In *Proc. of FMOODS'05*, LNCS 3535, pages 1–14. © Springer.

Pattern Matching over a Dynamic Network of Tuple Spaces

Rocco De Nicola[1] Daniele Gorla[2⋆] Rosario Pugliese[1]

[1]Dipartimento di Sistemi e Informatica, Università di Firenze
[2]Dipartimento di Informatica, Università di Roma "La Sapienza"
email : {denicola, pugliese}@dsi.unifi.it, gorla@di.uniroma1.it

Abstract. In this paper, we present recent work carried on μKLAIM, a core calculus that retains most of the features of KLAIM: explicit process distribution, remote operations, process mobility and asynchronous communication via distributed tuple spaces. Communication in μKLAIM is based on a simple form of pattern matching that enables withdrawal from shared data spaces of matching tuples and binds the matched variables within the continuation process. Pattern matching is orthogonal to the underlying computational paradigm of μKLAIM, but affects its expressive power. After presenting the basic pattern matching mechanism, inherited from KLAIM, we discuss a number of variants that are easy to implement and test, by means of simple examples, the expressive power of the resulting variants of the language.

1 Introduction

In the last decade, programming computational infrastructures available globally for offering uniform services has become one of the main issues in Computer Science. The challenges come from the necessity of dealing at once with issues like communication, co-operation, mobility, resource usage, security, privacy, failures, etc. in a setting where demands and guarantees can be very different for the many different components. KLAIM (*Kernel Language for Agents Interaction and Mobility*, [5]) is a tentative response to the call for innovative theories, computational paradigms, linguistic mechanisms and implementation techniques for the design, realization, deployment and management of global computational environments and their application.

KLAIM is an experimental language specifically designed to program distributed systems made up of several mobile components interacting through multiple distributed tuple spaces. Its communication model builds over, and extends, LINDA's notion of generative communication through a shared tuple space [11]. The LINDA model was originally proposed for parallel programming on isolated machines; multiple, possibly distributed, tuple spaces have been advocated later [12] to improve modularity, scalability and performance, and fit well in a global computing scenario.

KLAIM has proved to be suitable for programming a wide range of distributed applications with agents and code mobility [5, 6] and it has originated an actual programming language, X-KLAIM [1], that has been implemented by exploiting Java [2].

⋆ Most of the work presented in this paper was carried on while the second author was a PhD student at the University of Florence.

NETS	COMPONENTS
$N ::= \mathbf{0} \mid l :: C \mid N_1 \parallel N_2 \mid (\nu l)N$	$C ::= \langle t \rangle \mid P \mid C_1 \mid C_2$
TUPLES	TEMPLATES
$t ::= u \mid t_1, t_2$	$T ::= u \mid !\, x \mid T_1, T_2$
ACTIONS	
$a ::= \mathbf{in}(T)@u \mid \mathbf{read}(T)@u \mid \mathbf{out}(t)@u \mid \mathbf{eval}(P)@u \mid \mathbf{new}(l)$	
PROCESSES	
$P ::= \mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid *\, P$	

Table 1. μKLAIM Syntax

The main drawback of KLAIM is that it is not an actual programming language, nor a process calculus. The main aim of some our recent works (grouped together in [13]) has been the definition of a process calculus derived from KLAIM that retains all its distinctive features and expressive power, and develop over it the type theoretic and semantical foundations of the language. The resulting calculus has been called μKLAIM and, in [8], we have proved that it can reasonably encode KLAIM.

In this paper, we first describe μKLAIM (Section 2). Then, in Section 3, we present some recent enhancements of the basic formalism to deal with some low-level features, namely inter-node connections and failures. In Section 4, we argue on alternative forms of pattern matching for retrieving tuples. So far, KLAIM and its variants have used LINDA's original pattern matching, because of its simplicity. Nevertheless, other variants could be adopted without compromising language implementability, actually enhancing the overall expressive power. A novel contribution of this paper is the informal examination of this topic. Section 5 concludes the paper.

2 The Calculus μKLAIM

2.1 Syntax

The syntax of μKLAIM is reported in Table 1. A countable set $\mathcal{L}$ of *names* $l, l', \ldots, u, \ldots, x, y, \ldots$ is assumed. Names provide the abstract counterpart of the set of *communicable* objects and can be used as localities and variables: we do not distinguish between these kinds of objects. Notationally, we prefer letters $l, l', \ldots$ when we want to stress the use of a name as a locality and $x, y, \ldots$ when we want to stress the use of a name as a variable. We will use u for basic variables and localities.

