the number of hops necessary to perform actions.

A typed bisimulation for Dπ. The main contribution of [17] is in the definition of a typed bisimulation for Dπ networks. Indeed, since Dπ strongly relies on types to control the proper execution of programs (e.g. by blocking migrations over non-existing nodes, to ensure unique allocation of a channel name in a node or the consistent use of shared channel names by means of registered types), the behavioural semantics must take into account the global knowledge available. Moreover, it turns out that the equivalence of two nets strongly relies on the capabilities the system owns over the resources of the nets: indeed, a context in which no one can migrate over k and no $\mathtt{r}$ capability is present for channel a will never tell apart $k[\![\,\mathbf{stop}\,]\!]$ and $k[\![\,a!\langle\rangle\,]\!]$.

Thus, in [17] the notion of *knowledge-indexed relations* over networks is used. Such relations are families of binary relations over Dπ networks indexed by closed type environments, representing the knowledge of the user (i.e. a subset of the global knowledge of the system). In general, the user knowledge does not have enough information to type the systems considered, but it is consistent with (i.e. it is a supertype) the system knowledge, that on the contrary must type the systems. Hence, the standard notion of (weak) barbed congruence is extended to take into account also typing information.

Definition 4.3 $\cong^{rbc}$ *is the largest symmetric knowledge-indexed relation over Dπ networks such that:*

1. *it is* reduction closed, *i.e. whenever $\Delta \models M \cong^{rbc} N$ and $M \to M'$ there exists N' such that $N \to^* N'$ and $\Delta \models M' \cong^{rbc} N'$*
2. *it is* barb preserving, *i.e. $\Delta \models M \cong^{rbc} N$ and $\Delta \vdash M \Downarrow a@k$ implies $\Delta \vdash N \Downarrow a@k$ (where $\Delta \vdash M \Downarrow a@k$ holds true iff $\Delta \vdash_k k : \mathtt{loc}\{\mathbf{move}, a : \mathtt{res}\{\mathtt{r}\langle\rangle, \mathtt{w}\langle\rangle\}\}$ and $M \to^* (M' \mid k[\![\,a!\langle\rangle.P\,]\!])$)*
3. *it is* contextual, *i.e.*
   - *$\Delta \sqcap n : T \models M \cong^{rbc} N$ implies $\Delta \models (\nu n : T)M \cong^{rbc} (\nu n : T)N$*
   - *$\Delta \models M \cong^{rbc} N$ and $\Delta \vdash_k k : \mathtt{loc}\{\mathbf{move}\}$ and $\Delta \vdash k[\![\,P\,]\!]$ implies $\Delta \models M | k[\![\,P\,]\!] \cong^{rbc} N | k[\![\,P\,]\!]$*
   - *$\Delta \models M \cong^{rbc} N$ and $\Delta, \Delta' \vdash \mathbf{env}$ implies $\Delta, \Delta' \models M \cong^{rbc} N$*

According to this definition, a barb is now observable on a channel a at location k only if k is actually a reachable location containing channel a and it is possible to read and write values from it. For the sake of simplicity, only the **move** capability (and not the more complicated $\mathbf{move}_k$) is considered for the behavioural theory. Moreover, the contextuality requirement only accepts parallel contexts typeable w.r.t. the user knowledge; the latter, in turn, can increase by inventing new names (but not new capabilities for already known names) that, however, must preserve well-formedness of the knowledge.

The labelled bisimulation capturing this contextual congruence is built upon a typed LTS relating *configurations* of the form $\Delta \triangleright N$, where Δ is the user knowledge. We use $C, D, \ldots$ to range over configurations. Transitions take the form $\Delta \triangleright N \xrightarrow{\mu} \Delta' \triangleright N'$, since also the environment can change as a side effect of the action execution. Moreover, because of the possible limited knowledge, an external user may not be able to provoke these actions. The key rules are

