Programming Access Control: The KLAIM Experience

Rocco De Nicola[1] GianLuigi Ferrari[2] Rosario Pugliese[1]

[1]Dipartimento di Sistemi e Informatica, Università di Firenze
e-mail: {denicola,pugliese}@dsi.unifi.it
[2]Dipartimento di Informatica, Università di Pisa
e-mail: giangi@di.unipi.it

Abstract. In the design of programming languages for highly distributed systems where processes can migrate and execute on new hosts, the integration of security mechanisms is a major challenge. In this paper, we report our experience in the design of an experimental programming language, called KLAIM, which provides mechanisms to customize access control policies. KLAIM security architecture exploits a capability-based type system to provide mechanisms for specifying and enforcing policies that control uses of resources and authorize migration and execution of processes. By means of a few programming examples, we illustrate the flexibility of the KLAIM approach to support the specification of control policies and to guarantee their enforcement.

1 Introduction

Highly distributed systems and networks have now become a common platform for large scale distributed programming. Network applications distinguish themselves from traditional applications for *scalability* (huge number of users and nodes), *connectivity* (both availability and bandwidth), *heterogeneity* (operating systems and application software) and *autonomy* (of administration domains having strong control of their resources). These features call for programming languages that support mobility of code and of computations, and for effective infrastructures that support coordination of dynamically loaded (often untrusted) software modules. Current software computing environments exploit so called *security architectures* to monitor the execution of mobile code and protect hosts from external attacks. We refer the readers to [18] for a collection of papers dedicated to tackling these issues.

Recently, the possibility has been explored of considering security issues at the level of language design, aiming at embedding dynamic linking of code and protection mechanisms in programming languages. The challenge is that of designing the language together with its secure kernel and to implement the corresponding secure abstract machine. In other words, security issues are taken into account when designing the programming language and not later for addition to existing infrastructures. This provides the users with suitable programming

mechanisms for defining their own security policies. A partial example of this approach is given by the *Java 1.2 Security architecture* [24].

In this paper we report our experience in the design of a language which supports programming of security policies. We describe the security mechanisms of KLAIM (*a Kernel Language for Agents Interaction and Mobility*) [13], an experimental programming language specifically designed for programming network applications. KLAIM provides direct support for expressing and enforcing access control policies to resources and for authorizing migration and execution of mobile processes. KLAIM exploits a capability-based type system to specify and enforce access control policies.

KLAIM consists of core Linda [11, 10, 5] with multiple located tuple spaces. A KLAIM program, called a net, is structured as a collection of nodes. Each node has a name, and consists of a process component and a tuple space component. Sites are the *concrete* names of the nodes and are the main linguistic constructs to provide network references to nodes. Processes may access tuple spaces through explicit naming: operations over tuple spaces are indexed with their *locality*. Localities are the *symbolic* names of nodes and programmers need not to know the concrete network references, i.e. the precise association of localities with sites. The net primitives are designed to handle all issues related to physical distribution, scoping and mobility of processes: the visibility of localities, the allocation policies of tuple spaces, the scoping disciplines of mobile agents, etc. In other words, KLAIM nets provide the distributed infrastructure to coordinate processes that access and share resources over a highly distributed system.

KLAIM exploits a capability-based type system to specify and enforce access control policies. Capabilities provide information about the intentions/rights of processes: reading/consuming tuples, producing tuples, activating processes, and creating new nodes. Capabilities have a hierarchical structure; for example we assume that a processes authorized to consume a tuple has also the right of reading it. The hierarchy of capabilities is reflected in the subtype relation over access types. The underlying idea is that if a process `P` has access type `ac1` and this is a subtype of `ac2` then `P` could be granted access type `ac2` as well. In other words, `P` can be safely used in all contexts where a process of type `ac2` is expected. The (access) type of a process specifies the operations the process intends to perform at the localities of a net. The (access) type of a node specifies the access control policy of that node with respect to the nodes of the nets.

Enforcement of access control policies is performed by the *type checker* which controls that the loaded software modules match the access policies of the nodes. Only processes (stationary or mobile) that have successfully passed the type checking phase can be executed.

The clear distinction between specification and enforcement of access control policies is a key design element of the type system. Indeed, the development of KLAIM applications proceeds in two phases.

In the first phase, processes are programmed while ignoring the precise physical allocations of tuple spaces and the access rights of processes. By exploiting type annotations and mechanisms for code inspection, a type inference system

assigns types to processes and checks whether processes behave consistently with their type declarations.

In the second phase, processes are allocated over the nodes of the net. It is in this phase that access control policies are set up as a result of a coordination activity among the nodes of the net.

This paper illustrates the KLAIM access control model and shows how a capability-based type system is a useful and effective tool to write secure network applications. The main features of the approach are summarized below.

