On the Expressive Power of Klaim-based Calculi

Rocco De Nicola

Dip. di Sistemi ed Informatica, Univ. di Firenze, Italy

Daniele Gorla

Dip. di Informatica, Univ. di Roma “La Sapienza”, Italy
Dip. di Sistemi ed Informatica, Univ. di Firenze, Italy

Rosario Pugliese

Dip. di Sistemi ed Informatica, Univ. di Firenze, Italy

Abstract

In this work, we study the expressive power of variants of KLAIM, an experimental language with programming primitives for global computing that combines the process algebra approach with the coordination-oriented one. KLAIM has proved to be suitable for programming a wide range of distributed applications with agents and code mobility, and has been implemented on the top of a runtime system based on Java. The expressivity of its constructs is tested by distilling from it some (more and more foundational) calculi and studying the encoding of each of the considered languages into a simpler one. An encoding of the asynchronous $\pi$-calculus into one of these calculi is also presented.

Keywords: Global Computing, Code Migration, Tuple Spaces, Expressivity, Encodings.

1 This work has been partially supported by EU FET – Global Computing initiative, projects MIKADO IST-2001-32222. The funding body is not responsible for any use that might be made of the results presented here.

2 Email: denicola@dsi.unifi.it

3 Email: gorla@di.uniroma1.it

4 Email: pugliese@dsi.unifi.it

1571-0661/$ – see front matter © 2005 Elsevier B.V. All rights reserved.
1 Introduction

In the design of programming languages for global computing, a key research challenge is devising theoretical models and calculi with a clean formal semantics for specifying, programming and reasoning about global computing applications. These calculi could provide a basis for the design of systems which are “sound by construction” and behave in a predictable and analysable manner. The crux is to identify the more appropriate abstractions and to supply foundational and effective tools to support the development of global computing applications.

A suitable abstraction is mobility. This feature deeply increases flexibility and, thus, expressivity of programming languages. An evidence of the success gained by this programming style is given by the design of new commercial/prototype programming languages with suitable constructs. This activity has involved in the last years several important industrial and academic research institutions.

The first foundational calculus dealing with mobility has been the $\pi$-calculus [12], a simple and expressive calculus representing the essence of name passing with no redundant operation. The only operators of the $\pi$-calculus are the empty process, output and input prefix, parallel composition, name restriction and process replication; the exchanged values of the calculus are just names. From a global computing perspective, what $\pi$-calculus misses is an explicit notion of locality or of environment where computation takes place.

To deal with this deficiency several foundational languages, presented as process calculi or strongly based on them, have been developed and have improved the formal understanding of global computing systems. We mention, among the others, the Ambient calculus [4], the D$\pi$-calculus [11] and KLAIM [5]. As usual, a major problem in the development of a foundational language is to come out with abstractions that are a good compromise between expressivity, elegance and implementability. A paradigmatic example is the Ambient calculus: it is very elegant and expressive, but lacks of a reasonable distributed implementation.

We have been long working with KLAIM, an experimental language with programming constructs for global computing that combines the process algebraic paradigm with the coordination-oriented one. KLAIM has been 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. Communication takes place through distributed repositories (a very flexible model that well fits requirements of global computing) and remote operations (to supply a realistic abstraction level and avoid heavily resorting to code mobility).

**KLAIM** rests on an extension of the basic **LINDA** coordination model [9] with multiple distributed tuple spaces. A tuple space is a multiset of tuples that are sequences of information items. Tuples are anonymous and associatively selected from tuple spaces by means of a pattern-matching mechanism. Tuples can contain both values and code that can be subsequently accessed and evaluated. An allocation environment (associating logical and physical localities) is used to avoid the programmers to know the precise physical allocation of the distributed tuple spaces.

**KLAIM** has been upgraded to a full fledged programming language **X-KLAIM** [2] by relying on the implementation of a run-time system developed in Java for the sake of portability [3]. The linguistic constructs of **KLAIM** have proved to be very useful for programming a wide range of distributed applications with agents and code mobility [5,6] that, once compiled in Java, can be run over different platforms.

