Electronic Notes in Theoretical Computer Science 38 (2000)
URL: http://www.elsevier.nl/locate/entcs/volume38.html 13 pages

Object Oriented Languages with Abstractions for Mobile Computation

Marco Túlio de Oliveira Valente,

Institute of Informatics, Catholic University of Minas Gerais, Brazil,
E-mail: mtov@pucminas.br

Roberto da Silva Bigonha, Antônio Alfredo Ferreira Loureiro and Mariza Andrade da Silva Bigonha

Department of Computer Science, Federal University of Minas Gerais, Brazil,
E-mail: {bigonha,loureiro,mariza}@dcc.ufmg.br

Abstract

Recently, the notion of mobile computation has been proposed as an alternative to the construction of distributed applications in the Internet. Some processes calculi have already been designed to model this style of Computation. The Ambient Calculus of Cardelli and Gordon is the most distinguished among them. However, to transform this kind of distributed application in a reality in the Internet, it is necessary the design of programming languages supporting abstractions for mobile computation. This paper shows a proposal that introduces these kinds of abstractions in object oriented languages.

1 Introduction

Currently, one of the main difficulties to the dissemination of distributed applications that explore all the computational power of the Internet is the absence of abstractions that support their implementation [5]. Abstractions traditionally used in the client/server model of distributed applications, such as RPC [3] and CORBA [11], have been proposed to networks without problems of performance and availability, like local area networks. However, in a world-wide, opened and decentralized network, like the Internet, these two problems gain so great importance that any abstraction for computation in this network must be capable to treat and, if possible, to attenuate the effect of them on the applications. Therefore abstractions for programming distributed applications in the Internet must satisfy the following operational requirements:

- ability to support bandwidth fluctuations due to congestions in the network;
- ability to operate in a disconnect fashion, because it is not reasonable to assume uninterrupted connection to the network, mainly in the case of mobile devices, such as notebooks and personal digital assistants;
- ability to be executed in several architectures and operating systems that exist in a world-wide network like the Internet;
- ability to be automatically installed in client workstations, preventing this activity to be carried manually in the thousands of potencial clients of an Internet distributed application.

Recently, the notion of mobile computation has been proposed as a solution to the implementation of this type of application [5]. According to this notion, the execution of an application does not need to be restricted to a single workstation, but can roam over the network. This idea implies state mobility, i.e., not only the application code moves, as in Java, but also the application data. It is argued that this kind of mobility can reduce network load and latency and can also make it possible the construction of applications more robust to bandwidth fluctuations and that can execute disconnected from the Internet.

Currently there are some theoretical models specifically designed to express mobile computation. Among them, the Ambient Calculus [6] of Cardelli and Gordon is the most prominent. However, being a formal model, the Ambient Calculus possesses only the basic abstractions for specifying mobile computation and so its use is not recommended in practical applications. Therefore, to this kind of application become a reality in the Internet, it is necessary the design of programming languages with support to mobile computation. Such languages are called *Wide Area Languages* (WAL) [5].

In this work, we show a proposal that introduces these kinds of abstractions in an object oriented language. Instead of designing a new programming language with the Ambient Calculus as its unique computational model, we adopted a hybrid solution. In our approach, the applications continue to be composed by a set of objects and only the constructions for communication and mobility are inspired by the Ambient Calculus. As this solution does not require learning a new computational model, we believe that it can induce the dissemination of mobile applications in the Internet.

This paper is organized as follows. In Section 2, we describe the concept of *Wide Area Languages* and show the principles and features that a language must have to be classified as a WAL. In Section 3 we define a simple object based language, to which we incoporate several abstractions for mobile computation. We also show in this section examples of the use of these abstractions in applications such as plug ins, mobile agents and a conference reviewing system. In Section 4, we mention works related to the kind of mobility considered in this paper. Finally, Section 5 concludes and show possibilities of future work.

2 Wide Area Languages

A Wide Area Language (WAL) is a language with constructions that are semantically compatible with the principles of the Ambient Calculus and consequently with the Internet [5]. Amongst those principles, the most important are the following:

- WAN-Completeness: we must be able to easily implement in a WAL programs normally found in the Internet, such as applets, plug ins and mobile agents.
- WAN-Soundness: a WAL can not make use of constructions based in action-at-a-distance and continued connectivity.

In the Ambient Calculus terminology, *action-at-a-distance* denotes the idea that resources are transparently and continuously available, no matter how far away they are. This idea is largely used in languages for distributed programming in LANs and thus is the main difference between these languages and a WAL.

