In *Proc. of the 19th Symposium on Applied Computing (SAC'04)*, pages 1462–1467. 

# Controlling Data Movement in Global Computing Applications*

Daniele Gorla[1,2] Rosario Pugliese[2]

[1]Dipartimento di Informatica, Università di Roma "La Sapienza"

[2]Dipartimento di Sistemi e Informatica, Università di Firenze

e-mail: gorla@dsi.uniroma1.it , pugliese@dsi.unifi.it

## Abstract

We present a programming notation aiming at protecting the secrecy of both host and agent data in global computing applications. The approach exploits annotations with sets of node addresses, called regions. A datum can be annotated with a region that specifies the network nodes that are allowed to interact with it. Network nodes come equipped with two region annotations specifying the nodes that can send data and spawn processes over them. The language semantics guarantees that computation proceeds according to these region constraints. To minimize the overhead of runtime checks, a static compilation phase is exploited. The proposed approach is largely independent of a specific programming language; however, to put it in concrete form, here we focus on its integration within the process language $\mu$KLAIM. We prove that in compiled $\mu$KLAIM nets, data can be manipulated only by authorized users. We also give a more local formulation of this property, where only a subnet is compiled. Finally, we use our theory to model the secure behaviour of a UNIX-like multiuser system.

## 1 Introduction

In the design of programming languages for global computing applications, the integration of security mechanisms is a major challenge and a great effort has been recently devoted to embed such mechanisms within standard programming features. Several sensible language-based security techniques have been proposed in the literature, including type systems [12, 3, 5, 10], control and data flow analysis [11, 15, 6, 2], in-lined reference monitoring [8] and proof-carrying code [14]; some of these techniques are analyzed and compared in [19]. One major goal is to develop a language that is both flexible, expressive and safe; unfortunately, these requirements are often in contrast. For example, the possibility of exploiting mobile code deeply increased the flexibility of programming languages, but introduced new problems concerning the security of classified data. Indeed, since global computing has to take into account *open* networks, existence in the environment of malicious principals must be considered that can put security of data at risk. For instance, one can easily imagine malicious mobile processes attempting to access private data of the network node hosting them. Similarly, a malicious node can threaten a mobile process by trying to compromise its integrity through code modification or its secrecy through leak of sensitive data.

Therefore, to enhance its appropriateness for global computing applications, a programming language should come equipped with a solid foundational model that also encompasses security features. The proof that an application is 'safe' could then be done by relying on formal methods. To be realistic and useful for global computing, the language security model should $(a)$ consider existence of misbehaving entities in the execution environment of applications, and $(b)$ rely only on a local knowledge of such environment. Condition $(b)$ is necessary because it is impossible in practice to collect global information in a network of millions of users (like the Internet), that are under the control of different administration authorities and can be malicious.

The major contribution of this paper is the definition of an approach that permits protecting the secrecy of both host and agent data in global computing applications by relying on additional programming notation. The approach we propose, that is inspired by Confined-λ [13], exploits program annotations with *regions*, i.e. sets of node addresses, as follows. A datum can be annotated with a region that specifies the network nodes that are allowed to interact with it. This mechanism allows programmers to control the nodes that can share specific data, and to avoid them to be visible to other nodes. Moreover, nodes come equipped with two region annotations specifying the nodes that can send data and spawn processes over them. This mechanism allows the administrator of a node to control the data/processes the node can host, and to refuse malicious agents and undesired data. The language semantics guarantees that computation proceeds according to these region constraints. For example, a process $P$ can access a datum $t$ only if $P$'s execution does not export $t$ outside its region (namely, if $P$ does not write $t$ in a network node not included in $t$'s region or, similarly, if $P$ does not bring $t$ with itself while migrating to a node not included in $t$'s region). Enforcing similar requirements implies some form of code inspection, that would be too expensive to be entirely performed at runtime. Therefore, to minimize the runtime checks thus making the operational semantics as efficient as possible, a preliminary static compilation phase is exploited.

Our approach is largely independent of a specific programming language; however, to put it in concrete form, in this paper we focus on its integration within the process language $\mu$KLAIM [10]. $\mu$KLAIM is at the core of KLAIM (Kernel Language for Agents Interaction and Mobility, [5]), an experimental language specifically designed to program distributed systems made up of several mobile components interacting through multiple distributed tuple spaces. The tuple space paradigm, that was firstly introduced by the coordination language LINDA [9], defines a *tuple space* to be a multiset of *tuples*, that are sequences of information items. Tuples are *anonymous* and *associatively* selected from tuple spaces by means of a

*Work partially supported by EU FET - Global Computing initiative, project MIKADO IST-2001-32222, and by MIUR project NAPOLI.

| | | | |
|---|---|---|---|
| $N$ | ::= | $l\ {}_{r_d}{::}_{r_p}\ C \mid N_1 \parallel N_2$ | NETS |
| $C$ | ::= | $\langle et \rangle \mid P \mid C_1 \mid C_2$ | COMPONENTS |
| $P$ | ::= | $\mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid X \mid \mathbf{rec}\,X.P$ | PROCESSES |
| $a$ | ::= | | ACTIONS |
| | | $\mathbf{out}(t)@\ell$ | (output) |
| | $\mid$ | $\mathbf{in}(T)@\ell$ | (input) |
| | $\mid$ | $\mathbf{eval}(P)@\ell$ | (spawning) |
| | $\mid$ | $\mathbf{newloc}(u)$ | (creation) |
| $t$ | ::= | $[e]_r \mid [\ell]_r \mid t_1, t_2$ | TUPLES |
| $et$ | ::= | $[V]_r \mid [l]_r \mid et_1, et_2$ | EVALUATED TUPLES |
| $T$ | ::= | $e \mid \ell \mid !x \mid T_1, T_2$ | TEMPLATES |
| $e$ | ::= | $V \mid z \mid \ldots$ | EXPRESSIONS |

**Table 1. Syntax**

*pattern-matching* mechanism. KLAIM handles multiple distributed tuple spaces by placing a tuple space on each *node* of a *net*. Differently from other programming notations enabling process distribution and mobility, in KLAIM the network infrastructure (set up by some *administrators*) is clearly distinguishable from user processes (written by *programmers*) and is explicitly modelled, which we believe gives a more accurate description of the computer systems we are interested to. Moreover, in [7] several messaging models for mobile processes are examined and it is shown that the *blackboard* approach, that encompasses the one based on tuple spaces, is one of the most suitable, also because of its flexibility. General evidence of the success gained by the multiple tuple spaces paradigm is given by the many run-time systems that implement it, both from industries (e.g. SUN JavaSpaces [1] and IBM T Spaces [20]) and from universities (e.g. PageSpace [4], WCL [18], Lime [17] and TuCSoN [16]).