The main aim of our work is understanding the expressive power of tuple based communications and evaluating the theoretical impact of the linguistic primitives proposed for the language **KLAIM**. This task is carried on by distilling from **KLAIM** a few, more and more, foundational calculi and by studying the possibility of encoding each of the calculi in a more basilar one. A tight comparison between one of these calculi and the asynchronous \( \pi \)-calculus, see e.g. [14], is also provided. The first calculus we consider is \( \mu \text{KLAIM} \) [10]; it is obtained by eliminating from **KLAIM** the distinction between logical and physical localities (no allocation environment) and the possibility of higher order communication (no process code in tuples). The second calculus, \( c\text{KLAIM} \), is obtained from \( \mu \text{KLAIM} \) by only considering monadic communications and by removing one of the basic actions (read). The last calculus, \( l-c\text{KLAIM} \), is obtained by removing also the possibility of performing remote inputs and outputs; communications is only local and process migration is needed to use remote resources.

The rest of this extended abstract is organized as follows. **KLAIM** and the three calculi derived from it are presented in Section 2, while Section 3 briefly sums up the expressivity related results of our work. The formal developments of the latter Section are not reported in this extended abstract; the formal encodings and the actual proofs of their adequacy can be found in [7].
2 A Family of Process Languages

2.1 KLAIM: Kernel Language for Agents Interaction and Mobility

The syntax of KLAIM is given in Table 1. We assume two disjoint countable sets: \( \mathcal{L} \) of locality names \( l, l', \ldots \) and \( \mathcal{V} \) of variables \( x, y, \ldots, X, Y, \ldots, \text{self} \), where self is a reserved variable (see below). Notationally, we prefer letters \( x, y, \ldots \) when we want to stress the use of a variable as a basic variable, and \( X, Y, \ldots \) when we want to stress the use of a variable as a process variable. We will use \( u \) for basic variables and locality names.

Processes, ranged over by \( P \), are the KLAIM active computational units and may be executed concurrently either at the same locality or at different localities. Processes are built from the terminated process \( \text{nil} \) and from basic actions by using action prefixing, parallel composition and recursion. Actions, ranged over by \( a \), permit removing/accessing/adding data from/to node repositories, activating new threads of execution and creating new nodes. Action \( \text{new} \) is not indexed with an address because it always acts locally; all the other actions explicitly indicate the (possibly remote) locality where they will take effect. Tuples, \( t \), are the communicable objects: they are sequences of names and processes. Templates, \( T \), are patterns used to retrieve tuples and the pattern matching underlying the communication mechanism is the one used for LINDA [9].

Nets, ranged over by \( N \), are finite collections of nodes. A node is a triple \( l ::_{\rho} C \), where locality \( l \) is the address of the node, \( \rho \) is the allocation environment (a finite partial mapping from variables to names, used to implement dynamic binding of variables) and \( C \) is the program and data component located at \( l \). For the sake of simplicity, we assume that allocation environments act as the identity on locality names. Components, ranged over by \( C \), can be either processes or data, denoted by \( \langle t \rangle \). In the net \( (\nu l)N \), the scope of the name \( l \) is restricted to \( N \); the intended effect is that if one considers the

| \( N := 0 \) | \( l ::_{\rho} C \) | \( N_1 \parallel N_2 \) | \( (\nu l)N \) |
| \( C := \langle t \rangle \) | \( P \) | \( C_1 \parallel C_2 \) |
| \( P := \text{nil} \) | \( a.P \) | \( P_1 \parallel P_2 \) | \( X \) | \( \text{rec } X.P \) |
| \( a := \text{in}(T)@u \) | \( \text{read}(T)@u \) | \( \text{out}(t)@u \) | \( \text{eval}(P)@u \) | \( \text{new}(l) \) |
| \( t := u \) | \( P \) | \( t_1, t_2 \) |
| \( T := u \) | \( !x \) | \( !X \) | \( T_1, T_2 \) |
net $N_1 \parallel (\nu l)N_2$ then locality $l$ of $N_2$ cannot be immediately referred to from within $N_1$. We say that a net is well-formed if for each node $l ::_\rho C$ we have that $\rho(\text{self}) = l$, and, for any pair of nodes $l ::_\rho C$ and $l' ::_{\rho'} C'$, we have that $l = l'$ implies $\rho = \rho'$. Hereafter, we will only consider well-formed nets.

