In *Global Computing*, LNCS 2874, pages 88–150.

The KLAIM Project: Theory and Practice*

Lorenzo Bettini[1], Viviana Bono[2], Rocco De Nicola[1], Gianluigi Ferrari[3], Daniele Gorla[1], Michele Loreti[1], Eugenio Moggi[4], Rosario Pugliese[1], Emilio Tuosto[3], and Betti Venneri[1]

[1] Dip. Sistemi e Informatica, Univ. di Firenze, v. Lombroso 6/17, 50134 Firenze, Italy
[2] Dip. Informatica, Univ. di Torino, Corso Svizzera 185, 10149 Torino, Italy
[3] Dip. Informatica, Univ. di Pisa, v. Buonarroti 2, 56100 Pisa, Italy
[4] Dip. Informatica e Scienze dell'Informazione, Univ. di Genova, v. Dodecaneso 35, 16146 Genova, Italy

Abstract. KLAIM (*Kernel Language for Agents Interaction and Mobility*) is an experimental language specifically designed to program distributed systems consisting of several mobile components that interact through multiple distributed tuple spaces. KLAIM primitives allow programmers to distribute and retrieve data and processes to and from the nodes of a net. Moreover, localities are first-class citizens that can be dynamically created and communicated over the network. Components, both stationary and mobile, can explicitly refer and control the spatial structures of the network.
This paper reports the experiences in the design and development of KLAIM. Its main purpose is to outline the theoretical foundations of the main features of KLAIM and its programming model. We also present a modal logic that permits reasoning about behavioural properties of systems and various type systems that help in controlling agents movements and actions. Extensions of the language in the direction of object oriented programming are also discussed together with the description of the implementation efforts which have lead to the current prototypes.

Keywords: Process Calculi, Mobile Code, Distributed Applications, Network Awareness, Tuple Spaces, Type Systems, Temporal Logics, Java.

*This work has been partially supported by EU FET - Global Computing initiative, project AGILE IST-2001-32747, project DART IST-2001-33477, project MIKADO IST-2001-32222, project PROFUNDIS IST-2001-33100, and by MIUR project NAPOLI. The funding bodies are not responsible for any use that might be made of the results presented here.

1 Introduction

The distributed software architecture (model) which underpins most of the wide area network (WAN) applications typically consists of a large number of heterogeneous computational entities (sometimes referred to as nodes or sites of the network) where components of applications are executed. Network sites are generally managed by different authorities with different administrative policies and security requirements. Differently from traditional middlewares for distributed programming, the structure of the underlying network is made manifest to components of WAN applications. Indeed, a key design principle of WAN computing is *network awareness*. This is because often applications need to be aware of the administrative domains where they are currently located, and need to know how to cross administrative boundaries and move to other locations. Components of WAN applications are characterized by a highly dynamic behavior and have to deal with the unpredictable changes over time of the network environment (due to the unavailability of connectivity, lack of services, node failures, reconfiguration, and so on). Moreover, nomadic or mobile components must be designed to support heterogeneity and interoperability because they may disconnect from a node and reconnect later to a different node. Therefore, a distinguished feature of WANs and WAN applications is that their overall structure can change dynamically and unpredictably. We refer the interested reader to [Car99] for a comprehensive analysis of the issues related to the design and development of WAN applications.

The problems associated with the development of WAN applications have prompted the study of *new* paradigms and programming languages with mechanisms for handling code and agent mobility, for managing security, and for coordinating and monitoring the use of resources. Mobility provides a suitable abstraction to design and implement WAN applications. The usefulness of mobility emerges when developing both applications for nomadic devices with intermittent access to the network (*physical mobility*), and network services with different access policies (*logical mobility*). Mobility has produced new interaction paradigm [FPV98], that significantly differ from the traditional client-server pattern, and permit exchange of active units of behavior and not just of raw data:

- *Remote Evaluation*: processes send for execution to remote hosts;
- *Code On-Demand*: processes download code from remote hosts to execute it locally;
- *Mobile Agents*: processes can suspend and migrate to new hosts, where they can resume execution.

Among these design paradigms, Code On-Demand is probably the most widely used (e.g. Java Applets). The one based on mobile agents is, instead, the most challenging because it has a number of distinguishing features and poses a number of demands:

- an agent needs an *execution environment*: a server is needed that supplies resources for execution;
- an agent is *autonomous*: it executes independently of the user who created it (*goal driven*);
- an agent is able to detect changes in its operational environment and to act accordingly (*reactivity* and *adaptivity*).

Another interesting feature of mobile agents is the possibility of executing *disconnected operations* [PR98]: a software component may be remotely executed even if the user (its owner) is not connected; if this is the case, the agent may decide to "sleep" and to periodically try to reestablish the connection with its owner. Conversely, the user, when reconnected, may try to *retract* the component back home. In addition to this scenario, *ad hoc networks* [CMC99] allow connection of nomadic devices without a fixed network structure and *peer-to-peer architectures* (e.g. Napster and Gnutella) introduce a new pattern for Internet interaction by sharing information, that changes dynamically, among distributed components.

There are a few programming languages and systems that provide basic facilities for mobility. A well-known example is the Java programming language. Another interesting example is provided by Oracle [Ora99], which supports access to a database from a mobile device by exploiting mobile agents. However, current technologies provide only limited solutions to the general treatment of mobility.

At a foundational level, several process calculi have been developed to gain a more precise understanding of network awareness and mobility. We mention the Distributed Join-calculus [FGL$^+$96], the Distributed π-calculus [HR02], the Ambient calculus [CG00], the Seal calculus [CV99], and Nomadic Pict [WS99]. Other foundational models adopt a logical style toward the analysis of mobility. *MobileUnity* [MR98] and *MobAdtl* [FMSS02] are program logics specifically designed to specify and reason about mobile systems exploiting a Unity-like proof system. The aforementioned approaches have improved the formal understanding of the complex mechanisms underlying network awareness.

Some of the above mentioned calculi deal also with the key issue of security, namely *privacy* and *integrity* of data, hosts and agents. It is important to prevent malicious agents from accessing private information or modifying private data. Tools are thus needed that enable sites receiving mobile agents for execution to set demands and limitations to ensure that the agents will not violate privacy or jeopardize the integrity of the information. Similarly, mobile agents need tools to ensure that their execution at other sites will not disrupt them or compromise their security. The problem of modelling resource access control of highly distributed and autonomous components has been faced by exploiting suitable notions of type [HR02,BCC01,CGZ01].

1.1 The Klaim approach

KLAIM (*Kernel Language for Agents Interaction and Mobility*, [DFP98]) is an experimental language specifically designed to program distributed systems made up of several mobile components interacting through multiple distributed tuple spaces. KLAIM components, both stationary and mobile, can explicitly refer and control the spatial structures of the network at any point of their evolution. KLAIM primitives allow programmers to distribute and retrieve data and processes to and from the nodes of a net. Moreover, localities are first-class citizens, they can be dynamically created and communicated over the network and are handled via sophisticated scoping rules.

KLAIM communication model builds over, and extends, Linda's notion of *generative communication* through a single shared tuple space [Gel85]. A tuple space is a multiset of tuples that are sequences of information items. Tuples are *anonymous* and are picked up from tuple spaces by means of a *pattern-matching* mechanism (*associative selection*). Interprocess communication is *asynchronous*: producer (i.e. sender) and consumer (i.e. receiver) of a tuple do not need to synchronize. The Linda model, was originally proposed for parallel programming on isolated machines. Multiple, possibly distributed, tuple spaces have been advocated later [Gel89] to improve modularity, scalability and performance. The obtained communication model has a number of properties that make it appealing for WAN computing (see, e.g., [DWFB97,CCR96,Deu01]). The model permits *time uncoupling* (data life time is independent of the producer process life time), *destination uncoupling* (the producer of a datum does not need to know the future use or the destination of that datum) and *space uncoupling* (communicating processes need to know a single interface, i.e. the operations over the tuple space). The success of the tuple space paradigm is witnessed by the many tuple space based run-time systems, both from industries (e.g. SUN JavaSpaces [Sun99,AFH99] and IBM T Spaces [WMLF98]) and from universities (e.g. PageSpace [CTV$^+$98], WCL [Row98], Lime [PMR99] and TuCSoN [OZ99]).

KLAIM programming paradigm emphasizes a clear separation between the computational level and the net coordinator/administrator level. Intuitively, programmers design computational units (processes and mobile agents), while coordinators design nets. Hence, coordinators manage the initial distribution of processes and set the security policies for controlling access to resources and

mobility of processes. Coordinators have complete control over changes of configuration of the network that may be due to addition/deletion of software components and sites, or to transmission of programs and of sites references.

Thus, differently from other programming notations with explicit mechanisms for distribution and mobility, in KLAIM the network infrastructure is clearly distinguishable from user processes and explicitly modelled. We argue that this feature permits a more accurate handling of WAN applications. Indeed, structuring applications in terms of processes and coordinators provides a clean and a powerful abstraction device for WAN programming. In particular, it is instrumental to define security policies and their enforcement mechanisms.

KLAIM has been implemented [BDP02] by exploiting Java and has proved to be suitable for programming a wide range of distributed applications with agents and code mobility [DFP98,DFP00,BDL03,FMP03].

1.2 This paper

This paper reports our experience in the design and development of KLAIM. Its purpose is to outline the theoretical foundations of the main features of KLAIM and of its programming model together with the description of the implementation efforts which have lead to the current prototype.

The rest of the paper is organized as follows. Section 2 introduces, step by step, the foundations of KLAIM as a process calculus. We start by presenting CKLAIM (*Core* KLAIM), that can be seen as a variant of the π-calculus with process distribution, process mobility, and asynchronous communication of names through shared located repositories. We then continue by introducing μKLAIM (*Micro* KLAIM), that exploits the full power of Linda coordination primitives (tuples and pattern-matching), and move to introducing KLAIM, that is also equipped with higher-order communication and with a naming service facility. The section ends with the presentation of OPENKLAIM, a KLAIM dialect equipped with constructs for explicitly modelling connectivity between network nodes and for handling changes of the network topology. Section 3 defines a temporal logics for μKLAIM that permits specification and verification of dynamic properties of networks (e.g., resource allocation, access to resources and information disclosure). Section 4 introduces two type systems for controlling processes activities, namely access to resources and mobility, in μKLAIM networks. Section 5 introduces HOTKLAIM (*Higher-Order Typed* KLAIM), an enrichment of KLAIM with the powerful abstraction mechanisms and types of system F. This permits to conveniently deal with highly parameterized mobile components and to dynamically enforce host security policies. Section 6 presents O'KLAIM, a linguistic integration of object-oriented features with KLAIM, which is used as the coordination language for exchanging mobile object-oriented code among processes in a network. Section 7 presents X-KLAIM (*eXtended* KLAIM), an experimental programming language obtained by extending KLAIM with a high level syntax (including variable declarations, assignments, conditionals, sequential and iterative process composition). The pragmatics of the language is illustrated by means of simple programming examples which demonstrate how well established programming paradigms for mobile applications can be naturally programmed in KLAIM. Finally, in the last section we draw a few conclusions on our work on KLAIM.

2 Klaim as a process calculus

In this section, we present the foundations of KLAIM as a process calculus. We shall introduce the main features of KLAIM step by step, by defining appropriate process calculi of increasing complexity. The main advantage of the approach is that it provides a scalable context where the semantics of each construct is self-contained and simple. We start by presenting CKLAIM (*Core* KLAIM) a variant of the π-calculus [MPW92] with process distribution, process mobility, and

$N ::=$		NETS	$a ::=$		ACTIONS
	$l :: P$	*single node*		$\mathbf{out}(\ell')@\ell$	*output*
	$l :: \langle l' \rangle$	*located datum*		$\mathbf{in}(T)@\ell$	*input*
	$N_1 \parallel N_2$	*net composition*		$\mathbf{eval}(P)@\ell$	*migration*
				$\mathbf{newloc}(u)$	*creation*
$P ::=$		PROCESSES			
	$\mathbf{nil}$	*null process*	$T ::=$		TEMPLATES
	$a.P$	*action prefixing*		ℓ	*name*
	$P_1 \mid P_2$	*parallel composition*		$!\,u$	*formal*
	A	*process invocation*			

Table 1. cKLAIM syntax

asynchronous communication of names through shared located repositories instead of channel-based communication primitives. Then, we introduce μKLAIM (*Micro* KLAIM) which is obtained by enriching cKLAIM with the full power of Linda coordination primitives: tuples and pattern-matching. KLAIM is then obtained by extending μKLAIM with higher-order communication and with a naming service facility. Finally, we present OPENKLAIM, a KLAIM dialect with constructs for explicitly modeling connectivity between network nodes and for handling changes in the network topology.

2.1 cKlaim

The syntax of cKLAIM [GP03a] is reported in Table 1. We assume existence of two disjoint sets: the set $\mathcal{L}$, of *localities*, ranged over by $l, l', l_1, \ldots$, and the set $\mathcal{U}$, of *locality variables*, ranged over by $u, u', u_1, \ldots$. Localities are the addresses (i.e. network references) of nodes and are the syntactic ingredient used to express the idea of administrative domain: computations at a given locality are under the control of a specific authority. Moreover, localities provide the abstract counterpart of *resources* and are the cKLAIM communicable objects. The set of *names* $\mathcal{N}$, ranged over by $\ell, \ell', \ldots$, will denote the union of sets $\mathcal{L}$ and $\mathcal{U}$. Finally, we assume a set $\mathcal{A}$, of *process identifiers*, ranged over by $A, B, \ldots$.

Nets are finite collections of nodes where processes and data can be allocated. *Nodes* are pairs, the first component is a locality (l is the address of the node) and the second component is either a process or a datum.

Processes are the cKLAIM active computational units. They may be executed concurrently either at the same locality or at different localities and can perform four different basic operations, called *actions*. Two actions manage data repositories: adding/withdrawing a datum to/from a repository. One action activates a new thread of execution, viz. a process. The last action permits creation of new network nodes. The latter action is not indexed with an address because it always acts locally; all other actions indicate explicitly the (possibly remote) locality where they will take place. Action **in** exploits *templates* as patterns to select data in shared repositories.

Processes are built up from the special process **nil**, that does not perform any action, and from the basic operations by means of action prefixing, parallel composition and process definition. Recursive behaviours are modelled via process definitions. It is assumed that each process identifier A has a *single* defining equation $A \triangleq P$. Hereafter, we do not explicitly represent equations for process definitions (and their migration to make migrating processes complete), and assume that they are available at any locality of a net.

Names occurring in cKLAIM processes and nets can be *bound*. More precisely, action prefixes $\mathbf{in}(!\,u)@\ell.P$ and $\mathbf{newloc}(u).P$ bind u in P (namely, P is the scope of the bindings made by the

(Com)	$N_1 \parallel N_2 \equiv N_2 \parallel N_1$	(Assoc)	$(N_1 \parallel N_2) \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3)$
(Abs)	$l::P \equiv l::(P\|\mathbf{nil})$	(PrInv)	$l::A \equiv l::P \quad \textit{if } A \triangleq P$
(Clone)	$l::(P_1\|P_2) \equiv l::P_1 \parallel l::P_2$		

Table 2. Structural congruence

(Out)	$l::\mathbf{out}(l'')@l'.P \parallel l'::P' \succ\!\!\longrightarrow l::P \parallel l'::P' \parallel l'::\langle l''\rangle$
(Eval)	$l::\mathbf{eval}(Q)@l'.P \parallel l'::P' \succ\!\!\longrightarrow l::P \parallel l'::P'\|Q$
(In)	$l::\mathbf{in}(T)@l'.P \parallel l'::\langle l''\rangle \succ\!\!\longrightarrow l::P\sigma \parallel l'::\mathbf{nil}$ where $\sigma = \begin{cases} [l''/u] & \text{if } T = !\,u \\ \epsilon & \text{if } T = l'' \end{cases}$
(New)	$L \vdash l::\mathbf{newloc}(u).P \succ\!\!\longrightarrow L \cup \{l'\} \vdash l::P[l'/u] \parallel l'::\mathbf{nil}$ if $l' \notin L$
(Par)	$\dfrac{L \vdash N_1 \succ\!\!\longrightarrow L' \vdash N_1'}{L \vdash N_1 \parallel N_2 \succ\!\!\longrightarrow L' \vdash N_1' \parallel N_2}$
(Struct)	$\dfrac{N \equiv N_1 \quad L \vdash N_1 \succ\!\!\longrightarrow L' \vdash N_2 \quad N_2 \equiv N'}{L \vdash N \succ\!\!\longrightarrow L' \vdash N'}$

Table 3. cKlaim operational semantics

action). A name that is not bound is called *free*. The sets $fn(\cdot)$ and $bn(\cdot)$ (respectively, of free and bound names of a process/net term) are defined accordingly. The set $n(\cdot)$ of names of a term is the union of its sets of free and bound names. As usual, we say that two terms are α-*equivalent*, written $\equiv_\alpha$, if one can be obtained from the other by renaming bound names. Hereafter, we shall work with terms whose bound names are all distinct and different from the free ones. Moreover, we will use σ to range over *substitutions*, i.e. functions with finite domain from locality variables to localities, and write $\circ$ to denote substitutions composition and ϵ to denote the 'empty' substitution.

The operational semantics of cKlaim is given in terms of a structural congruence and of a reduction relation over nets. The *structural congruence*, $\equiv$, identifies nets which intuitively represent the same net. It is defined as the smallest congruence relation over nets that satisfies the laws in Table 2. The structural laws express that $\parallel$ is commutative and associative, that the null process can always be safely removed/added, that a process identifier can be replaced with the body of its definition, and that it is always possible to transform a parallel of co-located processes into a parallel over nodes. Notice that commutativity and associativity of '|' is somehow derived from rules (Com), (Assoc) and (Clone).

The *reduction relation*, $\succ\!\!\longrightarrow$, is the least relation induced by the rules in Table 3. Net reductions are defined over configurations of the form $L \vdash N$, where L is a finite set of names such that $fn(N) \subseteq L \subset \mathcal{N}$. Set L keeps track of the names occurring free in N and is needed to ensure global freshness of new network localities. Whenever a reduction does not generate any fresh addresses we write $N \succ\!\!\longrightarrow N'$ instead of $L \vdash N \succ\!\!\longrightarrow L \vdash N'$.

We now comment on the rules in Table 3. All rules for (possibly remote) process actions require existence of the target node. The assumption that all the equations for process definitions are available everywhere greatly simplifies rule (Eval) because it permits avoiding mechanisms for code inspection to find the process definitions needed by Q. Rule (In) requires existence of the chosen datum in the target node. Moreover, the rule says that action $\mathbf{in}(!\,u)@l'$ looks for any name l'' at l' that is then used to replace the free occurrences of u in the continuation of the process performing the input, while action $\mathbf{in}(l'')@l'$ looks exactly for the name l'' at l'; in both cases, the matched datum is consumed. With abuse of notation, we use **nil** to replace data that have

$N ::=$		NETS	$T ::= F \quad F, T$		TEMPLATES
	$l :: P$	*single node*	$F ::= f \quad !x \quad !u$		TEMPLATE FIELDS
	$l :: \langle et \rangle$	*located tuple*	$t ::= f \quad f, t$		TUPLES
	$N_1 \parallel N_2$	*net composition*	$f ::= e \quad \ell \quad u$		TUPLE FIELDS
			$et ::= ef \quad ef, et$		EVALUATED TUPLES
$a ::=$		ACTIONS	$ef ::= V \quad l$		EVALUATED TUPLE FIELDS
	$\mathbf{out}(t)@\ell$	*output*	$e ::= V \quad x \quad \ldots$		EXPRESSIONS
	$\mathbf{in}(T)@\ell$	*input*			
	$\mathbf{read}(T)@\ell$	*read*			
	$\mathbf{eval}(P)@\ell$	*migration*			
	$\mathbf{newloc}(u)$	*creation*			

Table 4. μKLAIM syntax

been consumed to avoid disappearance of the hosting node (whenever, in the initial configuration, it only contains tuples) due to data consumption. In rule (NEW), the premise exploits the set L to choose a fresh address l' for naming the new node. Notice that the address of the new node is not known to any other node in the net. Hence, it can be used by the creating process as a *private* name. Rule (PAR) says that if part of a net makes a reduction step, the whole net reduces accordingly. Finally, rule (STRUCT), that relates structural congruence and reduction, says that all structural congruent nets can make the same reduction steps.

Process interaction in cKLAIM is asynchronous: no synchronization takes place between sender and receiver processes (only existence of target nodes is checked). Moreover, communication is anonymous and associative because data have no names and are accessed via matching. Intuitively, data could be understood as *services* and matching provides a basic *service discovery* mechanism.

2.2 μKlaim

We now enrich cKLAIM with tuples and pattern-matching (and with a primitive for accessing tuples without consuming them) thus getting μKLAIM [GP03c]. Table 4 illustrates the syntactical categories for μKLAIM that differ from the corresponding ones in the syntax of cKLAIM. We shall use $x, y, z, \ldots$ as generic value variables, and still use ℓ to denote a locality l or a locality variable u.

In μKLAIM, communicable objects (the arguments of **out**) are *tuples*: sequences of actual fields. These contain expressions, localities or locality variables. The *tuple space* (TS, for short) of a node consists of the tuples located there. The precise syntax of *expressions* e is deliberately not specified. We assume that expressions contain, at least, basic values V and variables x. *Templates* are sequences of actual and formal fields, and are used as patterns to select tuples in a tuple space. Formal fields are written $!\,x$ or $!\,u$ and are used to bind variables to values. Notice that, syntactically, templates include tuples.

Processes can also read tuples without removing them from the tuple space by executing action $\mathbf{read}(T)@\ell$. Only evaluated tuples can be added to a TS and templates must be evaluated before they can be used for retrieving tuples. Template evaluation consists in computing the value of the expressions occurring in the template. Localities and formal fields are left unchanged by evaluation. Templates with variables in actual fields cannot be evaluated. We shall write $[\![\, T \,]\!]$ to denote the template resulting from evaluation of T when evaluation succeeds.