Nets are finite collections of nodes where processes and tuple spaces can be allocated. A *node* is a pair $l :: C$, where locality l is the address of the node and C is the parallel component located at l. *Components* can be processes or (located) tuples. *Located tuples*, $\langle t \rangle$, are inactive components representing tuples in a tuple space (TS, for short) that have been inserted either in the initial configuration or along a computation by executing an action **out**. The TS located at l results from the parallel composition of all located tuples residing at l. In $(\nu l)N$, name l is private to N; the intended effect is that, if one considers the term $N_1 \parallel (\nu l)N_2$, then locality l of N_2 cannot be referred from within N_1.

$match(l; l) = \epsilon$	$\dfrac{match(T_1; t_1) = \sigma_1 \quad match(T_2; t_2) = \sigma_2}{match(T_1, T_2; t_1, t_2) = \sigma_1 \circ \sigma_2}$
$match(!x; l) = [{}^{l}/_{x}]$	

Table 2. The Pattern Matching Function

Tuples are sequences of names. *Templates* are patterns used to select tuples in a TS. They are sequences of names and formal fields; the latter ones are written $!\,x$ and are used to bind variables to names.

Processes are the μKLAIM active computational units. They are built up from the inert process **nil** and from five basic operations, called *actions*, by using action prefixing, parallel composition and replication. The informal semantics of process actions is as follows. Action $\mathbf{in}(T)@u$ looks for a matching tuple $\langle t \rangle$ in the TS located at u; intuitively, a template matches against a tuple if both have the same number of fields and corresponding fields match, i.e. they are the same name, or one is a formal while the other one is a name. If $\langle t \rangle$ is found, it is removed from the TS, the formal fields of T are replaced in the continuation process with the corresponding names of t and the operation terminates. If no matching tuple is found, the operation is suspended until one is available. Action $\mathbf{read}(T)@u$ is similar but it leaves the selected tuple in u's TS. Action $\mathbf{out}(t)@u$ adds the tuple t to the TS located at u. Action $\mathbf{eval}(P)@u$ sends process P for execution at u. Action $\mathbf{new}(l)$ creates a new node in the net at the reserved address l. Notice that **new** is the only action not indexed with an address because it always acts locally; all the other actions explicitly indicate the (possibly remote) locality where they will take place.

Names occurring in terms can be bound by action prefixes or by restriction. More precisely, in processes $\mathbf{in}(T)@u.P$ and $\mathbf{read}(T)@u.P$ the prefixes bind the names in the formal fields of T within P; in process $\mathbf{new}(l).P$, the prefix binds l in P; in $(\nu l)N$, the restriction binds l in N. A name that is not bound is called *free*. The sets $bn(\cdot)$ and $fn(\cdot)$ (of bound and free names, resp., of term $\cdot$) are defined accordingly, and so is *alpha-conversion*. In the sequel, we shall assume that bound names in terms are all distinct and different from the free ones (by possibly applying alpha-conversion, this requirement can always be satisfied).

2.2 Operational Semantics

μKLAIM operational semantics is given in terms of a structural congruence and a reduction relation. The *structural congruence*, $\equiv$, identifies nets which intuitively represent the same net. It is inspired to π-calculus' structural congruence (see, e.g., [16]) and states that '$\|$' is a monoidal operator with **0** as identity, that **nil** is the identity for '$|$', that alpha-equivalent nets do coincide, and that the order of restrictions in a net is irrelevant. Moreover, the following laws are crucial to our setting:

$$\begin{array}{rll}
(\textsc{Clone}) & l :: C_1 | C_2 \;\equiv\; l :: C_1 \parallel l :: C_2 & \\
(\textsc{Repl}) & l :: *P \;\equiv\; l :: P \,|\, *P & \\
(\textsc{RepNil}) & l :: *\mathbf{nil} \equiv l :: \mathbf{nil} & \\
(\textsc{Ext}) & N_1 \parallel (\nu l)N_2 \equiv (\nu l)(N_1 \parallel N_2) & \text{if } l \notin fn(N_1)
\end{array}$$

(R-Out)

$$l :: \mathbf{out}(t)@l'.P \parallel l' :: \mathbf{nil} \longmapsto l :: P \parallel l' :: \langle t \rangle$$

(R-New)

$$l :: \mathbf{new}(l').P \longmapsto (\nu l')(l :: P \parallel l' :: \mathbf{nil})$$

(R-Eval)

$$l :: \mathbf{eval}(P_2)@l'.P_1 \parallel l' :: \mathbf{nil} \longmapsto l :: P_1 \parallel l' :: P_2$$

(R-Res)

$$\frac{N \longmapsto N'}{(\nu l)N \longmapsto (\nu l)N'}$$

(R-In)

$$\frac{match(T; t) = \sigma}{l :: \mathbf{in}(T)@l'.P \parallel l' :: \langle t \rangle \longmapsto l :: P\sigma \parallel l' :: \mathbf{nil}}$$

(R-Par)

$$\frac{N_1 \longmapsto N_1'}{N_1 \parallel N_2 \longmapsto N_1' \parallel N_2}$$