Names and variables occurring in K\textsc{aim} processes and nets can be bound. More precisely, prefix $\text{new}(l).P$ binds name $l$ in $P$, and, similarly, net restriction $(\nu l)N$ binds $l$ in $N$. Prefix $\text{in}(\ldots,!,\ldots)@u.P$ binds variable $u$ in $P$; this prefix is similar to the $\lambda$-abstraction of the $\lambda$-calculus. Finally, $\text{rec } X.P$ binds variable $X$ in $P$. A name/variable that is not bound is called free. The sets $\text{fn}(\cdot)$ and $\text{bn}(\cdot)$ (respectively, of free and bound names of a term) and $\text{fv}(\cdot)$ and $\text{bv}(\cdot)$ (of free/bound variables) are defined accordingly. As usual, we say that two terms are alpha-equivalent, written $=_{\alpha}$, if one can be obtained from the other by renaming bound names/variables.

Remark. The language presented so far slightly differs from [5]: the two differences are the absence of values and expressions, and the use of recursion instead of process definitions. Values (integers, strings, ...) and expressions are not included only to simplify reasoning: they can be easily encoded by following the classical translations in the $\pi$-calculus [14]. Recursion is easier than process definitions to deal with in a theoretical framework because the syntax of a recursive term already contains all the code needed to properly run the term itself.

The operational semantics relies on a structural congruence relation, $\equiv$, bringing the participants of a potential interaction to contiguous positions, and a reduction relation, $\rightarrow$, expressing the evolution of a net. The structural congruence is the least congruence closed under the axioms given in the upper part of Table 2; the reduction relation is given in the lower part of the same Table. There, we use two auxiliary functions:

(i) A tuple/template evaluation function, $\mathcal{E}[\_]_\rho$, to evaluate variables according to the allocation environment of the node performing the action whose argument is $. The main clauses of its definition are given below:

$$
\mathcal{E}[u]_\rho = \begin{cases} 
\text{UNDEF} & \text{otherwise} \\
u & \text{if } u \in \mathcal{L} \\ho(u) & \text{if } u \in \text{dom}(\rho)
\end{cases}
$$

where $P\{\rho\}$ denotes the process obtained from $P$ by replacing any free occurrence of a variable $x$ that is not within the argument of an eval with $\rho(x)$. Clearly, $\mathcal{E}[P]_\rho$ is UNDEF if $\rho(x)$ is undefined for some of these $x$. We shall write $\mathcal{E}[t]_\rho = t'$ to denote that the evaluation of $t$ using $\rho$
Axioms for Structural Congruence:

Monoid laws for "∥", i.e.

\[ N \parallel 0 \equiv N \parallel N_1 \equiv N_2 \parallel N_1 \parallel N_3 \equiv N_1 \parallel (N_2 \parallel N_3) \]

(ALPHA) \[ N \equiv N' \quad \text{if} \ N =_\alpha N' \]

(EXT) \[ (N_1 \parallel (\nu l)N_2) \equiv (\nu l)(N_1 \parallel N_2) \quad \text{if} \ l \not\in fn(N_1) \]

(ABS) \[ l ::_\rho C \equiv l ::_\rho (C | \text{nil}) \]

(CLONE) \[ l ::_\rho C_1 | C_2 \equiv l ::_\rho C_1 \parallel l ::_\rho C_2 \]

(REC) \[ l ::_\rho \text{rec} X.P \equiv l ::_\rho [\text{rec} X.P/X] \]

Reduction Relation:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RED-OUT</td>
<td>[ \rho(u) = l' \quad \mathcal{E}[t']_\rho = t' ]</td>
</tr>
<tr>
<td>RED-EVAL</td>
<td>[ l ::<em>\rho \text{out}(t)@u.P \parallel l' ::</em>\rho' \text{nil} \quad \rightarrow \quad l ::<em>\rho P \parallel l' ::</em>\rho' \langle t' \rangle ]</td>
</tr>
<tr>
<td>RED-IN</td>
<td>[ \rho(u) = l' \quad \text{match}(\mathcal{E}[T]_\rho, t) = \sigma ]</td>
</tr>
<tr>
<td>RED-READ</td>
<td>[ l ::<em>\rho \text{in}(T)@u.P \parallel l' ::</em>\rho' \langle t \rangle \quad \rightarrow \quad l ::<em>\rho P\sigma \parallel l' ::</em>\rho' \text{nil} ]</td>
</tr>
<tr>
<td>RED-NEW</td>
<td>[ l ::<em>\rho \text{new}(t').P \quad \rightarrow (\nu l')(l ::</em>\rho P \parallel l' ::_\rho'</td>
</tr>
<tr>
<td>RED-PAR</td>
<td>[ N_1 \rightarrow N'_1 \quad N_1 \parallel N_2 \rightarrow N'_1 \parallel N_2 ]</td>
</tr>
<tr>
<td>RED-RES</td>
<td>[ (\nu l)N \rightarrow (\nu l)N' ]</td>
</tr>
</tbody>
</table>