To define the operational semantics, we first formalize the pattern-matching mechanism which is used to select (evaluated) tuples from TSs according to (evaluated) templates. The *pattern-matching* function *match* is defined in Table 5. The meaning of the rules is straightforward: an

$$(\mathrm{M}_1)\ \ match(V,V) = \epsilon \qquad (\mathrm{M}_2)\ \ match(!\,x,V) = [V/x]$$

$$(\mathrm{M}_3)\ \ match(l,l) = \epsilon \qquad (\mathrm{M}_4)\ \ match(!\,u,l) = [l/u]$$

$$(\mathrm{M}_5)\ \ \frac{match(eF,ef) = \sigma_1 \qquad match(eT,et) = \sigma_2}{match(\ (eF,eT)\ ,\ (ef,et)\ ) = \sigma_1 \circ \sigma_2}$$

Table 5. Matching rules

evaluated template matches against an evaluated tuple if both have the same number of fields and corresponding fields do match; two values (localities) match only if they are identical, while formal fields match any value of the same type. A successful matching returns a substitution function associating the variables contained in the formal fields of the template with the values contained in the corresponding actual fields of the accessed tuple (of course, in μKLAIM, substitutions can also encompass values and value variables).

While the structural congruence is left unchanged, the reduction relation, $\succ\!\!\longrightarrow$ refines that given in Table 3 for cKLAIM. In the rest of this section, we comment on the different reduction rules. Rule (OUT) becomes

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

and expresses that the tuple resulting from the evaluation of the argument t of **out** is added to the TS at l' (therefore, the **out** can be performed only when t is evaluable). Rule (IN) becomes

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

The rule expresses that the process performing the operation can proceed only if the argument T of **in** is evaluable and pattern-matching succeeds. In this case, the tuple is removed from the TS and the returned substitution is applied to the continuation of the process performing the operation. A similar rule is introduced to model the semantics of action **read**, namely

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

that differs from (IN) just because the accessed tuple is still left in the TS.

2.3 Klaim

We are now able to introduce all features of KLAIM. Table 6 illustrates the syntax of the calculus that differs from the corresponding part of the syntax of μKLAIM; in particular, the productions for nets replace those in Table 1 and the productions for tuple fields replace those in Table 4. As a matter of notation, given a grammar such as $e ::= p_1 \mid \ldots \mid p_m$, we write $e \mathrel{+}= p_{m+1} \mid \ldots \mid p_{m+n}$ as a shorthand for $e ::= p_1 \mid \ldots \mid p_{m+n}$.

A *network node* becomes a term of the form $l::_\rho P$, where ρ is an *allocation environment* that binds the locality variables occurring free in P. Allocation environments provide a name resolution mechanism by mapping locality variables u into localities l. The distinguished locality variable `self` is used by processes to refer to the address of their current hosting node.

Remark 1. This is different from previous presentations of KLAIM where, besides (physical) localities and locality variables, we also used the syntactical category of *logical localities* and defined

$N ::=$		NETS	$P \mathrel{+}= X$	*process variable*
	$l ::_\rho P$	*single node*	$F \mathrel{+}= !\, X$	
	$l :: \langle et \rangle$	*located tuple*	$f \mathrel{+}= P$	
	$N_1 \parallel N_2$	*net composition*	$ef \mathrel{+}= P$	

Table 6. KLAIM syntax

allocation environments as maps from logical localities to (physical) localities. To simplify the resulting calculus, in this paper we preferred to incorporate the syntactical category of logical localities into that of locality variables. □

One significant design choice underlying KLAIM is abstraction of the exact physical allocation of processes and resources over the net. Indeed, in the initial configuration, localities cannot occur in templates/tuples argument of process actions because they cannot occur as actual fields anymore. Therefore, processes have no direct access to nodes and can get knowledge of a locality either through their (local) naming facilities, viz. allocation environment, or by communicating with other processes (which, again, exploit other allocation environments). To this aim, the operational semantics will use localities alike locality variables (i.e. it will be defined over nets generated from an extended syntax that allows localities to occur wherever we can have locality variables).

We say that a net is *well-formed* if for each node $l ::_\rho P$ we have that $\rho(\texttt{self}) = l$, and if for any pair of nodes $l ::_\rho P$ and $l' ::_{\rho'} P'$, $l = l'$ implies $\rho = \rho'$. Hereafter, we will only consider well-formed nets.

The second important extension with respect to μKLAIM is higher-order communication. This feature enables processes to exchange pieces of code through the communication actions. We will explain later how this form of code migration differs from the one provided by **eval**.

As far as the operational semantics is concerned, the structural congruence is modified in the obvious way (thus, it is not shown): the most significant law is $l ::_\rho (P_1 | P_2) \equiv l ::_\rho P_1 \parallel l ::_\rho P_2$. Allocation environments affects the evaluation of templates when evaluating locality variables. To this purpose, the template evaluation function takes as parameter the allocation environment of the node where evaluation takes place. The function has the form $[\![\cdot]\!]_\rho$ and the main clauses of its definition are given below:

$$[\![u]\!]_\rho = \begin{cases} \rho(u) & \text{if } u \in dom(\rho) \\ undef & \text{otherwise} \end{cases} \qquad [\![P]\!]_\rho = P\{\rho\}$$

where $P\{\rho\}$ denotes the process term obtained from P by replacing any free occurrence of a locality variable $u \in dom(\rho)$ that is not within the argument of an **eval** with $\rho(u)$. Process $[\![P]\!]_\rho$ is deemed to be well-defined only if $P\{\rho\}$ does not contain free locality variables outside the arguments of **eval**. Two examples of process evaluation are $[\![out(P)@\ell.Q]\!]_\rho = out([\![P]\!]_\rho)@\rho(\ell).Q\{\rho\}$ and $[\![eval(P)@\ell.Q]\!]_\rho = eval(P)@\rho(\ell).Q\{\rho\}$. We shall write $[\![t]\!]_\rho = et$ to denote that evaluation of tuple t using ρ succeeds and returns the evaluated tuple et.

The most significant rules of the reduction relation are reported in Table 7, where we write $\rho(\ell) = l$ to denote that either $\ell = l$ or ℓ is a locality variable that ρ maps to l. In rule (OUT), the local allocation environment is used both to determine the name of the node where the tuple must be placed and to evaluate the argument tuple. This implies that if the argument tuple contains a field with a process, the corresponding field of the evaluated tuple contains the process resulting from the evaluation of its locality variables. Hence, processes in a tuple are transmitted after the interpretation of their free locality variables through the local allocation environment. This corresponds to having a *static scoping* discipline for the (possibly remote) generation of tuples. A *dynamic linking* strategy is adopted for the **eval** operation, rule (EVAL). In this case the locality variables of the spawned process are not interpreted using the local allocation environment: the

$$\text{(Out)}\quad \frac{\rho(\ell) = l' \qquad [\![\, t \,]\!]_\rho = et}{l::_\rho \mathbf{out}(t)@\ell.P \parallel l'::_{\rho'} P' \succ\!\!\longrightarrow l::_\rho P \parallel l'::_{\rho'} P' \parallel l'::\langle et\rangle}$$

$$\text{(Eval)}\quad \frac{\rho(\ell) = l'}{l::_\rho \mathbf{eval}(Q)@\ell.P \parallel l'::_{\rho'} P' \succ\!\!\longrightarrow l::_\rho P \parallel l'::_{\rho'} P'|Q}$$

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

$$\text{(Read)}\quad \frac{\rho(\ell) = l' \qquad match([\![\, T \,]\!]_\rho, et) = \sigma}{l::_\rho \mathbf{read}(T)@\ell.P \parallel l'::\langle et\rangle \succ\!\!\longrightarrow l::_\rho P\sigma \parallel l'::\langle et\rangle}$$

$$\text{(New)}\quad \frac{l' \notin L}{L \vdash l::_\rho \mathbf{newloc}(u).P \succ\!\!\longrightarrow L \cup \{l'\} \vdash l::_\rho P[l'/u] \parallel l'::_{\rho[l'/\mathtt{self}]} \mathbf{nil}}$$

Table 7. KLAIM operational semantics

linking of locality variables is done at the remote node. Finally, in rule (NEW), the environment of a new node is derived from that of the creating one with the obvious update for the `self` variable. Therefore, the new node inherits all the bindings of the creating node.

We end this section with a simple example that should throw light on the differences between the two forms of mobility supplied by KLAIM. One form is mobility with static scoping: a process moves along the nodes of a net with a fixed binding of resources. The other form is mobility with dynamic linking: process movements break the links to local resources. For instance, consider a net consisting of two localities l_1 and l_2. A client process C is allocated at locality l_1 and a server process S is allocated at locality l_2. The server can accept processes for execution. The client sends process Q to the server. The code of processes is:

$$C \triangleq \mathbf{out}(Q)@u.\mathbf{nil}$$
$$Q \triangleq \mathbf{in}(''\mathrm{foo}'', !x)@\mathtt{self}.\mathbf{out}(''\mathrm{foo}'', x+1)@\mathtt{self}.\mathbf{nil}$$
$$S \triangleq \mathbf{in}(!X)@\mathtt{self}.X$$

The behaviour of the processes above depends on the meaning of u and `self`. It is the allocation environment that establishes the links between locality variables and localities. Here, we assume that the allocation environment of locality l_1, ρ_1, maps `self` into l_1 and u into l_2, while the allocation environment of locality l_2, ρ_2, maps `self` into l_2. Finally, we assume that the tuple spaces located at l_1 and l_2 both contain the tuple $\langle''\mathrm{foo}'', 1\rangle$. The following KLAIM program represents the net described above:

$$l_1::_{\rho_1} C|\langle''\mathrm{foo}'', 1\rangle \;\parallel\; l_2::_{\rho_2} S|\langle''\mathrm{foo}'', 1\rangle.$$

After the execution of $\mathbf{out}(Q)@u$, the tuple space at locality l_2 contains a tuple where the code of process Q is stored. Indeed, it is the process Q' that is stored in the tuple:

$$Q' \triangleq \mathbf{in}(''\mathrm{foo}'', !x)@l_1.\mathbf{out}(''\mathrm{foo}'', x+1)@l_1.\mathbf{nil}.$$

The locality variables occurring in Q are evaluated using the environment at locality l_1 where the action **out** has been executed. Hence, when executed at the server's locality the mobile process Q increases tuple $''\mathrm{foo}''$ at the client's locality.

In order to move process Q for execution at l_2 without keeping the original linkage to resources, the client code should be $\mathbf{eval}(Q)@u.\mathbf{nil}$. When $\mathbf{eval}(Q)@u$ is executed, Q is spawned at the remote

a	$::=$	ACTIONS	f	$+=$	TUPLE FIELDS
	$\mathbf{out}(t)@\ell$	*output*		$*l$	*Dereferentiation*
	$\mathbf{in}(T)@\ell$	*input*			
	$\mathbf{read}(T)@\ell$	*read*	$\mathbb{C}$	$::=$	NODECOORDINATORS
	$\mathbf{eval}(P)@\ell$	*migration*		P	*(standard) process*
	$\mathbf{bind}(u, l)$	*bind*		$pa.\mathbb{C}$	*action prefixing*
				$\mathbb{C}_1 \mid \mathbb{C}_2$	*parallel composition*
pa	$::=$	PRIVILEGED ACTIONS		$\mathbb{A}$	*node coordinator invocation*
	a	*(standard) action*			
	$\mathbf{newloc}(u, \mathbb{C})$	*creation*	N	$::=$	NETS
	$\mathbf{login}(\ell)$	*login*		$\mathbf{0}$	*empty net*
	$\mathbf{logout}(\ell)$	*logout*		$l ::_{\rho}^{S} \mathbb{C}$	*single node*
	$\mathbf{accept}(u)$	*accept*		$l :: \langle et \rangle$	*located tuple*
				$N_1 \parallel N_2$	*net composition*

Table 8. OPENKLAIM syntax

node *without* evaluating its locality variables according to the allocation environment ρ_1. Thus, the execution of Q will depend only on the allocation environment ρ_2 and Q will increase tuple $''\text{foo}''$ at the server's locality.

2.4 OpenKlaim

In this section, we present an extension of KLAIM, called here OPENKLAIM, that has been first presented in [BLP02] and was specifically designed for enabling users to give more realistic accounts of *open systems*. Indeed, open systems are dynamically evolving structures: new nodes can get connected or existing nodes can disconnect. Connections and disconnections can be temporary and unexpected. Thus, the assumption that the underlying communication network will always be available is too strong. Moreover, since network routes may be affected by restrictions (such as temporary failures or firewall policies), *naming* may not suffice to establish connections or to perform remote operations. Therefore, to make KLAIM suitable for dealing with open systems, the need arises to extend the language with constructs for explicitly modeling connectivity between network nodes and for handling changes in the network topology.

OPENKLAIM is obtained by equipping KLAIM with mechanisms to dynamically update allocation environments and to handle node connectivity, and with a new category of processes, called *nodecoordinators*, that, in addition to standard KLAIM operations, can execute privileged operations that permit establishing new connections, accepting connection requests and removing connections. The new privileged operations can also be interpreted as movement operations: entering a new administrative domain, accepting incoming nodes and exiting from an administrative domain. The KLAIM extensions that lead to OPENKLAIM are reported in Table 8.

OPENKLAIM processes can be thought of as user programs and differs from KLAIM processes in the following three respects.

– When tuples are evaluated, locality names resolution does not take place automatically anymore. Instead, it has to be explicitly required by putting the operator $*$ in front of the locality that has to be evaluated. For instance, $(3, l)$ and $(s, \mathbf{out}(s_1)@s_2.\mathbf{nil})$ are fully-evaluated while $(3, *l)$ and $(*l, \mathbf{out}(l)@\mathtt{self}.\mathbf{nil})$ are not.

– Operation **newloc** cannot be performed by user processes anymore. It is now part of the syntax of node coordinator processes because, when a new node is created, it is necessary to install one such process at it and, for security reasons, user processes cannot be allowed to do this.
– Operation **bind** has been added to enable user processes to enhance local allocation environments with name bindings. For instance, $\mathbf{bind}(u, l)$ enhances the local allocation environment with the pair (u, l).

NodeCoordinators can be thought of as processes written by node managers, a sort of superusers. Thus, in addition to the standard KLAIM operations, such processes can execute coordination operations to establish new connections (viz. $\mathbf{login}(\ell)$), to accept connection requests (viz. $\mathbf{accept}(u)$), and to remove connections (viz. $\mathbf{logout}(\ell)$). These operations are not indexed with a locality, since they always act locally at the node where they are executed. Node coordinators are stationary processes and cannot be used as tuple fields. They are installed at a node either when the node is initially configured or when the node is dynamically created, e.g. when a node coordinator performs $\mathbf{newloc}(u, \mathbb{C})$ (where $\mathbb{C}$ is a node coordinator).

A network node is now either a located tuple $l::\langle et\rangle$ or a 4-tuple of the form $l::^{S}_{\rho}\,\mathbb{C}$, where S gives the set of nodes connected to l and $\mathbb{C}$ is the parallel composition of user and node coordinator processes. A net can be an empty net $\mathbf{0}$, a single node or the parallel composition of two nets N_1 and N_2 with disjoint sets of node addresses (in this setting, we do not use structural congruence, thus we don't have an analogous of rule (CLONE) of Table 2).

If $l::^{S}_{\rho}\,\mathbb{C}$ is a node in the net, then we will say that the nodes in S are *logged in* l and that l is a *gateway* for those nodes. A node can have more than one gateway. Moreover, if l_1 is logged in l_2 and l_2 is logged in l_3 then l_3 is a gateway for l_1 too.

Remark 2. Our approach aims at a clean separation between the coordinator level (made up by node coordinator processes) and the user level (made up by standard processes). This separation has a considerable impact. From an abstract point of view, the coordinator level may represent the network operating system running on a specific computer and the user level may represent the processes running on that computer. The new privileged operations are then system calls supplied by the network operating system. From a more implementative point of view, the coordinator level may represent the part of a distributed application that takes care of the connections to a remote server (if the application is a client) or that manages the connected clients (if the application is a server). The user level then represents the remaining parts of the application that can interact with the coordinator by means of specific protocols. □

To save space, here we do not show the full operational semantics of OPENKLAIM (we refer the interested reader to [BLP02]), rather we show the most significant rules. The semantics of nets, given by the reduction relation $\succ\!\!\longrightarrow$ (partially) defined in Table 11, exploits two labelled transitions: $\xrightarrow[l]{\lambda}\!\!\!\!\rightarrow$, (partially) defined in Table 9, accounts for the execution of standard actions and for the availability of net resources (tuples and nodes); $\xrightarrow{\lambda}\!\!\!\!\rightarrow$, (partially) defined in Table 10, accounts for the execution of privileged actions. Within the transition labels, l indicates the gateway that makes an action possible, while λ represents the intended operation and has the form $\mathbf{x}(l_1, arg, l_2)$, where $\mathbf{x}$ is the operation, l_1 is the node performing the operation, l_2 is the target node, and arg is the argument of $\mathbf{x}$. For instance, $\mathbf{i}(l_1, [\![\, T \,]\!]_{\rho}, l_2)$ represents operation $\mathbf{in}(T)@l_2$ performed at l_1.

Rule (TUPLE) signals the presence of the tuple $\langle et\rangle$ in the tuple space of l and, similarly, rule (NODE) signals the presence of node $l::^{S}_{\rho}\,\mathbb{C}$ in the net. These information are used to enable execution of standard actions different from **bind**. Rule (ENV) permits changing the gateway used by an action. This is important for remote interaction because two nodes can interact only if there exists a node that acts as gateway for both. Moreover, rule (ENV) implements a *name*

$$l::\langle et\rangle \xrightarrow[l]{\langle et\rangle@l} \mathbf{0} \quad (\textsc{Tuple}) \qquad\qquad l::^S_\rho\, \mathbb{C} \xrightarrow[l]{l::^S_\rho \mathbb{C}} \mathbf{0} \quad (\textsc{Node})$$

$$\frac{N_1 \xrightarrow[l_1]{\lambda} N_1' \quad N_2 \xrightarrow[l_2]{l_2::^{\{l_1\}\cup S}_\rho \mathbb{C}} N_2'}{N_1 \parallel N_2 \xrightarrow[l_2]{\lambda\{\rho\}} N_1' \parallel N_2' \parallel l_2::^{\{l_1\}\cup S}_\rho \mathbb{C}} \quad (\textsc{Env})$$

$$l::^S_\rho \mathbf{bind}(u,l_1).\mathbb{C} \xrightarrow[l]{\mathbf{b}(l,u,l_1)} l::^S_{\rho[l_1/u]} \mathbb{C} \quad \textit{if } \rho(u) \textit{ is undefined} \quad (\textsc{Bind})$$

$$l::^S_\rho \mathbf{out}(t)@\ell.\mathbb{C} \xrightarrow[l]{\mathbf{o}(l,[\![\, t\,]\!]_\rho,\rho(\ell))} l::^S_\rho \mathbb{C} \quad (\textsc{Out})$$

$$l::^S_\rho \mathbf{in}(T)@\ell.\mathbb{C} \xrightarrow[l]{\mathbf{i}(l,[\![\, T\,]\!]_\rho,\rho(\ell))} l::^S_\rho \mathbb{C} \quad (\textsc{In})$$

Table 9. Process semantics (sample rules)

$$\frac{l_2 \notin L}{L \vdash l_1::^S_\rho \mathbf{newloc}(u,\mathbb{C}).\mathbb{C}' \xrightarrow{\mathbf{n}(l_1,\mathbb{C},l_2)} L\cup\{l_2\} \vdash l_1::^S_\rho \mathbb{C}'[l_2/u]} \quad (\textsc{Newloc})$$

$$l_1::^S_\rho \mathbf{login}(l_2).\mathbb{C} \xrightarrow{\mathbf{lin}(l_1,-,l_2)} l_1::^S_\rho \mathbb{C} \quad (\textsc{Login})$$

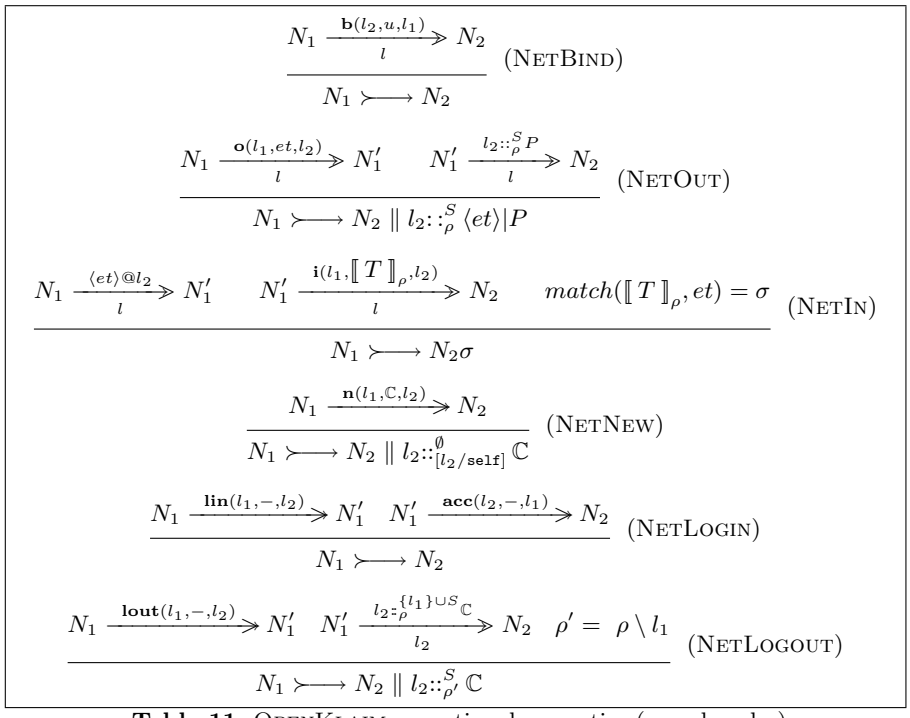
$$l_1::^S_\rho \mathbf{logout}(l_2).\mathbb{C} \xrightarrow{\mathbf{lout}(l_1,-,l_2)} l_1::^S_\rho \mathbb{C} \quad (\textsc{Logout})$$

$$l_1::^S_\rho \mathbf{accept}(u).\mathbb{C} \xrightarrow{\mathbf{acc}(l_1,-,l_2)} l_1::^{S\cup\{l_2\}}_\rho \mathbb{C}[l_2/u] \quad (\textsc{Accept})$$

Table 10. *Node coordinator* semantics (sample rules)

resolution mechanism (akin those of DNS servers): node l_1, that uses l_2 as a gateway, can exploit l_2's allocation environment for resolving localities that it is not able to resolve by itself (this is not shown in detail here but this is what notation $\lambda\{\rho\}$ means). Rules (Bind) and (NetBind) enhance the local allocation environment with the new alias u for l_1. Rules (Out) and (NetOut) model tuple output. To this aim, it is checked existence of the target node (by using rule (Node)) and existence of a gateway shared between the source and the target nodes (by using rule (Env)). Similarly, rules (In) and (NetIn) model communication; in this case, it is checked existence of a matching tuple at the target node (by using rule (Tuple)) and, again, existence of a shared gateway.