The results we prove ensure that execution of nets resulting from the compilation phase always respects the data annotations therein, thus data can be seen only by authorized users. However, we cannot assume knowledge of the whole net, thus we also prove a more general result stating that if only a subnet is compiled, then, during the evolution of the whole net, no violations of data annotations will ever occur in that subnet. The paper ends with an application of our approach to model the secure behaviour of a UNIX-like multiuser system and with comparisons to related work.

## 2 The Syntax of the Language

The language we use in this paper is a minor variation of $\mu$KLAIM [10] and its syntax is reported in Table 1. We assume the following, pairwise disjoint, countable sets: $\mathcal{X}$, *process variables*, ranged over by $X, X', X_1, \ldots$; $\mathcal{L}$, *localities*, ranged over by $l, l', l_1, \ldots$; $\mathcal{U}$, locality variables, ranged over by $u, u', u_1, \ldots$; $\mathcal{V}$, *basic values*, ranged over by $V, V', V_1, \ldots$; $\mathcal{Z}$, *value variables*, ranged over by $z, z', z_1, \ldots$. We let $\ell$ to range over $\mathcal{L} \cup \mathcal{U}$ and $x$ to range over $\mathcal{U} \cup \mathcal{Z}$.

The syntax of *expressions*, ranged over by $e$, is deliberately not specified; we just assume that expressions contain, at least, basic values and variables. *Localities* $l$ are the addresses of nodes. *Tuples* $t$ are sequences of annotated actual fields, that contain information items (expressions, localities or locality variables). *Templates* $T$ are used to select data in a tuple space (TS, for short); they are sequences of actual and formal fields. The latter ones are used to bind variables to values and are written $!z$ or $!u$. Data are *evaluated tuples*, $\langle et \rangle$, i.e. sequences of annotated values and localities.

Each actual field in a tuple is annotated with a *data region*, $r$, expressing the subnet where the field will be allowed to occur. A *region* can be either a finite subset of $\mathcal{L} \cup \mathcal{U}$ or a distinct element $\forall$ used to refer to the whole set $\mathcal{L} \cup \mathcal{U}$. The set of all regions $\mathcal{R}$ together with the relation $\subseteq$ forms a poset whose top element is $\forall$. Thus, e.g., $r_1 \cup r_2$ is $\forall$ if and only if $r_1$ or $r_2$ is $\forall$. Similarly, $\ell \in \forall$ holds always true. For the sake of readability, we shall omit the region annotation whenever it is $\forall$.

*Processes* are built up from the inactive process **nil** and from the basic operations by using prefixing, parallel composition and recursion. $\mu$KLAIM supplies four different basic operations, also called *actions*, to put/remove tuples from TSs, to spawn processes in execution and to create new nodes. The last operation is not indexed with an address because it always acts locally; all the other operations explicitly indicate the (possibly remote) address where they will take place.

Variables occurring in process terms can be *bound*; more precisely, prefix $\mathbf{in}(T)@\ell.P$ binds the variables in the formal fields of $T$, prefix $\mathbf{newloc}(u).P$ binds $u$ and $\mathbf{rec}\,X.P$ binds $X$. In all these cases, $P$ is the scope of the bindings. A variable that is not bound is called *free*. The sets $\mathrm{BV}(P)$ and $\mathrm{FV}(P)$ (of bound and free variables, resp., of $P$) are defined accordingly, and so is $\alpha$-*conversion*. In the sequel, we shall assume that bound variables in processes are all distinct and different from the free variables (this is always possible by using $\alpha$-conversion). Moreover, we extend functions $\mathrm{FV}(\cdot)$ and $\mathrm{BV}(\cdot)$ to templates in the obvious way.

*Nets* are finite collections of nodes where processes and data can be allocated. A *node* is a quadruple $l\ {}_{r_d}{::}_{r_p}\ C$, where locality $l$ is the address (i.e. network reference) of the node, $C$ is the (parallel) component located at $l$ and $r_d/r_p$ is the *data/process trust region* of the node (i.e. the set of localities of nodes that can respectively write data at $l$'s TS and spawn processes to $l$) as established by the node administrator. In general, $r_p \subseteq r_d$ since accepting processes is, in general, more dangerous than accepting data; however, we do not impose any restriction on this. In the sequel, we only take into account *closed* nets, i.e. nets only containing processes without free variables and whose node regions only contains localities (similarly to many real compilers, we consider terms containing free variables as programming errors).

The original presentation of $\mu$KLAIM [10] mainly differs from the present one in two aspects. Firstly, the typing annotations in the language are different (because the types in [10] were designed to control process actions). Secondly, to save space, we omitted the LINDA primitive **read** (to access data in TSs without removing them) because the **read** actions behave similarly to **in** actions.

## 3 A Preliminary Compilation

The language presented in the previous section is a mean to program applications where, during the computation, a datum can only appear in localities contained in its region annotation. The main goal of the runtime semantics is to enforce this requirement. However, in order to make the semantics as efficient as possible, a preliminary compilation phase is introduced. The activities of the static compilation deal with the following requirements:

1. a datum can be seen at (i.e. can pass through) $\ell$ if $\ell$ is contained in the region annotation of the datum;
2. a process retrieving a datum cannot send the datum outside its region.