(R-Read)

$$\frac{match(T; t) = \sigma}{l :: \mathbf{read}(T)@l'.P \parallel l' :: \langle t \rangle \longmapsto l :: P\sigma \parallel l' :: \langle t \rangle}$$

(R-Struct)

$$\frac{N \equiv N_1 \quad N_1 \longmapsto N_2 \quad N_2 \equiv N'}{N \longmapsto N'}$$

Table 3. μKLAIM Reduction Relation

Law (Clone) turns a parallel between co-located components into a parallel between nodes (by relying on this law, commutativity and associativity of '|' follows). Law (Repl) unfolds a replicated process; however, when the replicated process is **nil**, the unfolding is useless (see rule (RepNil)). Finally, law (Ext) is the standard π-calculus' rule for scope extension; it states that the scope of a restricted name can be extended, provided that no free name is captured.

The reduction relation is given in Table 3. It relies on the *pattern matching* function $match(\_\,;\,\_)$ that verifies the compliance of a tuple w.r.t. a template and associates values to variables bound in the template. Intuitively, a tuple matches a template if they have the same number of fields, and corresponding fields match. Formally, function *match* is defined in Table 2 where we let 'ϵ' be the empty substitution and '$\circ$' denote substitutions composition. Here, a substitution σ is a mapping of names for names; $P\sigma$ denotes the (capture avoiding) application of σ to P.

The operational rules of μKLAIM can be briefly motivated as follows. Rule (R-Out) states that execution of an output sends the tuple argument of the action to the target node. However, this is possible only if the target node does exist in the net. Rule (R-Eval) is similar, but deals with process spawning. Rules (R-In) and (R-Read) require existence of a matching datum in the target node. The tuple is then used to replace the free occurrences of the variables bound by the template in the continuation of the process performing the actions. With action **in** the matched datum is consumed while with action **read** it is not. Rule (R-New) states that action **new**(l') creates a new node at a reserved address l'. Rules (R-Par), (R-Res) and (R-Struct) are standard.

μKLAIM adopts a LINDA-like [11] communication mechanism: data are anonymous and associatively accessed via pattern matching, and communication is asynchronous. Indeed, even if there exist action prefixes for placing data to (possibly remote) nodes, no synchronization takes place between (sending and receiving) processes, because their interactions are mediated by nodes, that act as data repositories.

2.3 Observational Semantics

We now present a preorder on μKLAIM nets yielding sensible semantic theories. We follow the approach put forward in [10] and use *may testing* equivalence. Intuitively, two nets are may testing equivalent if they cannot be distinguished by any external observer taking note of the data offered by the observed net. More precisely, an *observer O* is a net containing a node whose address is a reserved locality name `test`. A computation reports *success* if, along its execution, a datum at node `test` appears; this is written $\stackrel{OK}{\Longrightarrow}$.

Definition 1 (May Testing Equivalence). May testing, $\sqsubseteq$, *is the least equivalence on* μKLAIM *nets such that, for every* $N \sqsubseteq M$, *it holds that* $N \parallel O \stackrel{OK}{\Longrightarrow}$ *if and only if* $M \parallel O \stackrel{OK}{\Longrightarrow}$, *for any observer O.*

The problem underneath the definition of may testing we have just presented is the universal quantification over observers. This makes it hard to prove equivalences in practice. In [13], we have developed an alternative characterisations of $\simeq$ as a trace-based equivalence and a co-inductive proof technique as a bisimulation-based equivalence. However, these definitions have been omitted from this paper: here, it sufficies to have a sensible notion of equivalence to equate nets.

3 Node Connections and Failures

In this section we present two enhancements of the basic framework presented so far. Such enhancements allow us to better model some global computing phenomena.

3.1 Modelling Connections

In [7], we developed the behavioural theory of a language derived from μKLAIM by introducing explicit inter-node connections and process actions to dynamically change them. The syntax of the resulting calculus, that is called TKLAIM (*topological* KLAIM), can be obtained by adding the following productions to those in Table 1:

$$N \;::=\; \cdots \;\mid\; \{l_1 \rightarrow l_2\} \qquad\qquad a \;::=\; \cdots \;\mid\; \mathbf{conn}(u) \;\mid\; \mathbf{disc}(u)$$

A *connection* (or *link*) is a pair of node addresses $\{l_1 \rightarrow l_2\}$ stating that the nodes at addresses l_1 and l_2 are directly linked. Actions $\mathbf{conn}(l_2)$ and $\mathbf{disc}(l_2)$ aim at changing the network topology: once executed at l_1 they create/remove a link $\{l_1 \rightarrow l_2\}$ from the net.