Rules (Newloc) and (NetNew) say that $\mathbf{newloc}(u,\mathbb{C})$ creates a new node in the net, binds its address to u in the local allocation environment and installs the node coordinator $\mathbb{C}$ at the new node. Differently from Klaim, the new node does not inherit the binders of the creating node (inheritance could be programmed by appropriately using **bind** in $\mathbb{C}$). We have also that a **newloc** does not automatically log the new node in the generating one. This can be done by installing in the new node a node coordinator that performs a **login**. Rule (Login) says that $\mathbf{login}(l_2)$ logs the executing node l_1 in l_2. Rules (Accept) and (NetLogin) say that, for a $\mathbf{login}(l_2)$ executed at l_1 to succeed, there must be at l_2 a node coordinator process of the form $\mathbf{accept}(l_1).\mathbb{C}'$. As a consequence of this synchronization, l_1 is added to the set S of nodes logged in l_2. Rules (Logout) and (NetLogout) say that $\mathbf{logout}(l_2)$ disconnects the executing node l_1 from l_2; as a consequence, l_1 is removed from the set S of nodes logged in l_2 and any *alias* for l_1 is removed from the allocation environment ρ of l_2 (notation $\rho \setminus l_1$). The second premise of rule (NetLogout) checks existence



$$\frac{N_1 \xrightarrow[l]{\mathbf{b}(l_2,u,l_1)} N_2}{N_1 \rightarrowtail N_2} \text{ (NetBind)}$$

$$\frac{N_1 \xrightarrow[l]{\mathbf{o}(l_1,et,l_2)} N_1' \qquad N_1' \xrightarrow[l]{l_2::^S_\rho P} N_2}{N_1 \rightarrowtail N_2 \parallel l_2::^S_\rho \langle et\rangle|P} \text{ (NetOut)}$$

$$\frac{N_1 \xrightarrow[l]{\langle et\rangle@l_2} N_1' \qquad N_1' \xrightarrow[l]{\mathbf{i}(l_1,[\![T]\!]_\rho,l_2)} N_2 \qquad match([\![T]\!]_\rho, et) = \sigma}{N_1 \rightarrowtail N_2\sigma} \text{ (NetIn)}$$

$$\frac{N_1 \xrightarrow{\mathbf{n}(l_1,\mathbb{C},l_2)} N_2}{N_1 \rightarrowtail N_2 \parallel l_2::^{\emptyset}_{[l_2/\mathtt{self}]} \mathbb{C}} \text{ (NetNew)}$$

$$\frac{N_1 \xrightarrow{\mathbf{lin}(l_1,-,l_2)} N_1' \quad N_1' \xrightarrow{\mathbf{acc}(l_2,-,l_1)} N_2}{N_1 \rightarrowtail N_2} \text{ (NetLogin)}$$

$$\frac{N_1 \xrightarrow{\mathbf{lout}(l_1,-,l_2)} N_1' \quad N_1' \xrightarrow[l_2]{l_2::^{\{l_1\}\cup S}_\rho \mathbb{C}} N_2 \quad \rho' = \rho \setminus l_1}{N_1 \rightarrowtail N_2 \parallel l_2::^S_{\rho'} \mathbb{C}} \text{ (NetLogout)}$$

Table 11. OpenKlaim operational semantics (sample rules)

of a node of the form $l_2::^{\{l_1\}\cup S}_\rho \mathbb{C}$ in N_1' and returns the net N_2 obtained by removing that node from N_1'.

Remark 3. OpenKlaim can be viewed as a core calculus to describe net infrastructures. The calculus can be easily extended with powerful constructs definable atop the basic primitives. For example, a few such constructs have been introduced in [BLP02]. In X-Klaim such derived operations are provided as primitives for efficiency reasons (see Section 7.1). □

The design principles underlying OpenKlaim have been exploited in [DFM+03] to define Kaos, a calculus that can be considered as an extension of μKlaim with OpenKlaim node coordinators. The main peculiarity of Kaos is that connections among nodes are labelled by *costs*, namely special values that abstract connection features. Costs are the formal tool for programming *Quality of Service* (QoS) attributes at the level of WAN applications. Indeed, Kaos costs measure *non-functional* properties (e.g., *timely response* and *security*) that programmers can specify and that depend on the application. The underlying algebraic structure of costs is a *constraint semiring* [BMR97] and this permits performing operations over costs, such as addition and comparison. Hence, it is possible to take into account costs when paths between nodes must be determined.

3 A modal logic for μKlaim

For agent-based calculi, as well as for other formalisms, it is crucial to have tools for establishing deadlock freeness, liveness and correctness with respect to given specifications. However for programs involving different actors and authorities it is also important to establish other properties such as resources allocation, access to resources and information disclosure. In [DL02,Lor02] a temporal logics has been proposed for specifying and verifying dynamic properties of mobile agents

$$\text{(Out)}\quad \frac{[\![\, t \,]\!] = et}{l::\mathbf{out}(t)@l'.P \parallel l::P' \succ\!\!\xrightarrow{\mathbf{o}(l,et,l')} l::P \parallel l::P' \parallel l'::\langle et\rangle}$$

$$\text{(Eval)}\quad l::\mathbf{eval}(Q)@l'.P \parallel l'::P' \succ\!\!\xrightarrow{\mathbf{e}(l,Q,l')} l::P \parallel l'::P'|Q$$

$$\text{(In)}\quad \frac{match([\![\, T \,]\!], et) = \sigma}{l::\mathbf{in}(T)@l'.P \parallel l'::\langle et\rangle \succ\!\!\xrightarrow{\mathbf{i}(l,et,l')} l::P\sigma \parallel l'::\mathbf{nil}}$$

$$\text{(Read)}\quad \frac{match([\![\, T \,]\!], et) = \sigma}{l::\mathbf{read}(T)@l'.P \parallel l'::\langle et\rangle \succ\!\!\xrightarrow{\mathbf{r}(l,et,l')} l::P\sigma \parallel l'::\langle et\rangle}$$

$$\text{(New)}\quad \frac{l' \notin L}{L \vdash l::\mathbf{newloc}(u).P \succ\!\!\xrightarrow{\mathbf{n}(l,-,l')} L \cup \{l'\} \vdash l::P[l'/u] \parallel l'::\mathbf{nil}}$$

$$\text{(Par)}\quad \frac{L \vdash N_1 \succ\!\!\xrightarrow{a} L' \vdash N_1'}{L \vdash N_1 \parallel N_2 \succ\!\!\xrightarrow{a} L' \vdash N_1' \parallel N_2}$$

$$\text{(Struct)}\quad \frac{N \equiv N_1 \qquad L \vdash N_1 \succ\!\!\xrightarrow{a} L' \vdash N_2 \qquad N_2 \equiv N'}{L \vdash N \succ\!\!\xrightarrow{a} L' \vdash N'}$$

Table 12. μKLAIM labelled operational semantics

specified in KLAIM. The inspiration for the proposal was Hennessy-Milner Logics [HM85] but it needed significant adaptations due to the richer operating context. In this section, we re-work on the logics of [DL02] and propose a simplified variant of the logic for μKLAIM.

In order to do this, we need to reconsider the operational semantics of μKLAIM that was given as a set of rewriting rules in Section 2. We need here a labelled operational semantics that makes evident the involved localities and the information transmitted over the net. Our labels carry information about the action performed, the localities involved in the action and the transmitted information. Transition labels have the following structure:

$$\mathbf{x}(l_1, arg, l_2),$$

where $\mathbf{x}$ denotes the action performed. The set Lab of transition labels a is defined by the following grammar:

$$a::= \mathbf{o}(l_1, et, l_2) \mid \mathbf{i}(l_1, et, l_2) \mid \mathbf{r}(l_1, et, l_2) \mid \mathbf{e}(l_1, P, l_2) \mid \mathbf{n}(l_1, -, l_2)$$

Locality l_1 denotes the node where the action is executed, while l_2 is the node where the action takes effect. Finally, *arg* is the argument of the action and can be either a tuple or a process. For instance, if a process running at l_1 inserts $[\![\, t \,]\!]$ in the tuple space located at l_2, by executing $\mathbf{out}(t)@l_2$, then the net evolves with a transition whose label is $\mathbf{o}(l_1, [\![\, t \,]\!], l_2)$. The rules of the labelled operational semantics are presented in Table 12. Notice that, the proposed semantics is completely in accordance with the one presented in Table 3. In fact, the rules are the same apart for the labels.

Temporal properties of nets are expressed by means of the *diamond* operator ($\langle\mathcal{A}\rangle\phi$) indexed with a predicate over transition labels. A net N satisfies a formula $\langle\mathcal{A}\rangle\phi$ if there exists a label a and a net N' such that we have: $N \succ\!\!\xrightarrow{a} N'$, a satisfies $\mathcal{A}$ and N' satisfies ϕ.

Specific process predicates are introduced to describe *static* properties of processes that are spawned to be evaluated remotely. These predicates permit specifying accesses to resources (data and nodes) by processes and the causal dependencies of their actions.

$$
\begin{array}{l}
\Phi ::= \mathtt{true} \mid \langle t \rangle @ \ell \mid \langle \mathcal{A} \rangle \phi \mid \kappa \mid \nu\kappa.\phi \mid \phi \vee \phi \mid \neg\phi \\
\mathcal{A} ::= \circ \mid \alpha \mid \mathcal{A}_1 \cap \mathcal{A}_2 \mid \mathcal{A}_1 \cup \mathcal{A}_2 \mid \mathcal{A}_1 - \mathcal{A}_2 \mid \forall u.\mathcal{A} \\
\alpha ::= \mathtt{O}(\ell_1, \ell, \ell_2) \mid \mathtt{I}(\ell_1, \ell, \ell_2) \mid \mathtt{R}(\ell_1, \ell, \ell_2) \mid \mathtt{E}(\ell_1, \mathtt{pp}, \ell_2) \mid \mathtt{N}(\ell_1, -, \ell_2) \\
\mathtt{pp} ::= \mathtt{1_P} \mid \mathtt{ap} \rightarrow \mathtt{pp} \mid \mathtt{pp} \wedge \mathtt{pp} \\
\mathtt{ap} ::= \mathtt{o}(\ell)@\mathtt{lp} \mid \mathtt{i}(T)@\mathtt{lp} \mid \mathtt{r}(T)@\mathtt{lp} \mid \mathtt{e}(\mathtt{pp})@\mathtt{lp} \mid \mathtt{n}(u)
\end{array}
$$

Table 13. The logic for μKLAIM

The logic provides also state formulae for specifying the distribution of resources (i.e. data stored in nodes) in the system.

Below, we introduce syntax and semantics of the logic. We let Φ be the set of logic formulae defined by the grammar of Table 3, where:

- ϕ is used to denote logical formulae that characterize properties of μKLAIM systems;
- κ belongs to the set of logical variables $VLog$;
- $\mathcal{A}$ denotes a *label predicate*, i.e. a predicate that finitely specifies an infinite set of transition labels;
- $\mathtt{pp}$ denotes a *process predicate* that express *static* properties of processes.

In the rest of this section, we explain first syntax and semantics of formulae, then introduce label predicates and their interpretation. We conclude the section with the definition of process predicates.

3.1 Logical formulae

A formula ϕ can be either $\mathtt{true}$, that is satisfied by any net, or a composed formula; a net N satisfies $\phi_1 \vee \phi_2$ if N satisfies either ϕ_1 or ϕ_2, while N satisfies $\neg\phi$ if N does not satisfies ϕ. Specific state formulae ($\langle t \rangle @ \ell$) are introduced for specifying properties related to the data placement over the nodes. N satisfies $\langle t \rangle @ \ell$ if and only if N contains node ℓ and tuple $\langle\!\langle\, t \,\rangle\!\rangle$ is stored in the tuple space located at ℓ. Dynamic properties of μKLAIM systems are specified using the operator *diamond* ($\langle \mathcal{A} \rangle \phi$) that is indexed with predicates specifying properties of transition labels. We will rely on the interpretation function $\mathbb{A}[\![\cdot]\!]$ that will be formally defined later. It interprets each label predicate $\mathcal{A}$ as a set of pairs $\langle a, \sigma \rangle$ where a is a transition label and σ is a substitution.

The intuitive interpretation of $\langle \mathcal{A} \rangle \phi$ will be:

- a net N satisfies $\langle \mathcal{A} \rangle \phi$ if there exist $\langle a, \sigma \rangle \in \mathbb{A}[\![\mathcal{A}]\!]$ and N' such that $N \stackrel{a}{\rightarrowtail} N'$ and N' satisfies $\phi\sigma$;

Recursive formulae $\nu\kappa.\phi$ are used to specify *infinite* properties of systems. To guarantee well that the interpretation function of formulae be well-defined, we shall assume that no variable κ occurs negatively (i.e. under the scope of an odd number of $\neg$ operators) in ϕ.

Other formulae like $[\mathcal{A}]\phi$, $\phi_1 \wedge \phi_2$ or $\mu\kappa.\phi$ can be expressed in ϕ. Indeed $[\mathcal{A}]\phi = \neg\langle \mathcal{A} \rangle \neg\phi$, $\phi_1 \wedge \phi_2 = \neg(\phi_1 \vee \phi_2)$ and $\mu\kappa.\phi = \neg\nu\kappa.\neg\phi[\neg\kappa/\kappa]$. We shall use these derivable formulae as *macros* in ϕ.

The interpretation function of formulae makes use of *logical environments*. A logical environment is a function that, given a logical variable and a substitution, yields a set of nets.

Definition 1. *Let $VLog$ be the set of logical variables, $Subst$ be the set of substitutions and Net be the set of μ*KLAIM *nets, we define the* logical environment *Env as*

$$Env \subseteq [VLog \rightarrow Subst \rightarrow 2^{Net}]$$

$$\mathbb{M}[\![\texttt{true}]\!]\varepsilon\sigma = Net$$
$$\mathbb{M}[\![\kappa]\!]\varepsilon\sigma = \varepsilon(\kappa)\sigma$$
$$\mathbb{M}[\![\langle t\rangle@l]\!]\varepsilon\sigma = \{N | N \equiv N_1 \parallel l::\langle[\![\, t\sigma \,]\!]\rangle\}$$
$$\mathbb{M}[\![\langle\mathcal{A}\rangle\phi]\!]\varepsilon\sigma = \{N | \exists a, \sigma', N'.\ N \overset{a}{\rightarrowtail} N', (a,\sigma') \in \mathbb{A}[\![\mathcal{A}\{\sigma\}]\!], N' \in \mathbb{M}[\![\phi]\!]\varepsilon\sigma' \cdot \sigma\}$$
$$\mathbb{M}[\![\phi_1 \vee \phi_2]\!]\varepsilon\sigma = \mathbb{M}[\![\phi_1]\!]\varepsilon\sigma \cup \mathbb{M}[\![\phi_2]\!]\varepsilon\sigma$$
$$\mathbb{M}[\![\neg\phi]\!]\varepsilon\sigma = Net - \mathbb{M}[\![\phi]\!]\varepsilon\sigma$$
$$\mathbb{M}[\![\nu\kappa.\phi]\!]\varepsilon\sigma = \bigcup\{g | g \subseteq f^{\phi}_{\kappa,\varepsilon}(g)\} \text{ where } f^{\phi}_{\kappa,\varepsilon}(g) = \mathbb{M}[\![\phi]\!]\varepsilon \cdot [\kappa \mapsto g]$$

Table 14. Interpretation function of formulae

We will use ε, sometime with indexes, to denote elements of Env.

The interpretation function $\mathbb{M}[\![\cdot]\!]: \Phi \to Env \to Subst \to 2^{Net}$ that, using a substitution environment and a logical environment, for each $\phi \in \Phi$, yields the set of nets that satisfy ϕ or, equivalently, the set of nets that are *models* for ϕ with respect to given substitution and logical environment. Function $\mathbb{M}[\![\cdot]\!]$ is formally defined in Table 14.

3.2 Label predicates

A label predicate $\mathcal{A}$ is built from *abstract actions* and $\circ$, that denotes the set of all transition labels, by using disjunction $(\cdot \cup \cdot)$, conjunction $(\cdot \cap \cdot)$ and difference $(\cdot - \cdot)$.

Abstract actions denote set of labels by singling out the kind of action performed (**out**, **in**, ...), the localities involved in the transition and the information transmitted. Abstract actions have the same structure of transition labels; but have *process predicates* instead of processes.

Finally, predicates $\forall u.\mathcal{A}$ is used to quantify over localities, where $\langle a, \sigma \cdot [l/u]\rangle$ belongs to $\mathbb{A}[\![\forall u.\mathcal{A}]\!]$ if and only if $\langle a, \sigma\rangle$ belongs to $\mathbb{A}[\![\mathcal{A}[l/u]]\!]$.

Formal interpretation of labels predicates is defined by means of interpretation function $\mathbb{A}[\![\cdot]\!]$. This function takes a label predicate $\mathcal{A}$ and yields a set of pairs $<$*transition label-substitution*$>$. Intuitively, $(a, \sigma) \in \mathbb{A}[\![\mathcal{A}]\!]$ if transition label *a satisfies* $\mathcal{A}$ with respect to the substitution σ. Function $\mathbb{A}[\![\cdot]\!]$ is defined in Table 15.

Notice that, $\forall u$ plays the role of *existential quantification* if it is used inside a $\langle\cdot\rangle$, while it works like an *universal quantification* when used inside $[\cdot]$. Moreover, in $\langle\mathcal{A}\rangle\phi$, $\mathcal{A}$ acts as binder for quantified variables in $\mathcal{A}$ that appear in ϕ.

Process predicates are used for characterizing properties processes involved in the transition. For instance:

- $\mathcal{A}_1 = \texttt{I}(l_1, l, l_2)$ is satisfied by a transition label if a process, located at l_1, retrieves locality l from the tuple space at l_2;
- $\mathcal{A}_2 = \forall u_1 \texttt{I}(u_1, l, l_2)$ is satisfied by a transition label if a process, located at a generic locality, retrieves locality l from the tuple space at l_2;
- $\mathcal{A}_2 - \mathcal{A}_1$ is satisfied by a transition label if a process, that is not located at l_1, retrieves locality l from the tuple space at l_2.

3.3 Process predicates

Process predicates shall be used to specify the kind of accesses to the resources of the net (data and nodes) that a process might perform in a computation. These accesses are composed for specifying their causal dependencies. The causal properties we intend to express for processes are of the form *"first read something and then use the acquired information in some way"*.

$$\mathbb{A}[\![\circ]\!] = Lab$$

$$\mathbb{A}[\![\mathtt{O}(\ell_1, t, \ell_2)]\!] = \{(\mathbf{o}(l_1, t, l_2); \emptyset)\}$$

$$\mathbb{A}[\![\mathtt{I}(\ell_1, T, \ell_2)]\!] = \{(\mathbf{i}(l_1, T, l_2); \emptyset)\}$$

$$\mathbb{A}[\![\mathtt{R}(\ell_1, T, \ell_2)]\!] = \{(\mathbf{i}(l_1, T, l_2); \emptyset)\}$$

$$\mathbb{A}[\![\mathtt{E}(\ell_1, \mathtt{pp}, \ell_2)]\!] = \{(\mathbf{e}(l_1, P, l_2); \emptyset) | P \in \mathbb{P}[\![\mathtt{pp}]\!]\}$$

$$\mathbb{A}[\![\mathtt{N}(\ell_1, -, \ell_2)]\!] = \{(\mathbf{n}(l_1, -, l_2); \emptyset)\}$$

$$\mathbb{A}[\![\mathcal{A}_1 \cup \mathcal{A}_1]\!] = \mathbb{A}[\![\mathcal{A}_1]\!] \cup \mathbb{A}[\![\mathcal{A}_2]\!]$$

$$\mathbb{A}[\![\mathcal{A}_1 \cap \mathcal{A}_1]\!] = \{(a; \sigma_1 \cdot \sigma_2) | (a; \sigma_1) \in \mathbb{A}[\![\mathcal{A}_1]\!], (a; \sigma_2)\mathbb{A}[\![\mathcal{A}_2]\!]\}$$

$$\mathbb{A}[\![\mathcal{A}_1 - \mathcal{A}_2]\!] = \{(a; \sigma) | (a; \sigma) \in \mathbb{A}[\![\mathcal{A}_1]\!], \forall \sigma' \ (a; \sigma') \notin \mathbb{A}[\![\mathcal{A}_2]\!]\}$$

$$\mathbb{A}[\![\forall u.\mathcal{A}]\!] = \bigcup_{l \in \mathcal{L}} \{(a; \sigma \cdot [u/l]) | (a; \sigma) \in \mathbb{A}[\![\mathcal{A}[l/u]]\!]\}$$

Table 15. Label predicates interpretation

We use $\mathtt{1_P}$ for a generic process and $\mathtt{pp}_1 \wedge \mathtt{pp}_2$ for the set of processes that *satisfy* $\mathtt{pp}_1$ and $\mathtt{pp}_2$. A process satisfies $\mathtt{ap} \rightarrow \mathtt{pp}$ if it may perform an access (i.e. an action) that satisfies $\mathtt{ap}$ and use the acquired information as specified by $\mathtt{pp}$. The satisfaction relation between actions ($\mathtt{act}$) and access predicates ($\mathtt{ap}$) is quite intuitive and can be defined inductively as follows:

$\mathbf{out}(t)@\ell_2$ satisfies $\mathtt{o}(t)@\ell_2$
$\mathbf{in}(T)@\ell$ satisfies $\mathtt{i}(T)@\ell$
$\mathbf{read}(T)@\ell$ satisfies $\mathtt{r}(T)@\ell$
$\mathbf{eval}(P)@\ell$ satisfies $\mathtt{e}(\mathtt{pp})@\ell \Leftrightarrow P$ satisfies $\mathtt{pp}$
$\mathbf{newloc}(u)$ satisfies $\mathtt{n}(u)$

Process predicates can be thought of as types that reflect the possible accesses a process might perform along its computation; they also carry information about the possible use of the acquired resources.