- Access rights are explicitly recorded in type specification, thereby providing declarative specifications of access control policies.
- Subtyping of access rights: alternative access policies can be defined by replacing the default policy with a new, more restrictive policy that is a subtype of the default policy.
- Mobile processes are typed by their access control requirements; these are automatically generated by the type inference procedure.
- Processes access control requirements are clearly separated from nodes access control policies: the node access control policy is formulated by the authority (the owner) of the node and is dynamically enforceable at run-time.
- KLAIM type system is sufficiently powerful to express access patterns of policies for customizing and confining the route of process migration.

Figure 1 below, illustrates the basic ingredients of our approach.

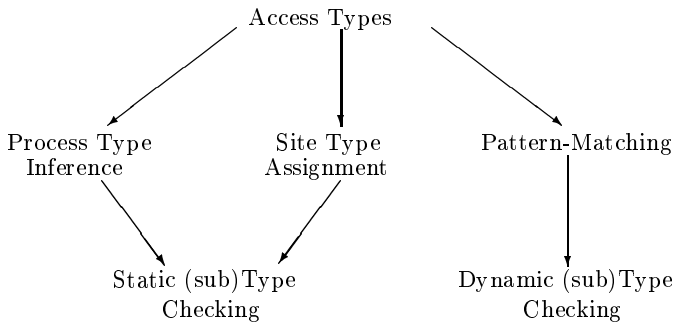


Fig. 1. KLAIM type system: main ingredients

The language and the design philosophy underlying KLAIM are presented in [13]. The mathematical foundations (decidability and soundness) of the kernel of KLAIM type system can be found in [15] (a preliminary presentation appeared in [12]). The prototype implementation of the language is described in [2].

The rest of the paper is organized as follows. In the next section we briefly review KLAIM and its capability-based type system. Section 3 illustrates how KLAIM primitives allow programming of complex interaction protocols; more

specifically, we consider three well-known paradigms for distributed computing, namely, mobile agent, code-on-demand and remote evaluation. We also show how the capability-based type system can be exploited to address the security needs of the three paradigms. Section 4 shows how the run-time type-checking mechanisms of KLAIM can be exploited to impose access control policies. In particular, we show how to impose restrictions on mobile processes concerning:

- the access to tuples from one or more localities;
- the exchange of tuples among groups of localities;
- the trajectory of process migrations.

The final section summarizes our approach and suggests future extensions.

2 The KLAIM programming model

We begin this section by summarizing the Linda programming model. Then, we introduce the main features of KLAIM by means of simple examples. The complete syntax of the language is reported in the Appendix A.

Linda is a coordination language that relies on an asynchronous and associative communication mechanism based on a shared global environment called Tuple Space (TS). A tuple space is a multiset of tuples, that are sequences of actual fields (expressions or values) and formal fields (variables). *Pattern-matching* is used to select tuples in a TS. Linda provides four main primitives for handling tuples: two (non-blocking) operations add tuples to a TS, two (possibly blocking) operations read/withdraw tuples from the TS. The Linda asynchronous communication model allows programmers to explicitly control interactions among processes via shared data and to use the same set of primitives both for data manipulation and for process synchronization.

2.1 Nets and Processes

We now introduce the KLAIM primitives without considering typing issues. KLAIM programs are structured around the notions of *localities*, *tuples*, *tuple spaces*, *nets* and *processes*.

Localities (`l, l', ...`) can be thought of as the symbolic names for sites. Sites (`s, s', ...`) are the addresses (network references) of nodes. The bindings between localities and sites are stored in the *allocation environment* of each node of the net. Existence of a distinguished locality `self` is assumed: the `self` locality is used by processes to refer to their current execution site.

Tuples contain information items; their fields can be actual fields (i.e. expressions, localities, processes) and formal fields (i.e. variables). Syntactically, a formal field takes the form `!ide`, where `ide` is an identifier. For instance, the sequence `(''foo'', 25, !u)` is a tuple with three fields. The first two fields are basic values (a string value and an integer value) while the third field is a locality variable. Similarly, `(Agent<Itinerary, RetLloc>, !List, Site)` is a tuple whose first field is a process (with two parameters).

Tuple spaces are collections (multisets) of tuples. *Pattern-matching* is used to select elements from a tuple space. Two tuples match if they have the same number of fields and corresponding fields have matching values or variables. Variables match any value of the same type, and two values match only if they are identical.

KLAIM *nets* provide the infrastructure to coordinate users accessing and sharing a set of resources over a configurable distributed system. Nets are sets of nodes; each node consists of a site `s`, an allocation environment `e`, a set of running processes `P` and a tuple space `T`. Sites stand for physical places with a boundary (e.g. host machines, firewalls, administration domain).

A node (an active KLAIM abstract machine) embodies both the active computational units (processes) and the resources (tuples). KLAIM communication paradigm is anonymous (tuples have no name) and associative (tuples are content addressable). This form of communication mechanism has been recognized as a suitable tool to program network services where the set of available services (coded as tuples in the tuple spaces) at any given moment may dynamically change (see [1, 23]).