In order to fulfill these principles, a WAL must have the following features:

- A WAL must allow migration between nodes of the network of hierarquically structured entities that contain both data and computation. This make a WAL different from languages that allow, for instance, only object migration.
- A WAL must only allow communication between entities that share the same location. In order to prevent action-at-a-distance, communication between remote entities must be implemented through mobility .
- In a WAL, references to local entities must be done through names and not through physical addresses of memory. In this way, the mobility of these entities is not restricted by static bindings between them and their context of execution.
- In a WAL, entities are accessed through blocking semantics, i.e., in case that an entity is not locally available at the moment that another entity tries to access it, the execution of the latter remains blocked until the entity become available. This type of semantics simplifies the implementation of dynamic linking and reconfiguration.
- In a WAL, access to resources of the context of execution must be done using dynamic binding. In this way, when an entity migrates to another context, it can automatically restablish the links to the resources it needs.

3 Abstractions for Mobile Computation

As mentioned in the introduction, the abstractions proposed in this paper will be defined for an object oriented language. However, instead of using a traditional language, such as C++ or Java, we have chosen to add these abstractions to an object based language. The reason is that these kind of languages are simpler and more flexible than object oriented languages [1]. The language used in this paper is based in Obliq [4]. However, Obliq's notions of distributed lexical scope and transparent handling of network references have not been considered in our language, because they are unsuitable for computation in the Internet [5].

Applications in Obliq are organized as a set of objects. Being a object based language, it does not have classes, inheritance and dynamic method dispatching. These features are inherent to object oriented languages but they have no semantic implication to mobile computation, which is based on the notion of objects.

In Obliq, an object with fields $x_1, x_2, \ldots, x_n$ is created in the following way:

`{` x_1 `=>` a_1`,` x_2 `=>` a_2`, ...,` x_n `=>` a_n `}`

where each a_i can be an attribute or method. An attribute is defined as in the following example:

```
x => 3
```

A method definition has the form:

`x =>` **meth** `(`$y, y_1, y_2, \ldots, y_m$`) b` **end**

where y is the *self* parameter, $y_1, y_2, \ldots, y_m$ are the remaining parameters and b is the method body.

Obliq variables do not have types. However, values carry their types to execution time, thus allowing the language run-time to be strongly typed.

3.1 Containers, Contexts and Mobility

One of the main decisions in the design of Internet distributed systems should be how to divide such applications in mobile and autonomous parts. Thus, we propose a language construction to delimit these parts. In this paper, we call this construction a *container*. A container is a wrapper of objects that makes them mobile, that is, containers can move to *contexts* located in other network nodes. Contexts are services available in nodes for remote execution of containers, as showed in Figure 1. A context can also offer *resources* to the execution of containers, like, for example, a window system or a data structure. Contexts are identified by URLs of the form: *host/name*, where *host* is the name of the workstation and *name* is the name of the context.

A container with objects $p_1, p_2, \ldots, p_n$ is created by the following command: *new_container* $(m, p_1, p_2, \ldots, p_n)$, where m is an optional parameter that denotes the name of the container. An object cannot belong simultaneously to more than one container.

Container mobility is implemented by the following context operations:

- `insert_object (c, a)`: insert the object a in the container c;
- `context_jump (d)`: subjective mobility of containers to context d;

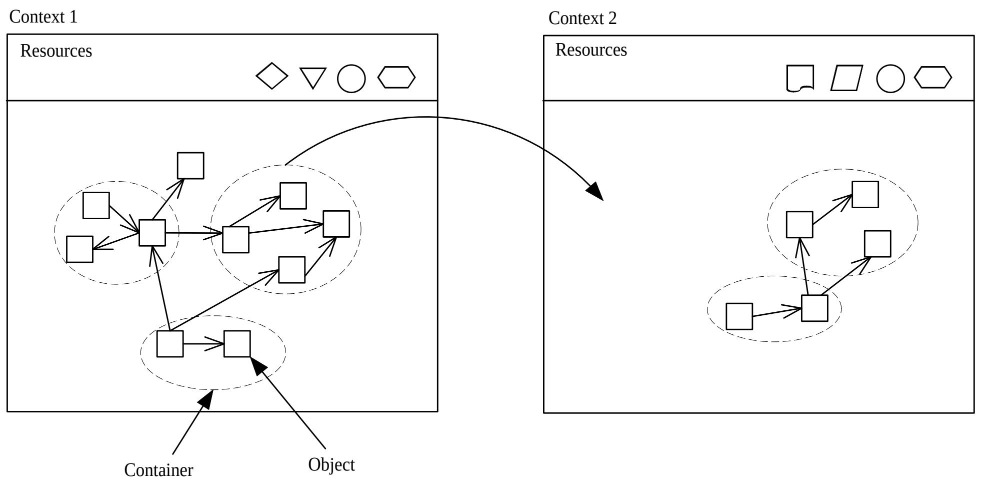

Fig. 1. Abstractions for Mobile Computation in the Internet

- `move (c, d, p)`: objective mobility of container c to context d;
- `this_container`: returns the name of the current container;
- `is_container_available (n)`: tests if there is a container with name n in the local context;
- `is_object_available (n)`: tests if there is an object with name n in the local context;
- `new_name`: returns a container name that is unique in the network;
- `here`: returns the URL of the current context.