The operational semantics of TKLAIM is obtained by modifying that of μKLAIM to take into account information on existing connections. First, the following structural rules are added

$$l :: \mathbf{nil} \;\equiv\; \{l \rightarrow l\} \qquad\qquad \{l_1 \rightarrow l_2\} \;\equiv\; \{l_1 \rightarrow l_2\} \parallel l_1 :: \mathbf{nil} \parallel l_2 :: \mathbf{nil}$$

They state that nodes are self-connected and that connections are established only between actual nodes. Second, the reduction relation of Table 3 is modified so that axioms check existence of proper connections enabling process actions. For example, rule (R-OUT) now becomes

$$l :: \mathbf{out}(t)@l'.P \parallel \{l \to l'\} \longmapsto l :: P \parallel \{l \to l'\} \parallel l' :: \langle t \rangle$$

Thus, the sending operation is enabled only if the source and the target nodes are directly connected. Analogous modifications are needed for rules (R-EVAL), (R-IN) and (R-READ). Of course, we also need two new axioms for the two new primitives

$$\text{(R-CONN)} \quad l :: \mathbf{conn}(l').P \parallel l' :: \mathbf{nil} \longmapsto l :: P \parallel \{l \to l'\}$$

$$\text{(R-DISC)} \quad l :: \mathbf{disc}(l').P \parallel \{l \to l'\} \longmapsto l :: P \parallel l' :: \mathbf{nil}$$

The behavioural theory of TKLAIM presented in Section 2.3, and modified to take connections into account, has been used in [9] to state and prove the properties of a well-known routing protocol for mobile systems, namely the *handover protocol* [15] proposed by the European Telecommunication Standards Institute (ETSI) for the GSM Public Land Mobile Network.

TKLAIM can be easily accommodated to model a finer scenario where connections must be activated by a handshaking between the nodes involved (this feature is similar to the so-called *co-capabilities* of Safe Ambients [14]). This mechanism can be implemented by introducing a new action **acpt** that, by synchronizing with an action **conn**, authorises the creation of a new connection either from a specific node or from any node. An enabling action corresponding to **disc** seems to be less reasonable, but could be handled similarly.

Action $\mathbf{acpt}(l)$ by a process located at l' means that l' is ready to activate a connection with l; thus, the operational rule for activating a connection now becomes

$$\text{(R-CONN}_1\text{)} \quad l :: \mathbf{conn}(l').P \parallel l' :: \mathbf{acpt}(l).Q \longmapsto l :: P \parallel \{l \to l'\} \parallel l' :: Q$$

Similarly, action $\mathbf{acpt}(!x)$ by a process located at l' means that l' is ready to activate a connection with any node, whose address will be bound to x in the continuation. In this case, the operational rule for activating a connection is

$$\text{(R-CONN}_2\text{)} \quad l :: \mathbf{conn}(l').P \parallel l' :: \mathbf{acpt}(!x).Q \longmapsto l :: P \parallel \{l \to l'\} \parallel l' :: Q[{}^{l}/_{x}]$$

3.2 Modelling Failures

In [9], we enriched TKLAIM with some simple but realistic ways to model failures in global computing systems. We model failures of nodes and of node components by adding the annihilating rule

$$\text{(R-FAILN)} \quad l :: C \longmapsto \mathbf{0}$$

to TKLAIM's operational rules that serves different purposes. Indeed, axiom (R-FAILN) models one of the following:

– *message omission*, if C represents a part of the tuple space at l (i.e. C is of the form $\langle t_1 \rangle | \ldots | \langle t_n \rangle$);
– node *fail-silent* failure, if, in the overall net, l occurs as address only in $l :: C$;
– *abnormal termination* of some processes running at l, if in the overall net there are other nodes with address l.

Modelling failures as disappearance of a resource (a datum, a process or a whole node) is a simple, but realistic, way of representing failures in a global computing scenario [3]. Indeed, while the presence of data/nodes can be ascertained, their absence cannot because there is no practical upper bound to communication delays. Thus, failures cannot be distinguished from long delays and should be modelled as totally asynchronous and undetectable events.

Clearly, our failure model can be easily adapted to deal with link failures too. To this aim, we only need to add the operational rule

$$\text{(R-FailC)} \quad \{l_1 \rightarrow l_2\} \longmapsto \mathbf{0}$$

that models the (asynchronous and undetectable) failure of the link between nodes l_1 and l_2.

The behavioural theory of tKlaim presented in Section 2.3 can be adapted to cope with failures. In [9], we used some resulting equational laws to prove the properties of a well-known distributed fault-tolerant protocol, namely the *k–set agreement* [4], and of a simplified routing task, namely discovering the neighbours of a given node.

4 Experimenting with Pattern Matching

The pattern matching function adopted by Klaim and its variants is essentially that of Linda, that was introduced by Gelernter in its seminal paper [11]. It enables withdrawal from the shared data space of matching tuples and binds the matched variables within the continuation process. This choice was driven both by historical and simplicity reasons. To be precise, Klaim's pattern matching differs from Linda's original one in that it does not allows tuples to contain formal fields. This feature, called *inverse structured naming*, was introduced for widening matching possibilities (tuples' formal fields can be matched by any value of the same type), rather than for communication purposes (indeed, tuples' formal fields are never replaced by corresponding values).