Technically, tuple spaces are modelled as processes emitting tuples, namely processes of the form **out**`(t)` where `t` is an *evaluated tuple*, i.e. a tuple where all fields are ground (they do not include variables). For instance, **out**`(''foo'', 1)` defines the tuple `(''foo'', 1)`.

The allocation environment `e` constraints network connectivity: it basically behaves as a proxy mechanism for the processes allocated at a certain node.

The following code

```
node s :: e {P | T}
```

defines a node, where `s` is the site, `e` the allocation environment, `P` the set of running processes and `T` the tuple space. Similarly, the following code

```
node s1 :: e1 {P1 | T1} ||
node s2 :: e2 {P2 | T2} ||
node s3 :: e3 {P3 | T3} .
```

defines a net with 3 nodes.

Processes are the active computational units. They can perform five different basic operations, called *actions*, that permit reading from a tuple space, withdrawing from a tuple space, writing in a tuple space, activating new threads of execution and creating new nodes.

The operation for retrieving information from a node has two variants: **in**`(t)@l` and **read**`(t)@l`. Action **in**`(t)@l` evaluates the tuple `t` and looks for a matching tuple `t'` in the tuple space located at `l` (`l` is the logical address of the tuple space). Whenever the matching tuple `t'` is found, it is removed from the tuple space. The corresponding values of `t'` are assigned to the variables in the formal fields of `t` and the operation terminates; the new bindings are used by the continuation of the process that has executed **in**`(t)@l`. If no matching tuple is found, the operation is suspended until one becomes available.

Action **read**(t)@l differs from **in**(t)@l only because the tuple t' selected by pattern-matching is not removed from the tuple space.

The operation for placing information on a node has, again, two variants: **out**(t)@l and **eval**(P)@l. The operation **out**(t)@l adds the tuple resulting from the evaluation of t to the tuple space located at l. The operation **eval**(P)@l spawns a process (whose code is given by P) at the node located at l.

The operation for creating new nodes is **newloc** which takes as arguments a list of locality variables. For instance, **newloc**(u1, u2, u3) dynamically creates 3 distinct new sites that can only be accessed via locality variables u1, u2, u3.

Notice that variables occurring in KLAIM processes can be bound by actions. More precisely, the actions **in**(t)@l and **read**(t)@l act as binders for variables in the formal fields of the tuple t. The action **newloc** binds the locality variables taken as arguments.

We now provide some simple examples of KLAIM programs; more advanced programming examples will be presented later.

KLAIM provides two mechanisms of mobility. The action **eval** moves a process code to a remote site "cutting" the links to the local resources, i.e. by adopting a dynamic scoping discipline. The action **out** can move a piece of code to a remote site while maintaining the links to the resources allocated at the original site, i.e. by adopting a static scoping discipline.

Our first example illustrates a process that moves along the nodes of a net with a fixed binding of localities to sites. We consider a net consisting of two sites s1 and s2. A client process Client is allocated at site s1 and a server process Server is allocated at site s2. The server process can accept clients for execution. The client process sends process Q to the server. This is implemented by the following code:

```
def Client = out(Q)@l1 ; nil
def Q = in(''foo'', !x)@self; out(''foo'', x+1)@self; nil
def Server = in(!X)@self; eval(X)@self; nil
```

The behaviour of the above processes depends on the meaning of l1 and self. It is the allocation environment that establishes the links between localities and sites. Here, we assume that the allocation environment of node s1, namely e1, associates self to s1 and l1 to s2, while the allocation environment of site s2, e2, associates self to s2. Finally, we assume that the tuple spaces located at s1 and s2 both contain the tuple (''foo'', 1). The following KLAIM code defines the net outlined above:

```
node s1 :: e1 { Client | out(''foo'', 1)} ||
node s2 :: e2 { Server | out(''foo'', 1)}.
```

The client process Client sends process Q for execution at the server node (locality l1 is bound to s2 in the allocation environment e1). After the execution of the action **out**(Q)@l1, the tuple space at site s2 contains a tuple where the code of process Q is stored. Indeed, the process stored in the tuple is

```
Q' = in(''foo'', !x)@s1; out(''foo'', x+1)@s1 ; nil
```

because the localities occurring in Q are evaluated using the environment at s1 where the action **out** has been executed. Hence, when executed at the server's site the mobile code increases the tuple at the client's site. Fig. 2 gives a pictorial representation of this alternative.