The operation *context_jump* migrates the current container to the context whose name is passed as a parameter. After this operation, the execution continues in the next instruction, but inside the target context. In the Ambient Calculus, this form of mobility is called subjective, since it is the object itself that decides to move the container in which it is executing[6]. The operation *move* is called in a container that is not the current one, being, therefore, called objective mobility. After this operation, the execution in the source context continues asynchronously in the next instruction. In the target context, the execution begins by the method *start* of the object p and finishes when this method returns.

In every context there is a pre-defined container named *Context*. Every object that has not been inserted in another container belongs to this pre-defined container.

3.2 Objects

Objects are handled by *name semantics*. Every object has an implicit name that uniquely identifies it in every context of the network. A definition of the form **let** x = { . . . } associates with the identifier x the name of the object created. When an operation of the form $x.op$ is executed, it is first verified if an object with the name denoted by x exists in the local context. If such object exists, the operation *op* of this object is executed. In case that it does not exist, the call remains *blocked* until this object become locally available. Therefore, objects located in the current context are called *available objects*, while remote objects are called *unavailable objects*.

This semantics for object manipulation keeps the language faithfull to the Ambient Calculus principles, as it does not entail action-at-a-distance, i.e., it does not allow transparent handling of network references as usual in distributed languages for local area networks. Moreover, this name semantics helps the free migration of objects wrapped in containers, as it does not create static links between those objects and their context of execution.

There is also the operation *is_object_available* (r), which returns *true* if the object denoted by r is available in the local context, and *false* otherwise.

Example: A Conference Reviewing System
We show next, as an example of an Internet application, the implementation of a conference reviewing system, used by the program committee of a conference for evaluation of the papers. This system is cited in [5] as example of an application that can take benefit from the abstractions of a WAL. A feature desired in such a system is the possibility of operation disconnected from the network. This feature will allow a member of the program committee to evaluate papers in his *notebooks* or in workstations that does not have a dedicated connection to the Internet.

Suppose an article that has been sent for evaluation to a given reviewer. As we intend to fulfill the requirement of disconnected operation, the solution is to create a *container* with the objects needed for the evaluation and then send it to the reviewer's context, so that the evaluation can be done locally. A possible implementation is presented below:

```
let cont= new_container (article, evalform);
move (cont, reviewer.context, evalform);
```

In this code, *article*, *evalform* and *reviewer* are objects already created in the program and that denote, respectively, the article to be evaluated, the evaluation form and the reviewer. After the creation and initialization of the *container cont* with these objects, we asynchronously send it to execution in the context of the reviewer. In this context, as defined by the operation *move*, the execution will begin by the method *start* of the object *evalform*. This method can, for example, show in the workstation the evaluation form of the paper. After that, the reviewer can decide to repass the article to a second reviewer. To do so, the following operation should be executed:

```
context_jump (n);
```

where n is the URL associated with the context of the new reviewer. This second reviewer evaluates the article and then he executes a new *context_jump* to return the container to the context of the first reviewer. This reviewer, after checking the evaluation form, decides to send it to the context of the conference and to notify the chair of the program committee that the evaluation is finished. This is accomplished by the execution of the following sequence of code:

```
context_jump (conference_context);
chair.evaluationEnd (article.id);
```

Note that *chair* denotes the object that represents the chair of the program committee. This object has remained unavailable while the container roamed by the contexts of the reviewers. However, after the execution of this code, the container is back to its source context, and therefore the object denoted by *chair* become again available. Now it is possible to send without blocking a message such as *evaluationEnd*.

3.3 Synchronization

The operation to coordinate the migration of container is:

```
let i= wait (n);
```

The *wait* operation blocks the execution until a container with name n become available in the local context. When this happens, the execution continues and i denotes a special object of the container, called *interface object*. The interface object p of a container c is defined by the operation *set_interface_object* (c, p). Another way to do this is through the flag "i", passed as a parameter to an insertion operation like the following: *insert_object* (*c, p, "i"*).