To formally define functions $\mathbb{P}[\![\cdot]\!]$ that yields the set of process satisfying a given process predicates, we need to introduce a transition relation for describing possible computations of processes. The operational semantics proposed in Table 12, is not adequate, because it describes the actual computation of nets and processes. The relation we need, instead, has to describe, using a sort of *abstract interpretation*, the structured sequences of actions a process might perform during its computation.

Let $\mathcal{V}$ be a set of variables, we will write $\mathcal{V} \vdash P \xRightarrow{act} Q$ whenever:

- the process P, at some point of its computation, might perform the action act;
- all the actions that syntactically precede act in P, that are execute before act, do not bind variables in $\mathcal{V}$.

Let $P \rightarrow_{\mathcal{V}} Q$ be the relation defined in Table 16, $\mathcal{V} \vdash P \xRightarrow{act} Q$ is inductively defined as follows:

- for every $\mathcal{V}$,

$$\mathcal{V} \vdash act.P \xRightarrow{act} P$$

$$\begin{array}{|ll|}\hline act.P \rightarrow_{\mathcal{V}} P\ (1) & A \rightarrow_{\mathcal{V}} P\ (2) \\ P|Q \rightarrow_{\mathcal{V}} P & P|Q \rightarrow_{\mathcal{V}} Q \\ P+Q \rightarrow_{\mathcal{V}} P & P+Q \rightarrow_{\mathcal{V}} Q \\ \hline \multicolumn{2}{|l|}{(1)\ act \text{ does not bind variables in } \mathcal{V}} \\ \multicolumn{2}{|l|}{(2)\ A \stackrel{def}{=} P} \\ \hline \end{array}$$

Table 16. Abstract interpretation of processes

$$\begin{array}{|l|}\hline \mathbb{P}[\![\mathtt{1_p}]\!] = Proc \\ \\ \mathbb{P}[\![\mathtt{ap} \rightarrow \mathtt{pp}]\!] = \{P | \exists act, Q_1, Q_2: \\ \qquad P \equiv_\alpha Q_1, \mathtt{fv}(\mathtt{ap} \rightarrow \mathtt{pp}) \vdash Q_1 \stackrel{act}{\Longrightarrow} Q_2, act \in \mathbb{AC}[\![\mathtt{ap}]\!], Q_2 \in \mathbb{P}[\![\mathtt{pp}]\!]\} \\ \\ \mathbb{P}[\![\mathtt{pp}_1 \wedge \mathtt{pp}_2]\!] = \mathbb{P}[\![\mathtt{pp}_1]\!] \cap \mathbb{P}[\![\mathtt{pp}_2]\!] \\ \\ \mathbb{AC}[\![\mathtt{o}(t)@\ell]\!] = \{\mathbf{out}(t)@\ell\} \qquad \mathbb{AC}[\![\mathtt{i}(T)@\ell]\!] = \{\mathbf{in}(T)@\ell\} \\ \\ \mathbb{AC}[\![\mathtt{r}(T)@\ell]\!] = \{\mathbf{read}(T)@\ell\} \\ \mathbb{AC}[\![\mathtt{e}(\mathtt{pp})@\ell]\!] = \{\mathbf{eval}(Q)@\ell | Q \in \mathbb{P}[\![\mathtt{pp}]\!]\sigma\} \\ \\ \mathbb{AC}[\![\mathtt{n}(u)]\!] = \{\mathbf{newloc}(u')\} \\ \hline \end{array}$$

Table 17. Process predicates interpretation functions

– if $P \rightarrow_{\mathcal{V}} P'$ and $\mathcal{V} \vdash P' \stackrel{act}{\Longrightarrow} Q$ then

$$\mathcal{V} \vdash P \stackrel{act}{\Longrightarrow} Q$$

The process predicates interpretation function $\mathbb{P}[\![\cdot]\!]$ is inductively defined in Table 17. We will write $P{:}\,\mathtt{pp}$ to denote that $P \in \mathbb{P}[\![\mathtt{pp}]\!]$. Conversely, we will write $\neg(P{:}\,\mathtt{pp})$ whenever $P \notin \mathbb{P}[\![\mathtt{pp}]\!]$. Furthermore, we assume that process predicates are equal up to contraction (i.e. $\mathtt{pp} \wedge \mathtt{pp} = \mathtt{pp}$), commutative and associative properties; for instance $\mathtt{pp}_1 \wedge (\mathtt{pp}_2 \wedge \mathtt{pp}_1) = \mathtt{pp}_1 \wedge \mathtt{pp}_2$.

We would like to remark that process predicates represent *set* of *causal dependent* sequences of *accesses* that a single process *might* perform and not actual computational sequences.

That follows is a typical properties that one can prove using the logic. Let us consider the set of processes that, after reading the name of a locality from l_1, spawn a process to the read locality:

$$\mathtt{i}(!u)@l_1 \rightarrow \mathtt{e}(\mathtt{1_p})@u \rightarrow \mathtt{1_p}$$

This predicate is by

$$\mathbf{in}(!u_1)@l_1.\mathbf{in}(!x)@u_1.\mathbf{eval}(P)@u_1.Q$$

but it is not satisfied by

$$\mathbf{in}(!u_2)@l_1.\mathbf{read}(!u_2)@l_2.\mathbf{eval}(P)@u_2.\mathbf{nil}$$

since no process is evaluated at the locality retrieved from l_1. Indeed a locality from l_1 is retrieved, but the one used to evaluate P is the locality read from l_2.

The process predicate above could be used for specifying a security policies. For instance, one could ask that *never a process that, after reading the name of a locality from l_1, spawns a process to the read locality, is evaluated at site l_2*. This property can be formalized using the following formula:

$$\nu\kappa.[\forall u.\mathtt{E}(u, \mathtt{i}(!u)@l_1 \rightarrow \mathtt{e}(\mathtt{1_p})@u \rightarrow \mathtt{1_p}, l_2)]\mathtt{false} \ \wedge [\circ]\kappa$$

3.4 An automatic tool for supporting analysis

To simplify the analysis of μKLAIM programs, we use the framework KLAIML [Lor02] that permits simulating an μKLAIM program and generating its reachability graph. Moreover, using KLAIML, it is possible to verify whether a program satisfies a formula.

The core of the system, which is implemented in OCaml [LRVD99], consists of two components: `klaimlgraph` and `klaimlprover`. The first one permits analyzing the execution of μKLAIM programs and generating their reachability graphs. The second one, after loading a net N and a formula ϕ, tests the satisfaction of ϕ by N. If the analyzed program has a finite reachability graph, `klaimlprover` exhibits the actual tree structure of the proof either for ϕ or for $\neg\phi$.

The results produced by `klaimlgraph` and `klaimlprover` are stored in XML format. These files can be visualized using the *front-end* components of the system: `jgraphviewer` and `jproofviewer`.

4 Types for access and mobility control in μKlaim

In the design of programming languages for mobile agents, the integration of security mechanisms is a major challenge; indeed, a great effort has been recently devoted to embed security issues within standard programming features. Several sensible language-based security techniques have been proposed in literature, including type systems, control and data flow analysis, in-lined reference monitoring and proof-carrying code; some of these techniques are analyzed and compared in [SMH00].

An important topic deeply investigated for KLAIM is the use of type systems for security [DFP97,DFP99,DFPV00,DFP00,GP03c,GP03b], namely for controlling accesses to tuple spaces and mobility of processes. To better clarify the problems we were faced with, let us consider a simple scenario. Imagine that a publisher P has an on-line repository (implemented by a node whose address is l_R) containing all its available papers. It is then reasonable to want enforcing some minimal security requirements, like, e.g., that only authorized users can read P papers (*secrecy* of P's data) and no user other than P can put/remove papers in l_R (*integrity* of P's data). The only (unsatisfactory) mechanism available in KLAIM for protecting P publications is to make l_R a reserved address; in this way, P can communicate it only to trusted entities. However, the behaviour of this "trusted" entities is out of P's control: they could (maliciously or incidentally) make l_R public and, from then onwards, no security property on P's data can be ensured.

The idea of statically controlling the execution of a program via types dates back in time. The traditional property enforced by types, i.e. *type safety*, implies that every data will be used consistently with its declaration during the computation (e.g., an integer variable will always be assigned integer values). However, to better deal with global computing problems, we generalized traditional types to *behavioural types*. Intuitively, behavioural types are abstractions of process behaviours and provide information about the *capabilities of processes*, namely the operations processes can perform at a specific locality (downloading/consuming a tuple, producing a tuple, activating a process, and creating a new node). By using behavioural types, each KLAIM node comes equipped with a security policy, specified by a net coordinator in terms of execution privileges: the policy of node l describes the actions processes located at l are allowed to execute. Type checking will guarantee that only processes whose intentions match the rights granted by coordinators are allowed to proceed.

In this section we shall summarize the type theory developed for KLAIM and illustrate how to use type systems to enforce the policies mentioned above. For the sake of presentation, we concentrate on μKLAIM and leave aside the treatment of equations for process definitions (we refer the interested reader to the original papers for a full account of the theories presented).

4.1 A capability-based type system

In this section, we illustrate the basic ideas underlying various type systems, increasingly more powerful, developed for μKLAIM. The development of μKLAIM applications proceed in two phases. In the first phase, node administrators assign policies to the nodes of the net, and processes are programmed while ignoring the access rights of the hosting nodes. In the second phase, processes are allocated over the nodes of the net, while type checking their intentions against the policy of the hosting node. Finally, through a mix of both static and dynamic typing, μKLAIM type system guarantees that only processes with intentions that match the access rights as granted by the net coordinators are allowed to proceed.

We start by presenting a basic framework for our type theory; further developments are given is Sections 4.2 and 4.3. As we already said, 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 and n to indicate *capabilities*, where each symbol stands for the operation whose name begins with it; e.g., r denotes the capability of executing a **read** action. We let π to range over subsets of $\{r, i, o, e, n\}$. *Types*, ranged over by δ, are functions mapping localities (and locality variables) into subsets of capabilities. For the sake of readability, types will be written according to the following notation $[\ell_1 \mapsto \pi_1, \ldots, \ell_n \mapsto \pi_n]$. By taking advantage of the fact that types are functions, we express *subtyping* in terms of the standard pointwise inclusion of functions. Hence, we write $\delta_1 \preceq \delta_2$ if $\delta_1(\ell) \subseteq \delta_2(\ell)$ for every $\ell \in \mathcal{L} \cup \mathcal{U}$.

Each node can be decorated with a type, set by the node administrator, that determines the access policy of the node in terms of access rights on the other nodes of the net. For example, 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 at l to perform **eval** actions over l'. From this perspective, subtyping formalizes degrees of restrictions, i.e. if $\delta_1 \preceq \delta_2$, then δ_1 expresses a less permissive policy than δ_2. Hence, the syntax of μKLAIM nets becomes

$$N \ ::= \ l::^{\delta} P \quad \mid \quad l::\langle et \rangle \quad \mid \quad N_1 \parallel N_2$$

Nodes of the form $l::\langle et \rangle$ represent located resources. We assume that the located resources in the initial configuration have been produced by the net coordinator and, then, are reliable (i.e. no checks are needed).

A static type checker verifies whether the processes in the net do comply with the security policies of the nodes where they are allocated. To this aim, two syntactic constructs are now explicitly typed. Firstly, the **newloc** construct becomes **newloc**$(u{:}\,\delta)$, where δ specifies the security policy of the new node. Moreover, template formal parameters are now of the shape $!\,u{:}\,\pi$, where π specifies the access rights corresponding to the operations that the receiving process 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.

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

To give the flavour of our typing inference system, we show and comment on three significant typing rules concerning **eval**, **in** and **newloc** actions. The rules are

$$\frac{e \in \delta(\ell) \qquad \delta|\!\!\vdash_l P}{\delta|\!\!\vdash \mathbf{eval}(Q)@\ell.P} \qquad\qquad \frac{i \in \delta(\ell) \qquad \delta[\widetilde{u \mapsto \pi}]_{!u:\pi \in T}|\!\!\vdash_l P}{\delta|\!\!\vdash \mathbf{in}(T)@\ell.P}$$

$$\frac{n \in \delta(l) \qquad \delta' \preceq \delta[u \mapsto \delta(l)] \qquad \delta[u \mapsto \delta(l)]|\!\!\vdash_l P}{\delta|\!\!\vdash \mathbf{newloc}(u{:}\,\delta').P}$$

In $\delta|\!\!\vdash_l P$, the δ is called *typing environment*; it records the privileges granted to P and provides information about P's free variables. In all rules, the static checker must verify the existence of the privilege for executing the checked action in the current typing environment. When typing $\mathbf{eval}(Q)@\ell.P$, notice that in general nothing can be statically said about the legacy of Q at ℓ. Indeed, ℓ can be a locality variable and, thus, the locality name replacing it (and hence its associated policy) will be known only at run-time. When typing $\mathbf{in}(T)@\ell.P$, the continuation process P can intend to perform actions on the locality variables bound by T. Thus, P must be typed in the environment obtained from δ by adding information about such variables, as stated by T; this is written $\delta[\widetilde{u \mapsto \pi}]_{!u:\pi \in T}$, where $\delta_1[\delta_2]$ denotes the pointwise union of functions δ_1 and δ_2. Thus, the static checking of P in this extended environment will verify that the declarations contained in T for its bound variables will be respected by P. When typing $\mathbf{newloc}(u{:}\,\delta').P$, we assume that the creating node owns over the created one all the privileges it owns on itself (thus, the continuation process P will be typed in the environment δ extended with the association $[u \mapsto \delta(l)]$). Moreover, the check $\delta' \preceq \delta[u \mapsto \delta(l)]$ verifies that the access policy δ' for the new node is in agreement with the policy δ of the node executing the operation.[1]

Type information contained in processes play a crucial role in the operational semantics, thus enabling/disabling process migrations and data communications. This fact is expressed by modifying the new operational rules for actions **eval** and **in**/**read** as follows. The new reduction rule for **eval** is

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

Notice that the process Q must be dynamically typechecked against the policy of node l', now that the target of the migration (and hence its security policy) is known. The reduction rule for creation of new nodes is

$$\frac{l' \notin L}{L \vdash l::^{\delta} \mathbf{newloc}(u{:}\,\delta').P \rightarrowtail L \cup \{l'\} \vdash l::^{\delta[l' \mapsto \delta(l)]} P[l'/u] \parallel l'::^{\delta'[l'/u]} \mathbf{nil}}$$

Notice that in this case all checks have been made statically, but the point here is that a new node with its policy is added and that the policy of the creating of node change accordingly. The reduction rule for action **in** (the corresponding rule for **read** is omitted) becomes:

$$\frac{match_{\delta}(\llbracket\, T \,\rrbracket, et) = \sigma}{l::^{\delta} \mathbf{in}(T)@l'.P \parallel l'::\langle et \rangle \rightarrowtail l::^{\delta} P\sigma \parallel l'::\mathbf{nil}}$$

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 $match_{\delta}$

[1] This check prevents a malicious node l from forging capabilities by creating a new node with more powerful privileges (where, e.g., sending a malicious process that takes advantage of capabilities not owned by l).

simply relies on the following variant of rule (M_4) in Table 5:

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

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'.

By relying on static and dynamic typechecking, we can prove that the type system is *sound*, namely that well-typedness is an invariant of the operational semantics (*subject reduction*) and that well-typed nets are free from run-time errors, caused by misuse of access rights (*type safety*).

Let's now see the impact of type soundness in practice. The protection of P on-line publications can now be programmed very easily: to preserve data secrecy it suffices to assign the privilege $[l_R \mapsto \{r\}]$ only to the authorized nodes, and to preserve data integrity it suffices to assign the privilege $[l_R \mapsto \{i, o\}]$ only to the node associated to P (say a node with address l_P). Indeed, type soundness ensures that P's papers will be read only by processes running in authorized nodes, and that only processes running at l_P will be allowed to modify the repository l_R.

4.2 Dynamic privileges management

The above modeling of the publisher scenario satisfies the requirements that motivated our approach to type discipline μKLAIM. However, it is far from being realistic and usable, especially in e-commerce applications, because of its static nature. In this section, we 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 [GP03c] for full details and additional 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, tuple fields take now the form

$$f ::= \; e \quad \big| \quad \ell{:}\,\mu$$

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 $[\![\,\cdot\,]\!]_\delta$, whose most significant definition rule is

$$\frac{\mu = [l_1 \mapsto \pi_1, \ldots, l_m \mapsto \pi_m] \qquad \mu' = [l_1 \mapsto \pi_1 \cap \delta(l), \ldots, l_m \mapsto \pi_m \cap \delta(l)]}{[\![\, l{:}\,\mu \,]\!]_\delta = l{:}\,\mu'}$$

The operational rule for **out** now becomes

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

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. The new rule for **in** (the rule for **read** is similar) is:

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

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 typechecks $P\sigma$ by also considering such privileges. Its definition relies on the following variants of rules (M_3), (M_4) and (M_5) of Table 5:

$$match_l^\delta(l'{:}\,\mu, l'{:}\,\mu') = \langle [\,], \epsilon\rangle \qquad \frac{\pi \subseteq \delta(l') \cup \mu(l)}{match_l^\delta(!\,u{:}\,\pi, l'{:}\,\mu) = \langle [l' \mapsto \pi], [l'/u]\rangle}$$

$$\frac{match_l^\delta(F, f) = \langle \delta_1, \sigma_1\rangle \quad match_l^\delta(T, t) = \langle \delta_2, \sigma_2\rangle}{match_l^\delta(\ (F,T)\ ,\ (f,t)\ ) = \langle \delta_1[\delta_2], \sigma_1 \circ \sigma_2\rangle}$$

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. Hence, in the new setting the node $l::^{[l'\mapsto\{r\}]}\mathbf{read}(!u{:}\,\{o\})@l'.\mathbf{out}(t)@l'$ turns out to be legal. Action $\mathbf{out}(t)@l'$ can be marked and checked at run-time since, if u would be dynamically replaced with l', l will acquire the privilege $[l' \mapsto \{o\}]$ and the process running at l could proceed; otherwise, the process will be suspended. In this type system, the dynamic acquisition of privileges is exploited exactly for relaxing the static type checking and admitting nodes like l while requiring on (part of) them a dynamic checking.

The static semantics now is built up over the judgement $\delta|\!\!\vdash_l P \ \triangleright\ P'$, 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. Since the new static checker cannot reject anymore those processes intending to perform statically illegal actions, the rules for typing processes shown before must be slightly modified. Thus, e.g., the rule for typing **eval** actions becomes

$$\frac{\delta|\!\!\vdash_l P \ \triangleright\ P'}{\delta|\!\!\vdash_l \mathbf{eval}(Q)@\ell.P \ \triangleright\ mark_\delta(\mathbf{eval}(Q)@\ell).P'}$$

where $mark_\delta(\mathbf{eval}(Q)@\ell)$ is $\underline{\mathbf{eval}}(Q)@\ell$ if $e \notin \delta(\ell)$ and is $\mathbf{eval}(Q)@\ell$ otherwise.

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 $\delta|\!\!\vdash_l P \ \triangleright\ P$ (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 so that the process that is actually sent for execution is that resulting (if any) from the typechecking of the original incoming process, thus such a process contains all necessary marks. Moreover, for taking into account execution of marked actions, the following rules must be added to the previous ones

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

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

where $tgt(a)$ and $cap(a)$ denote, resp., the target locality and the capability associated to action a. In substance, these rules say 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.

By exploiting this more sophisticated type theory, the publisher example can be formulated by using the following net, that models both user and publisher behaviour:

$$\begin{array}{l} l_U ::^{[l_U \mapsto \{i\},\, l_P \mapsto \{o\}]} \mathbf{out}(\text{“}Subsc\text{”}, l_U\colon [l_P \mapsto \{o\}])@l_P.\mathbf{in}(\text{“}Access\text{”}, u\colon \{r\})@l_U.C \;\; \| \\ l_P ::^{[l_P \mapsto \{i\},\, l_R \mapsto \{i,o\}]} *\,\mathbf{in}(\text{“}Subsc\text{”}, !u'\colon \{o\})@l_P.\mathbf{out}(\text{“}Access\text{”}, l_R\colon [u' \mapsto \{r\}])@u' \;\; \| \\ l_R ::^{[\,]} \; \langle paper1\rangle \mid \langle paper2\rangle \mid \ldots \end{array}$$

Process $*P$ stands for $P|P|\ldots$ (i.e. the π-calculus *replication* operator) and can be easily encoded via process definitions. C represents the user usage of the on-line publications, thus it may contain operations like $\mathbf{read}(\ldots)@l_R$ that would be marked by the static inference. This setting is more realistic because the only privileges statically assigned are $[l_R \mapsto \{i, o\}]$ to l_P (to implement data integrity) and $[l_P \mapsto \{o\}]$ to allow the user l_U to require (by possibly paying a certain fee) the subscription to P's publications. It is then l_P that gives l_U the possibility of accessing l_R. Upon completion of the protocol, the net will be

$$\begin{array}{l} l_U ::^{[l_U \mapsto \{i\},\, l_P \mapsto \{o\},\, l_R \mapsto \{r\}]} \; C \quad \| \quad l_R ::^{[\,]} \; \langle paper1\rangle \mid \langle paper2\rangle \mid \ldots \quad \| \\ l_P ::^{[l_P \mapsto \{i\},\, l_C \mapsto \{i,o\},\, l_U \mapsto \{o\}]} *\,\mathbf{in}(\text{“}Subscr\text{”}, !u'\colon \{o\})@l_P.\mathbf{out}(\text{“}Access\text{”}, l_R\colon [u' \mapsto \{r\}])@u' \end{array}$$

Notice that all processes eventually spawned at l_U are then enabled to use the privilege $[l_R \mapsto \{r\}]$.