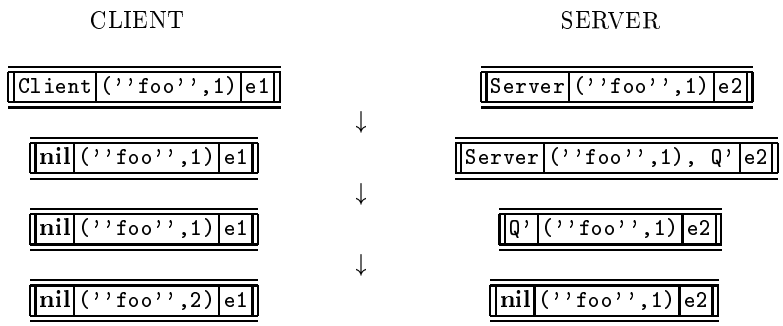


Fig. 2. Agent Mobility: Static Scoping

Our second example illustrates how mobile agents migrate with a *dynamic scoping* strategy. In this case the client process Client is defined by the code **eval**(Q)@l1 ; **nil**. When action **eval**(Q)@l1 is executed, the code of the process Q is spawned at the remote node *without* evaluating its localities according to the allocation environment e1. Thus, the execution of Q will depend only on the allocation environment e2 and, hence, Q will increase the tuple at the server's site. Fig. 3 illustrates this alternative.

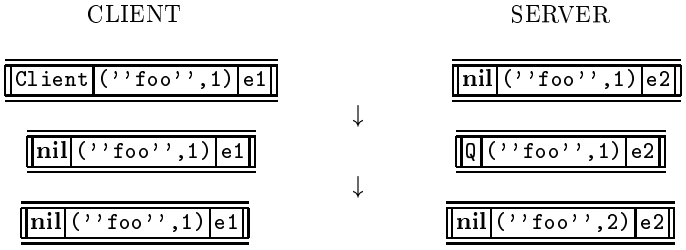


Fig. 3. Agent Mobility: Dynamic Scoping

2.2 Types for Access Control

KLAIM makes use of a capability-based type system to express and enforce access control policies. Capability-based types provide information about permissions

of processes: downloading/consuming tuples, producing tuples, activating processes, and creating new nodes.

We use { `r, i, o, e, n` } to indicate the set of permissions, where each symbol stands for the operation whose name begins with it (e.g. `r` denotes the permission of executing a **read** action).

The type `Bottom` is used to express no requirement (no action) by processes. Conversely, the type `Top` denotes the intention of performing any kind of operations. Moreover, **atype** and **ttype** denote access types and tuples types, respectively.

A type of the form `l --> r[ttype] --> Bottom`, an *arrow type*, describes the permission/intension of performing, at location `l`, the action of reading a tuple of type **ttype** without imposing any further constraint on the remaining process (the continuation). The arrow type `l --> e -->` **atype** describes the permission of executing of process of type **atype** at location `l`. We often adopt the name *capability* to indicate the action permission of arrow types. For instance, `r[int, string]` is the capability of the arrow type `l --> r[int, string] --> Bottom`.

The union type "**atype, atype**" is used to join permissions. Recursive types are used for typing migrating recursive processes (we often use `'a` to denote type variables).

In general, permissions have a hierarchical structure: a process authorized to read a tuple with a `real` value has also the right of reading a tuple with an `int` value. Similarly, a process authorized to perform an **in** action may also perform a **read** action. The hierarchy of access rights is reflected in the subtype relation. The underlying idea is that if a process `P` has type `ac1` and `ac1` is a subtype of `ac2` then `P` could be considered as having type `ac2` as well. In other words, `P` can be safely used whenever a process of type `ac2` is expected.

The subtype relation $\preceq$ formalizes this intuition. The type `Bottom` semantically corresponds to the smallest type, and the type `Top` denotes the greatest type. Several typing judgments formally characterize the subtype relation. For instance, the following clause

`l --> r[int] --> Bottom` $\preceq$ `l --> r[real] --> Bottom`

states that a process looking for a tuple with an integer field can also be viewed as a process looking for a tuple with a real field. Motivations and formal properties of the subtype relation $\preceq$ can be found in [15].

KLAIM tuples are typed. Tuple types basically are record types. The distinguished tuple type **any** is used to have a form of *genericity* of tuples.

To express user-based access policies, localities and locality variables may be annotated with type specifications which are pairs of the form $\langle \lambda, \mathtt{ac} \rangle$ called *access lists* that are lists of bindings from localities to capabilities. The distinguished locality **all** is used to denote all nodes of a net. Access lists are exploited for different purposes:

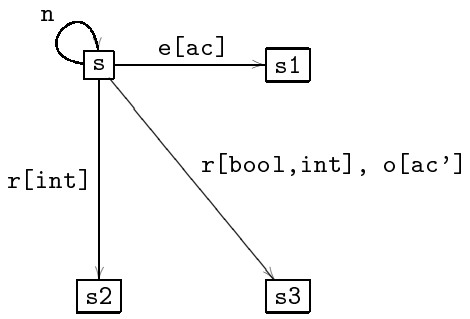
- To restrict the kind of actions that a process can perform at (the site corresponding to) a specific locality that is transmitted in a tuple.

– To specify the access rights of newly created nodes with respect to the nodes of the net, and the vice versa.
– To specify the permissions of a process on actual arguments of process definitions.