These activities require some form of code inspection that is too expensive to be performed when the action is executed. The compilation phase relieves the runtime from such inspection by performing check 1. statically and by annotating template formal fields with regions to dynamically perform check 2. more efficiently. Hence, the syntax of templates becomes

$$T ::= e \mid \ell \mid [!x]^r \mid T_1, T_2$$

To better distinguish the annotations put by the programmers/administrators from those put by the compiler, we shall write

$$
\begin{array}{rcll}
N_1 \parallel N_2 & \succ & N_1' \parallel N_2' & \text{if } N_i \succ N_i' \text{ for } i=1,2 \\
l\ {}_{r_d}{::}_{r_p}\ C & \succ & l\ {}_{r_d}{::}_{r_p}\ C' & \text{if } C \succ_l C' \wedge (r_d = \forall \ \vee \ r_d \subset \mathcal{L}) \wedge (r_p = \forall \ \vee \ r_p \subset \mathcal{L}) \\
C_1 | C_2 & \succ_l & C_1' | C_2' & \text{if } C_i \succ_l C_i' \text{ for } i=1,2 \\
\langle et \rangle & \succ_l & \langle et \rangle & \text{if } l \in reg(et) \\
P & \succ_l & P' & \text{if } \emptyset \# P \succ_l \emptyset \# P' \\
\\
\Gamma \# P & \succ_\ell & \Gamma \# P & \text{if } P = \mathbf{nil} \vee P = X \\
\Gamma \# \mathbf{rec}\, X.P & \succ_\ell & \Gamma' \# \mathbf{rec}\, X.P' & \text{if } \Gamma \uplus \{X:\emptyset\} \# P \succ_\ell \Gamma' \uplus \{X:\emptyset\} \# P' \\
\Gamma_1 \# P_1 | P_2 & \succ_\ell & \Gamma_3 \# P_1' | P_2' & \text{if } \Gamma_i \# P_i \succ_\ell \Gamma_{i+1} \# P_i' \text{ for } i=1,2 \\
\Gamma \# \mathbf{newloc}(u).P & \succ_\ell & \Gamma' \# \mathbf{newloc}(u).P' & \text{if } \Gamma \uplus \{u:\emptyset\} \# P \succ_\ell \Gamma' \uplus \{u:r'\} \# P' \\
\Gamma \# \mathbf{out}(t)@\ell'.P & \succ_\ell & \Gamma' + \{x:r\}_{x \in \mathrm{FV}(t)} \# \mathbf{out}(t)@\ell'.P' & \text{if } \{\ell,\ell'\} \subseteq reg(t) = r \wedge \Gamma \# P \succ_\ell \Gamma' \# P' \\
\Gamma_1 \# \mathbf{eval}(P_1)@\ell'.P_2 & \succ_\ell & \Gamma_3 + \{x:\{\ell\}\}_{x \in \mathrm{FV}(P_1)} \# \mathbf{eval}(P_1')@\ell'.P_2' & \text{if } \ell \in reg(P_1) \wedge \Gamma_1 \# P_1 \succ_{\ell'} \Gamma_2 \# P_1' \wedge \Gamma_2 \# P_2 \succ_\ell \Gamma_3 \# P_2' \\
\Gamma \# \mathbf{in}(T)@\ell'.P & \succ_\ell & \Gamma' \nearrow^{\mathrm{BV}(T)} \# \mathbf{in}(T')@\ell'.P' & \text{if } \Gamma \uplus \{x:\{\ell\}\}_{x \in \mathrm{BV}(T)} \# P \succ_\ell \Gamma' \uplus \{x:r_x\}_{x \in \mathrm{BV}(T)} \# P' \\
 & & & \quad \wedge \{x:r_x'\}_{x \in \mathrm{BV}(T)} = \big( \{x:(r_x - \{x\})\}_{x \in \mathrm{BV}(T)} \big) \nearrow^{\mathrm{BV}(T)} \\
 & & & \quad \wedge \ T' = T[{}^{[!x]^{r_x'}}/{!x}]_{x \in \mathrm{BV}(T)}
\end{array}
$$

where

$$
\begin{array}{rcl@{\qquad}rcl}
reg([e]_r) & = & reg([\ell]_r) \triangleq r & reg(t_1,t_2) & \triangleq & reg(t_1) \cap reg(t_2) \\
reg(\mathbf{nil}) & = & reg(X) \triangleq \forall & reg(P_1|P_2) & \triangleq & reg(P_1) \cap reg(P_2) \\
reg(\mathbf{out}(t)@\ell.P) & \triangleq & reg(t) \cap reg(P) & reg(\mathbf{eval}(P_1)@\ell.P_2) & \triangleq & reg(P_1) \cap reg(P_2) \\
reg(\mathbf{newloc}(u).P) & = & reg(\mathbf{in}(T)@\ell.P) = reg(\mathbf{rec}\, X.P) \triangleq reg(P) & & &
\end{array}
$$

**Table 2. The Compilation Procedure**

the latter ones as superscripts and the former ones as subscripts. Intuitively, $[!x]^r$ states that the datum replacing $x$ will pass through the localities in $r$.

The compilation procedure is given in Table 2 and is written $N \succ N'$; intuitively, this judgement means that the procedure takes a net $N$ (written according to the syntax in Table 1) and returns a net $N'$ obtained from $N$ by annotating all the template formal fields with a region containing the nodes where the values received will pass through. E.g., in process $\mathbf{in}(!z)@l.\mathbf{out}([z]_r)@l'$ the declaration $!z$ of variable $z$ must be associated to region $r$. Moreover, the compiler verifies that each component located in a node $l$ contains only data that can be seen by $l$ (this is done by the judgement $\succ_l$). Finally, it also verifies that actions **out** and **eval** send data/code to nodes where the data/code can appear without violating the region annotations. Of course, if any of the performed checks fails, the compilation fails too (namely, not all syntactically legal nets are compilable).