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. In our example, the user might just buy an 'individual licence'. 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 subscription could have an expiration date, e.g., it could be an annual subscription. Timing information can easily be accommodated in the framework presented by simply assigning privileges a validity duration and by updating these information for taking into account time passing. 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. In our running example, this corresponds to purchasing the right of accessing P's publications a given number of times. 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 to deal with this new scenario. Finally, the granter of a privilege could decide to revoke the privilege previously granted. In our example, P could prohibit l_U from accessing its publications 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 privileges revocation we could annotate privileges dynamically acquired with the granter identity and enable processes to use a new 'revoke' operation.

4.3 Other uses of types

We conclude this short overview on KLAIM types for security by mentioning two variants. The first one enables a more efficient static checking (but is less realistic and heavier to deal with); the second one allows for a finer control of processes activities (but is more complicated). In both

cases, a static type checker is exploited to minimize the number of run-time checks; type soundness is then formulated in terms of the corresponding notion of well-typedness.

The types for KLAIM originally proposed in [DFP99,DFPV00,DFP00] 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 typechecks 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. 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 system presented in Section 4.1 is 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, types can be *recursive*. 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 $[l \mapsto \{i\} \mapsto \bot, l' \mapsto \{o\} \mapsto \bot]$.

In [GP03b] the type system of Section 4.1 has been 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 TS. 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 $\mathbf{in}(!x, !y)@...$.

5 HotKlaim

This section introduces *Higher-Order Typed* KLAIM (HOTKLAIM), and extension of system F [Gir72,Rey74] with primitives from KLAIM. The purpose of HOTKLAIM is to enhance KLAIM with general purpose features, namely the powerful abstraction mechanisms and types of system F, which are orthogonal to network-aware programming. These features allow to deal with highly parameterized mobile components and to dynamically enforce host security policies: types are metadata extracted at run-time and used to express trustiness guarantees. A further extension, called METAKLAIM, is described in [FMP03]. METAKLAIM supports the interleaving of computational activities with meta-programming activities, like dynamic linking and assembling and customization of components, through the use of METAML-style *staging annotations* [TS00,MHP00].

HOTKLAIM borrows the computational paradigm from μKLAIM: a *net* is a collection of *nodes*, and each node is addressed by a *locality* and consists of a multi-set of active *processes* and passive

Types	$t \in \mathsf{T} ::= X \mid L \mid t_1 \to t_2 \mid (t_i \mid i \in m) \mid \forall X.t \mid U \Rightarrow t$
Contexts	$\Gamma \in \mathsf{Ctx} ::= \emptyset \mid \Gamma, X \mid \Gamma, x{:}\,t$
Terms	$e \in \mathsf{E} ::= x \mid l \mid \lambda x{:}\,t.e \mid e_1\ e_2 \mid \mathsf{fix}\ x{:}\,t.e \mid (e_i \mid i \in m) \mid \pi_j\ e \mid op\ e \mid \Lambda X.e \mid e\{t\} \mid p \Rightarrow e$
Patterns	$p \in \mathsf{P} ::= x!t \mid\ = e \mid (p_i \mid i \in m)$

Table 18. Syntax of types and terms

tuples. In HotKlaim terms include localities, processes and tuples, while types include the types L and $(t_i \mid i \in m)$ of localities and tuples. There is no type for processes[2], because process actions can be performed by terms of any type. The primitives of HotKlaim take the following form:

- $spawn(e)$ activates a process (obtained from e) in a parallel thread. Thus $P|Q$ of μKlaim corresponds to $spawn(\lambda\_{:}\,().P); Q$.
- $new(e)$ creates a new locality l, activates a process (obtained from e) at l, and returns l. Thus $new(\lambda u{:}\,L.P)$ corresponds to the sequence of actions $\mathbf{newloc}(u).\mathbf{eval}(P)@u$ of μKlaim.
- $output(l, e)$ adds the value of e to the tuple space (TS) at l (*output* is non-blocking). Thus $\mathbf{out}(t)@\ell.P$ of μKlaim corresponds to $output(\ell, t); P$
- $input(l, p \Rightarrow e)$ accesses the TS located at l to fetch a value v matching p. If such a v exists, it is removed from the TS, and the variables $x!t$ declared in p are replaced within e by the corresponding values in v. Otherwise, the operation is suspended until one becomes available. Thus $\mathbf{in}(T)@\ell.P$ of μKlaim correspods to $input(\ell, T \Rightarrow P)$.

Remark 4. In Klaim the variables declared in a template pattern have no type annotation, because there are only three *types* of variables (values $!x$, localities $!u$, and processes $!X$). In HotKlaim variables can have any type, thus the *input* primitive performs dynamic type-checking, to ensure that a matching v is consistent with the types of variables declared in the pattern. In Klaim there is a primitive $eval(l, e)$ for activating a process at a remote locality l. This primitive for asynchronous process mobility has not been included in HotKlaim for the following reasons:

- *eval* relies on dynamic scoping (a potentially dangerous mechanism), which is not available in HotKlaim, since in a functional setting one can use (the safer mechanism of) parametrization.
- with *eval* a node may activate a process on another node, but the target node has no control over the incoming process. This can be a source of security problems. In particular, Local Type Safety (see below) fails, if *eval* is added.

In HotKlaim process mobility occurs only by "mutual agreement", i.e. a node can *output* a process abstraction in any TS, but the abstraction becomes an active process (at l) only if a process (at l) *inputs* it. Higher-order remote communication is essential to implement this form of mobility. □

In the rest of this section, we will use the following notations and conventions.

- m, n range over the set N of natural numbers. Furthermore, $m \in \mathsf{N}$ is identified with the set $\{i \in \mathsf{N} \mid i < m\}$ of its predecessors.
- $\mathrm{FV}(e)$ is the set of free variables in e. If E is a set of syntactic entities, then E_0 indicates the set of entities in E without free variables.
- $\overline{e}$ ranges over finite sequences of e. $\overline{e}{:}\,t$ is a shorthand for $e_i{:}\,t$ for each e_i in the sequence $\overline{e}$.
- $\mu(A)$ is the set of multisets with elements in A, and $\uplus$ is multiset union.

Table 18 summarizes the syntax of HotKlaim, which uses the following primitive categories

- a numerable set XT of *type variables*, ranged over by $X, \ldots$;

[2] One could identify processes with terms of type ().

– a numerable set X of *term variables*, ranged over by $x, \ldots$;
– a numerable set L of *localities*, ranged over by $l, \ldots$;
– a finite set $\mathsf{Op} = \{spawn, new, output, input\}$ of *local operations*, ranged over by op.

The syntax of HotKlaim can be explained in terms of system F and Klaim (in the following we assume that t, Γ, e, p respectively range over T, Ctx, E and P).

– From system F we borrow functional types $t_1 \to t_2$, abstraction $\lambda x{:}\,t.e$ and application $e_1\ e_2$, and polymorphic types $\forall X.t$, type abstraction $\Lambda X.e$ and instantiation $e\{t\}$.
– From Klaim we borrow localities l of type L, tuples $(e_i | i \in m)$ of type $(t_i | i \in m)$, and the construct $p{\Rightarrow}e$ of type $U{\Rightarrow}t$, which performs pattern matching and dynamic type-checking on *untrusted* values deposited in a TS (in Klaim this construct is bundled with the *input* primitive); the primitives *spawn*, *new*, *output* and *input* are among the local operations Op.
– Finally, we have recursive definitions $\mathsf{fix}\ x{:}\,t.e$ and projections $\pi_j\ e$.

In HotKlaim, we perform a dynamic type check, when we input an untrusted value from a TS, in order to ensure some trustiness guarantees. The type system of HotKlaim is relatively simple, and the guarantees we can express are limited. For instance, we cannot express constraints on the computational effects of a term, such as the ability to spawn new threads or to perform input/output. We circumvent this limitation by allowing only input of *global* values.

$$\text{A term } e \in \mathsf{E}_0 \text{ is } \textbf{global} \overset{\Delta}{\iff} \text{ it has no occurrences of } \textit{local} \text{ operations } op \in \mathsf{Op}. \tag{1}$$

Thus the only way we can turn a global value v into a process (interacting with its environment) is by passing some local operations (possibly in customized form), in other words v must be a higher-order abstraction representing processes parameterized w.r.t. customized local operations.

Remark 5. The use of dynamic type dispatching in a distributed polymorphic programming language has been strongly advocated in [Dug99]. For simplicity, we have chosen not to include dynamic type dispatching in HotKlaim, but it would be a very appropriate extension. One might wonder whether $input(x!t{\Rightarrow}e)$ of HotKlaim is *semantically equivalent* to $\mathsf{typecase}\ \_\ \mathsf{of}\ (x{:}\,t)e$ of [ACPP91,ACPR95]. In fact, they are different! To simplify the comparison we consider a type U of untrusted values, and replace the *input* primitive with a construct $\mathsf{check}\ \_\ \mathsf{against}\ (x!t)e$.

– The type U of untrusted values has the following introduction and elimination rules
$$\frac{\Gamma \vdash}{\Gamma \vdash \langle e\rangle : U} \qquad \frac{\Gamma \vdash v : U \qquad \Gamma, x{:}\,t \vdash e : t'}{\Gamma \vdash \mathsf{check}\ v\ \mathsf{against}\ (x{:}\,t)e : t'}$$
the reduction semantics is $\mathsf{check}\ \langle v\rangle\ \mathsf{against}\ (x{:}\,t)e \longrightarrow e[v/x]$ provided $\emptyset \vdash v : t$, thus at runtime we have to check that v has type t (in the empty context).
– In [ACPP91,ACPR95] the type D of dynamics has similar introduction and elimination rules
$$\frac{\Gamma \vdash e : t}{\Gamma \vdash d(e{:}\,t) : D} \qquad \frac{\Gamma \vdash v : D \qquad \Gamma, x{:}\,t \vdash e : t'}{\Gamma \vdash \mathsf{typecase}\ v\ \mathsf{of}\ (x{:}\,t)e : t'}$$
the reduction semantics is $\mathsf{typecase}\ d(v{:}\,t'')\ \mathsf{of}\ (x{:}\,t)e \longrightarrow e[v/x]$ provided $t'' \equiv t$, thus at runtime we only need to check equality of types.

Therefore, the two mechanisms accomplish different useful tasks. For instance, if we have an untrusted dynamic value $\langle d(v{:}\,t)\rangle$, we must first check that $d(v{:}\,t) : D$ (or equivalently that $v{:}\,t$), and only then we can compare t with other types to decide how to use v safely. □

$$\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash}{\Gamma, X \vdash}\ X \text{ fresh} \qquad \frac{\Gamma \vdash t}{\Gamma, x{:}\, t \vdash}\ x \text{ fresh} \qquad X\ \frac{\Gamma \vdash}{\Gamma \vdash X}\ X \in \Gamma \qquad L\ \frac{\Gamma \vdash}{\Gamma \vdash L}$$

$$\to\ \frac{\Gamma \vdash t_1 \quad \Gamma \vdash t_2}{\Gamma \vdash t_1 \to t_2} \qquad (\_)\ \frac{\Gamma \vdash t_i \quad i \in m}{\Gamma \vdash (t_i | i \in m)} \qquad \forall\ \frac{\Gamma, X \vdash t}{\Gamma \vdash \forall X.t} \qquad U{\Rightarrow}\ \frac{\Gamma \vdash t}{\Gamma \vdash U{\Rightarrow}t}$$

$$\textbf{var}\ \frac{\Gamma \vdash}{\Gamma \vdash x{:}\, t}\ x{:}\, t \in \Gamma \qquad \textbf{loc}\ \frac{\Gamma \vdash}{\Gamma \vdash l{:}\, L} \qquad \textbf{fun}\ \frac{\Gamma, x{:}\, t_1 \vdash e{:}\, t_2}{\Gamma \vdash \lambda x{:}\, t_1.e{:}\, t_1 \to t_2} \qquad \textbf{app}\ \frac{\Gamma \vdash e_1{:}\, t_1 \to t_2 \quad \Gamma \vdash e_2{:}\, t_1}{\Gamma \vdash e_1\ e_2{:}\, t_2}$$

$$\textbf{fix}\ \frac{\Gamma, x{:}\, t \vdash e{:}\, t}{\Gamma \vdash \mathsf{fix}\ x{:}\, t.e{:}\, t} \qquad \textbf{tuple}\ \frac{\Gamma \vdash \quad \{\Gamma \vdash e_i{:}\, t_i \mid i \in m\}}{\Gamma \vdash (e_i | i \in m){:}\, (t_i | i \in m)} \qquad \textbf{proj}\ \frac{\Gamma \vdash e{:}\, (t_i | i \in m)}{\Gamma \vdash \pi_j\ e{:}\, t_j}\ j < m$$

$$\textbf{spawn}\ \frac{\Gamma \vdash e{:}\, () \to t}{\Gamma \vdash \mathit{spawn}\ e{:}\, ()} \qquad \textbf{new}\ \frac{\Gamma \vdash e{:}\, L \to t}{\Gamma \vdash \mathit{new}\ e{:}\, L} \qquad \textbf{input}\ \frac{\Gamma \vdash e{:}\, (L, U{\Rightarrow}t)}{\Gamma \vdash \mathit{input}\ e{:}\, t} \qquad \textbf{output}\ \frac{\Gamma \vdash e{:}\, (L, t)}{\Gamma \vdash \mathit{output}\ e{:}\, ()}$$

$$\textbf{poly}\ \frac{\Gamma, X \vdash e{:}\, t}{\Gamma \vdash \Lambda X.e{:}\, \forall X.t} \qquad \textbf{spec}\ \frac{\Gamma \vdash e{:}\, \forall X.t_2 \quad \Gamma \vdash t_1}{\Gamma \vdash e\{t_1\}{:}\, t_2[{}^{t_1}\!/_X]} \qquad \textbf{case}\ \frac{\Gamma \vdash \overline{e}(p){:}\, L \quad \Gamma, \Gamma(p) \vdash e{:}\, t}{\Gamma \vdash p{\Rightarrow}e{:}\, U{\Rightarrow}t}$$

Table 19. Type system

5.1 A type system

The type system derives judgments of the following forms

- $\Gamma \vdash$, i.e. Γ is a well-formed context
- $\Gamma \vdash t$, i.e. t is a well-formed type
- $\Gamma \vdash e{:}\, t$, i.e. e is a well-formed term of type t

The declarations in a context Γ have the following meaning: X means that the type variable X ranges over types t, while $x{:}\, t$ means that the term variable x ranges over *values* of type t.

Table 19 gives the typing rules. They are standard, except rule **(case)**, which uses some auxiliary notation, namely a context $\Gamma(p)$ and a sequence $\overline{e}(p)$ of terms, defined by induction on $p \in \mathsf{P}$

$p \in \mathsf{P}$	$\Gamma(p) \in \mathsf{Ctx}$	$\overline{e}(p) \in \mathsf{E}^*$
$x!t$	$x{:}\, t$	$\emptyset$
$= e$	$x{:}\, L$	e
$(p_i \mid i \in m)$	$\Gamma(p_0), \ldots, \Gamma(p_{m-1})$	$\overline{e}(p_0), \ldots, \overline{e}(p_{m-1})$

5.2 Operational semantics

A net $N \in \mathsf{Net} \stackrel{\Delta}{=} \mu(\mathsf{L} \times (\mathsf{E}_0 + \mathsf{V}_0 + \{\mathsf{err}\}))$ is a multi-set of pairs consisting of a locality l and either a process term e, or a value $\langle v \rangle$ in the TS, or err indicating that a process at l has crashed. The dynamics of a net is given by a relation $N \succ\!\!\longrightarrow N'$ defined in terms of two transition relations $e \stackrel{a}{\longmapsto} e'$ and $e \longmapsto \mathsf{err}$ for terms: err means that a process has crashed, this is different from node failure (that we do not model), and from a deadlocked process (e.g. a process that is waiting to input a tuple that never arrives). The transitions relations are defined in terms of evaluation contexts (see [WF94]) and reductions $r \stackrel{a}{\longrightarrow} e'$ (and $r \longrightarrow \mathsf{err}$).

Table 20 summarizes the syntactic categories for the operational semantics. Redexes are the subterms where rewriting takes place. Evaluation contexts identify which of the redexes in a term should be evaluated first, namely the hole [] gives the position of such a redex.

In the following we let v, vp, r, E, a range over V, VP, R, EC and A, respectively.

Table 21 defines the reduction $\longrightarrow$ and uses an auxiliary function $match(p, v)$, which returns a closed substitution $\sigma{:}\, \mathsf{X} \stackrel{fin}{\to} \mathsf{V}_0$ or $fail$. Its definition is by induction on $p \in \mathsf{P}$. The base cases

Values	$\mathsf{V} ::= l \mid \lambda x{:}t.e \mid (v_i \mid i \in m) \mid \Lambda X.e \mid vp \Rightarrow e$
Evaluated Patterns	$\mathsf{VP} ::= x!t \mid \ = v \mid (vp_i \mid i \in m)$
Redexes	$\mathsf{R} ::= v_1 v_2 \mid \mathsf{fix}\, x{:}t.e \mid \pi_j\ v \mid op\ v \mid v\{t\}$
Evaluation Contexts	$\mathsf{EC} ::= [] \mid Ee \mid v\ E \mid (\overline{v}, E, \overline{e}) \mid \pi_j\ E \mid op\ E \mid E\{t\} \mid Ep \Rightarrow e$
Evaluation Contexts for patterns	$Ep ::= (\overline{vp}, Ep, \overline{p}) \mid \ = E$
Actions	$\mathsf{A} ::= \tau \mid l{:}e \mid s(e) \mid i(v)@l \mid o(v)@l$ with $e \in \mathsf{E}_0$ and $v \in \mathsf{V}_0$

Table 20. Values, redexes and evaluation contexts

$(\lambda x{:}t.e)\ v_2 \xrightarrow{\tau} e[v_2/x]$	$v_1\ v_2 \longrightarrow \mathsf{err}$ if $v_1 \not\equiv \lambda x{:}t.e$
$\pi_j\ (v_i \mid i \in m) \xrightarrow{\tau} v_j$ if $j < m$	$\pi_j\ v \longrightarrow \mathsf{err}$ if $v \not\equiv (v_i \mid i \in m)$ with $j < m$
$\mathsf{fix}\, x{:}t.e \xrightarrow{\tau} e[\mathsf{fix}\, x{:}t.e/x]$	–
$spawn\ v \xrightarrow{s(v())} ()$	–
$new\ v \xrightarrow{l:(vl)} l$	–
$output\ (l, v) \xrightarrow{o(v)@l} ()$	$output\ v \longrightarrow \mathsf{err}$ if $v \not\equiv (l, v_1)$
$input\ (l, vp \Rightarrow e) \xrightarrow{i(v)@l} e\sigma$ if $match(vp, v) = \sigma$	$input\ v \longrightarrow \mathsf{err}$ if $v \not\equiv (l, vp \Rightarrow e)$
$(\Lambda X.e)\{t\} \xrightarrow{\tau} e[t/X]$	$v\{t\} \longrightarrow \mathsf{err}$ if $v \not\equiv \Lambda X.e$

Table 21. Reductions for actions and symbolic evaluation

are:

p	$match(p, v)$	
$x!t$	$[v/x]$	if $\emptyset \vdash v{:}t$ and v global, otherwise $fail$
$= e$	$\emptyset$	if $v \equiv e \in \mathsf{L}$, otherwise $fail$

match is used by *input* for dynamic type checking of *global* values (see (1), page 28).

We just comment on some of the reduction rules in Table 21 (the others are standard):

- The rules for *spawn*, *new*, *output* and *input* come from KLAIM.
- *input* requires pattern matching and dynamic type-checking of global values. Moreover, *input* may get stuck, e.g. $input(l, x!X \Rightarrow e)$ is stuck because there are no closed values of type X.
- All reductions to err correspond to type-errors.

The transition relation $\longmapsto$ is defined (in terms of $\longrightarrow$) by the following standard rules

$$\frac{r \xrightarrow{a} e'}{E[r] \xmapsto{a} E[e']} \qquad \frac{r \longrightarrow \mathsf{err}}{E[r] \longmapsto \mathsf{err}}$$

The relation $\succ\!\!\longrightarrow$ is defined (in terms of $\longmapsto$) by the following rules

$$\frac{e \longmapsto \mathsf{err}}{N \uplus (l{::}e) \succ\!\!\longrightarrow N \uplus (l{::}\mathsf{err})} \qquad \frac{e \xmapsto{\tau} e'}{N \uplus (l{::}e) \succ\!\!\longrightarrow N \uplus (l{::}e')}$$

$$\frac{e \xmapsto{i(v)@l_2} e'}{N \uplus (l_1{::}e) \uplus (l_2{::}\langle v \rangle) \succ\!\!\longrightarrow N \uplus (l_1{::}e') \uplus (l_2{::}())} \qquad \frac{e \xmapsto{o(v)@l_2} e'}{N \uplus (l_1{::}e) \succ\!\!\longrightarrow N \uplus (l_1{::}e') \uplus (l_2{::}\langle v \rangle)}$$

$$\frac{e \xmapsto{s(e_2)} e_1}{N \uplus (l{::}e) \succ\!\!\longrightarrow N \uplus (l{::}e_1) \uplus (l{::}e_2)} \qquad \frac{e \xmapsto{l_2::e_2} e_1}{N \uplus (l_1{::}e) \succ\!\!\longrightarrow N \uplus (l_1{::}e_1) \uplus (l_2{::}e_2)}\ l_2 \notin L(N) \cup \{l_1\}$$

where $L(N) \stackrel{\Delta}{=} \{l \mid (l{::}\_) \in N\} \subseteq_{fin} \mathsf{L}$ is the set of localities in the net N. The rules have an obvious meaning, we just remark that the side condition in the last rule ensures freshness of l_2.

5.3 Type safety

In order to express the type safety results we introduce two notions of well-formed net: one is *global*, the other is relative to a subset L of nodes.

Global: A net N is well-formed $\stackrel{\Delta}{\Longleftrightarrow}$ $(l::\mathsf{err}) \notin N$, and for every $(l::e) \in N$ exists t s.t. $\emptyset \vdash e:t$.