Several other alternatives could be considered that simplify the task of programming. In the rest of this section, we will present a number of variants and briefly discuss, by means of simple examples, the expressive power of the resulting variants of the language. We shall limit our interest to variants of the matching function of Table 2 that can be 'easily' implemented also in a distributed setting.

For each variant, we shall present a simple motivating example and show that the suggested modification simplifies programming when compared with the same task written in μKlaim. In the examples, wherever we find it convenient, we shall use basic data values (e.g. strings) to improve readability.

4.1 Enforcing Name Difference

KLAIM's pattern matching permits selecting a tuple that contains a specific value (name), say l, in a specific field, say the i-th one: it sufficies to use a template containing l in its i-th field. But one could be, instead, interested in selecting those tuples that have a precise structure but do not contain a l in their i-th field. To this aim, we extend the syntax of templates as

$$T \quad ::= \quad \cdots \;\big|\; \neg\, u\,!x$$

and, correspondingly, we extend the pattern matching function of Table 2 by adding the axiom

$$match(\neg\, l\,!x; l') = [^{l'}/_x] \qquad \text{if } l' \neq l \tag{1}$$

Clearly, this extension of the pattern matching function does not compromise implementability.

Let μKLAIM$^{\neq}$ be μKLAIM with the two modifications just presented. Then, we can easily implement in μKLAIM$^{\neq}$ a standard if-then-else construct, as follows

$$\textbf{if } l_1 = l_2 \textbf{ then } P_1 \textbf{ else } P_2 \;\triangleq\; \textbf{new}(l').\textbf{out}(l_1)@l'.(\textbf{in}(l_2)@l'.P_1 \mid \textbf{in}(\neg\, l_2\,!x)@l'.P_2)$$

with $l' \notin fn(l_1, l_2, P_1, P_2)$ and $x \notin fn(P_2)$. By relying on may-testing, we can easily state and prove the soundness of this implementation as follows:

$$\begin{array}{lll} l_1 = l_2 & \text{implies that} & l :: \textbf{if } l_1 = l_2 \textbf{ then } P_1 \textbf{ else } P_2 \;\simeq\; l :: P_1 \\ l_1 \neq l_2 & \text{implies that} & l :: \textbf{if } l_1 = l_2 \textbf{ then } P_1 \textbf{ else } P_2 \;\simeq\; l :: P_2 \end{array}$$

In μKLAIM such a construct is not finitary implementable, assuming (as usual) that the set of names is infinite. At most, we can use process

$$\textbf{new}(l').\textbf{out}(l_1)@l'.(\textbf{in}(l_2)@l'.P_1 \mid \textbf{in}(!x)@l'.P_2)$$

where, however, P_2 could also be executed whenever $l_1 = l_2$.

Notice that the implementation in μKLAIM$^{\neq}$ of the if-then-else we have just presented can be achieved with a simpler formulation of templates and pattern matching. Indeed, it sufficies to add fields of the form $\neg u$, with rule (1) replaced by

$$match(\neg l; l') = \epsilon \qquad \text{if } l' \neq l$$

However, the general formulation exploiting fields of the form $\neg\, u\,!x$ enables us to program more sophisticated applications. As an example, we consider a 'fair server', that never serves the same client two consecutive times. The code required for this task is the following:

$$P \triangleq \textbf{in}(!x)@l.\textbf{new}(l').\textbf{out}(x)@l'.(< \textit{Serve client } x > \;\mid\; * Q)$$

$$Q \triangleq \textbf{in}(!y)@l'.\textbf{in}(\neg\, y\,!z)@l.\textbf{out}(z)@l'.\; < \textit{Serve client } z >$$

The fair server is located at l and it runs process P. Client processes invoke the service by sending to l the address of the node where they run. Then, process P retrieves the first service request (coming from x), creates a new node l' to store the currently served

client, serves x and then activates the replicated process Q. The latter one retrieves from l' the last served client y and waits for a new request coming from a client z different from y; it then stores z in l' and serves z.