The auxiliary function $reg(\_)$ returns the intersection of the data regions occurring as its arguments. Moreover, $\succ$ and $\succ_l$ rely on an auxiliary procedure $\Gamma \# P \succ_\ell \Gamma' \# P'$ where $\Gamma$, called *environment*, is a finite mapping $(\mathcal{Z} \cup \mathcal{U} \cup \mathcal{X}) \to \mathcal{R}$ such that $\mathrm{FV}(P) \subseteq dom(\Gamma)$. Thus, the procedure $\emptyset \# P \succ_\ell \emptyset \# P'$ is defined only if $P$ is closed; in that case, for each template formal field in $P$, a region annotation describing the use of that field in the continuation process (i.e. where it will be sent) is determined and used to decorate $P$ (thus obtaining $P'$). Such regions are calculated by the compiler by considering the locality where the process runs (the $\ell$ decorating $\succ_\ell$) and examining the localities where the variables can appear upon execution of actions **out** and/or **eval**. Notice, however, that some care must be taken when annotating fields because otherwise closed nets can become open upon compilation. As an example, consider the nodes (that we both consider legal)

$$
\begin{array}{ll}
l :: \mathbf{in}(!z)@l'.\mathbf{in}(!u)@l''.\mathbf{out}([z]_{\{l,u\}})@l & (\star) \\
l :: \mathbf{in}(!u)@l'.\mathbf{out}([u]_{\{l,u\}})@u & (\star\star)
\end{array}
$$

Blindly annotating these processes would result in

$$
\begin{array}{l}
l :: \mathbf{in}([!z]^{\{l,u\}})@l'.\mathbf{in}([!u]^{\{l\}})@l''.\mathbf{out}([z]_{\{l,u\}})@l \\
l :: \mathbf{in}([!u]^{\{l,u\}})@l'.\mathbf{out}([u]_{\{l,u\}})@u
\end{array}
$$

that are open because of the occurrence of $u$ in the regions of $!z$ and $!u$, resp.. The solution we designed to accept $(\star)$ is to assign $!z$ the region annotation $\forall$. This is reasonable since $\mathbf{in}([!z]^{\{l,u\}})@l'$ means 'retrieve a datum from $l'$ and share it with *a generic* locality of the net' (because $u$ can be dynamically replaced with any locality name). The solution we designed to accept $(\star\star)$ is to remove $u$ from $!u$ region annotation and assume that a locality can always occur in the node having that locality as address.

In Table 2, we write $\{\ldots\}_{i \in I}$ to mean $\bigcup_{i \in I}\{\ldots\}$. Function $\uplus$ denotes union between environments with disjoint domains. Function $\Gamma \nearrow^S$, where $S \subset_{\mathrm{fin}} \mathcal{U}$, is inductively defined as

$$
\begin{array}{rcl@{\qquad}l}
\emptyset \nearrow^S & \triangleq & \emptyset & \\
(\Gamma \uplus \{x:r\}) \nearrow^S & \triangleq & \begin{cases} \Gamma \nearrow^S \uplus \{x:r\} & \text{if } r \cap S = \emptyset \\ \Gamma \nearrow^S \uplus \{x:\forall\} & \text{otherwise} \end{cases} & \\
(\Gamma \uplus \{X:\emptyset\}) \nearrow^S & \triangleq & \Gamma \nearrow^S \uplus \{X:\emptyset\} &
\end{array}
$$

and is used to eliminate anomalies like the annotation for $z$ in $(\star)$. Function $+$ extends the information of an environment; formally

$$
\begin{array}{rcl@{\qquad}l}
\Gamma + \emptyset & \triangleq & \Gamma & \\
\Gamma + \{x:r\} & \triangleq & \Gamma' \uplus \{x:r \cup r'\} & \text{if } \Gamma = \Gamma' \uplus \{x:r'\} \\
\Gamma + (\{x:r\} \uplus \Gamma') & \triangleq & (\Gamma + \{x:r\}) + \Gamma' &
\end{array}
$$

Before concluding this section, we briefly comment on some compilation rules. The compilation of $N$ also verifies that $N$ is closed. The regions associated to process variables and to locality variables bound by action **newloc** are useless: they are just put to give the environment a uniform structure. When dealing with process $\mathbf{out}(t)@\ell'.P$ at $\ell$, the procedure firstly calculates the intersection of the regions occurring in $t$; let us call $r$ the resulting region. Then, it verifies that both the hosting locality $\ell$ and the target locality $\ell'$ can see $t$. Finally, the continuation process $P$ is compiled in the environment $\Gamma$, thus obtaining the annotated process $P'$ and the environment $\Gamma'$. Hence, the result of the compilation will be $\mathbf{out}(t)@\ell'.P'$ together with $\Gamma'$ extended with the information that the variables occurring in $t$ could be seen at $r$. Similar observations also hold when compiling process $\mathbf{eval}(P_1)@\ell'.P_2$; additionally, notice that the target locality $\ell'$ is ensured to occur in $P_1$'s region by the judgement $\succ_{\ell'}$ (that also calculates further annotations for $\mathrm{FV}(P_1)$). Finally, when dealing with process $\mathbf{in}(T)@\ell'.P$ at $\ell$, the procedure should compile $P$ in the environment $\Gamma$ extended by associating the variables bound by $T$ to region $\{\ell\}$. At the end of this compilation, the region annotations calculated for such variables are put in $T$, obtaining the (annotated) template $T'$. Notice that, since some of these variables can occur in $\Gamma'$ region annotations (because of anomalies like $(\star)$), the environment resulting

from this compilation must be $\Gamma' \nearrow^{\mathrm{BV}(T)}$. Similarly, since some locality variable $u$ bound by $T$ can occur in $u$'s region or in the region of some other variable in $\mathrm{BV}(T)$ (thus generating anomalies like $(\star\star)$ and $(\star)$), the annotated template $T'$ is obtained from $T$ by using $r'_x$ instead of $r_x$, where $r'_x$ is obtained from $r_x$ by ruling out all potential anomalies.