Local: A net N is well-formed w.r.t. $L \subseteq L(N)$ $\stackrel{\Delta}{\Longleftrightarrow}$ $(l::\mathsf{err}) \notin N$ when $l \in L$, and for every $(l::e) \in N$ with $l \in L$ exists t s.t. $\emptyset \vdash e:t$.

In the definition of well-formed net nothing is said about values $\langle v \rangle$ in a TS, since they are considered untrusted. In fact, processes can fetch such values only through the *input* primitive, which performs dynamic type-checking. Indeed, we have the following theorem about type safety:

If $N \succ\!\!\longrightarrow N'$, then
Global: N well-formed implies N' well-formed
Local: N well-formed w.r.t. L implies N' well-formed w.r.t. L

The type safety theorem then guarantees that a well-formed net will never give rise to type errors. Together with dynamic type checking performed with input operations, these imply that our type system can be used for protecting hosts from imported code, thus ensuring various kinds of host security properties (as in [YH99]).

Remark 6. The local type safety property is enforced by two features of HotKlaim: the dynamic type-checking performed by the input operation (namely *match*), which prevents ill-typed values in a TS to pollute well-typed processes; the absence of Klaim's *eval* primitive, which would allow processes external to L to spawn ill-typed processes at a locality in L. For instance, with an *eval* primitive similar to a 'remote' *spawn* the following net transition would become possible

$$l_{bad}::eval(l_{good}, v_{bad}), l_{good}::\langle v \rangle \succ\!\!\longrightarrow l_{bad}::(), l_{good}::v_{bad}(), l_{good}::\langle v \rangle$$

where v_{bad} is any closed value such that $v_{bad}() \longmapsto \mathsf{err}$. □

5.4 An example: nomadic data collector

We address the issue of protecting host machines from malicious mobile code. Consider a scenario where a user wants to assemble information about a specific item (e.g. the price of certain devices). Part of the behaviour of the user's application strictly depends on this information. However, there are some activities which are independent of it. The user's application exploits the mobility paradigm: a mobile component travels among hosts of the net looking for the required information. For simplicity, we assume that each node of the distributed database contain tuples either of the form `(i,d)`, where `i` is the search key and `d` is the associated data, or of the form `(i,l)`, where `l` is a locality where more data associated to `i` can be searched. We freely use ML-like notations for functions and sequential composition, and write `fn x:t.e` instead of $\lambda x{:}t.e$ and `V X.t` instead of $\forall X.t$.

```
L                    (* localities *)
type Key  = ...      (* authorization keys *)
type Data = ...
(* polymorphic types of local operations input, output, spawn *)
type I = V X. (L,U=>X) -> X
type O = V X. (L,X) -> ()
type S = V X. (() -> X) -> ()
(* polymorphic types of customized operations for input, output *)
type KI = Key -> I
```

```
type KO = Key -> O
(* process abstractions with security checks *)
type EnvK = (L,KI,KO,S)
type CAK = EnvK -> ()
```

The type `CAK` of (mobile) process abstractions is parameterized with respect to the locality where the process will be executed and the customized operations. In other words, the type `EnvK` can be interpreted as the network environment of the process. This environment must be fed with the information about the current location and its local operations. We want to emphasize that the customized operations for communication require an authorization key. In such a way, depending on the value of the key `k` (that below is checked by a function `safe`), the customized operation `in' k` could generate an actual `input` or `()` when the key does not allow to read anything. Customization of the output operation is done similarly.

```
fun in' (k:Key):I = if safe k then input else ()
```

We now discuss the main module of our mobile application: the nomadic data collector. Process abstraction `pca(k,i,u)` is the mobile process which retrieves the required information on the distributed database. The parameter `k` is an authorization key, `i` is a search key, and `u` is the locality where all data associated to `i` should be collected. The behavior of `pca(k,i,u)` is rather intuitive. After being activated, `pca(k,i,u)` spawns a process that perform a *local query* (which removes from the local database data associated to the search key `i`). Then the mobile process forwards the result of the query to the TS located at `u`, and sends copies of itself (i.e. of `pca(k,i,u)`) to localities that may contain data associated to `i`.

```
fun pca(k:Key, i:Data, u:L):CAK =
  fix ca:CAK. fn (self', in', out', spawn'):EnvK .
    spawn' {()} (() => fix p:().
        (in' k) {()} (self', (_=i, x!Data) => (out' k) {Data} (u,x)) ; p);
     fix q:(). (in' k) {()} (self', (_=i, l!L) => (out' k) {CAK} (l,ca)) ; q
```

The process abstraction `pca(k,i,u)` is instantiated and activated by process `execute`, which fetches values of type `CAK`, and activates them by providing a customized environment `env`

```
fun execute (self:L, env:EnvK):() =
  fix exec:(). input (self, X!CAK => spawn (() => X env) ; exec)
```

6 O'Klaim: an object-oriented Klaim

O'KLAIM is a linguistic integration of KLAIM with object-oriented features. The coordination part and the object-oriented part are orthogonal, so that, in principle, such an integration would work for any extension/restriction of KLAIM, from cKLAIM onward, and also for other calculi for mobility and distribution, such as *DJoin* [FGL+96]. O'KLAIM is built following the design of the core calculus MOMI (Mobile Mixins).

6.1 MoMi and O'Klaim

MOMI was introduced in [BBV02] and extended in [BBV03b]. The underlying motivating idea is that standard class-based inheritance mechanisms, which are often used to implement distributed systems, do not scale well to distributed contexts with mobility. MOMI's approach consists in structuring mobile object-oriented code by using mixin-based inheritance (a mixin is an incomplete class parameterized over a superclass, see [BC90,FKF98,ALZ03]); this fits the dynamic and open nature of a mobile code scenario. For example, a downloaded mixin, describing a mobile agent that must access some files, can be completed with a base class in order to provide access

methods specific of the local file system. Conversely, critical operations of a mobile agent, enclosed in a downloaded class, can be redefined by applying a local mixin to it (e.g., in order to restrict access to sensible resources, as in a *sand-box*)[3]. Therefore, MoMi is a combination of a core coordination calculus and an object-oriented mixin-based calculus equipped with types. The key rôle in MoMi's typing is played by a *subtyping* relation that guarantees safe, yet flexible and scalable, code communication, and lifts type soundness of local code to a global type safety property. In fact, we assume that the code that is sent around has been successfully compiled and annotated with its static type. When the code is received on a site (under the hypothesis that the local code has been successfully compiled, too), it is accepted only if its type is subtyping-compliant with the expected one. If the code is accepted, it can be integrated with the local code under the guarantee of no run-time errors, and without requiring any further type checking of the whole code. MoMi's subtyping relation involves not only object subtyping, but also a form of class subtyping and mixin subtyping: therefore, subtyping hierarchies are provided along with the inheritance hierarchies. It is important to notice that we are not violating the design rule of keeping inheritance and subtyping separated, since mixin and class subtyping plays a pivotal role only during the communication, when classes and mixins become genuine run-time polymorphic values.

In synthesis, MoMi consists of:

1. the definition of an object-oriented "surface calculus" with types called Sool (*Surface Object-Oriented Language*), that describes the essential features that an object-oriented language must have to write mixin-based code;
2. the definition of a subtyping relation on the class and mixin types of the above calculus, to be exploited dynamically at communication time;
3. a very primitive coordination language based on a synchronous send/receive mechanism, to put in practice the communication of the mixin-based code among different site.

O'Klaim is the integration of Sool and its subtyping (both described in the next section), within Klaim, which offers a much more sophisticated, complete, and effective coordination mechanism than the toy one of MoMi.

6.2 Sool: syntax, types, and subtyping

In this section we present the object-oriented part of O'Klaim, called Sool. Sool is defined as a standard class-based object-oriented language supporting mixin-based class hierarchies via *mixin definition* and *mixin application*. It is important to notice that specific incarnations of most object-oriented notions (such as, e.g., functional or imperative nature of method bodies, object references, cloning, etc.) are irrelevant in this context, where the emphasis is on the structure of the object-oriented mobile code. Hence, we work here with a basic syntax of the kernel calculus Sool (shown in Table 22), including the essential features a language must support to be O'Klaim's object-oriented component.

Sool expressions offer object instantiation, method call and mixin application; $\diamond$ denotes the mixin application operator. A Sool value, to which an expression reduces, is either an object, which is a (recursive) record $\{m_i = f_i \ ^{i\in I}\}$, or a class definition, or a mixin definition, where $[m_i = f_i \ ^{i\in I}]$ denotes a sequence of method definitions, $[m_k{:}\,\tau_{m_k} \ \mathsf{with} \ f_k \ ^{k\in K}]$ denotes a sequence of method re-definitions, and I, J and K are sets of indexes. Method bodies, denoted here with f (possibly with subscripts), are closed terms/programs and we ignore their actual structure. A mixin can be seen as an abstract class that is parameterized over a (super)class. Let us describe informally the mixin use through a tutorial example:

[3] A more complete example of use of mixins can be found in Section 7.2, implemented in the object-oriented version of X-Klaim.

$$
\begin{array}{rll}
exp ::= & v & \text{(value)} \\
| & \mathsf{new}\ exp & \text{(object creation)} \\
| & exp \Leftarrow m & \text{(method call)} \\
| & exp_1 \diamond exp_2 & \text{(mixin appl.)} \\
v ::= & \{m_i = f_i\ ^{i\in I}\} & \text{(record)} \\
| & x & \text{(variable)} \\
| & \mathsf{class}\ [m_i = f_i\ ^{i\in I}]\ \mathsf{end} & \text{(class def)} \\
| & \mathsf{mixin} & \\
& \quad \mathsf{expect}[m_i{:}\,\tau_{m_i}\ ^{i\in I}] & \\
| & \quad \mathsf{redef}[m_k{:}\,\tau_{m_k}\ \mathsf{with}\ f_k\ ^{k\in K}] & \text{(mixin def)} \\
& \quad \mathsf{def}[m_j = f_j\ ^{j\in J}] & \\
& \mathsf{end} &
\end{array}
$$

Table 22. Syntax of SOOL.

$$
\begin{array}{rl}
\tau ::= & \Sigma \\
| & \mathsf{class}\langle\Sigma\rangle \\
| & \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle \\
\Sigma ::= & \{m_i{:}\,\tau_{m_i}\ ^{i\in I}\}
\end{array}
$$

Table 23. Syntax of types.

$$
\begin{array}{l}
\mathrm{M} = \mathsf{mixin} \\
\quad \mathsf{expect}\ [n{:}\,\tau] \\
\quad \mathsf{redef}\ [m_2{:}\,\tau_2\ \mathsf{with}\ \ldots\ next\ \ldots] \\
\quad \mathsf{def}\ [m_1 = \ldots n() \ldots] \\
\mathsf{end}
\end{array}
\qquad
\begin{array}{l}
\mathrm{C} = \mathsf{class} \\
\quad [n = \ldots \\
\quad m_2 = \ldots] \\
\mathsf{end}
\end{array}
\qquad (\mathsf{new}\ (\mathrm{M} \diamond \mathrm{C})) \Leftarrow m_1()
$$

Each mixin consists of three parts:

1. methods defined in the mixins, like m_1;
2. *expected methods*, like n, that must be provided by the superclass;
3. *redefined methods*, like m_2, where *next* can be used to access the implementation of m_2 in the superclass.

The application $M \diamond C$ constructs a class, which is a subclass of C.

The set $\mathcal{T}$ of types is defined in Table 23. Σ (possibly with a subscript) denotes a record type of the form $\{m_i{:}\,\tau_{m_i}\ ^{i\in I}\}$. As we left method bodies unspecified (see above), we must assume that there is a type system for the underlying part of SOOL to type method bodies and records. We will denote this type derivability with $\Vdash$. Rules for $\Vdash$ are obviously not specified, but $\Vdash$-statements are used as assumptions in other typing rules. The typing rules for SOOL values are in Table 24.

Mixin types, in particular, encode the following information:

1. record types Σ_{new} and Σ_{red} contain the types of the mixin methods (new and redefined, respectively);
2. record type Σ_{exp} contains the *expected* types, i.e., the types of the methods expected to be supported by the superclass;
3. well typed mixins are well formed, in the sense that name clashes among the different families of methods are absent (the last three clauses of the (*mixin*) rule).

The typing rules for SOOL expressions are in Table 25.

Rule (*mixin app*) relies strongly on a subtyping relation $<:$. The subtyping relation rules depend obviously on the nature of the SOOL calculus we choose, but an essential constraint is that it must contain the *subtyping-in-width* rule for record types: $\Sigma_2 \subseteq \Sigma_1 \Rightarrow \Sigma_1 <: \Sigma_2$.

$$\frac{}{\Gamma, x{:}\tau \vdash x{:}\tau}\ (proj) \qquad \frac{\Gamma \quad \{m_i = f_i\ ^{i\in I}\}{:}\{m_i{:}\tau_{m_i}\ ^{i\in I}\}}{\Gamma \vdash \{m_i = f_i\ ^{i\in I}\}{:}\{m_i{:}\tau_{m_i}\ ^{i\in I}\}}\ (rec)$$

$$\frac{\Gamma \vdash \{m_i = f_i\ ^{i\in I}\}{:}\{m_i{:}\tau_{m_i}\ ^{i\in I}\}}{\Gamma \vdash \mathsf{class}\ [m_i = f_i\ ^{i\in I}]\ \mathsf{end}{:}\,\mathsf{class}\langle\{m_i{:}\tau_{m_i}\ ^{i\in I}\}\rangle}\ (class)$$

$$\frac{\begin{array}{c}\Gamma, \bigcup_{i\in I} m_i{:}\tau_{m_i}, \bigcup_{k\in K} m_k{:}\tau_{m_k} \vdash \{m_j = f_j\ ^{j\in J}\}{:}\{m_j{:}\tau_{m_j}\ ^{j\in J}\} \\ \Gamma, \bigcup_{i\in I} m_i{:}\tau_{m_i}, \bigcup_{k\in K} m_k{:}\tau_{m_k}, \bigcup_{j\in J} m_j{:}\tau_{m_j}, \mathsf{next}{:}\tau_{m_r} \quad f_r{:}\tau'_{m_r} \quad \tau'_{m_r} <: \tau_{m_r} \quad \forall r \in K \\ Subj(\Sigma_{new}) \cap Subj(\Sigma_{exp}) = \emptyset \quad Subj(\Sigma_{new}) \cap Subj(\Sigma_{red}) = \emptyset \\ Subj(\Sigma_{red}) \cap Subj(\Sigma_{exp}) = \emptyset\end{array}}{\begin{array}{l}\mathsf{mixin} \\ \quad \mathsf{expect}[m_i{:}\tau_{m_i}\ ^{i\in I}] \\ \Gamma \vdash \quad \mathsf{redef}[m_k{:}\tau_{m_k}\ \mathsf{with}\ f_k\ ^{k\in K}]\ :\ \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle \\ \quad \mathsf{def}[m_j = f_j\ ^{j\in J}] \\ \mathsf{end}\end{array}}\ (mixin)$$

where $\Sigma_{new} = \{m_j{:}\tau_{m_j}\ ^{j\in J}\}, \Sigma_{red} = \{m_k{:}\tau_{m_k}\ ^{k\in K}\}, \Sigma_{exp} = \{m_i{:}\tau_{m_i}\ ^{i\in I}\}$

Table 24. Typing rules for SOOL values

$$\frac{\Gamma \vdash exp{:}\{m_i{:}\tau_{m_i}\ ^{i\in I}\} \qquad j \in I}{\Gamma \vdash exp \Leftarrow m_j{:}\tau_{m_j}}\ (lookup) \qquad \frac{\Gamma \vdash exp{:}\,\mathsf{class}\langle\{m_i{:}\tau_{m_i}\ ^{i\in I}\}\rangle}{\Gamma \vdash \mathsf{new}\ exp{:}\{m_i{:}\tau_{m_i}\ ^{i\in I}\}}\ (new)$$

$$\frac{\begin{array}{l}\Gamma \vdash exp_1{:}\,\mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle \\ \Gamma \vdash exp_2{:}\,\mathsf{class}\langle\Sigma_b\rangle \\ \Sigma_b <: (\Sigma_{exp} \cup \Sigma_{red}) \\ Subj(\Sigma_b) \cap Subj(\Sigma_{new}) = \emptyset\end{array}}{\Gamma \vdash exp_1 \diamond exp_2{:}\,\mathsf{class}\langle\Sigma_b \cup \Sigma_{new}\rangle}\ (mixin\ app)$$

Table 25. Typing rules for SOOL expressions.

An extension of SOOL with *subtyping-in-depth* can be found in a preliminary form in [BBV03b]. Subtyping-in-depth offers a much more flexible communication pattern, but it complicates the object-oriented code exchange for problems similar to the "subtyping-in-depth versus override" matter of the object-based languages (see [AC96,BBV03b] for examples).

We consider $m{:}\tau_1$ and $m{:}\tau_2$ as distinct elements, and $\Sigma_1 \cup \Sigma_2$ is the standard record union. Σ_1 and Σ_2 are considered *equivalent*, denoted by $\Sigma_1 = \Sigma_2$, if they differ only for the order of their pairs $m_i{:}\tau_{m_i}$.

In the rule (*mixin app*), Σ_b contains the type signatures of all methods supported by the superclass to which the mixin is applied. The premises of the rule (*mixin app*) are as follows:

i) $\Sigma_b <: (\Sigma_{exp} \cup \Sigma_{red})$ requires that the superclass provides all the methods that the mixin expects and redefines.

ii) $Subj(\Sigma_b) \cap Subj(\Sigma_{new}) = \emptyset$ guarantees that no name clash will take place during the mixin application.

Notice that the superclass may have more methods than those required by the mixin constraints. Thus, the type of the mixin application expression is a class type containing both the signatures of all the methods supplied by the superclass (Σ_b) and those of the new methods defined by the mixin (Σ_{new}).

$$\frac{\Sigma' <: \Sigma}{\mathsf{class}\langle\Sigma'\rangle \sqsubseteq \mathsf{class}\langle\Sigma\rangle}\ (\sqsubseteq \mathit{class})$$

$$\frac{\Sigma'_{new} <: \Sigma_{new} \qquad \Sigma_{exp} <: \Sigma'_{exp} \qquad \Sigma'_{red} = \Sigma_{red}}{\mathsf{mixin}\langle\Sigma'_{new}, \Sigma'_{red}, \Sigma'_{exp}\rangle \sqsubseteq \mathsf{mixin}\langle\Sigma_{new}, \Sigma_{red}, \Sigma_{exp}\rangle}\ (\sqsubseteq \mathit{mixin})$$

Table 26. Subtype on class and mixin types.

$P ::=$	**nil**			(null process)
	$act.P$			(action prefixing)
	$P_1 \mid P_2$			(parallel composition)
	X			(process variable)
	$A\langle arg_1, \ldots, arg_n\rangle$			(process invocation)
	$\mathsf{def}\, A(id_1{:}\,\tau_1, \ldots, id_n{:}\,\tau_n) = P\ \mathsf{in}\ \ P'$			(process definition)
	$\mathsf{def}\ x = exp\ \mathsf{in}\ \ P$			(object-oriented expression)
$act ::=$	$\mathbf{out}(t)@\ell$	$\mathbf{in}(t)@\ell$	$\mathbf{read}(t)@\ell$	$\mathbf{eval}(P)@\ell$
$t ::=$	f	f, t		
$f ::=$	arg	$!\, id{:}\,\tau$		
$id ::=$	x	X		
$arg ::=$	e	P	ℓ	v

Table 27. O'KLAIM process syntax (see Table 22 for the syntax of *exp* and v; types τ are defined in Table 23). **newloc** is not relevant in this context as it is the same of Section 2.3, so it is omitted.

The key idea of SOOL's typing is the introduction of a novel subtyping relation, denoted by $\sqsubseteq$, defined on class and mixin types. This subtyping relation is used to match dynamically the actual parameter's types against the formal parameter's types during communication. The part of the operational semantics of O'KLAIM, which describes communication formally, is presented in Section 6.5. The subtyping relation $\sqsubseteq$ is defined in Table 26; rule ($\sqsubseteq$ *class*) is naturally induced by the (width) subtyping on record types, while rule ($\sqsubseteq$ *mixin*): permits the subtype to define more 'new' methods; prohibits to override more methods; and enables a subtype to require less expected methods.