The application we have just presented can be useful in a distributed system to avoid starvation of client processes. If we want to extend it to the case where n client processes must be regularly alternated, we need a more general form of pattern matching. This can be obtained by defining a small language for name expressions like

$$\xi \;::=\; u \;\mid\; \neg u \;\mid\; \xi_1 \vee \xi_2 \;\mid\; \xi_1 \wedge \xi_2$$

where the only operations on names are tests for equality and difference combined by logical connectors and/or. Now, templates are defined as

$$T \;::=\; \xi\,!x \;\mid\; T_1, T_2$$

Notice that the old field $!x$ would be an abbreviation for $(l \vee \neg l)!x$ (for a generic l) and the old field u would be an abbreviation for $u!x$ (for an unused variable x). The pattern matching rule (1) is now replaced by rule

$$match(\xi\,!x; l) = [{}^{l}/_{x}] \qquad \text{if } l \models \xi$$

where the compatibility check $l \models \xi$ is defined as expected

$$\begin{array}{llll} l \models l & & l \models \xi_1 \vee \xi_2 & \text{if } l \models \xi_1 \text{ or } l \models \xi_2 \\ l \models \neg l' & \text{if } l \neq l' & l \models \xi_1 \wedge \xi_2 & \text{if } l \models \xi_1 \text{ and } l \models \xi_2 \end{array}$$

4.2 Scope of Name Binders

In the previous sections, we have assumed that the scope of name binders contained within templates is the process following the action that has the template as argument. However, it is possible to consider as part of the scope of a name also those template fields that syntactically follow the binder of the name. This feature could be exploited for retrieving tuples that contain multiple occurrences of the same name (value), whatever it is.

For example, consider the data base of a travel agency storing information about clients. This can be modelled by associating to the agency a locality l whose TS hosts the data base as tuples of the form

$$\langle\ Name\ ,\ TripID\ ,\ Departure\_Date\ ,\ Return\_Date\ ,\ Destination\ \rangle$$

Consider now a query for the record of a client that has planned with the agency a one-day trip to Rome (e.g., this could be needed to perform a market research). In the new formulation of the calculus, this can be very easily implemented by action

$$\mathbf{read}(!x_n, !x_i, !x, x, \text{“}Rome\text{”})@l$$

To extend the scope of a name binder to the remaining part of the template where it occurs, the pattern matching function in Table 2 has to be modified. We must reformulate function *match* in order to apply the partial substitution calculated after the analysis

of the first i fields to the analysis of the $(i+1)$-th field. To formalise this idea, let p range over template fields, i.e.

$$p \quad ::= \quad u \;\mid\; !x$$

Then, the pattern matching rules of Table 2 now become

$$\frac{match_\sigma(T;t) = \sigma_1 \quad match_{\sigma_1}(p;l) = \sigma_2}{match_\sigma(T,p;t,l) = \sigma_2} \qquad match_\sigma(u;l) = \sigma \quad \text{if } u = l \text{ or } \sigma(u) = l$$

$$match_\sigma(!x;l) = \sigma \circ [{}^{l}/_{x}]$$

Function *match* is invoked in rules (R-In) and (R-Read) as $match_\epsilon(T;t)$.

It should be apparent that the effect of the matching mechanism above, that permits enforcing selection of tuples where the very same field occurs repeatedly, cannot be achieved in its full generality in μKLAIM. At the best, it could be somehow simulated under the (restrictive) assumption that the duplicated values are known in advance or can be guessed. Coming back to the travel agency example, we could write, e.g.,

$$\mathbf{read}(!x_n, !x_i, 1/1/05, 1/1/05, \text{“}Rome\text{”})@l$$

but in this way we would only select those clients that went to Rome on January the 1st, 2005.

Also in a language with the full power of the if-then-else, like μKLAIM$^{\neq}$, achieving the effect of the pattern matching above poses some problems. Indeed, consider the process

$$A \triangleq \mathbf{in}(!x, !y)@l.\mathbf{if}\ x = y\ \mathbf{then}\ P\ \mathbf{else}\ \mathbf{out}(x,y)@l.A$$

where, for simplicity, we have used recursive process definitions (that can be simulated by relying on replication, see e.g. [16]). This encoding of action $\mathbf{in}(!x, x)@l.P$ is not fully satisfactory because it introduces divergence: process A can repeatedly access at l a tuple of the form $\langle l_1, l_2\rangle$, with $l_1 \neq l_2$.

4.3 Exact Matching

LINDA's pattern matching enables the consumer of a datum to specify some constraints over the accessed tuple (i.e., the values occurring at some precise positions of the tuple). A symmetric capability is not available for the producer of a tuple, i.e. it cannot specify any constraint over the template used for retrieving the tuple. This fact forbids the storing of reserved data at public tuple spaces. For example, let d be a reserved datum stored at l, e.g.

$$l :: \langle d \rangle$$

Then, any process knowing l can easily retrieve d with action $\mathbf{in}(!x)@l$.

To provide data producers with the capability of controlling data retrievals, we slightly extend the syntax of tuples from Table 1 to become

$$t \quad ::= \quad \cdots \;\mid\; u!$$

Intuitively, a name marked with a ' !' occurring within a tuple can only be matched by the very same name occurring at the corresponding position within a template and not

by a formal field. Hence, the pattern matching function of Table 2 must also include the axiom

$$match(l; l!) = \epsilon$$

In particular, both $match(!x; l!)$ and $match(l'; l!)$, with $l' \neq l$, will fail. Let μKLAIM$^{!}$ be the variant of μKLAIM with exact name matching.