DEFINITION 3.1 (COMPILED NETS). *A net $N$ is deemed* compiled *if there exists a net $N'$ (written according to the syntax of Table 1) such that $N' \succ N$.*

# 4 The Operational Semantics

Nets are executed according to the reduction relation $\succ\!\!\rightarrow$ defined in Table 3. $\succ\!\!\rightarrow$ relates configurations of the form $L \triangleright N$, where $L$ is such that $loc(N) \subseteq L \subset_{fin} \mathcal{L}$ and function $loc(N)$ returns the set of localities occurring in $N$. In a configuration $L \triangleright N$, $L$ is needed to ensure global freshness of new addresses. For the sake of readability, when a reduction does not generate any fresh addresses, we write $N \succ\!\!\rightarrow N'$ instead of $L \triangleright N \succ\!\!\rightarrow L \triangleright N'$. The semantics exploits the following auxiliary functions and relations:

- a *structural congruence* relation, $\equiv$, equating $\alpha$-convertible processes, equating processes obtained by folding/unfolding recursive definitions, stating that "$\|$" is commutative and associative, and that **nil** acts as the identity for "$|$";
- a tuple/template evaluation function $\mathcal{E}[\![\cdot]\!]$ that turns basic expressions into basic values (whenever possible);
- *substitutions* are functions from value and locality variables to values and localities. We use '$\varepsilon$' and '$\circ$' to denote the empty substitution and substitutions composition. We want to remark that, when applied to a process $P$, a substitution also acts on the region annotations in $P$.
- a function *match* that verifies whether a (evaluated) tuple matches against a (evaluated) template; this happens whenever they both have the same number of fields and corresponding fields match. Two actual fields match if they are identical, while a formal field matches any actual of the same sort provided that the use of the formal (i.e. the region put by the compiler) respects the specifications of the actual (i.e. its data region). When matching succeeds, *match* returns a substitution associating the variables in the formals to the corresponding actuals. Formally, function *match* is defined as:

$$match(V,[V]_r) = \varepsilon \qquad \frac{r \subseteq r'}{match([!z]^r,[V]_{r'}) = [V/z]}$$
$$match(l,[l]_r) = \varepsilon \qquad match([!u]^r,[l]_{r'}) = [l/u]$$

$$\frac{match(T_1,t_1) = \sigma_1 \qquad match(T_2,t_2) = \sigma_2}{match((T_1,T_2),(t_1,t_2)) = \sigma_1 \circ \sigma_2}$$

Some comments on the operational semantics are now in order. We put the dynamic checks $l \in r'_d$ and $l \in r'_p$ as premises of rules for actions **out**/**eval** to prevent an untrusted node $l$ to send data/code over $l'$. Notice that no static check could enforce this property without loss of expressivity: e.g., in $\mathbf{in}(!u)@l.\mathbf{eval}(\ldots)@u$, it is statically impossible to know which locality will replace $u$ and, thus, it is impossible to determine whether the locality executing the **eval** is trusted by the target locality or not. Moreover, we assume that a node $l$ trusts every node $l'$ it creates. This seems us reasonable since, once created, $l'$ is not known to any other node in the net; thus, $l$ can use it as a sort of *private* resource and can decide the nodes of the net that can know it (by also exploiting region annotations).[1] Finally, the fifth rule turns a parallel between components into a parallel between nodes; this is necessary to present the semantics in a simpler form.

$$\frac{l \in r'_d \qquad et = \mathcal{E}[\![t]\!]}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{out}(t)@l'.P \,\|\, l\ {}_{r'_d}{::}_{r'_p}\ C' \succ\!\!\rightarrow l\ {}_{r_d}{::}_{r_p}\ P \,\|\, l'\ {}_{r'_d}{::}_{r'_p}\ C' \,|\, \langle et\rangle}$$

$$\frac{l \in r'_p}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{eval}(Q)@l'.P \,\|\, l'\ {}_{r'_d}{::}_{r'_p}\ C' \succ\!\!\rightarrow l\ {}_{r_d}{::}_{r_p}\ P \,\|\, l'\ {}_{r'_d}{::}_{r'_p}\ C' \,|\, Q}$$

$$\frac{match(\mathcal{E}[\![T]\!],et) = \sigma}{l\ {}_{r_d}{::}_{r_p}\ \mathbf{in}(T)@l'.P \,\|\, l'\ {}_{r'_d}{::}_{r'_p}\ \langle et\rangle \succ\!\!\rightarrow l\ {}_{r_d}{::}_{r_p}\ P\sigma \,\|\, l'\ {}_{r'_d}{::}_{r'_p}\ \mathbf{nil}}$$

$$\frac{l' \notin L}{L \triangleright l\ {}_{r_d}{::}_{r_p}\ \mathbf{newloc}(u).P \ \succ\!\!\rightarrow\ L \cup \{l'\} \triangleright l\ {}_{r_d \cup \{l'\}}{::}_{r_p \cup \{l'\}} P[l'/u] \,\|\, l'\ {}_{r_d \cup \{l'\}}{::}_{r_p \cup \{l'\}}\ \mathbf{nil}}$$

$$\frac{L \triangleright l\ {}_{r_d}{::}_{r_p}\ C_1 \,\|\, l\ {}_{r_d}{::}_{r_p}\ C_2 \,\|\, N \succ\!\!\rightarrow L' \triangleright l\ {}_{r'_d}{::}_{r'_p}\ C'_1 \,\|\, l\ {}_{r_d}{::}_{r_p}\ C'_2 \,\|\, N'}{L \triangleright l\ {}_{r_d}{::}_{r_p}\ C_1 \,|\, C_2 \,\|\, N \ \succ\!\!\rightarrow\ L' \triangleright l\ {}_{r'_d}{::}_{r'_p}\ C'_1 \,|\, C'_2 \,\|\, N'}$$

$$\frac{L \triangleright N_1 \ \succ\!\!\rightarrow\ L' \triangleright N'_1}{L \triangleright N_1 \,\|\, N_2 \ \succ\!\!\rightarrow\ L' \triangleright N'_1 \,\|\, N_2}$$

$$\frac{N_1 \equiv N'_1 \qquad L \triangleright N'_1 \ \succ\!\!\rightarrow\ L' \triangleright N'_2 \qquad N'_2 \equiv N_2}{L \triangleright N_1 \ \succ\!\!\rightarrow\ L' \triangleright N'_2}$$

**Table 3. Operational Semantics**

A straightforward property of the operational semantics ensures *integrity* of the components located at a node.