Example: Plug in Installation
Consider the conference reviewing system discussed in Section 3.2. In some point of the reviewing process, papers have to be visualized. Using the proposed abstractions we can easily define a plug in that provides a viewer for a document format not available in the local context. This plug in can be represented by a container and its installation can be implemented in the following way:

```
1:  if not is_container_available ("acrobat")
2:     then ....  (* download the plug in from a remote context *)
3:  let plugin= wait ("acrobat");
4:  plugin.display (article.text);
```

In this program fragment, we suppose that will be visualized a document in the pdf format. Initially, the pre-defined operation *is_container_available* (line 1) verifies if there is in the local context a container named *"acrobat"*. It must be previously known by the programmer that containers with this name can show pdf documents. If there is no such container locally, we must download it from a remote context (line 2). Then the operation *wait* (line 3) suspends the execution until such container is locally available. When this happens, the operation returns the *interface object* of it. We can then use the method *display* of this object to show the text of the paper (line 4).

3.4 Communication

Communication between objects of the same container works as in traditional object oriented systems, that is, by exchanging synchronous messages. The same occurs between objects of different containers but located in the same context. However communication between objects of containers located in different contexts is not done with the same degree of transparency, since messages sent to unavailable objects are handled with a blocking semantics. As considered in the Ambient Calculus, this type of communication must be implemented by means of messenger containers that migrate to the remote context, send locally the message to the receiving object and then return with the result. In order to free the programmer from implementing all this procedure, an abstraction for sending messages through containers migration is proposed in this section.

Suppose that r denotes an unavailable object, located in context t. A message *msg* can be sent to this object with the following syntax:

```
let x= t::r.msg (y);
```

Messages as this one, qualified with the name of the target context, are sent through a messenger container, generated automatically in the following way:

```
1:   let p = { home => here,
2:             local_r => r,
3:             start => meth (s)
4:                        let res = local_r.msg (y);
5:                        insert_object (this_container, res, "i");
6:                        context_jump (home);
7:                      end
8:           };
9:   let messenger = new_container (p, y);
10:  move (messenger, t, p);
11:  let x= wait (messenger);
```

In the code above, a messenger container is created with an object p to send the message and with the object y that represents the message parameter (line 9). The created container is sent to the context of the remote object (line 10), where then the message *msg* is sent locally without the risk of blocking (line 4). Once sent the message, its result is inserted in the messenger container (line 5) and then this container returns to the source context (line 6). Then, in this context, x denotes the object with the result (line 11).

It should be noted in the code above that y is a free identifier in the object denoted by p (line 4). In Obliq, due to the notion of distributed lexical scope, this identifier would remain associated with its source location, even when the container migrates. However, in our approach such fact does not happen. The reason is that objects are handled by means of name semantics and operations on unavailable objects remain blocked. Thus, to prevent a blocking in any operation executed on the parameter y, it is inserted in the messenger container (line 9).

Example: Conference Reviewing System (continuation)
In the previous conference reviewing system, the reviewer in charge of the evaluation of an article can need an extra period of time to produce its report. To get this time, he must request it to the chair of the program committee. However, as the object that represents this chair is unavailable in the context of the reviewer, it is not possible to immediately send a message to it. The solution then is to send a messenger to the context of the chair with the message *requestExtraTime (n)*, where n is the number of days. As described, this messenger can be sent in the following way:

```
let r= conference_context::chair.requestExtraTime (3);
```

The message above is qualified with the "address" to be followed by messenger container.

3.5 Resources

A container can also move to another context to locally access the resources it needs, reducing in this way the demand for network communication. First, to indicate that an object p implements the interface to access a resource with name n and to export this resource to the current context, we use the following operation:

```
export_resource (n, p);
```

As the resources of a context must be always available, containers that export resources are not allowed to move.

In a client container, a reference r for a resource with name n can be defined in the following way:

```
resource r= n;
```

Finally, it is proposed that resources are accessed through dynamic binding, that is, when a container moves to another context, references to resources are automatically established with those of the same name in the new context. In case that a resource with the required name does not exist in the target context, all references for this resource will have a *nil* value.

Example: A Mobile Agent for E-Commerce
We show next the implementation of a container that represents a mobile agent who searches the price of a book in a set of Internet bookstores. Each

bookstore is represented by a context. In each bookstore, the agent, i.e., the container, gets the price of the desired book by accessing a resource that represents the price list of the bookstore. After visiting the last bookstore, the agent returns to its original context, where it then verifies which bookstore has the lowest price.