\( N \equiv M \rightarrow M' \equiv N' \)

Table 2

Kbaim Operational Semantics

(ii) A pattern matching function, \( \text{match} \), to verify the compliance of a tuple w.r.t. a template and to associate values (i.e. names and processes) to variables bound by the template. Intuitively, a tuple matches against a template if they have the same number of fields, and corresponding
fields match. Formally, it is defined by the following rules:

\[
\text{match}(l, l) = \epsilon \\
\text{match}(!x, l) = [l/x] \\
\text{match}(!X, P) = [P/x] \\
\text{match}(T_1, t_1) = \sigma_1 \\
\text{match}(T_2, t_2) = \sigma_2 \\
\text{match}(T_1, T_2, t_1, t_2) = \sigma_1 \circ \sigma_2
\]

where we let ‘\(\epsilon\)’ to be the empty substitution and ‘\(\circ\)’ to denote substitutions composition. Here, a substitution \(\sigma\) is a mapping of names and processes for variables; \(P\sigma\) denotes the (capture avoiding) application of \(\sigma\) to \(P\). Moreover, we assume that \(P\sigma\) yields a process written according to the syntax of Table 1.

The intuition beyond the operational rules of \(\mu\text{KLM}\) is the following. In rule (\textsc{Red-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. Notice that processes in a tuple are transmitted after the interpretation of their free 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 \textsc{eval} operation, rule (\textsc{Red-Eval}). In this case the free variables of the spawned process are not interpreted using the local allocation environment: the linking of variables is done at the remote node. Rules (\textsc{Red-In}) and (\textsc{Red-Read}) require existence of a matching tuple 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 action. With action \textsc{in}, the matched tuple is consumed while with action \textsc{read} it is not. Finally, in rule (\textsc{Red-New}), the environment of a new node is derived from that of the creating one with the obvious update for the \texttt{self} variable. Therefore, the new node inherits all the bindings of the creating node.
2.2 \( \mu \text{KLAIM: micro KLAIM} \)

The calculus \( \mu \text{KLAIM} \) has been derived in [10] from KLAIM by removing allocation environments and the possibility of having pieces of code as tuple fields (and, then, process variables as template fields). Its syntax is given in Table 3. The removal of allocation environments makes it possible to merge together names and variables. Thus, we only assume a countable set \( \mathcal{N} \) of names \( l, l', \ldots, u, \ldots, x, y, \ldots, X, Y, \ldots \). Like before, we prefer using letters \( l, l', \ldots \) when we want to stress the use of a name as a locality, \( x, y, \ldots \) when we want to stress the use of a name as a locality variable, and \( X, Y, \ldots \) when we want to stress the use of a name as a process variable. We will use \( u \) for locality variables and localities. Process variables will be used only for defining recursive processes.

Notice that \( \mu \text{KLAIM} \) can be considered as the largest sub-calculus of KLAIM where tuples do not contain any process, templates do not contain any process variable, allocation environments are empty and all processes are closed. These modifications sensibly simplifies the operational semantics of the language. The structural congruence is readily adapted from Table 2; the key laws to define the reduction relation are given in Table 4. Notice that now tuples/templates evaluation function is useless and substitutions are (standard) mappings of names for names. Hence, the definition of function \( \text{match} \) is given by the following laws:

\[
\begin{align*}
\text{(RED-OUT)} & \quad l :: \text{out}(t)@l'.P \parallel l' :: \text{nil} \rightarrow l :: P \parallel l' :: \langle t \rangle \\
\text{(RED-EVAL)} & \quad l :: \text{eval}(P_2)@l'.P_1 \parallel l' :: \text{nil} \rightarrow l :: P_1 \parallel l' :: P_2 \\
\text{(RED-IN)} & \quad \text{match}(T, t) = \sigma \\
& \quad l :: \text{in}(T)@l'.P \parallel l' :: \langle t \rangle \rightarrow l :: P \sigma \parallel l' :: \text{nil} \\
\text{(RED-READ)} & \quad \text{match}(T, t) = \sigma \\
& \quad l :: \text{read}(T)@l'.P \parallel l' :: \langle t \rangle \rightarrow l :: P \sigma \parallel l' :: \langle t \rangle \\
\text{(RED-NEW)} & \quad l :: \text{new}(l').P \rightarrow (\nu l')(l :: P \parallel l' :: \text{nil})
\end{align*}
\]

Table 4
\( \mu \text{KLAIM Distinctive Reduction Rules} \)
2.3 cCLAIM: core Klaim

The calculus cCLAIM has been introduced in [8] by eliminating from µKlaim the action read and by only considering monadic communications (i.e. tuples and templates containing only one field). The formal syntax of cCLAIM is given in Table 5. Notice that cCLAIM is a sub-calculus of µKlaim and thus it inherits from µKlaim the operational semantics.

2.4 l-cCLAIM: local core Klaim

l-cCLAIM is the version of cCLAIM where actions out and in can be only performed locally, i.e. the only remote primitive is the action eval. This calculus is introduced here for the first time and permits only local (intra-node) communications. Communication between processes at different nodes would have to be implemented by moving processes. The syntax of l-cCLAIM can be derived from the syntax of cCLAIM given in Table 5 by replacing the productions for process actions with

\[
\begin{align*}
a & ::= \text{in}(T)@u \mid \text{out}(t)@u \mid \text{eval}(P)@u \mid \text{new}(l) \\
t & ::= u \\
T & ::= u \mid !x
\end{align*}
\]

The operational semantics of l-cCLAIM is obtained by replacing rules (RED-OUT) and (RED-IN) of Table 4 with

\[
\begin{align*}
(\text{RED-OUT}) & \quad l :: \text{out}(l').P \leftrightarrow l :: P \mid \langle l' \rangle \\
(\text{RED-IN}) & \quad l :: \text{in}(T).P \mid \langle l' \rangle \leftrightarrow l :: P\sigma \quad \text{if} \; \text{match}(T,l') = \sigma
\end{align*}
\]
3 Expressivity: An Overview of the Results

In the full paper [7], we describe encodings between the languages presented so far, and discuss their relations with the asynchronous $\pi$-calculus [14], written $\pi_a$-calculus in the sequel. Here, we only sum up the results obtained and briefly comment upon them.

To study the correspondences established by the encodings, we consider a variety of semantic equivalences based on barbed bisimilarity. The latter is a uniform notion of equivalence among language terms, whose definition only relies on the existence of a reduction relation and of an observation predicate. Starting from barbed bisimilarity, more refined notions of equivalence can be "uniformly" obtained. In fact, given a family $F$ of language contexts, we can declare two processes $F$-equivalent if they are barbed bisimilar whenever plugged in any context of $F$.

We start by introducing barbed bisimilarity and congruence for the KLAIM-based process description languages presented so far. The observation predicate, or barb, for our languages is $N \downarrow l$ which is defined as follows

$$N \downarrow l \text{ iff } N \equiv (\nu \tilde{l})(N' \parallel l ::_\rho (t)) \text{ for some } \tilde{l}, N', \rho \text{ and } t \text{ s.t. } l \not\in \tilde{l}$$

where, of course, a non empty $\rho$ is present only in the case of KLAIM. As usual, $N \downarrow l$ stands for $\exists N' : N \redd N' \downarrow l$, where $\redd$ is the reflexive and transitive closure of $\rightarrow$. A context $C[\cdot]$ is a net with a hole $[\cdot]$ to be filled with any net. \footnote{In the case of KLAIM, we implicitly assume that the hole of $C[\cdot]$ can be filled only with those nets $N$ that guarantee that $C[N]$ is well-formed, i.e. allocation environments of nodes having the same address do coincide.} Formally,

$$C[\cdot] ::= [\cdot] \mid N \parallel C[\cdot] \mid (\nu l)C[\cdot]$$

Finally, we say that a binary relation $\Re$ between nets is

- **barb preserving**, if $N \Re M$ and $N \downarrow l$ imply $M \downarrow l$;
- **reduction closed**, if $N \Re M$ and $N \redd N'$ imply $M \redd M'$ and $N' \Re M'$;
- **$F$-context closed**, if $N \Re M$ implies $C[N] \Re C[M]$ for every context $C[\cdot]$ of $F$.