$$\frac{N \to N'}{\Delta \triangleright N \xrightarrow{\tau} \Delta \triangleright N'} \qquad \frac{\Delta \vdash_k k : \mathtt{loc}\{\mathbf{move}, a : \mathtt{res}\{\mathtt{w}\langle T\rangle\}\} \qquad \Delta \vdash_k U : T}{\Delta \triangleright k[\![\,a?(X : T).P\,]\!] \xrightarrow{k.a?\langle V\rangle} \Delta \triangleright k[\![\,P[^{U}/_{X}]\,]\!]}$$

$$\frac{\Delta \vdash_k k : \mathtt{loc}\{\mathbf{move}, a : \mathtt{res}\{\mathtt{r}\langle T\rangle\}\} \qquad \Delta \sqcap_k U : T \text{ exists}}{\Delta \triangleright k[\![\,a!\langle U\rangle.P\,]\!] \xrightarrow{k.a!\langle V\rangle} \Delta \sqcap_k U : T \triangleright k[\![\,P\,]\!]}$$

By using this LTS, the definition of (weak) labelled bisimulation is standard.

Definition 4.4 *A symmetric binary relation $\Re$ over Dπ configurations is said to be a* bisimulation *if $C \;\Re\; D$ implies that, whenever $C \xrightarrow{\mu} C'$, there exists D' such that $D \overset{\hat{\mu}}{\Longrightarrow} D'$ and $C' \;\Re\; D'$.*

M and N are bisimilar in the environment Δ, *written $\Delta \models M \approx^{bis} N$, if $\Delta \triangleright M \;\Re\; \Delta \triangleright N$ for some bisimulation $\Re$.*

Finally, a more complex scenario is considered in [17], where the context can already put threads in localities where it does not have migration rights; thus, for example, the context $k[\![\, a?().\mathbf{go}\, l.b!\langle\rangle \,]\!]$ can distinguish the nets

$$l[\![\ldots]\!] \mid k[\![\,\mathbf{stop}\,]\!] \qquad \text{and} \qquad l[\![\ldots]\!] \mid k[\![\, a!\langle\rangle \,]\!]$$

by relying on the user knowledge $\Delta \triangleq l : \mathtt{loc}\{\mathbf{move}, b : \mathtt{res}\{\mathtt{r}\langle\rangle, \mathtt{w}\langle\rangle\}\}, k : \mathtt{loc}\{a : \mathtt{res}\{\mathtt{r}\langle\rangle, \mathtt{w}\langle\rangle\}\}$. The definitions in this more elaborated setting are heavier, and thus are left for the interested reader.

An asynchronous bisimulation for *lsd*π. As we said, in [33] is given an asynchronous bisimulation for *lsd*π; but it does not capture any contextual equivalence. Hence, the bisimulation can be useful to prove some interesting equations or to formulate a fully abstract encoding of *lsd*π in Dπ(or viceversa). However, apart from congruence properties, it has not a strong theoretical relevance. The main labelled rules it exploits are

$$a!\langle\tilde{u}\rangle \xrightarrow{a!\langle\tilde{u}\rangle} \mathbf{0} \qquad\qquad a?(\tilde{x}).P \xrightarrow{a?\langle\tilde{u}\rangle} P[^{\tilde{u}}/_{\tilde{x}}]$$

$$\frac{r \neq s}{r[\![\, a@s!\langle\tilde{u}\rangle \,]\!] \xrightarrow{\tau} s[\![\, (a@s!\langle\tilde{u}\rangle)\sigma_r \,]\!]} \qquad \frac{r \neq s}{r[\![\, a@s?(\tilde{u}).P \,]\!] \xrightarrow{\tau} s[\![\, (a@s?(\tilde{u}).P)\sigma_r \,]\!]}$$

$$\frac{P \xrightarrow{(\nu\tilde{n})a!\langle\tilde{u}\rangle} P' \qquad Q \xrightarrow{a?\langle\tilde{u}\rangle} Q' \qquad subj(\tilde{n}) \cap fn(Q) = \emptyset}{P \mid Q \xrightarrow{\tau} (\nu\tilde{n})(P' \mid Q')}$$

where $subj(\tilde{n})$ collects the (possibly compound) names occurring in $\tilde{n}$ that are caught by the restriction on $\tilde{n}$. The labelled bisimulation is a standard asynchronous bisimulation and is defined as follows:

Definition 4.5 *A symmetric binary relation $\Re$ between lsdπ nets is a* labelled bisimulation *if, whenever $N_1 \Re N_2$, it holds that:*

1. *if $N_1 \xrightarrow{\tau} N_1'$ then $N_2 \xrightarrow{\tau} N_2'$ and $N_1' \Re N_2'$*
2. *if $N_1 \xrightarrow{(\nu\tilde{n})u!\langle\tilde{u}\rangle} N_1'$ with $subj(\tilde{n}) \cap fn(N_2) = \emptyset$ then $N_2 \xrightarrow{(\nu\tilde{n})u!\langle\tilde{u}\rangle} N_2'$ and $N_1' \Re N_2'$*
3. *if $N_1 \xrightarrow{u?\langle\tilde{u}\rangle} N_1'$ then*
   a. either $N_2 \xrightarrow{u?\langle\tilde{u}\rangle} N_2'$ and $N_1' \Re N_2'$,
   b. or $N_2 \xrightarrow{\tau} N_2'$ and $N_1' \Re (N_2' \mid s[\![\, a!\langle\tilde{u}\rangle \,]\!])$, where $u = a@s$.

Labelled bisimilarity $\sim$ *is the largest labelled bisimulation.*

This relation is used in [33] to prove the motto of *lsd*π: "what you see is what you get". This is expressed by the equation

$$s[\![P]\!] \;\sim\; r[\![P]\!]$$

that holds true whenever P only contains global names.

5 Related Work

In this paper we analysed three languages for programming global computing applications that rely on a flat net architecture. To the best of our knowledge, there are two other such languages studied in literature: $\pi_{1\ell}$-calculus [1] by Roberto Amadio and NOMADIC PICT [36] by Peter Sewell *et al*. In this section we shall briefly mention their characteristics and, thus, we will compare their features with the work presented in this paper for μKLAIM, Dπ and *lsd*π.

$\pi_{1\ell}$-calculus. The calculus is an asynchronous π–calculus with features for processes allocation (nodes are organised in a flat architecture) and movement, together with primitives for killing executing nodes and detecting such failures. Each node comes equipped with a *location process* providing information about the status of the corresponding node (dead or alive). The migration primitive, $\mathtt{spawn}(P,l)$, is very closed to action **go** in Dπ and **eval** in μKLAIM: the movement is objective and the mobility is weak (actually, the latter primitives were inspired by the former one).

Channels are not located in a node and the communication is by default remote: a message sent to a channel *a* enters in the network (thus exiting from the site where it has been emitted) and then reaches without any explicit routing its corresponding receiver (if any). For any channel, there is a *unique* receiving process for that channel and names transmitted as arguments of messages cannot be used to create receivers. This restriction greatly simplifies the algebraic theory and is enforced by a type system.

Nodes are thus the unit of failure: they cannot move (since there is no hierarchical structure assumed), they do not offer capabilities for inter-process communication (channels are reachable from everywhere) and there is no notion of security policy associated to a node. The calculus is encoded in an asynchronous π–calculus enjoying the unicity of receivers, called π_1-calculus, thus hiding all the features related to distribution.

NOMADICT PICT. The language is a distributed and agent-based version of PICT [32], a concurrent language based on the asynchronous π–calculus. Differently from all the languages for distribution with flat net architecture, the language relies on a net (a collection of named sites) where *named agents* can roam. Both agents and sites are uniquely named.

Channels are not located, but the communication between two agents can take place only if they are located in the same node (thus no low-level remote communication is allowed). However, the language also provides a (high-level) primitive for remote communication, that transparently delivers a message to an agent even if the latter is not co-located with the message sender. This primitive is then encoded in the low-level calculus by a central forwarding server, implemented by only using the low-level primitives.

Agents are the unity of mobility, while sites are the unity of communication. The migration primitive $\mathtt{migrate\ to}\ \_$ is subjective in that it moves all the agent performing the action to the specified site $\_$. Moreover, the migration primitive is strong and asynchronous in that an agent brings its state in the remote site where it migrates on and can still receive messages while moving. These aspects are in sharp contrast with the design choice of Dπ and μKLAIM.