PROPOSITION 4.1. *Let $loc(N) \triangleright N \succ\!\!\rightarrow L' \triangleright N'$, $l \notin L' - loc(N)$ and $l\ {}_{r_d}{::}_{r_p}\ C$ be a node of $N'$. Then, for any parallel component $C'$ in $C$ it holds that: $(i)$ either $C'$ was located at $l$ in the initial configuration $N$, or $(ii)$ $C'$ is a datum written at $l$ by a node in $r_d$, or $(iii)$ $C'$ is a process spawned to $l$ by a node in $r_p$.*

Our main results state that compiled nets always reduce to compiled nets and that compiled nets do respect region annotations. The former result can be viewed as a form of subject reduction where the property that remains invariant during reduction is the fact that a net is compiled, while the latter result can be viewed as a form of safety where the property guaranteed by the fact that a net is compiled is that there are no immediate violations of data regions. Together, these results imply soundness of our theory, i.e. no violation of data regions will ever occur during the evolution of compiled nets.

THEOREM 4.1 (SUBJECT REDUCTION). *If $N$ is compiled and $loc(N) \triangleright N \succ\!\!\rightarrow L' \triangleright N'$ then $N'$ is compiled.*

DEFINITION 4.1. *A net $N$ is* safe *if for any $l\ {}_{r_d}{::}_{r_p}\ C$ in $N$, it holds that $l$ occurs in the region of each datum in $C$.*

THEOREM 4.2 (SAFETY). *If $N$ is compiled then $N$ is safe.*

The results given above can be generalized by requiring only a subnet of the whole net to be compiled[2]. We call *r-subnet* of $N$ the net formed by all the nodes $l\ {}_{r_d}{::}_{r_p}\ C$ in $N$ such that $\{l\} \cup r_d \cup r_p \subseteq r$.

[1] For the sake of simplicity, we assigned $l'$ the trust regions of $l$. It is easy to extend the language for allowing the programmer to explicitly specify the trust regions of a newly created node.

[2] Indeed, by using the convention that absence of a region annotation means $\forall$, a not compiled net can be executed according to the rules in Table 3 by (safely) considering all its template annotations as $\forall$.

Notice that such a net is not necessarily defined for all $r$; of course it is always defined for $r = \forall$ and coincides with $N$. By denoting with $\succ\!\!\rightarrow^*$ the reflexive and transitive closure of $\succ\!\!\rightarrow$, we obtain

THEOREM 4.3 (SOUNDNESS). *Let the $r$-subnet of $N$ be defined and compiled. If $loc(N) \triangleright N \succ\!\!\rightarrow^* L' \triangleright N'$ then the $r'$-subnet of $N'$ is defined and safe, where $r' = r \cup (L' - loc(N))$.*

**Dynamic trust.** We can handle trust regions more dynamically by extending the language with two actions **trust**$(l)$ and **warn**$(l)$ to, respectively, add/remove $l$ from the trust region of the node executing the action (in this way, e.g., a node can choose whether trusting or not a newly created node). However, more runtime checks are needed in this more expressive framework. In particular, none of the two requirements given at the beginning of Section 3 can be statically enforced. Thus, the compiler can only attach regions to the arguments of actions **in**, **out** and **eval** to make the dynamic checks more efficient. We omit the details from this extended abstract.

# 5 An Example: a Multiuser System

In this section we use the framework presented so far to program a simple but meaningful example. For the sake of readability, in the rest of this section we will omit trailing occurrences of process **nil**, and use parameterized process definitions and strings. Moreover, we borrow from [10] the primitive **read** that behaves similarly to **in** but, after its execution, it leaves the accessed datum in the TS.

We present the behaviour of a simple UNIX-like multiuser system, where users can login (exploiting a password-based approach) and use the system functionalities, which consist in reading/writing files or executing programs. For the sake of clarity, we shall present the system in three steps and, finally, we shall merge them together. Let $l_S$ be the address of the server, $\forall$ be its data trust region and $\emptyset$ be its starting trust region (thus no user can spawn code on $l_S$).

**User Identification.** We start with programming the identification of different users via passwords. Let $l_p$ be a private repository used by $l_S$ to record the users known and their passwords. Thus, $l_p$ hosts the component

$$\langle l_1, [pwd_1]_{\{l_1,l_p,l_S\}} \rangle \;\mid\; \ldots \;\mid\; \langle l_n, [pwd_n]_{\{l_n,l_p,l_S\}} \rangle$$

Let $l$ be a user wanting to log in $l_S$. If $l$ is already known to $l_S$ (i.e. it is one of the $l_i$s), then $l$ can use a process like

$$\mathbf{out}(\text{“}login\text{”}, l, [pwd]_{\{l,l_S\}})@l_S.\mathbf{in}(\text{“}logged\text{”})@l_S.\ldots$$

communicating with the server process

$$Login(l_p) \triangleq \mathbf{rec}X.\,\mathbf{in}(\text{“}login\text{”}, !u, !z)@l_S.\big( X \mid \mathbf{read}(u,z)@l_p.\mathbf{out}([\text{“}logged\text{”}]_{\{l_S,u\}})@l_S \big)$$

Intuitively, $l$ requires a connection by sending its user ID (its locality) and its password; the server checks whether this information is correct and sends back an ack, activating the remainder of $l$ computation. Notice that the region annotations to *pwd* and “*logged*” rule out denial of service attacks of a nasty intruder (aimed at cancelling the request of login or the corresponding ack).