Access types are also used to define the access control policy of nodes. For instance, the following access type, where `ac` and `ac'` are access types and tuple types respectively, describes the access policy of node `s`, i.e. the permissions of processes located at `s`

```
s --> n --> Bottom,
s1 --> e --> ac,
s2 --> r[int] --> Bottom,
s3 --> r[bool,int] --> Bottom,
s3 --> o[ac'] --> Bottom
```

For example, a process located at `s` is allowed to spawn a process of type `ac` at site `s1` and it is not allowed to perform any other action at site `s1`. Nodes access types have a natural representation as labelled graphs. The following graph represents the access type described above.



We now give a description of the use of types for access control. Let us consider a system consisting of a process `Server` and two identical `Client` processes. The server process

```
def Server = out(l:<void, Top>)@self; nil
```

adds a tuple, containing locality `l`, to its local tuple space and evolves to the terminated process **nil**. In our example the server process does not take advantage of the possibility of imposing further restrictions on the access policy of `l`: the pair `<void, Top>` states that the access restrictions at `l` are left unchanged.

The client process

```
def Client = read(!u:<[self--> e], ac>)@l-S ;
             eval(P)@u ; nil.
```

first accesses the tuple space located at `l-S` to read an address. Then, it assigns the value read to the locality variable `u` and, sends process `P` for execution at `u`. The pair `<[self--> e], ac>` specifies that `Client` needs a locality where it

is possible to send for execution a process with (access) type `ac` from the site where `Client` is running[1].

Let us now consider a net where `Server` is allocated at site `s` and the two, identical, `Client` processes are at sites `s1` and `s2`, where `l-S` is bound to `s` to allow clients to interact with the server.

Under the assumption that code `P` does not violate the access requirements specified by the access type `ac` (i.e. the type inferred for `P` is a subtype of `ac`), the inference system determines the following type `c` for the process `Client`

```
c = l-S --> r[<[self--> e], ac>] --> Bottom
```

This type `c` is an abstraction of `Client` behaviour stating that the process aims at reading the address of a node where a code of type `ac` is allowed to run.

Notice that an important part of the programming task consists of adding type annotations to the code in order to specify the access policies. Our inference system inspects the annotated code to automatically determine the access policy requirements of the code.

Let us consider our running example and assume the following types for the permissions associated to the sites `s1` and `s2` where `Client` is allocated:

```
ac-s1 = s --> r[<[self--> e], ac1>] --> Bottom
ac-s2 = s --> r[<[self--> e], ac2>] --> Bottom
```

Both permissions state that a process can be executed at the address read at site `s`. However, in the case of site `s1` processes with privileges smaller that `ac1` can be executed at the read address; while processes with type smaller than `ac2` can be executed in the case of site `s2`.

The enforcement is performed by the *type checker* that verifies that the types of the processes loaded on a node are valid instances of the type which specifies the access control policy of the node. Processes are allocated and executed only if they are accepted by the type checker. In the type checking phase, the type of the process is obtained by evaluating its localities according to the allocation environment of the site where the process is allocated. Hence, access requests for sites which are not "visible" from the node lead to type failures.

In our running example assume that `c1` and `c2` are the instances of the type of the `Client` process in sites `s1` and `s2`, respectively. Further, assume that `c1` is a subtype of `ac-s1` and that `c2` requires more privileges than `ac-s2` then, only the `Client` process at site `s1` successfully passes the type checking phase and has the right of reading tuples at the server's site and, consequently, of sending processes for execution.

3 Paradigms for Mobility

Mobile Code Applications are applications running over a network whose distinctive feature is the exploitation of forms of "mobility". According to the classification proposed in [9], we can single out three *paradigms* that, together with

[1] In KLAIM `self` is a reserved keyword that indicates the current execution site

the traditional *client-server* paradigm, are largely used to build mobile code applications. These paradigms are: *Mobile Agent*, *Code-On-Demand* and *Remote Evaluation*.

The first paradigm can be easily implemented in KLAIM by using the actions **eval** or **out**, in dependence on the chosen binding strategy (dynamic or static, respectively) adopted for localities. We did comment on this at the end of Section 3. In the rest of this section, we analyze the two remaining paradigms with specific attention devoted to the security aspects.

Code-on-Demand A component of an application running over a network at a given node, can dynamically download some code from a remote node to perform a given task.

To download and execute code stored in the tuple space located at `l` with certain access privileges (specified by the access type `ac`), we can use the process

read(!X:ac)@l ; eval(X)@self; **nil**.

The downloaded code is checked for access violations at run-time by the pattern-matching mechanism that checks the types of processes to ensure that they satisfy the type annotations specified by the programmer. Hence, process codes are checked before being downloaded. In other words, the pattern-matching operation performs a *run-time* type checking of incoming codes.

Remote Evaluation A component of a network application can invoke services from other components by transmitting both the data needed to perform the service and the code that describes how to perform the service.

To transmit both code `P` and data `v` of type `bt` at the locality `l` of the server, we can use the code

out(**in**(!y:bt)@l ; A<y>, v)@l

where `def A(x) = P`.