A type system and a typed LTS are then defined to verify that the programs written respects the typing information specified also during their computations. Types record information about the values a channel can carry on (both in read and in write mode) and the fact that a name refers to a site or to an agent (and in this last case, it also verifies whether the agent is mobile or static). This use of types is not related to access control: it is closer to standard π–calculus-like type safety.

6 Conclusions

The present paper has collected several results about the most popular languages for programming global computing applications under the assumption that the net model is just a collection of nodes hosting executing processes and resources. The main case-study languages were those included in the MIKADO project, namely Dπ, *lsd*π and μKLAIM. The aim of this paper was to put in evidence the commonalities between the languages and the foundational theories developed by the project members during the last years. Section 2 has compared their language design issues, mainly in terms of the communication and mobility primitives: it should be clear now that, even if the languages are somehow different, they are closely related and are easily interchangeable. Sections 3 and 4 have sketched some of the theory developed for them. It should be clear from the exposition their type theories in particular are clearly inspired by the same principles.

We believe that these commonalities can been transferred in the core programming model; indeed, the choice of having a flat net architecture only simplifies the semantic theory underlying the languages. We think that at least the type related work can be adapted to languages with a hierarchical net structure. On the other hand, the behavioural theories of the latter languages are harder because of the more complicated interaction that can arise in network executions (see, e.g., the problems of *grave interferences* pointed out in [24] for the Ambient calculus or the complications present in the definition of coinductive proof techniques for the Ambient calculus [26, 27], the Boxed Ambient calculus [5] or the Seal calculus [7]). However, we believe that the theories developed for the (simpler) flat net languages can help in understanding the essence of a bisimulation in a global computing scenario. They can be a good testing environment for developing new semantic notions (like, e.g., failures, connections/disconnections, routing information) for more general frameworks.

Acknowledgements: Matthew Hennessy, Vladimiro Sassone and Rosario Pugliese carefully read a first draft of this work and helped me in improving it. Antonio Ravara suggested some good improvements and explained me several *lsd*π subtleties.

References

[1] R. M. Amadio. An asynchronous model of locality, failure, and process mobility. In D. Garlan and D. Le Metayer, editors, *Proceedings of COORDINATION '97*, volume 1282 of *LNCS*. Springer, 1997. Extended version as Rapport de Recherche RR-3109, INRIA Sophia-Antipolis, 1997.

[2] R. M. Amadio, I. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous π-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998. An extended abstract appeared in *Proceedings of CONCUR '96*, LNCS 1119: 147–162.

[3] R. M. Amadio, G. Boudol, and C. Lhoussaine. The Receptive Distributed Pi-Calculus. *Trans. on Programming Languages and Systems*, 25(5):1-29. ACM, 2003.

[4] G. Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.

[5] M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication interference in mobile boxed ambients. In Agrawal and Seth editors, *Proc. of FST-TCS'02*, volume 2556 of *LNCS*, pages 71–84. Springer, 2002.

[6] L. Cardelli and A. D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. An extended abstract appeared in *Proceedings of FoSSaCS '98*, number 1378 of LNCS, pages 140-155, Springer, 1998.

[7] G. Castagna and F. Z. Nardelli. The Seal Calculus Revisited: contextual equivalence and bisimilarity. In Agrawal and Seth editors, *Proc. of FST-TCS'02*, volume 2556 of *LNCS*, pages 85–96. Springer, 2002.

[8] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[9] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science*, 240(1):215–254, 2000.

[10] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *Proceedings of CONCUR '96*, volume 1119 of *LNCS*, pages 406–421. Springer, 1996.

[11] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[12] D. Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89*, volume 366 of *LNCS*, pages 20–27. Springer, 1989.

[13] J. Godskesen, T. Hildebrandt, and V. Sassone. A calculus of mobile resources. In L. Brim, P. Jancar, M. Kretínský, and A. Kucera, editors, *CONCUR*, volume 2421 of *LNCS*, pages 272–287. Springer, 2002.