In μKLAIM$^{!}$, tuple fields marked with a ' !' act as passwords (of a symmetric cryptographic system) that retrievers must exhibit in order to access the tuple. In this way, secret data can be freely stored at public TSs; for example, the node

$$(\nu n)(l :: \langle n!, d\rangle \mid P)$$

is safe in that no other process than P can (immediately) access d, whatever be the rest of the net.

4.4 Nested Tuples

In KLAIM and all its variants tuples and templates are plain sequences of fields; roughly speaking, they are *lists* of fields. A straightforward generalisation of this definition is allowing *nested* tuples/templates, i.e. lists of fields that can contain other lists.

To model nested tuples/templates, we overload notation $\langle\cdot\rangle$ and modify the syntax of tuples and templates from Table 1 to become

$$t \;::=\; \cdots \;\big|\; \langle t\rangle \qquad\qquad T \;::=\; \cdots \;\big|\; \langle T\rangle$$

The pattern matching function is smoothly adapted to deal with nested arguments. We just need to add such rules as the following ones

$$\frac{match(T; t) = \sigma}{match(\langle T\rangle\,;\, \langle t\rangle) = \sigma} \qquad\qquad match(!x; t) = [t/x]$$

to the definition of function *match* given in Table 2. The first rule extends pattern matching by still requiring that matching templates and tuples have the same structure. The second rule allows to match entire tuples with a single variable; in such a setting, it should also be possible to assign entire tuples to variables and to use projection operators for retrieving each tuple field.

Notice that the two rules must not be necessarily used both at the same time. For instance, let μKLAIM^{nt} be the variant of μKLAIM with nested tuples and pattern matching extended using the first rule above. A simple application is the modelling of tree-like structures, similar to XML documents. For example, binary trees can be easily obtained by restricting the syntax of nested tuples as follows

$$t \;::=\; u \;\big|\; \langle t_1\rangle, u, \langle t_2\rangle$$

Clearly, trees can be somehow modelled in μKLAIM by using tuples corresponding to a preorder visit of the tree. To univocally identify the tree

$$\langle a\rangle, b, \langle c\rangle$$

in μKLAIM we could use the tuple

$$b, \textbf{left}, a, \textbf{right}, c$$

where **left** and **right** are a reserved names used to delimit the two subtrees. However, the exact name matching of μKLAIM$^!$ is needed in order to faithfully simulate the pattern matching function of μKLAIM^{nt}. Thus, μKLAIM^{nt} can be encoded in μKLAIM$^!$, but the ease of programming makes μKLAIM^{nt} a valid proposal as well.

4.5 Collecting Multisets of Tuples

We conclude this section with another variant of the matching function, called *matchAll*, that permits matching a template T and a multiset of tuples $\mathcal{M}$, and returning the multiset of substitutions induced by all matchings. Notationally, given a component C of the form $\langle t_1 \rangle \mid \cdots \mid \langle t_n \rangle$, we shall use $\mathcal{M}(C)$ to denote the multiset of tuples $\{\!|t_1, \cdots, t_n|\!\}$; we will use $\uplus$ to denote multiset union.

Function $matchAll(T; \mathcal{M})$ returns a pair consisting of:

1. the multiset Σ of substitutions, containing the elements $\sigma_1, \ldots, \sigma_n$ (corresponding to the single tuples t_i in $\mathcal{M}$ that match template T), and
2. the multiset $\mathcal{M}'$ of the tuples t_j in $\mathcal{M}$ that do not match T.

Function $matchAll(T; \mathcal{M})$ can be defined in terms of function *match* given in Table 2 as follows:

$$matchAll(T; \{\!|\}\!|) = \langle \{\!|\}\!|, \{\!|\}\!| \rangle \qquad \frac{matchAll(T; \mathcal{M}) = \langle \Sigma, \mathcal{M}' \rangle}{matchAll(T; \mathcal{M} \uplus \{\!|t|\!\}) = \begin{cases} \langle \Sigma \uplus \{\!|\sigma|\!\}, \mathcal{M}' \rangle & \text{if } match(T; t) = \sigma \\ \langle \Sigma, \mathcal{M}' \uplus \{\!|t|\!\} \rangle & \text{otherwise} \end{cases}}$$