If the user is not registered in $l_S$ yet, he can send an “hello” request to the server containing its address and wait for a password

$$\mathbf{out}([\text{“}hello\text{”}]_{\{l,l_S\}}, l)@l_S.\mathbf{in}(\text{“}registered\text{”}, !pwd)@l_S.\ldots$$

The server then handles this request with the process

$$NewUser(l_p) \triangleq \mathbf{rec}X.\,\mathbf{in}(\text{“}hello\text{”}, !u)@l_S.\big( X \mid \textit{create a fresh } pwd\,.\,\mathbf{out}(u, [pwd]_{\{u,l_S,l_p\}})@l_p.\ \mathbf{out}(\text{“}registered\text{”}, [pwd]_{\{l_S,u\}})@l_S \big)$$

Notice that a locality $l'$ different from $l$ can send $l_S$ a request for a new password pretending to be $l$: the only difference with the “hello” message given above is that the message now should contain also $l'$ in the data region. However, the server will report the new password to $l$ and the region put on the password will ensure that *pwd* will not leave $l$. Thus, $l'$ can withdraw *pwd* only by sending a process to $l$ and then acting in $l$ with the new password. This can be possible only if $l$ trusts $l'$, implying that $l$ accepts this ‘suspicious’ activity of $l'$.

**The File System.** We now consider a server handling a file system where different users can write/read data. Let $l_f$ be a private repository used by $l_S$ to store the files. A file named $N$, whose content is the string $S$, readable by users in $r$ and writtable by users in $r'$, is stored in $l_f$ as the component

$$C_N \triangleq \langle N, [\text{“}read\text{”}]_{r \cup \{l_S,l_f\}}, [\text{“}written\text{”}]_{r' \cup \{l_S,l_f\}} \rangle \;\mid\; \langle N, S \rangle$$

Intuitively, “*read*” and “*written*” are just dummy data used to properly store the regions $r$ and $r'$. Then, the server handles requests for reading and writing files with the following processes

$$Read(l_f) \triangleq \mathbf{rec}X.\,\mathbf{in}(\text{“}read\text{”}, !u, !n)@l_S.\big( X \mid \mathbf{read}(n, !z_r, !z_w)@l_f.\mathbf{read}(n, !z)@l_f.\ \mathbf{out}([z_r]_{\{l_f,u\}}, n, z)@u \big)$$

$$Write(l_f) \triangleq \mathbf{rec}X.\,\mathbf{in}(\text{“}write\text{”}, !u, !n, !z)@l_S.\big( X \mid \mathbf{read}(n, !z_r, !z_w)@l_f.\mathbf{in}(n, !z')@l_f.\ \mathbf{out}(n, z)@l_f.\mathbf{out}([z_w]_{\{u\}}, n)@u \big)$$

Intuitively, the first **in** action collects the request for reading/writing the file named $n$ performed by locality $u$; then the following **read** action, once compiled[3], verifies whether the locality replacing $u$ has the read/write privilege over file $n$. Finally, the required operation is performed (the content of the file is read or the old content is replaced with the new one) and an acknowledgement (containing the kind of operation performed, the name of the file and, in the “read” case, also its content) is reported to $u$.

**Executing Code-on-Demand.** In this last scenario, a user can dynamically download some code from the server to perform a given task. The server stores all the downloadable processes in a private locality $l_c$. For each process named $N$, whose code is $P$ and that is downloadable by nodes in $r$, the server stores in $l_c$ the component

$$C_N \triangleq \langle N, [\text{“}downloaded\text{”}]_{r \cup \{l_S,l_c\}} \rangle \;\big|\; \mathbf{rec}X.\,\mathbf{in}(N, !u)@l_c.\big( X \mid \mathbf{read}(N, !z_e)@l_c.\ \mathbf{eval}(\,\mathbf{eval}(\mathbf{out}([z_e]_{\{l_c,l_S,u\}}, N)@u.P)@u\,)@l_S \big)$$

Then, when a user wants to download some code, the server handles his request with the process

$$Execute(l_c) \triangleq \mathbf{rec}X.\,\mathbf{in}(\text{“}execute\text{”}, !u, !n)@l_S.\mathbf{out}(n, u)@l_c.X$$

Notice that $l_c$ cannot directly send $P$ for execution to $u$ because (the locality associated to) $u$ cannot have $l_c$ in its trust region (since $l_c$ is fresh). Thus, $P$ must firstly pass through $l_S$ and then, if $l_S$ is in the trust region of $u$ (which we assume it is the case), the code-on-demand procedure successfully terminates, by also reporting an ack to the user.

**The System.** Finally, we can put together the activities shown so far to obtain the implementation of the complete activity of the server. Thus, the (not yet compiled) initial configuration of $l_S$ should be

$$l_S \,{}_{\forall}{::}_{\emptyset}\; \mathbf{newloc}(u_1).\mathbf{newloc}(u_2).\mathbf{newloc}(u_3).\ \textit{set up } u_1 \textit{ with the identites and passwords of the users}.\ \textit{set up } u_2 \textit{ with the data of the file system}.\ \textit{set up } u_3 \textit{ with the processes for the downloads}.\big( NewUser(u_1) \mid Login(u_1) \mid Read(u_2) \mid Write(u_2) \mid Execute(u_3) \big)$$

[3] Indeed, the compilation annotates these actions as

$$\mathbf{read}(n, [!z_r]^{\{l_S,l_f,u\}}, [!z_w]^{\{l_S,l_f\}})@l_f$$
$$\mathbf{read}(n, [!z_r]^{\{l_S,l_f\}}, [!z_w]^{\{l_S,l_f,u\}})@l_f$$

and, thus, they will be successfully executed at runtime only if the locality replacing $u$ is in the region annotating the dummy items “*read*” and “*written*” respectively.