**Definition 3.1** Barbed bisimilarity, $\cong$, is the largest symmetric, barb preserving and reduction closed relation between nets.

**Definition 3.2** Barbed congruence, $\cong$, is the largest symmetric, barb preserving, reduction and context closed relation between nets.

The main results of our work are depicted in Table 6. There, a labelled arrow between two calculi, $X \xrightarrow{P} Y$, means that the language $X$ can be encoded...
in the language \( \mathcal{Y} \) and the encoding enjoys property \( \mathcal{P} \). The main properties we shall consider for the encodings will be based on the following two notions:

**Operational Correspondence:** An encoding \( \text{enc}(\cdot) \) of language \( \mathcal{X} \) into language \( \mathcal{Y} \) satisfies this property if for every reduction of an \( \mathcal{X} \)-term \( T_1 \) into a \( \mathcal{X} \)-term \( T_2 \) there exists a (possibly weak) reduction from the \( \mathcal{Y} \)-term \( \text{enc}(T_1) \) leading to the \( \mathcal{Y} \)-term \( \text{enc}(T_2) \). Viceversa, for every reduction of a \( \mathcal{Y} \)-term \( \text{enc}(T_1) \) into a \( \mathcal{Y} \)-term \( T \) there exists a reduction from the \( \mathcal{X} \)-term \( T_1 \) leading a \( \mathcal{X} \)-term \( T' \) such that \( T \) can be reduced (or even better, is equivalent) to \( \text{enc}(T') \).

**Full Abstraction w.r.t. \( EQ \):** An encoding \( \text{enc}(\cdot) \) of language \( \mathcal{X} \) into language \( \mathcal{Y} \) satisfies this property if for every pair of \( \mathcal{X} \)-terms \( T_1 \) and \( T_2 \) it holds that \( T_1 \equiv_{\mathcal{X}} T_2 \) if and only if \( \text{enc}(T_1) \equiv_{\mathcal{Y}} \text{enc}(T_2) \).

**Semantical Equivalence w.r.t. \( EQ \):** An encoding \( \text{enc}(\cdot) \) of language \( \mathcal{X} \) into language \( \mathcal{Y} \) satisfies this property if for every \( \mathcal{X} \)-term \( T \) it holds that \( T \equiv_{\mathcal{Z}} \text{enc}(T) \), for some language \( \mathcal{Z} \) containing both \( \mathcal{X} \) and \( \mathcal{Y} \).

Notice that the equivalence \( EQ \) in the above definitions is not a specific equivalence but a family of equivalences that has to be properly instantiated to the various languages considered, yielding \( EQ_{\mathcal{X}} \), \( EQ_{\mathcal{Y}} \) and \( EQ_{\mathcal{Z}} \). The stronger the equivalence the better the encoding, in that it more strongly attests that the target language has similar expressive power to the source one. Moreover, if an encoding is semantical equivalent w.r.t. \( EQ \) then it is also fully abstract w.r.t. the same equivalence. Thus, an encoding enjoying semantical equivalence is ‘better’ than an encoding enjoying fully abstraction, once the equivalence has been fixed. Finally, in the definition of semantical equivalence, the language \( \mathcal{Z} \) is useless whenever \( \mathcal{Y} \) is a sub-language of \( \mathcal{X} \). In this case, it suffices to require that \( T \equiv_{\mathcal{X}} \text{enc}(T) \) for every \( \mathcal{X} \)-term \( T \).

The equivalences we use in this paper are barbed bisimilarity and barbed congruence; the first one does not prescribe any contextual property, while the
second one requires that equivalences be preserved under all possible language contexts. As usual, see e.g. [14], barbed bisimilarity is coarser than barbed congruence. It often turns out that a ‘half-way’ solution between the two notions above is the appropriate one; it relies on what we call translated barbed congruence, written $\congruence$,:

An encoding $\mathit{enc}(\cdot)$ from language $\mathcal{X}$ to language $\mathcal{Y}$ is fully abstract w.r.t. $\congruence$ whenever the set of contexts in $\mathcal{Y}$ considered for context closure is formed by the translation via $\mathit{enc}(\cdot)$ of contexts in $\mathcal{X}$.