In a system like this, each virtual bookstore must define a resource named *price_list*, representing its price list.

```
let p = { ..., search => meth (s, name) ...end, ...};
export_resource ("price_list", p);
```

The container that will carry out the search can be implemented in the following way:

```
1:  let q = { home => here,
2:            number_bookstores => 3,
3:            bookstore => [ "amazon", "bookpool", "iBS" ],
4:            price => [nil, nil, nil],
5:            bookstore_search => meth (s, name)
6:                                  resource pl = "price_list";
7:                                  for i= 1 to number_bookstores do
8:                                    context_jump (bookstore [i]);
9:                                    price [i]:= pl.search (name);
10:                                 end;
11:                                 context_jump (home);
12:                               end,
13:
14:           cheapest_price=> meth (s)
15:                                 (* returns the cheapest price *)
16:                            end
17:         };
18: let agent= new_container (q);
19: q.bookstore_search ("the art of computer programming");
20: let x= q.cheapest_price ();
```

In this example, the variable *pl* is dynamically associated with the resource named "price_list" in each bookstore visited by the mobile agent (line 6). In this way, we certify that the search operation (line 9) will always perform the search for the book in the correct bookstore.

4 Related Work

Mobility was always present in abstractions for the construction of distributed applications. In RPC [3], for example, we have control mobility and a limited form of data mobility, usually only of basic data types. RPC is then an abstraction used in the implementation of traditional client/server systems and such that it does not fulfill any of the requirements of an Internet distributed application as mentioned in Section 1. CORBA [11], in turn, has as objective the introduction of control mobility in object oriented languages. CORBA also supports interoperation between diverse systems and provides location transparency in the access to objects. However, being an evolution of the RPC model, CORBA also does not take care of the operational requirements of an Internet distributed application.

There are also languages that support object mobility. Among then Emerald [7] is probably the most known. However, in the same way as Obliq [4], these languages have been designed for local networks, allowing only the implementation of applications based in action-at-a-distance and, therefore, inadequate to a network like the Internet.

Java [2] is currently the language most used in the implementation of Internet applications. Although Java supports code mobility, it does not support state mobility. With code mobility, we can guarantee execution in diverse operating systems and prevent the manual installation of applications in the client workstations. However, as the application still depends on the network to get the information needed to its execution, the model is not robust enough to deal with communication failures, nor capable to operate disconnected from the network. Java also support control mobility through the library Java RMI [10].

Recently, mobile agents have been considered as an alternative to the construction of Internet distributed systems. A mobile agent is a program that can roam through the machines of a network, carrying the state of its execution. Telescript [12] was the first language to support the implementation of mobile agents. Later, with the dissemination of Java, other systems based on this language appeared, like Aglets [8] and Voyager [9]. Although these languages support state mobility, that is, the same form of mobility proposed in this work, mobile agents are monolithic and stand alone programs, designed to realize specific tasks. In general, these systems use a primitive mechanism for communication between agents, based on message exchange and action-at-a-distance. In none of them there is a higher level abstraction for composition of diverse agents in a distributed application of realistic size.

5 Conclusions

This paper has showed a proposal for introducing abstractions for mobile computation, inspired in the Ambient Calculus, in an object oriented language. The proposed language fulfills the following principles of a Wide Area Language (WAL):

- Completeness: by means of the examples included in the paper, we showed the usability of the proposed abstractions in the implementation of typical Internet applications, like plug ins and mobile agents, and also in the implementation of applications that require operation in disconnected mode, like the conference reviewing system.

- Consistency: the notion of containers and the use of name semantics do not allow the construction of applications based on action-at-a-distance or that require continuous connectivity.

Before the implementation of a real language that incorporates the proposed abstractions, we intend to study the following features:

- An exception handling mechanism capable to handle exceptions related to communication and mobility. This mechanism must, for example, allow a certain degree of flexibility in the blocking semantics used in the execution of operations on unavailable objects.
- Constructions to allow dynamic reconfiguration of containers, making possible to change the implementation of a remote container without restarting its execution. This feature is useful in applications designed to execute continuously.

References

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

[2] K. Arnold and J. Gosling. *The Java Programming Language.* Addison Wesley, 2nd edition, 1997.

[3] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Feb. 1984.

[4] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[5] L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer-Verlag, 1999.

[6] L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[7] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[8] D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets.* Addison-Wesley, 1998.

[9] Object Space. Voyager core package technical overview. Technical report, Object Space Inc., 1997.

[10] Sun Microsystems. Java Remote Method Invocation Specification, Oct. 1998.

[11] S. Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 14(2), Feb. 1997.

[12] J. E. White. Mobile agents. In J. Bradshaw, editor, *Software Agents*, pages 437–472. AAAI Press/MIT Press, 1997.