Here, we assume that the server adopts the following (code-on-demand) protocol

in(!X:ac, !x:bt)@self; **out**(x)@self; **eval**(X)@self; **nil**

To prevent "damages" from `P`, the server loads and executes code `P` only if the instance of the type of the code is a subtype of `ac`. Again, dynamic type checking is extensively used to ensure that dynamically executable codes adhere to the access policy of the node.

Type `ac` may give only minimal access permissions on the server's site, for instance, only the capability of reading a tuple consisting of a string and an integer value, and giving back the results of the execution. In this case, the access type is of the form

```
s --> r[string, int] --> Bottom,
s0 --> o[string, int] --> Bottom,
s1 --> o[Top] --> Bottom
```

where s is the server's site and s0 is the site which invokes server's facilities and s1 is another site.

Notice, however, that this does not prevent P from visiting other sites. In particular, code P may be programmed in such a way that it transmits some code Q with access type ac-Q at l1:

```
def P = read(!y:string, !x:int)@l ;
        out(op2(y), op1(x))@self;
        out(Q)@l1; nil
```

It is immediate to see that the instance of the type of code P at the server site is

```
s --> r[string, int] --> Bottom,
s0 --> o[string, int] --> Bottom,
s1 --> o[ac-Q] --> Bottom
```

Hence, the instance of the code P satisfies the access policy of node s0. However, code Q is only stored in the tuple space at s1, no new thread of execution is activated at that site. Before being executed code Q must be read and verified (dynamic type checking). Therefore, process P cannot silently activate a *Trojan horse* at the remote site s1.

4 Controlling Use of Resources

In the previous section, we examined the support offered by the KLAIM capability-based type system for access control by considering well-known paradigms of mobility. In this section, we show how to exploit KLAIM capability-based types to restrict the resources usage within a net. In particular, we show that KLAIM types are powerful enough to program and enforce policies which are usually handled by *low-level* techniques (e.g. sand-boxing and firewall). Moreover, the explicit typing of sites can be also used to define policies which limit mobile processes to specific route.

Restricting Interactions KLAIM action primitives operate on the whole net not just at the process current site. From the point of view of security mechanisms, communications among different sites of the net (i.e. remote communications) could be controlled and regulated. This corresponds to place over a node a form of *sandboxing* which treats all processes as potentially suspicious.

For instance, to make sure that a process running on a certain site s gets only local information, it suffices to constrain the type specifying the access control policy of the node to allow local communications only.

To this purpose, it suffice to state that for no site s' different from s, the type `s' --> r[any] --> Bottom` (or any of its subtypes) is a subtype of ac-s, the access type of site s. Hence, a process P allocated at site s that is willing to perform remote **read**/**in** operations violates the access rights. To access tuples at a remote tuple space, a well-typed process must first move (if it has the required rights) to the remote site. Using a similar strategy also output actions can be forced to be local.

Firewall We now show how to specify a firewall by means of suitable typing. The idea is that the type of the site where the firewall is allocated specifies the access policy of network services. Assume that the firewall is set up at site `s` to protect sites `s-1`, ..., `s-n` from sites `S-1`, ..., `S-m`.

A first requirement is that processes located at site `S-j` cannot directly interact and ask for services at any site `s-i`. To this purpose we impose that for each site `S-j` it cannot be the case that `s-i --> cap --> Bottom`, where `cap` is any capability, is a subtype of the access type of the node (`ac-S-j`).

On the other hand, type `ac-S-j` may have subtypes of the form

```
s --> o[any] --> Bottom,
```

since each site `S-j` may send service requests to the firewall. Here we assume that the service request is stored in a tuple and we do not impose any restriction on the type of the tuple. This setting guarantees that all connections from sites `S-1`, ..., `S-m` to sites `s-1`, ..., `s-n` have to pass through site `s`, the firewall.

For instance, process

```
def P = out(eval(Q)@s-i)@s ; nil
```

complies with the access control policy, while process

```
def P' = eval(Q)@s-k ; nil
```

violates the access control policy of site `s-k`.

The firewall can be programmed to handle the requests according to certain policies. Typically, the firewall handles a service request according to the following pattern

```
read(!Request:ac-r) ; <RequestHandler>
```

where type `ac-r` specifies the access policy that the service request must satisfy. For instance, if

```
ac-r = s-1 --> i[any] --> Bottom,
       s-2 --> e --> (s-2 --> r[any] --> Bottom,
                      S-k --> o[any] --> Bottom)
```

then the two service requests

```
eval(read(!x:int)@self; out(op(x))@S-k ; nil)@s-2 ; nil
read(!x:int)@s-1 ; nil
```

both satisfy the type requirements and will be accepted. The service request

```
read(!x:int)@s-1 ; out(op(x))@S-k ; nil
```

violates the access policy and will be rejected.

The examples above do not specify any constraint on the tuples read. More refined access policies can be obtained by specifying the type of tuples. For instance, one can fix a policy where the read operations are constrained to get

from the tuple space only 'plain' data tuples, namely tuples without localities or process codes.