Indeed, if we consider the encoding as a protocol (i.e. a precise sequence of message exchanges), translated contexts represent opponents conform to the protocol. For most purposes, this result suffices since it precisely says that the source language can be faithfully compiled in the target one.

We now briefly conclude by commenting on the obtained results.

- **KLAIM vs $\mu$KLAIM.** As we said, the latter language is obtained from the former by removing higher-order data and dynamic binding of free locality variables. The first feature has already been implemented in a first-order language with name passing and restriction [13]. The encoding of the second feature is an example of the compilation of a dynamic naming discipline into a static one in presence of higher-order constructs, like the primitive $\mathit{eval}$. As witnessed by [15], this is a very difficult task. Thus, it is reasonable that the proposed encoding enjoys a property expressed only in terms of barbed bisimilarity, that is quite coarse. However, since barbed bisimilarity is a reduction closed relation, semantical equivalence implies operational correspondence. Thus, our result states that a KLAIM net and its encoding can simulate each other step-by-step, without altering the observation predicate.

- **$\mu$KLAIM vs cKLAIM.** The encoding has to properly implement actions $\mathit{read}$ and polyadic data exchanges. The fact that, in an asynchronous setting, a $\mathit{read}$ can be simulated by an $\mathit{in}$ followed by an $\mathit{out}$ of the datum accessed should not be surprising (see [1]). On the other hand, the fact that, in a LINDA setting, polyadic tuples can be replaced by monadic ones is something new; as far as we know, our result is the first one on this topic. The basic idea for the encoding is the implementation of a polyadic tuple by means of a process that sequentially produces the fields of the tuple. The receiving process accesses these (monadic) fields in an exclusive and ordered way. If the $i$-th tuple field matches against the $i$-th template field, the retrieving procedure goes on; otherwise, it stops and rolls back the involved processes. Notice that, like in the $\pi$-calculus [16], the encoding is not fully abstract w.r.t. barbed congruence. Indeed, since (atomic) polyadic actions are trans-
lated into sequences of monadic actions, contexts not abiding by the protocol schema imposed in the encoding can usually acquire enough information to distinguish two terms corresponding to the encoding of equivalent source terms.

- **cKlaim vs L-cKlaim.** In this case, the encoding translates remote actions into a mixture of migrations and local actions. The fact that the encoding enjoys semantical equivalence w.r.t. barbed congruence means that remote actions, although notationally convenient, do not change the expressive power of the considered calculi.

- **π_{a}-calculus vs cKlaim.** To conclude, we compare one of our languages with the asynchronous π-calculus [14]. The main idea is translating channels into localities. Thus, the message passing paradigm of the π_{a}-calculus can be compiled in the shared memory paradigm underlying LINDA. We could not find an encoding enjoying full abstraction w.r.t. barbed congruence. Although, we have not been able to exhibit impossibility result, proving full abstraction is in our view a very hard task. The problem is that, in the π_{a}-calculus, each free name is always associated to a channel; thus, the knowledge of a name implies that actions over a channel with that name can be always performed. This is not the case of KLAIM (and in the calculi derived from it); it is possible that a free name is not associated to a locality. Thus, since each KLAIM action is executed only if the target locality exists (see the operational rules for actions **in/read/out/eval**), full abstraction can be violated.

**Concluding Assessment**

The results we have summarised in this paper show that some design choices in a language are often driven by the aim of easing programming activity. Indeed, most of the existing calculi or languages for mobile processes are Turing complete (in particular, this applies to all the calculi presented in this paper); they mostly differ in the way specific tasks can be described or programmed. Moreover, some design choices greatly improve the possibilities of developing encodings enjoying stronger properties, while others hinder this possibility.

In our view, the present work improves the understanding of the language KLAIM and the appreciation of its specific design choices that make it significantly different from the standard process calculi. We believe that the results presented here can be exploited also for assessing expressiveness of other calculi with a communication paradigm similar to the one of KLAIM.
References