Notice that our example simplifies UNIX behaviour in two major aspects. Firstly, we did not require that a user must login before using the functionalities offered by the system; secondly, the files/programs are put by the system and not by the users. Both these choices were driven by the aim of simplifying the presentation; however, our simplified setting could be easily enriched with more refined and realistic features.

Finally, we want to remark that, by exploiting the dummy data "*read*", "*written*" and "*downloaded*", we have enforced an access control policy by only using region annotations. This confirms that, in spite of its simplicity, the approach we presented in this paper is very powerful.

# 6 Related Work

In the last years, a lot of work has been devoted to design languages for mobile processes that come equipped with security mechanisms (at compile-time and/or at run-time) based on, e.g., type systems [12, 3, 5, 10], control and data flow analysis [11, 15, 6, 2] and proof carrying code [14]. The approach we presented in this paper is related to all these techniques. It strongly follows the idea of a dynamic type system, where the programmer specifies an annotation (i.e. a "type") for some elements of the calculus, and the semantics, by relying on a static compilation phase, respects the annotations in the net[4]. Our compilation phase keeps track of where the process data will appear during the execution of the process itself; this is very similar to control flow analysis. However, differently from e.g. [6, 11], we do not use over-approximations of regions that will access data: the annotation of our template fields is precise. Finally, outputs and migrations are allowed only from trusted nodes; thus, we assume that the sender of a datum/process can be reliably determined. This assumes an authentication mechanism (e.g., the agent travels with a certificate giving evidence of its origin): this is a form of proof carrying code.

We deeply drew inspiration from Confined-$\lambda$ [13], a higher-order functional language that supports distributed computing by allowing expressions at different localities to communicate via channels. Authors of code can assign regions (i.e. subsystems) to values in order to limit the part of a system where a value can freely move; a type system is defined that guarantees that each value can roam only within the corresponding region. This is very similar to our approach; there are however some differences. First of all, exploiting channels greatly simplifies the static semantics because, as usual, channels are assumed to transmit values of a certain type (i.e. values visible within a certain region). This is not the case in the approach we presented, because a TS can host every possible kind of datum; thus, no static information about the types of the data appearing in the TS can be assumed. Moreover, our annotations are only associated to the relevant data. In [13], a programmer must declare a type (i.e. a region) for any constant, function and channel; this is clearly heavier. Finally, when compiling a net, we do not rely on any form of global knowledge of the system; only the annotations in the process are considered. On the contrary, the type system in [13] assumes a global typing environment for handling shared channels; this somehow contrasts with the features a global computing scenario.

We want to conclude by saying that the group types for the Ambient calculus [3] aim at purposes very similar to ours. A group can be seen as a set of ambients (i.e. localities names) and is used to express properties of ambient movement (like, e.g., "an ambient whose name is in group *G* can enter an ambient whose name is in group *H*"). This can be used to control ambient movement and, thus, the visibility of ambients (i.e. data) in different regions of a net. However, also this approach uses global knowledge about the execution environment and, moreover, it also relies on typing the whole net.

[4] However, our approach is simpler than most of the typed process calculi proposed in literature since our annotations are very intuitive: a programmer wanting a certain datum to be restricted to a certain region has only to annotate the datum with the localities in that region. This is simpler than, e.g., the channel types of D$\pi$ [12], the Ambient types [3] or the recursive process types of KLAIM [5].

**Acknowledgments** We are grateful to Rocco De Nicola and the anonymous referees for helpful comments.

# 7 References

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

[2] C. Braghin, A. Cortesi, and R. Focardi. Security Boundaries in Mobile Ambients. *Computer Languages*, 28(1):101–127, Nov 2002.

[3] L. Cardelli, G. Ghelli, and A. D. Gordon. Types for the ambient calculus. *Journal of Information and Computation*, 177(2):160–194, 2002.

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

[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] P. Degano, F. Levi, and C. Bodei. Safe ambients: Control flow analysis and security. In *Proc. of ASIAN'00*, LNCS 1961, pp. 199–214. Springer, 2000.

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

[8] Ú. Erlingsson and F. B. Schneider. SASI enforcement of security policies: A retrospective. In *New Security Paradigms Workshop*, pp. 87–95, Caledon Hills, Ontario, Canada, Sept. 1999. ACM SIGSAC.

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

[10] D. Gorla and R. Pugliese. Resource Acces and Mobility Control with Dynamic Privileges Acquisition. In J. Parrow, editor, *Proc. of ICALP'03*, LNCS 2719, pp. 119–132. Springer, 2003.

[11] R. R. Hansen, J. G. Jensen, F. Nielson, and H. R. Nielson. Abstract interpretation of mobile ambients. In *Proc of SAS'99*, LNCS 1694, pp. 134–148. Springer, 1999.

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

[13] Z. D. Kirli. Confined mobile functions. In *Proc. of the 14th CSFW*, pp. 283–294. IEEE, 2001.

[14] G. Necula. Proof-Carrying Code. In *Proceedings of POPL '97*, pages 106–119. ACM, 1997.

[15] F. Nielson, H. R. Nielson, R. R. Hansen, and J. G. Jensen. Validating firewalls in mobile ambients. In J. C. Baeten and S. Mauw, editors, *Proc. of CONCUR'99*, LNCS 1664, pp. 463–477. Springer, 1999.

[16] A. Omicini and F. Zambonelli. Coordination for internet application development. *Autonomous Agents and Multi-agent Systems*, 2(3):251–269, 1999.

[17] G. Picco, A. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st ICSE'99*, pp. 368–377. ACM Press, 1999.

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

[19] F. B. Schneider, G. Morrisett, and R. Harper. A language-based approach to security. In *Informatics: 10 Years Ahead, 10 Years Back*, LNCS 2000, pp. 86–101. Springer, 2000.

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