KLAIM applications might need to define their own firewall, i.e. a user node which acts as a filter for the network traffic, without changing the default policies of the nodes of the net. Let us consider, for instance, the following user process:

```
def P = newloc( u-1: <al-1, ac-1>,
                u-2: <al-2, ac-2>,
                u-F: <al-F, ac-F>) ;
        <Firewall Handler>
```

Now, assume that both access lists `al-1` and `al-2` are of the form

```
[u-F --> cap]
```

for a suitable capability `cap`.

Process `P` creates a subnet whose sites are associated to locality variables `u-1` and `u-2` and the type annotations ensure that the subnet can be reached only through site `u-F` (the firewall). The user process specifies in the type specification of locality `u-F` the access policy to the firewall. For instance, setting `al-F = [self--> o[any]]` means that the firewall site can be accessed only from the site where the user process is running.

Fares and Tickets A primary access control policy consists of controlling the route of a mobile agent traveling in the net. For instance, if one has to configure a set of sites with new software, a mobile agent can be programmed to travel among the sites to install the new release of the software.

If the starting site of the trip is site `s-0` and sites `s-1`, `s-2`, ..., `s-n` have to be visited visited before getting back to the starting site, the following type can be used to specify the access policy of the trip:

```
ac-0 = ac, s-1 --> e --> ac-1
ac-1 = ac, s-2 --> e --> ac-2
                 ⋮
ac-n-1 = ac, s-n --> e --> ac'
ac' = s-0 --> o[string] --> Bottom
```

The idea is that at each site type `ac` specifies the allowed operations (e.g. installing the new release of a software package); the remaining type information specifies the structure of the trip (which is the next site of the trip). When it reaches the last site of the trip, the agent has the rights of returning to the original site the results of the trip (e.g. the notification that the installation was successful).

The type discussed above can be properly interpreted as the *fare* of the trip: an agent `M` can perform the trip provided that its type matches the fare, namely its instance at site `s-0` is a subtype of the type of the trip. Notice that this ensures that a malicious agent cannot modify the itinerary of the trip to visit sites other than those listed in its ticket.

5 Concluding Remarks

We have described the security architecture of the experimental network programming language KLAIM. This language provides users with a programmable support to configure application-specific access control policies by exploiting a capability-based type system. Although, the type system we designed is tailored to the KLAIM language, the general spirit of the approach can be applied to define types for access control of programming languages for highly distributed systems. We summarize below the main points:

- Declarative specification of access control policies via type annotations and type inference.
- Structured hierarchy of access rights based on subtyping.
- Clear separation of process requirements from node requirements.
- Static and dynamic type checking.
- Programmability and customization of access control policies.

The prototype implementation of KLAIM [2] is built on top of Java and Java security architecture (the sandbox model). The implementation of KLAIM access control models (in Java version 1.2) is in progress, however preliminary implementations have been already exploited to validate some design choices of the type system.

We plan to extend KLAIM access types to handle history dependent access control. In history dependent access control access requested are granted on the basis of what happened in the past of the ongoing computation. Moreover, the development of network applications raises other issues related to security. We plan to integrate in KLAIM other security mechanisms (both at the foundation level and at the implementation level). These include mechanisms for secure communication and authentication, and agent code security. As for related work, we do refer the interested reader to [14] where specific comments on [3, 4, 8, 6, 7, 16, 17, 19–22] can be found.

Acknowledgments

The authors would like to thank Lorenzo Bettini, Michele Loreti and Betti Venneri for interesting discussions and comments. This work has been partially supported by Esprit working groups *CONFER2*, and *COORDINA*, and by MURST projects TOSCA and SALADIN.

A KLAIM syntax

Basic Types
btype ::= `int` | `real` | `string` | `bool` | ...

Field Types
ftype := **btype** | **<alist, atype>** | **atype**

Tuple Types
ttype :: = **ftype** | **any** | **ftype, ttype** | **ttype, ftype**

Access Lists
alist ::= **void** | `[l -->e]` | `[l -->n]` | `[l --> r[ttype]]` |
`[l --> i[ttype]]` | `[l --> o[ttype]]` |
`[all -->e]` | `[all -->n]` | `[all --> r[ttype]]` |
`[all --> i[ttype]]` | `[all --> o[ttype]]` | **alist,alist**

Access Types
atype ::= `Bottom` | `Top` | `l --> r[ttype] --> Bottom` |
`l --> i[ttype] --> Bottom` | `l --> o[ttype] --> Bottom` |
`l --> n --> Bottom` | `l --> e --> atype`|
atype, atype| `'a` | `rec 'a atype`

Locality Patterns
lpattern ::= `u:`**<alist, atype>** | `u:`**<alist, atype>, lpattern**

Actions
`Act` :: = **out**`(t)@L` | **in**`(t)@L` | **read**`(t)@L` | **eval**`(P)@L` |
newloc(lpattern)