To show its usefulness, we use function *matchAll* to model the semantics of the construct **forall** used in the programming language X-KLAIM [1]. Intuitively, process

$$\textbf{forall } \textbf{in}(T)@l \textbf{ do } P$$

retrieves all the tuples $t_1, \ldots, t_n$ located at l that match T, then uses the substitutions $\sigma_i = match(T, t_i)$ to execute n instances of P with the different substitutions (i.e., $P\sigma_1, \ldots, P\sigma_n$). To formalize the semantics of **forall**, we find it convenient to make use of a construct for sequential composition of processes, that we shall write $P_1; P_2$. The operational semantics of sequential composition is modelled by the following rules where, to avoid name capture, we assume that $bn(P_1) \cap fn(P_2) = \emptyset$:

$$\frac{l :: P_1 \;\equiv\; l :: \textbf{nil}}{l :: P_1; P_2 \;\equiv\; l :: P_2} \qquad \frac{l :: P_1 \parallel N \;\longmapsto\; l :: P_1' \parallel N'}{l :: P_1; P_2 \parallel N \;\longmapsto\; l :: P_1'; P_2 \parallel N}$$

Now, the semantics of **forall** can be modelled as follows:

$$\frac{matchAll(T; \mathcal{M}(\langle t_1\rangle \mid \cdots \mid \langle t_n\rangle)) = \langle \{\!|\sigma_{i_1}, \cdots, \sigma_{i_k}|\!\}, \mathcal{M}(C')\rangle}{l :: \textbf{forall in}(T)@l' \textbf{ do } P \parallel l' :: \langle t_1\rangle \mid \cdots \mid \langle t_n\rangle \;\longmapsto_{l'}\; l :: P\sigma_{i_1}; \cdots ; P\sigma_{i_k} \parallel l' :: C'}$$

$$\frac{N_1 \longmapsto_l N_1'}{N_1 \parallel N_2 \longmapsto_l N_1' \parallel N_2} \; N_2 \text{ does not contain tuples located at } l$$

The two rules above define a new transition relation $\longmapsto_l$ that is parameterized with respect to the address l of the node where the tuple space is located. This parametrization is necessary for ensuring that the entire tuple space at l is used as a parameter of *matchAll*. The operational semantics of the resulting language is given by the union of relations $\longmapsto$ (defined in Table 3) and $\longmapsto_l$, for any l.

5 Conclusions

We have briefly presented μKLAIM, a simple calculus that retains the main features of KLAIM, and have summarised some recent linguistic extensions that permit the explicit modelling of inter-node connections and of nodes and links failures. We have then sketched a research that we are currently pursuing that aims at assessing the impact of plugging into the calculus more powerful pattern matching mechanisms. By means of simple examples, we have shown how flexible (but still implementable) pattern matching policies can ease the task of programming global computing application. Clearly, the study of relative expressiveness, possible encodings and minimality deserves a deeper attention and will be the subject of future investigations.

References

1. 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*, pages 110–115. IEEE Computer Society Press, 1998.
2. 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.
3. L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, number 1603 in LNCS, pages 51–94. Springer, 1999.
4. S. Chaudhuri. More Choices Allow More Faults: Set Consensus Problems in Totally Asynchronous Systems. *Information and Computation*, 105(1):132-158, 1993.
5. 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.
6. 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.
7. R. De Nicola, D. Gorla, and R. Pugliese. Basic observables for a calculus for global computing. Research Report 07/2004, Dipartimento di Informatica, Università di Roma "La Sapienza". Available at `http://www.dsi.uniroma1.it/~gorla/publications.htm`.

8. R. De Nicola, D. Gorla, and R. Pugliese. On the expressive power of KLAIM-based calculi. Research Report 09/2004, Dipartimento di Informatica, Università di Roma "La Sapienza". Available at `http://www.dsi.uniroma1.it/~gorla/publications.htm`. An extended abstract appeared in *Proc. of EXPRESS'04*, ENTCS.
9. R. De Nicola, D. Gorla, and R. Pugliese. Global computing in a dynamic network of tuple spaces. In J. Jacquet and G. Picco, editors, *Proc. of COORDINATION'05*, number 3454 in LNCS, pages 157–172. Springer, 2005.
10. R. De Nicola and M. Hennessy. Testing equivalence for processes. *Theoretical Computer Science*, 34:83–133, 1984.
11. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
12. D. Gelernter. Multiple Tuple Spaces in Linda. In J. G. Goos, editor, *Proceedings, PARLE '89*, volume 365 of *LNCS*, pages 20–27, 1989.
13. D. Gorla. *Semantic Approaches to Global Computing Systems*. PhD thesis, Dip. Sistemi ed Informatica, Univ. di Firenze, 2004.
14. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings of POPL '00*, pages 352–364. ACM, 2000.
15. F. Orava and J. Parrow. An algebraic verifi cation of a mobile network. *Journal of Formal Aspects of Computing*, 4:497–543, 1992.
16. J. Parrow. An introduction to the pi-calculus. In J. Bergstra, A. Ponse, and S. Smolka, editors, *Handbook of Process Algebra*, pages 479–543. Elsevier Science, 2001.