6.3 O'Klaim: syntax

O'KLAIM processes are defined formally in Table 27. The reader may note that the operations of **in** (and **read**) and **out**, retrieving from and inserting into a tuple space, are relevant for our purpose (so they need to be modified accordingly), while net composition and the **newloc** operation are not (and therefore they are not discussed here, because they are the same as the ones of KLAIM). In order to obtain O'KLAIM, we extend the KLAIM syntax of tuples t (presented in Section 2.3) to include any object-oriented value v (defined in Table 22). In particular, formal fields are now explicitly typed (typing rules for typing O'KLAIM are in Section 6.4). Indeed, since types are crucial in O'KLAIM, the scope of process definitions is now made explicit by means of the construct $\mathsf{def}\, A(id_1{:}\,\tau_1, \ldots, id_n{:}\,\tau_n) = P\ \mathsf{in}\ \ P'$, where also process formal parameters are explicitly typed. Actions $\mathbf{in}(t)@\ell$ (and $\mathbf{read}(t)@\ell$) and $\mathbf{out}(t)@\ell$ can be used to move object-oriented code (together with the other KLAIM items) from/to a locality ℓ, respectively. Moreover, we add to KLAIM processes the construct $\mathsf{def}\ x = exp\ \mathsf{in}\ \ P$ in order to pass to the sub-process P the result of computing *exp* (for *exp* syntax see Table 22).

$$\frac{}{\Gamma, X\colon \mathsf{proc} \vdash X\colon \mathsf{proc}}\ (proj) \qquad \frac{}{\Gamma \vdash \mathbf{nil}\colon \mathsf{proc}}\ (nil)$$

$$a \equiv \mathbf{in}, \mathbf{read}, \mathbf{out} \qquad \frac{\begin{array}{l}\Gamma \vdash \ell\colon \mathsf{loc} \\ \Gamma \vdash t_i\colon \tau_i \qquad i = 1, \ldots, n \\ \Gamma \cup \mathit{ftypes}(t_1, \ldots, t_n) \vdash P\colon \mathsf{proc}\end{array}}{\Gamma \vdash a(t_1, \ldots, t_n)@\ell.P\colon \mathsf{proc}}\ (receive)$$

$$\mathit{ftypes}(f, t) = \begin{array}{ll}\{id\colon \tau\} \cup \mathit{ftypes}(t) & \text{if } f \equiv !id\colon \tau \\ \mathit{ftypes}(t) & \text{otherwise}\end{array}$$

$$\frac{\Gamma \vdash Q\colon \mathsf{proc} \qquad \Gamma \vdash \ell\colon \mathsf{loc}}{\Gamma \vdash \mathbf{eval}(Q)@\ell.P\colon \mathsf{proc}}\ (\textsc{Eval})$$

$$\frac{\Gamma \vdash P_1\colon \mathsf{proc} \qquad \Gamma \vdash P_2\colon \mathsf{proc}}{\Gamma \vdash (P_1 \mid P_2)\colon \mathsf{proc}}\ (comp) \qquad \frac{\Gamma \vdash exp\colon \tau \qquad \Gamma, x\colon \tau \vdash P\colon \mathsf{proc}}{\Gamma \vdash \mathsf{def}\ x = exp\ \mathsf{in}\ \ P\colon \mathsf{proc}}\ (let)$$

$$\frac{\Gamma, id_1\colon \tau_1, \ldots, id_n\colon \tau_n \vdash P\colon \mathsf{proc} \quad \Gamma, A(id_1\colon \tau_1, \ldots, id_n\colon \tau_n)\colon \mathsf{proc} \vdash P'\colon \mathsf{proc}}{\Gamma \vdash \mathsf{def}\, A(id_1\colon \tau_1, \ldots, id_n\colon \tau_n) = P\ \mathsf{in}\ \ P'\colon \mathsf{proc}}\ (defproc)$$

$$\frac{\Gamma \vdash A(id_1\colon \tau_1, \ldots, id_n\colon \tau_n)\colon \mathsf{proc} \quad \Gamma \vdash arg_i\colon \tau_i' \quad \tau_i' <\colon \tau_i \quad i = 1, \ldots, n}{\Gamma \vdash A\langle arg_1, \ldots, arg_n\rangle\colon \mathsf{proc}}\ (proccall)$$

Table 28. Typing rules for processes

6.4 Typing for O'Klaim

Typing rules for processes are defined in Table 28. O'Klaim type system is not concerned with access rights and capabilities, as it is instead the type system for Klaim presented in Section 4. In the O'Klaim setting, types serve the purpose of avoiding the "message-not-understood" error when merging local and foreign object-oriented code in a site. Thus, we are not interested in typing actions inside processes: from our perspective, an O'Klaim process is well typed when it has type proc, which only means that the object-oriented code that the process may contain is well typed. O'Klaim requires that every process is statically type-checked separately on its site and annotated with its type. In particular, every tuple item t_i that takes part in the information exchange (which may be an object-oriented value) must be decorated with type information, denoted by ${t_i}^{\tau_i}$ (see Table 30). The types of the tuples are built statically by the compiler: notice that only the types of formal arguments in the process definition $\mathsf{def}\, A(id_1\colon \tau_1, \ldots, id_n\colon \tau_n) = P\ \mathsf{in}\ \ P'$ must be given explicitly by the programmer. In a process of the form $\mathbf{in}(!\, id\colon \tau)@\ell.P$, the type τ is used to statically type check the continuation P, where id is possibly used.

6.5 Operational semantics for O'Klaim

The operational semantics of O'Klaim involves two sets of rules. The first set of rules describes how Sool object-oriented expressions reduce to values and is denoted by $\twoheadrightarrow$. We omit here most of the rules because they are quite standard; they can be found in [BBV03a]. However, we want to describe the rule concerning mixin application, that produces a new class containing all the methods which are added and redefined by the mixin and those defined by the superclass. The rule (*mixinapp*) is presented in Table 29. The function *override*, defined below and used by rule

$$\frac{\begin{array}{c} exp_1 \twoheadrightarrow \left(\begin{array}{l}\mathsf{mixin}\\ \mathsf{expect}[m_i\colon \tau_{m_i}\ ^{i\in I}]\\ \mathsf{redef}[m_k\colon \tau_{m_k}\ \mathsf{with}\ f_k\ ^{k\in K}]\\ \mathsf{def}[m_j = f_j\ ^{j\in J}]\\ \mathsf{end}\end{array}\right)\\ exp_2 \twoheadrightarrow \mathsf{class}\ [m_l = f_l\ ^{l\in L}]\ \mathsf{end}\end{array}}{exp_1 \diamond exp_2 \twoheadrightarrow \left(\begin{array}{l}\mathsf{class}\\ [m_j = f_j\ ^{j\in J}] \cup\\ override([m_k = f_k\ ^{k\in K}], [m_l = f_l\ ^{l\in L}])\\ \mathsf{end}\end{array}\right)}$$

Table 29. The (*mixinapp*) operational rule

$$\frac{match(\tau, \tau_i)}{match(!id\colon \tau, t_i{}^{\tau_i})}$$

$$match(\tau_1, \tau_2) = \begin{array}{ll} \tau_1 \sqsubseteq \tau_2 & \text{if } \tau_1 \text{ and } \tau_2 \text{ are mixin or class types}\\ \tau_1 <\colon \tau_2 & \text{otherwise}\end{array}$$

Table 30. Additional matching rules (with $\mathsf{proc} <\colon \mathsf{proc}$)

(*mixinapp*), takes care of introducing in the new class the overridden methods, and of binding the special variable *next* to the implementations provided by the super class in the mixin's redefined method bodies: these "old" method implementations are given a fresh name, denoted by $m_{i'}$. Dynamic binding is then implemented for redefined methods, and old implementations from the super class are basically hidden in the derived class, since they are given a fresh name (this is reflected in the X-Klaim implementation, presented in Section 7.2).

Definition 2. *Given two method sets, ϱ_1 and ϱ_2, the result of override(ϱ_1, ϱ_2) is the method set ϱ_3 defined as follows:*

- *for all $m_i = f_i \in \varrho_2$ such that $m_i \neq m_j$ for all $m_j = f_j \in \varrho_1$, then $m_i = f_i \in \varrho_3$;*
- *for all $m_i = f_i \in \varrho_1$ such that $m_i = f'_i \in \varrho_2$, let $m_{i'}$ be a fresh method name: then $m_{i'} = f'_i \in \varrho_3$ and $m_i = f_i[m_{i'}/next] \in \varrho_3$.*

Notice that name clashes among methods during the application will never take place, since they have already been solved during the typing of mixin application.

The second set of rules for O'Klaim concerns processes and it is a simple extension of the operational semantics of Klaim, presented in Section 2.3. Notice that the O'Klaim's operational semantics must be defined on typed (compiled) processes, because the crucial point is the dynamic matching of types. Indeed, an **out** operation adds a tuple decorated with a (static) type to a tuple space. Conversely, a process can perform an **in** action by synchronizing with a process which represents a matching typed tuple. To this aim, the standard matching predicate for tuples, *match* (Table 5), is extended with an additional rule presented in Table 30.

The additional matching rule uses the static type information, delivered together with the tuple items, in order to dynamically check that the received item is correct with respect to the type of the formal field, say τ. Therefore, an item is accepted if and only if is subtyping-compliant with the expected type of the formal field. Informally speaking, one can accept any class containing more resources than expected. Conversely, any mixin with weaker requests about methods expected from the superclass can be accepted. Note that the subtyping checking is analogous to the one we would perform in a sequential language where mixins and classes could be passed as parameters to

$$\frac{exp \twoheadrightarrow v}{\ell ::_\rho \mathsf{def}\ x = exp\ \mathsf{in}\ \ P \succ\!\!\longrightarrow \ell ::_\rho P[v/x]}\ (def)$$

Table 31. The (def) operational rule

Fig. 1. The framework for X-KLAIM.

methods. In a sequential setting, this dynamic checking might look as a burden, but in a distributed mobile setting the burden seems well-compensated by the added flexibility in communications. Finally, in order to obtain the full O'KLAIM's operational semantics, we must add a rule for $\mathsf{def}\ x = exp\ \mathsf{in}\ \ P$ to KLAIM's operational semantics. The additional rule is presented in Table 31. This rule relies on the reduction relation for object-oriented expressions $\twoheadrightarrow$. No further modification to the semantics of KLAIM is required.

Type safety of the communication results from the (static) type soundness of local and foreign code; there is no need of additional type-checking after a communication takes place. Therefore, we can reuse most of the meta-theory of MOMI (presented in [BBV03a]) to prove O'KLAIM's soundness. This shows the modularity of our approach. A complete proof of a "global soundness property" for O'KLAIM in the spirit of the one proved for MOMI is in progress.

In Section 7.2, an implementation of O'KLAIM in X-KLAIM is sketched. The reader should have noticed that O'KLAIM was left parametric with respect to the object-oriented language underlying SOOL, but a working implementation of O'KLAIM necessitates of: a coordination language; and a language underneath SOOL. Therefore, it is important to notice that X-KLAIM enriched with object-oriented features (following SOOL's design) plays the double role of the coordination language, and of the object-oriented language underneath SOOL (i.e., this means essentially that method bodies are expressed in X-KLAIM).

7 The programming language X-Klaim

X-KLAIM (*eXtended* KLAIM) is an experimental programming language that extends KLAIM with a high level syntax for processes: it provides variable declarations, enriched operations, assignments, conditionals, sequential and iterative process composition.

The implementation of X-KLAIM is based on KLAVA, a Java package that provides the run-time system for X-KLAIM operations, and on a compiler, which translates X-KLAIM programs into Java programs that use KLAVA. The structure of the KLAIM framework is outlined in Figure 1. X-KLAIM can be used to write the higher layer of distributed applications while KLAVA can be seen both as a middleware for X-KLAIM programs and as a Java framework for programming according to the KLAIM paradigm. With this respect, by using KLAVA directly, the programmer is able to exchange, through tuples, any kind of Java object, and implement a more fine-grained kind of mobility. X-KLAIM and KLAVA are available on line at `http://music.dsi.unifi.it`. KLAVA is briefly described in [BDFP98] and presented in detail in [BDP02,Bet03].

RecProcDefs	::= **rec** id formalparams procbody
	| **rec** id formalparams **extern**
	| RecProcDefs **;** RecProcDefs
formalParams	::= [paramlist]
paramlist	::= ϵ | id **:** type | **ref** id **:** type | paramlist **,** paramlist
procbody	::= declpart **begin** proc **end**
declpart	::= ϵ | **declare** decl
decl	::= **const** id **:=** expression
	| **locname** id
	| **var** idlist **:** type
	| decl **;** decl
idlist	::= id | idlist **,** idlist
proc	::= KAction | **nil**
	| id **:=** expression | **var** id **:** type | proc **;** proc
	| **if** boolexp **then** proc **else** proc **endif**
	| **while** boolexp **do** proc **enddo**
	| **forall** Retrieve **do** proc **enddo**
	| procCall | **call** id | **(** proc **)** | **print** exp
KAction	::= **out(** tuple **)@**id | **eval(** proc **)@**id | Retrieve
	| **go@**id | **newloc(** id **)**
Retrieve	::= Block | NonBlock
Block	::= **in(** tuple **)@**id | **read(** tuple **)@**id
NonBlock	::= **inp(** tuple **)@**id | **readp(** tuple **)@**id | Block **within** numexp
boolexp	::= NonBlock | *standard bool exp*
tuple	::= expression | proc | **!** id | tuple **,** tuple
procCall	::= id **(** actuallist **)**
actuallist	::= ϵ | expression | proc | id | actuallist **,** actuallist
expression	::= $*$ expression | *standard exp*
id	::= *string*
type	::= **int** | **str** | **loc** | **logloc** | **phyloc** | **process** | **ts** | **bool**

Table 32. X-KLAIM process syntax. Syntax for other standard terms is omitted.

Remark 7. In [Tuo99] a similar approach has been adopted for `Klada` that is an `Ada95` implementation of a KLAIM-based prototype language. The main peculiarity of `Klada` regards the treatment of dynamic evolving nets. In particular, implementing the **newloc** primitive requires the use of remote access type objects of `Ada95` and the introduction of a unique globally shared name manager. □

X-KLAIM syntax is shown in Table 32. We just describe the more relevant features. Local variables of processes are declared in the **declare** section of the process definition. Standard base types are available (**str**, **int**, etc...) as well as X-KLAIM typical types, such as **loc** for locality variables, **process** for process variables and **ts**, i.e., tuple space, for implementing data structures by means of tuple spaces, e.g. lists, that can be accessed through standard tuple space operations. Finally, comments start with the symbol #.

A locality variable can be initialized with a string that will correspond to its actual value. We distinguish between two kinds of localities (see the remark 1 in Section 2.3): *logical localities* are symbolic names for nodes (the distinct logical locality, `self`, can be used by processes to refer to their execution node); *physical localities* are identifiers through which nodes can be uniquely identified within a net and must have the form <IP_address>:<port>. Thus, a physical locality variable has to be initialized with a string corresponding to an Internet address. The type **loc** represents a generic locality, without specifying whether it is logical or physical, while **logloc**

(resp. **phyloc**) represents a logical (resp. physical) locality. A simple form of subtyping is supplied for locality variables in that **logloc** <: **loc** and **phyloc** <: **loc**. Logical localities that are used as "destination" are evaluated automatically, i.e., if the locality used after the @ is a logical one, it is first translated to a physical locality. Conversely, when tuples are evaluated, locality names resolution does not take place automatically: it has to be explicitly invoked by putting the operator $*$ in front of the locality that has to be evaluated:

l := ∗output; *# retrieve the physical locality associated to output*
out(∗output)@`self`; *# insert the physical locality associated to output*

Apart from standard KLAIM operations, X-KLAIM also provides non-blocking version of the retrieval operations, namely **readp** and **inp**; these act like **read** and **in**, but, in case no matching tuple is found, the executing process does not block but `false` is returned. Indeed, **readp** and **inp** can be used where a boolean expression is expected. These variants, used also in some versions of Linda [CG89], are useful whenever one wants to search for a matching tuple in a tuple space with no risk of blocking. For instance, **readp** can be used to test whether a tuple is present in a tuple space.

Furthermore, a *timeout* (expressed in milliseconds) can be specified for **in** and **read**, through the keyword **within**; the operation is then a boolean expression that can be tested to determine whether the operation succeeded:

if in(!x, !y)@l **within** 2000 **then** ... **else** ... **endif**

Time-outs can be used when retrieving information for avoiding that processes block due to network latency bandwidth or to absence of matching tuples.

It is often useful to iterate over all elements of a tuple space matching a specific template. However, due to the inherent nondeterministic selection mechanism of pattern matching a subsequent **read** (or **readp**) operation may repeatedly return the same tuple, even if several other tuples match. For this reason X-KLAIM provides the construct **forall** that can be used for iterating actions through a tuple space by means of a specific template. Its syntax is:

forall Retrieve **do** proc **enddo**

The informal semantics of this operation is that the loop body "proc" is executed each time a matching tuple is available. Even duplicate tuples are repeatedly retrieved by the **forall** primitive; it is however guaranteed that each tuple is retrieved only once. Notice however that the tuple space is not blocked when the execution of the **forall** is started, thus this operation is not atomic: the set of tuples matching the template can change before the command completes. A locked access to such tuples can be explicitly programmed. Our version of **forall** is different from the one proposed in [BWA94] and is similar to the **all** variations of retrieval operations in *PLinda* [AS92].

Data structures can be implemented by means of the data type **ts**; a variable declared with such type can be considered as a tuple space and can be accessed through standard tuple space operations, apart from **eval** that would not make sense when applied to variables of type **ts**. Furthermore **newloc** has a different semantics when applied to a variable of type **ts**: it empties the tuple space. Then, **forall** can be used to iterate through such data structures.

eval$(P)@l$ starts the process P on the node at locality l; P can be either a process name (and its arguments):

eval(P(`"foo"`, 10))@l

or the code (i.e., the actions) of the process to be executed:

eval(**in**(!i)@`self`; **out**(i)@l2)@l

Processes can also be used as tuple fields, such as in the following code:

NodeDefs	::= ϵ | **nodes** nodedefs **endnodes**
ProcDefs	::= ϵ | RecProcDefs
nodedefs	::= id **::** **{** environment **}** nodeoptions nodeprocdefs
	| nodedefs **;** nodedefs
environment	::= ϵ | id $\sim$ id | environment **,** environment
nodeprocdefs	::= procbody | nodeprocdefs || nodeprocdefs
nodeoptions	::= **class** id | **port** num

Table 33. X-KLAIM node syntax.

out(P("`foo`", 10), **in**(!i)@`self`; **out**(i)@l2)@l

However, in this case, these processes are not started automatically at l: they are simply inserted in its tuple space. They can be retrieved (e.g., by another process executing at l) and explicitly evaluated:

in(!P1, !P2)@`self`;
eval(P1)@`self`;
eval(P2)@`self`

Thus, basically, **eval** provides *remote evaluation* functionalities, while **out** can be used to implement the *code on-demand* paradigm.

X-KLAIM also provides *strong mobility* by means of the action **go**@l [BD01] that makes an agent migrate to l and resume its execution at l from the instruction following the migration action. Thus in the following piece of code an agent retrieves a tuple from the local tuple space, then it migrates to the locality l and inserts the retrieved tuple into the tuple space at locality l:

in(!i, !j)@`self`;
go@l;
out(i, j)@`self`

Also I/O operations in X-KLAIM are implemented as tuple space operations. For instance the logical locality *screen* can be attached (mapped) to the output device. Hence, operation **out**("`foo\n`")@*screen* corresponds to printing the string "`foo\n`" on the screen. Similarly, the locality *keyboard* can be attached to the input device, so that a process can read what the user typed with a **in**(!s)@*keyboard*. Further I/O devices, such as files, printers, etc., can also be handled through the locality abstraction.

A process can execute only on a KLAVA node since in KLAIM nodes are the execution engines. The syntax for defining a node in X-KLAIM is in Table 33. A node is defined by specifying its name (*id*), its allocation environment, some options (described later) and a set of processes running on it. An allocation environment contains the mapping from logical localities to physical localities of the form

logical_locality_variable $\sim$ physical_locality_constant

thus it also implicitly declares the logical locality variables for all the processes defined in the node. Processes defined in a node have the same syntax of Table 32 but they do not have a name, since these processes are visible and accessible only from within the node where they were defined and not in the whole program. Basically the processes defined in a node correspond to the `main` entry point in languages such as Java and C.

With the option **class** it is possible to specify the actual Java class that has to be used for this node, and the option **port** can be used to specify the Internet port where the node is listening. Remember that, together with the IP address of the computer where the node will run, the port number defines the physical locality of the node.

```
rec NewsGatherer[ item : str, retLoc : loc ]
 declare
  var itemVal : str ;
  var nextLoc : loc ;
  var again : bool
 begin
  again := true;
  while again do
   if read( item, !itemVal )@self within 10000 then
    go@retLoc;
    print "found " + itemVal;
    again := false;
   else
    if readp( item, !nextLoc )@self then
     go@nextLoc
    else
     go@retLoc;
     print "search failed";
     again := false
    endif
   endif
  enddo
 end
```

Listing 1: X-KLAIM implementation of a news gatherer using strong mobility.

Now we show a programming example dealing with mobility, implemented in X-KLAIM, namely, a *news gatherer*, that relies on mobile agents for retrieving information on remote sites. We assume that some data are distributed over the nodes of an X-KLAIM net and that each node either contains the information we are searching for, or, possibly, the locality of the next node to visit in the net. This example is inspired by the one of [DFP98].

The agent *NewsGatherer* first tries to read a tuple containing the information we are looking for, if such a tuple is found, the agent returns the result back home; if no matching tuple is found within 10 seconds, the agent tests whether a link to the next node to visit is present at the current node; if such a link is found the agent migrates there and continues the search, otherwise it reports the failure back home. The implementation of this agent exploiting strong mobility (by means of the migration operation **go**) is reported in Listing 1.

7.1 Connectivity actions

X-KLAIM relies on the hierarchical model of OPENKLAIM, presented in Section 2.4. Thus, it also provides all the primitives for explicitly dealing with node connectivity. Consistently with the hierarchical model of KLAIM such actions can be performed only by *node coordinators*.

The syntax of node coordinators is shown in Table 34, and is basically the same of standard X-KLAIM processes (Table 32) apart from the new privileged actions. We briefly comment these actions:

- **login**(*loc*) logs the node where the node coordinator is executing at the node at locality *loc*; **logout**(*loc*) logs the node out from the net managed by the node at locality *loc*. **login** returns **true** if the login succeeds and **false** otherwise.
- **accept**(*l*) is the complementary action of **login** and indeed, the two actions have to synchronize in order to succeed; thus a node coordinator on the server node (the one at which other nodes

NodeCoordinator	::=	**rec** NodeCoordDef
NodeCoordDef	::=	**nodecoord** id formalparams declpart nodecoordbody
	|	**nodecoord** id formalparams **extern**
nodecoordbody	::=	**begin** nodecoordactions **end**
nodecoordaction	::=	*standard process action* | **login(** id **)** | **logout(** id **)**
	|	**accept(** id **)** | **disconnected(** id **)** | **disconnected(** id **,** id **)**
	|	**subscribe(** id **,** id **)** | **unsubscribe(** id **,** id **)**
	|	**register(** id **,** id **)** | **unregister(** id **)**
	|	**newloc(** id **)** | **newloc(** id **,** nodecoordactions **)**
	|	**newloc(** id **,** nodecoordactions **,** num **,** classname **)**
	|	**bind(** id **,** id **)** | **unbind(** id **)**
	|	**dirconnect(** id **)** | **acceptconn(** id **)**

Table 34. X-KLAIM node coordinator syntax. This syntax relies on standard process syntax shown in Table 32

want to log) has to execute **accept**. This action initializes the variable l to the physical locality of the node that is logging. **disconnected**(l) notifies that a node has disconnected from the current node; the physical locality of such node is stored in the variable l. **disconnected** also catches connection failures. Notice that both **accept** and **disconnected** are blocking in that they block the running process until the event takes place. Instead, **logout** does not have to synchronize with **disconnected**.