Processes
`P` :: = **nil** | `Act ; P` | `P | P` | `A<P, L, v>`

Agent Definitions
`A ::= def A(P,L,v) = P`

Evaluated Tuples
`T :=` **out**`(t)` | `T | T`

Nets
`N ::= node s :: e {` `P | T` `}` **where atype** | `N || N`

Tuples
`t ::= f | (f , t)`

Fields
`f ::= v | P | l:`**<alist, atype>** |
`!x:`**btype** | `!X:`**atype** | `!u:`**<alist, atype>**

References

1. K. Arnold, B. Osullivan, R.W. Scheifler, J. Waldo, A. Wollrath, B. O'Sullivan. *The Jini specification*. Addison Wesley, 1999.
2. L. Bettini, R. De Nicola, G. Ferrari, R. Pugliese. Interactive Mobile Agents in X-KLAIM. *IEEE Seventh International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, *Proceedings* (P. Ciancarini, R. Tolksdorf, Eds.), IEEE Computer Society Press, 1998.
3. C. Bodei, P. Degano, F. Nielson, H.R. Nielson. Control Flow Analysis for the π-calculus. *Concurrency Theory* (CONCUR'98), *Proceedings* (D. Sangiorgi, R.de Simone, Eds.), *LNCS* 1466, pp.611-638, Springer, 1998.
4. G. Boudol. Typing the use of resources in a Concurrent Calculus. *Advances in Computing Science* (ASIAN'97), *Proceedings* (R.K. Shyamasundar, K. Ueda, Eds.), *LNCS* 1345, pp.239-253, Springer, 1997.
5. N. Carriero, D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444-458, 1989.
6. L. Cardelli, A. Gordon, Mobile Ambients. *Foundations of Software Science and Computation Structures* (FoSSaCS'98), *Proceedings* (M. Nivat, Ed.), *LNCS* 1378, pp.140-155, Springer, 1998.
7. L. Cardelli, A. Gordon, Types for Mobile Ambients. *Proc. of the ACM Symposium on Principles of Programming Languages*, ACM Press, 1999.
8. L. Cardelli, G. Ghelli, and A. Gordon, Mobile Types for Mobile Ambients To appear *ICALP'99*, *LNCS* , Springer, 1999.
9. A. Fuggetta, G.P. Picco, G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, Vol.24(5):342-361, IEEE Computer Society Press, 1998.
10. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80-112, ACM Press, 1985.
11. D. Gelernter, N. Carriero, S. Chandran, et al. Parallel Programming in Linda. *Proc. of the IEEE International Conference on Parallel Programming*, pp. 255-263, IEEE Computer Society Press, 1985.
12. R. De Nicola, G. Ferrari, R. Pugliese. Coordinating Mobile Agents via Blackboards and Access Rights. *Coordination Languages and Models* (COORDINATION'97), *Proceedings* (D. Garlan, D. Le Metayer, Eds.), *LNCS* 1282, pp. 220-237, Springer, 1997.
13. R. De Nicola, G. Ferrari, R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, Vol.24(5):315-330, IEEE Computer Society Press, 1998.
14. R. De Nicola, G.-L. Ferrari, R. Pugliese. "Types as Specifications of Access Policies", In J. Vitek, C. Jensen (Eds) [18], pp.117-146.
15. R. De Nicola, G. Ferrari, R. Pugliese, B. Venneri. Types for Access Control. To appear, *Theoretical Computer Science*, 2000. Available at `http://rap.dsi.unifi.it/papers.html`.
16. G. Necula. Proof-carrying code. *Proc. of the ACM Symposium on Principles of Programming Languages*, ACM Press, 1997.
17. J. Riely, M. Hennessy. Trust and Partial Typing in Open Systems of Mobile Agents. *Proc. of the ACM Symposium on Principles of Programming Languages*, ACM Press, 1999.
18. J. Vitek, C. Jensen (Eds). *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, LNCS State-Of-The-Art-Survey, *LNCS* 1603, Springer, 1999.

19. R. Stata, M. Abadi. A Type System for Java Bytecode Verifier. *Proc. of the ACM Symposium on Principles of Programming Languages*, ACM Press, 1998.
20. J. Vitek, G. Castagna. A Calculus of Secure Mobile Computations. *Proc. of Workshop on Internet Programming Languages*, Chicago, 1998.
21. D. Volpano, G. Smith. A typed-based approach to program security. *Theory and Practice of Software Development* (TAPSOFT'97), *Proceeding* (M. Bidoit, M. Dauchet, Eds.), *LNCS* 1214, pp.607-621, Springer, 1997.
22. D. Volpano, G. Smith. Secure Information Flow in a Multi-threaded Imperative Language. *Proc. of the ACM Symposium on Principles of Programming Languages*, ACM Press, 1998.
23. Sun Microsystems. The JavaSpace Specifications. `http://java.sun.com/products/javaspaces`, 1997.
24. Li Gong. The Java Security Architecture (JDK 1.2). `http://java.sun.com/`, 1998.