[14] D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. Research report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2002. Available at `http://rap.dsi.unifi.it/~pugliese/DOWNLOAD/muklaim-full.pdf`. An extended abstract appeared in the Proceedings of *ICALP'03*, *LNCS* 2719.

[15] D. Gorla and R. Pugliese. A semantic theory for global computing system. Research report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2003. Available at `http://rap.dsi.unifi.it/~pugliese/DOWNLOAD/bis4k-full.pdf`.

[16] D. Gorla and R. Pugliese. Enforcing security policies via types. In D. Hutter, G. Mueller, W.Stephan, and M. Ullman, editors, *Proc. of 1st Intern.Conf. on Security in Pervasive Computing (SPC'03)*, volume 2802 of *LNCS*, pages 88–103. Springer-Verlag, 2003.

[17] M. Hennessy, M. Merro, and J. Rathke. Towards a behavioural theory of access and mobility control in distributed systems. In *Proceedings of FoSSaCS '03*, volume 2620 of *LNCS*, pages 282–299. Springer, 2003. Full version as COGS Computer Science Technical Report, 2002:01.

[18] M. Hennessy, J. Rathke, and N. Yoshida. SAFE-Dpi: a language for controlling mobile code. In *Proceedings of FoSSaCS '04*, to appear. Full version as COGS Computer Science Technical Report, 2003:02.

[19] M. Hennessy and J. Riely. Type-Safe Execution of Mobile Agents in Anonymous Networks. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *LNCS*, pages 95–115. Springer, 1999.

[20] M. Hennessy and J. Riely. Resource Access Control in Systems of Mobile Agents. *Information and Computation*, 173:82–120, 2002.

[21] K. Honda and M. Tokoro. An object calculus for asynchronous communication. In P. America, editor, *Proceedings of ECOOP '91*, volume 512 of *LNCS*, pages 133–147. Springer, July 1991.

[22] K. Honda and N. Yoshida. On Reduction-Based Process Semantics. *Theoretical Computer Science*, 152(2): 437–486, 1995.

[23] U. Nestmann and B. Pierce. Decoding Choice Encodings. *Information and Computation*, 163:1–59, 2000.

[24] F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings of POPL '00*, pages 352–364. ACM, Jan. 2000.

[25] F. Martins. Mobility policies for *lsd*π (provisory title). Draft, 2003.

[26] M. Merro and M. Hennessy. Bisimulation congruences in Safe Ambients. In *Proceedings of POPL '02*. ACM, 2002.

[27] M. Merro and F. Z. Nardelli. Bisimulation proof methods for mobile ambients. Technical Report 2003:1, COGS, University of Sussex, Brighton, 2003. An extended abstract appeared in the *Proceedings of ICALP'03*.

[28] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[29] R. Milner, J. Parrow, and J. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.

[30] R. Milner and D. Sangiorgi. Barbed Bisimulation. Proceedings of *ICALP'92*, LNCS, pagg. 685–695. Springer, 1992.

[31] B. C. Pierce and D. Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–454, 1996. An extract appeared in *Proceedings of LICS '93*: 376–385.

[32] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. In G. Plotkin, C. Stirling, and M. Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing. MIT Press, May 2000.

[33] A. Ravara, A. G. Matos, V. T. Vasconcelos, and L. Lopes. A Lexically Scoped Distributed π-Calculus. Technical Report TR02-4, Department of Computer Science, University of Lisbon, 2002. An extended abstract appeared in Proceedings of *Foundations of Global Computing 2003*, ICALP'03 Satellite Workshop.

[34] J. Riely and M. Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of POPL '99*, pages 93–104. ACM, 1999. Full version to appear in *Journal of Automated Reasoning, 2003*.

[35] J. B. Stefani. A calculus of kells. An extended abstract appeared in Proceedings of *Foundations of Global Computing 2003*, ICALP'03 Satellite Workshop.

[36] A. Unyapoth and P. Sewell. Nomadic Pict: Correct communication infrastructures for mobile computation. In *Proceedings of POPL '01*, pages 116–127. ACM, Jan. 2001.