- **subscribe**(*loc*, *logloc*) is similar to **login**, but it also permits specifying the logical locality (*logloc* is an expression of type **logloc**) with which a node wants to become part of the net coordinated by the node at locality *loc*; this request can fail also because another node has already subscribed with the same logical locality at the same server. **unsubscribe**(*loc*, *logloc*) performs the opposite operation. Notice that, in OPENKLAIM (Section 2.4), these operations are not part of the syntax of the dialect, but they are derived operations; in the implementation we preferred to supply them as primitives.
- **register**(pl, ll), where pl is a physical locality variable and ll is a logical locality variable, is the complementary action of **subscribe** that has to be performed on the server; if the subscription succeeds pl and ll will respectively contain the physical and the logical locality of the subscribed node. The association $pl \sim ll$ is automatically added to the allocation environment of the server. **unregister**(pl, ll) records the unsubscriptions.

bind(*logloc*, *phyloc*) allows to dynamically modify the allocation environment of the current node: it adds the mapping *logloc* $\sim$ *phyloc*. On the contrary, **unbind**(*logloc*) removes the mapping associated to the logical locality *logloc*. **newloc** is a privileged action and is supplied in three forms in order to make programming easier: apart from the standard form that only takes a locality variable, where the physical locality of the new created node is stored, also the form **newloc**(l, *nodecoordinator*) is provided. Since **newloc** does not automatically logs the new created node in the net of the creating node, this second form allows to install a node coordinator in the new node that can perform this action (or other privileged actions). Notice that this is the only way of installing a node coordinator on another node: due to security reasons, node coordinators cannot migrate, and cannot be part of a tuple. In order to provide better programmability, this rule is slightly relaxed: a node coordinator can perform the **eval** of a node coordinator, provided that the destination is `self`. Finally the third form of **newloc** takes two additional arguments: the port number where the new node is going to be listening and the (Java) class of the new node.

```
rec nodecoord SimpleLogin[ server : loc ]
 begin
  if login( server ) then
   print "login successful";
   out("logged", true)@self
  else
   print "login failed!"
  endif
 end

rec nodecoord SimpleLogout[ server : loc ]
 begin
  in("logged", true)@self;
  logout(server);
  print "logged off."
 end
```

```
rec nodecoord SimpleAccept[]
 declare
  var client : phyloc
 begin
  # waiting for clients...
  accept(client);
 end

rec nodecoord SimpleDisconnected[]
 declare
  var client : phyloc
 begin
  # waiting for disconnections...
  disconnected(client);
 end
```

Listing 2: An example showing **login** and **logout** (left) and the corresponding **accept** and **disconnected**.

class	::= **class** id { **declare** fields } methods **end**
mixin	::= **mixin** id { **declare** fields } mixinmethods **end**
field	::= **const** id **:=** expression
	| **locname** id
	| **var** idlist **:** type
type	::= xklaimtype | **object** id | **class** id | **mixin** id
method	::= id **(** parameters **)** { **:** type } { localvars }**begin** methodactions **end**
mixinmethod	::= (**def** | **redef** | **expect**) method
methodaction	::= processaction | **return** exp
exp	::= xklaimexp | **new** exp | exp <> exp | methodcall
processaction	::= xklaimaction | methodcall
methodcall	::= exp **.** id **(** arguments **)** | **next (** arguments **)**

Table 35. X-KLAIM syntax for MOMI features. Symbols of the shape *xxxs*, such as "parameters" and "arguments", are intended as (possibly empty) lists of *xxx*, separated by the appropriate separator.

7.2 Object-oriented features

The object-oriented part of X-KLAIM, based on O'KLAIM (described in Section 6.3), is shown in Table 35 and it is to be considered complementary to the one shown in Table 32. So, the syntax of X-KLAIM processes is extended with object-oriented operations, and basically the syntax of method bodies is the same of the one of an X-KLAIM process (apart from the **return** statement). The syntax of X-KLAIM is extended in order to include mixins, classes, the mixin application operation <>, objects, the object instantiation operation **new**, and the method call, and, by consequence, the Java code generated by the compiler will interact, apart from KLAVA, also with `momi` (a Java package implementing the virtual machine for MOMI). It is important to remark that the package `momi` is independent from the specific mobile code framework (e.g., KLAVA, in our case). Class and mixin fields are always considered *private* and **this** is used to refer to the host object.

Class and mixin names, that are used for specifying a type (e.g., **object** id, **class** id, **mixin** id) in a variable or parameter declaration, are only a shortcut for their actual interface. Thus, when performing type checking and structural subtyping, internally, the compiler replaces a class (resp. mixin) name with the corresponding class (resp. mixin) type. This enables a remote site, for instance, to ask for a class providing specific methods with specific types, without any requirements

on the name of such a class. The same obviously holds for mixins. Obviously there must be some types on which all nodes that want to exchange object-oriented code have to agree upon, and by default these are the basic types.

An object can be declared as follows:

var my_obj : object C

that declares `my_obj` as an object of a class with the interface of `C`. Thus it can be assigned also an object of a class whose interface is a subtype of the one of `C`, as hinted above.

Class and mixins names can be used as expressions for creating objects, for specifying a type (as in the above declaration) and for delivering code to a remote site. Higher-order variables of kind class and mixin can be declared similarly:

var my_class : **class** C;
var my_mixin : **mixin** M

The above declarations state that `my_class` (`my_mixin`) represents a class (mixin) with the same interface of `C` (`M`). Once initialized, these variables can be used where class and mixin names are expected:

my_class := C;
my_obj := **new** my_class; *# same as new C*
my_obj := **new** (my_class <> M); *# provided that the application is well−typed*
my_mixin := M;
my_obj := **new** (my_class <> my_mixin); *# same as above*

An object declaration such as

var my_obj : object M

is correct even when `M` refers to a mixin definition. This does not mean that an object can be instantiated from a mixin directly; however, such a declared object can be instantiated with a class created via the application of `M` to a (correct) superclass:

my_obj = **new** (M <> C) *# OK, provided the application is well typed*

Indeed, the declared object will be considered as having the same interface of the mixin `M`, that is the interface of a class having all the defined, redefined and expected methods of `M`.

One of the most interesting feature of MoMi is the ability of sending classes and mixins as mobile code as shown in the following example. We assume that a site provides printing facilities for local and mobile agents. The access to the printer requires a driver that the site itself has to provide to those that want to print, since it highly depends on the system and on the printer. Thus, the agent that wants to print is designed as a mixin, that expects a method for actually printing, `print_doc`, and defines a method `start_agent` through which the execution engine can start its execution. The actual instance of the printing agent is instantiated from a class dynamically generated by applying such mixin to a local superclass that provides the method `print_doc` acting as a wrapper for the printer driver.

However the system is willing to accept any agent that has a compatible interface, thus any mixin that is a subtype to the one used for describing the printing agent. Thus any client wishing to print on this site can send a mixin that is subtyping compliant to the one expected. In particular such a mixin can implement finer printing formatting capabilities.

Listing 3 presents a possible implementation of the printing client node (on the left) and of the printer server node (on the right). The printer client sends to the server a mixin `MyPrinterAgent` that respects (it is a subtype of) the mixin that the server expects to receive, `PrinterAgent` (not shown here). In particular this mixin will print a document on the printer of the server after preprocessing it. On the server, once the mixin is received, it is applied to the local (super)class

```
    # this is the mixin actually sent to the remote site
    # MyPrinterAgent <: PrinterAgent
    mixin MyPrinterAgent
      expect print_doc(doc : str) : str;
      def start_agent() : str
      begin
        return
         this.print_doc(this.preprocess("my document"))
      end;
      def preprocess(doc : str) : str
      begin
        return "preprocessed(" + doc +")"
      end
    end

    rec SendPrinterAgent[server : loc]
      declare
        var response : str;
        var sent_mixin : mixin MyPrinterAgent
      begin
        print "sending printer agent to " + server;
        sent_mixin := MyPrinterAgent;
        out(sent_mixin)@server;
        in(!response)@server;
        print "response is " + response
      end

  # the following class provides print_doc, so a PrinterAgent can be
  # applied to it.  Notice that it also provides another method, init()
  # that is ignored by the mixin
  class LocalPrinter
    print_doc(doc : str) : str
    begin
      # real printing code omitted :-)
      return "printed " + doc
    end;
    init()
    begin
      nil # foo init
    end
  end

  rec ReceivePrinterAgent[]
    declare
      var rec_mixin : mixin PrinterAgent;
      var result : str
    begin
      print "waiting for a PrinterAgent mixin...";
      in(!rec_mixin)@self;
      print "received " + rec_mixin;
      result := (new rec_mixin <> LocalPrinter).start_agent();
      print "result is " + result;
      out(result)@self
    end
```

Listing 3: The printer agent example (above: the sender site - the printer client, below: the receiver site - the printer server).

`LocalPrinter`, and an object (the agent) is instantiated from the resulting class, and started so that it can actually print its document. The result of the printing task is then retrieved and sent back to the client.

We observe that the sender does not actually know the mixin name `PrinterAgent`: it only has to be aware of the mixin type expected by the server (remember that in X-KLAIM class and mixin definition names are only shortcut for their actual types). Furthermore, the sent mixin can also define more methods than those specified in the receiving site, thanks to the mixin subtyping relation (Table 26). This adds a great flexibility to such a system, while hiding these additional methods to the receiving site (since they are not specified in the receiving interface they are actually unknown statically to the compiler), and also avoiding dynamic name clashes.

8 Conclusions

The work on KLAIM began in 1995 with the aim of combining the work on process algebras with localities and the one based on asynchronous generative communication mechanisms. The idea was that of building on the clean modelling of concurrency achieved with process algebras and on the success of the Linda coordination model [DP96]. Moreover, it was prompted by the intuition that network aware programming was calling for new languages and paradigms that would consider localities as first-class citizens, that should be dynamically created and handled via appropriate scoping rules [DFP97]. Since then, a lot of work has been done, also by other groups, by trying to enrich the model and the linguistic primitives to face the new challenges posed by the continuously evolving scenario of global computing.

In this paper we have tried to give a brief account on the work on KLAIM, by privileging the foundational aspects. We have thus presented a series of variants of the languages aiming, on one hand, at finding the appropriate tool for understanding and modelling processes behaviour over evolving wide area networks and, on the other hand, at finding the appropriate linguistic constructs to actually design and develop applications for such a challenging environment. We have also described programming logics for proving properties of widely distributed programs and prototype implementations of the proposed abstractions.

We want to conclude by saying that we do not see this as the end of our work on this topics, convinced as we are, that much work remains to be done and that the model and the language for network aware programming is still far to come, and that for sure will not be KLAIM. But, we hope that some of the ideas outlined here will help the search.

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[ACPP91] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[ACPR95] M. Abadi, L. Cardelli, B. Pierce, and D. Remy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[AFH99] K. Arnold, E. Freeman, and S. Hupfer. *JavaSpaces Principles, Patterns and Practice*. Addison-Wesley, 1999.

[ALZ03] D. Ancona, G. Lagorio, and E. Zucca. Jam - designing a java extension with mixins. *ACM Transaction on Programming Languages and Systems*, 2003. To appear.

[AS92] B. G. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In J. P. Banatre and D. Le Metayer, editors, *Proc. of Research Directions in High–Level Parallel Programming Languages*, volume 574 of *LNCS*, pages 93–109. Springer, 1992.

[BBV02] L. Bettini, V. Bono, and B. Venneri. Coordinating Mobile Object-Oriented Code. In F. Arbarb and C. Talcott, editors, *Proc. of Coordination Models and Languages*, number 2315 in LNCS, pages 56–71. Springer, 2002.

[BBV03a] L. Bettini, V. Bono, and B. Venneri. MoMi - A Calculus for Mobile Mixins. Manuscript, 2003.

[BBV03b] L. Bettini, V. Bono, and B. Venneri. Subtyping Mobile Clasees and Mixins. In *Proc. of Foundation of Object Oriented Languages (FOOL10)*, 2003.

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *Proc. OOPSLA '90*, pages 303–311, 1990.

[BCC01] M. Bugliesi, G. Castagna, and S. Crafa. Reasoning about security in mobile ambients. In *Concur 2001*, number 2154 in LNCS, pages 102–120. Springer, 2001.

[BD01] L. Bettini and R. De Nicola. Translating Strong Mobility into Weak Mobility. In G. P. Picco, editor, *Mobile Agents*, number 2240 in LNCS, pages 182–197. Springer, 2001.

[BDFP98] L. Bettini, R. De Nicola, G. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-KLAIM. In P. Ciancarini and R. Tolksdorf, editors, *Proc. of the 7th Int. IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 110–115, Stanford, 1998. IEEE Computer Society Press.

[BDL03] L. Bettini, R. De Nicola, and M. Loreti. Formulae meet programs over the net: a framework for correct network aware programming. submitted for pubblication, available at `http://music.dsi.unifi.it/`, 2003.

[BDP02] L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java Package for Distributed and Mobile Applications. *Software — Practice and Experience*, 32:1365–1394, 2002.

[Bet03] L. Bettini. *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*. PhD thesis, Dip. di Matematica, Università di Siena, 2003. Available at `http://music.dsi.unifi.it`.

[BLP02] L. Bettini, M. Loreti, and R. Pugliese. An Infrastructure Language for Open Nets. In *Proc. of ACM SAC 2002, Special Track on Coordination Models, Languages and Applications*, pages 373–377. ACM, 2002.

[BMR97] S. Bistarelli, U. Montanari, and F. Rossi. Semiring-based constraint satisfaction and optimization. *Journal of the ACM*, 44(2):201–236, March 1997.

[BWA94] P. Butcher, A. Wood, and M. Atkins. Global Synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.

[Car99] L. Cardelli. Abstractions for Mobile Computation. In Vitek and Jensen [VJ99], pages 51–94.

[CCR96] S. Castellani, P. Ciancarini, and D. Rossi. The ShaPE of ShaDE: a coordination system. Technical Report UBLCS 96-5, Dip. di Scienze dell'Informazione, Univ. di Bologna, Italy, 1996.

[CG89] N. Carriero and D. Gelernter. How to Write Parallel Programs: A Guide to the Perplexed. *ACM Computing Surveys*, 21(3):323–357, 1989.

[CG00] 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.

[CGZ01] G. Castagna, G. Ghelli, and F. Zappa Nardelli. Typing mobility in the seal calculus. In *Concur 2001*, number 2154 in LNCS, pages 82–101. Springer, 2001.

[CMC99] M. Scott Corson, Joseph P. Macker, and Gregory H. Cirincione. Internet-based mobile ad hoc networking. *Internet Computing*, 3(4), 1999.

[CTV+98] P. Ciancarini, R. Tolksdorf, F. Vitali, D. Rossi, and A. Knoche. Coordinating multiagent applications on the WWW: A reference architecture. *IEEE Transactions on Software Engineering*, 24(5):362–366, 1998.

[CV99] G. Castagna and J. Vitek. Seal: A framework for secure mobile computations. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, number 1686 in LNCS, pages 47–77. Springer, 1999.

[Deu01] D. Deugo. Choosing a Mobile Agent Messaging Model. In *Proc. of ISADS 2001*, pages 278–286. IEEE, 2001.

[DFM+03] R. De Nicola, G. Ferrari, U. Montanari, R. Pugliese, and E. Tuosto. A formal basis for reasoning on programmable qos. In *International Symposium on Verification – Theory and Practice – Honoring Zohar Manna's 64th Birthday*, LNCS. Springer-Verlag, 2003.

[DFP97] R. De Nicola, G. Ferrari, and R. Pugliese. Locality based Linda: Programming with explicit localities. In Michel Bidoit and Max Dauchet, editors, *TAPSOFT '97: Theory and Practice of Software Development*, volume 1214 of *LNCS*, pages 712–726. Springer-Verlag, 1997.

[DFP98] 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.

[DFP99] R. De Nicola, G. Ferrari, and R. Pugliese. Types as Specifications of Access Policies. In Vitek and Jensen [VJ99], pages 117–146.

[DFP00] R. De Nicola, G. Ferrari, and R. Pugliese. Programming Access Control: The Klaim Experience. In C. Palamidessi, editor, *Proc. of the 11th International Conference on Concurrency Theory (CONCUR'00)*, volume 1877 of *LNCS*, pages 48–65. Springer-Verlag, 2000.

[DFPV00] R. De Nicola, G. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science special issue on Coordination*, 240(1):215–254, 2000.

[DL02] R. De Nicola and M. Loreti. A Modal Logic for Mobile Agents. *ACM Transactions on Computational Logic*, 2002. To appear. Available at `http://music.dsi.unifi.it/`.

[DP96] R. De Nicola and R. Pugliese. A process algebra based on linda. In P. Ciancarini and C. Hankin, editors, *Proceedings of the First International Conference on Coordination Models and Languages (COORDINATION'96)*, volume 1061 of *Lecture Notes in Computer Science*, pages 160–178. Springer, 1996.

[Dug99] D. Duggan. Dynamic typing for distributed programming in polymorphic languages. *ACM Transactions on Programming Languages and Systems*, 21(1):11–45, 1999.

[DWFB97] N. Davies, S. Wade, A. Friday, and G. Blair. Limbo: a tuple space based platform for adaptive mobile applications. In *Int. Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP'97)*, 1997.

[FGL+96] C. Fournet, G. Gonthier, J. J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. of 7th Int. Conf. on Concurrency Theory (CONCUR'96)*, volume 1119 of *LNCS*, pages 406–421. Springer-Verlag, 1996.

[FKF98] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Proc. POPL '98*, pages 171–183, 1998.

[FMP03] G. Ferrari, E. Moggi, and R. Pugliese. MetaKlaim: A type safe multi-stage language for global computing. *Mathematical Structures in Computer Science*, 2003.

[FMSS02] G. Ferrari, C. Montangero, L. Semini, and S. Semprini. Mark, a reasoning kit for mobility. *Automated Software Engineering*, 9:137–150, 2002.

[FPV98] A. Fuggetta, G. Picco, and G. Vigna. Understanging code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[Gel89] D. Gelernter. Multiple Tuple Spaces in Linda. In J.Hartmanis G. Goos, editor, *Proceedings, PARLE '89*, volume 365 of *LNCS*, pages 20–27, 1989.

[Gir72] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Thèse de doctorat d'etat, University of Paris VII, 1972.

[GP03a] D. Gorla and R. Pugliese. Behavioural equivalences for distributed and mobile systems. Research report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2003. Available at `http://rap.dsi.unifi.it/~pugliese/DOWNLOAD/bis4klaim.pdf`.

[GP03b] D. Gorla and R. Pugliese. Enforcing Security Policies via Types. In *Proc. of the 1st International Conference on Security in Pervasive Computing (SPC'03)*, LNCS. Springer-Verlag, 2003.

[GP03c] D. Gorla and R. Pugliese. Resource access and mobility control with dynamic privileges acquisition. Research report, Dipartimento di Sistemi e Informatica, Università di Firenze, 2003. Available at `http://rap.dsi.unifi.it/~pugliese/DOWNLOAD/muklaim-full.pdf`. An extended abstract appeared in *Proceedings of ICALP'03*, LNCS, Springer, 2003.

[HM85] M. Hennessy and R. Milner. Algebraic Laws for Nondeterminism and Concurrency. *Journal of the ACM*, 32(1):137–161, January 1985.

[HR02] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. *Information and Computation*, 173(1):82–120, 2002.

[Lor02] M. Loreti. *Languages and Logics for Network Aware Programming*. PhD thesis, Università di Siena, 2002. Available at `http://music.dsi.unifi.it`.

[LRVD99] X. Leroy, D. Rémy, J. Vouillon, and D. Doligez. The Objective Caml system, documentation and user's guide. `http://caml.inria.fr/ocaml/htmlman/`, 1999.

[MHP00] The MetaML Home Page, 2000. Provides source code and documentation online at `http://www.cse.ogi.edu/PacSoft/projects/metaml/index.html`.

[MPW92] 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.

[MR98] P.J. McCann and G-C. Roman. Compositional programming abstraction for mobile computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, 1998.

[Ora99] Oracle. Oracle 9*i*AS application server lite web page. In *http://www.oracle.com/*, 1999.

[OZ99] A. Omicini and F. Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-agent Systems*, 2(3):251–269, 1999. Special Issue on Coordination Mechanisms and Patterns for Web Agents.

[PMR99] G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conference on Software Engineering (ICSE'99)*, pages 368–377. ACM Press, 1999.

[PR98] A.S. Park and P. Reichl. Personal Disconnected Operations with Mobile Agents. In *Proc. of 3rd Workshop on Personal Wireless Communications, PWC'98*, Tokyo, 1998.

[Rey74] J.C. Reynolds. Towards a theory of type structure. In *Proceedings Colloque sur la Programmation, Paris*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425, New York, NY, june 1974. Springer-Verlag. Extension of typed lambda calculus to user-defined types and polymorphic functions.

[Row98] A. Rowstron. WCL: A web co-ordination language. *World Wide Web Journal*, 1(3):167–179, 1998.

[SMH00] F.B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Ahead, 10 Years Back. Conference on the Occasion of Dagstuhl's 10th Anniversary*, number 2000 in Lecture Notes in Computer Science, pages 86–101. Springer-Verlag, 2000.

[Sun99] Sun Microsystems. Javaspace specification. available at: `http://java.sun.com/`, 1999.

[TS00] W. Taha and T. Sheard. MetaML: Multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2), 2000.

[Tuo99] E. Tuosto. An ada95 implementation of a network coordination language with code mobility. In M. G. Harbour and J. A. de la Puente, editors, *Intl. Conference on Reliable Software Technologies - Ada-Europe'99*, volume 1622 of *LNCS*, pages 199–210, Santander, Spain, June 1999. Springer-Verlag.

[VJ99] J. Vitek and C. Jensen, editors. *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS. Springer-Verlag, 1999.

[WF94] A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.

[WMLF98] P. Wyckoff, S. McLaughry, T. Lehman, and D. Ford. TSpaces. *IBM Systems Journal*, 37(3):454–474, 1998.

[WS99] M. Wand and I. Siveroni. Constraint systems for useless variable elimination. In *In proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, pages 291–302, 1999.

[YH99] N. Yoshida and M. Hennessy. Assigning types to processes. CogSci Report 99.02, School of Cognitive and Computing Sciences, University of Sussex, UK, 1999.