Modelling Mobile Objects in Open Distributed Systems

Position Paper for ECOOP'95 Workshop on Mobility and Replication

Thilo Kielmann
Univ. of Siegen, Germany
kielmann@informatik.uni-siegen.de

1 Introduction

Programming of open distributed systems is primarily concerned with coordinating concurrently operating active entities. Concurrent programming languages based on the concept of *generative communication* initiated the research area of *coordination* [3]. Today, the interaction between active entities is typically investigated based on this notion, and by introducing a variety of non–conventional computing models.

Coordination as the key concept for modelling concurrent systems is concerned with managing the communication which is necessary due to the distributed nature of a system, with the expression of parallel and distributed algorithms, as well as with all aspects of the composition of concurrent systems. We can characterize coordination by the following notions: *Agents* are active, self–contained entities performing *actions* on their own behalf. The actions can be divided into two different classes: (1) *Inter–Agent actions* which perform the communication between different agents and hence are the subject of coordination models. (2) *Intra–Agent actions* which are all actions belonging to a single agent like e.g. computations. We call a collection (or a system of) interacting agents a *configuration*. Finally, *coordination* can be defined as managing the inter–agent activities of agents collected in a configuration.

Coordination of agents can be expressed in terms of coordination models and languages, where *"A coordination model is the glue that binds separate activities into an ensemble"* [3] and a *coordination language* is *"the linguistic embodiment of a coordination model"* [3]. In other words, coordination models allow to describe how agents interact whereas coordination languages allow to express the modelling in terms of language constructs which are (favourably orthogonally) combined with a programming language.

Object–orientation has been well established as an approach to the design and implementation of large application systems. Its primary objectives concern program modularization, encapsulation, and reuse of software components. The obvious idea of exploiting the encapsulation property for concurrent programming has generated a lot of research work [1]. The most promising approach to concurrent object–oriented programming seems to be based on *active objects* which unify the notions of (passive) objects and processes. More specifically, an active object contains its own thread of control while it is still protected by its interface. Furthermore, the active–object approach enables generative communication to be included in object–oriented concurrent systems.

Open distributed processing is a currently evolving field which is characterized by the upcoming ISO standard on open distributed processing (ODP) [4]. In the ODP definition, *distributed systems* have to cope inherently with *remoteness* of components, with *concurrency*, the *lack of a global state*, and *asynchrony* of state changes. In addition, *open distributed systems* are characterized by *heterogeneity* in all parts of the involved systems, *autonomy* of various management or control authorities and organizational entities, *evolution* of the system configuration, and *mobility* of programs and data.

In the ODP model, communication is performed by providing and requesting services. The model enforces a request–reply communication structure which directly reflects objects as providers of services defined by their interface specification. Offering and using services is done by communicating with a trader, which uses a repository of type definitions in order to identify offered and requested service types. After the trader has offered the identification of an object capable to provide the requested service, client and server (in this model called importer and exporter) directly connect to each other.

Communication based on such object identifiers inherently bears the potential of dangling references in the case of dynamically changing configurations due to disappearing (or moving) objects. As will be

shown later, generative communication is needed to overcome these problems.

2 Objective Linda

As we have seen so far, the service–oriented communication model inherently uses request–reply pairs of messages which typically force direct connections between client and server in order to relate a reply to its corresponding request. The restricted RPC–like communication style is one disadvantage of this modelling. Even worse is the connection–based communication which hampers dynamically changing configurations with agents eventually appearing in and disappearing from a system.

Reflecting the needs of dynamically changing configurations of mobile agents, we can identify the following four basic elements of coordination models for open distributed systems:

1. Active Objects

   Objects are the building blocks of concurrent systems. More specifically, objects denote instances of abstract data types which enable the exploitation of software design and reuse known from classical object–oriented technology. Making objects active directly enables them to model active agents.

2. Generative Communication

   Generative Communication, known from the Linda coordination model, enables uncoupled (connectionless) communication with anonymous peers because it is based on creating and consuming first–class communication objects. It hence introduces the possibility to model mobile agents which dynamically enter and leave running configurations.

3. Homogeneity

   A simple homogeneous model should be introduced in order to support human programmers in building large systems. Ideally, there shall be only one sort of objects which unifies the notions of agent, data, and unit of application structuring.

4. Hierarchical Abstractions

   Whole configurations should be treatable like single agents. This demands for hierarchies of nested object spaces which represent application structure.

In the following, we briefly introduce the coordination model Objective Linda [5] which combines the basic elements identified so far and demonstrate its usefulness for modelling systems of mobile active objects.

The original Linda coordination model has been introduced to incorporate the idea of generative communication [2]. In Linda, processes (the coordination entities) communicate by putting tuples (basic data items like numbers and strings) into the so–called "tuple space" (by the `out` operation) and by reading or removing tuples from it (by the `read` and `in` operations). Synchronization is performed by forcing processes to wait until a suitable tuple to be read has been inserted into the tuple space. Furthermore, new processes can be invoked by putting active tuples into the tuple space (by the `eval` operation) for subsequent evaluation. Active tuples produce results in the form of passive tuples to which they are converted upon termination of their computations.

Objective Linda replaces Linda's notions of tuples and tuple spaces by objects. The objects themselves are instances of abstract data types which are defined by class hierarchies in a language–independent notation, called *Object Interchange Language* (OIL). Actual programs may then be written in traditional (sequential) object–oriented languages to which a language binding of OIL classes can be declared. Objective Linda has a strong emphasis on using the abstract data types for selecting objects to be consumed from an object space which, on one hand, introduces a higher abstraction level compared to Linda's pattern matching, and on the other hand enables interoperability between heterogeneous systems. For this purpose, every OIL object has to provide an interface method called `match` the implementation of which determines how objects of the corresponding class are identified in object spaces. The `in` and `read` operations are then supplied with a so–called reader object which specifies class and properties of an object to be consumed. In the following, the basic principles of Objective Linda are summarized:

- An Objective Linda computation is performed in a hierarchy of objects containing other objects.
- Coordination between objects is *only* performed by producing and consuming other objects (in a generative manner).
- Objects may be passive data or may be active, executing their own activity. Due to hierarchical decompositions, it is possible to have several activities concurrently operating on complex objects.

- Objects also act as object spaces (short: OS). These object spaces form the coordination part of the objects whereas the activities make up the computational part of the objects.

- Every object knows two object spaces it can operate with: its own OS, called `self`, and the OS it is directly included in, called its `context`.

- Objects residing in the same object space are called *peer objects.*

Figure 1 shows a configuration consisting of six nested objects. Objects are depicted by ovals and are either active or passive. The latter ones simply contain data. Active objects are invisible from their outside and may only be noticed by monitoring their passive peers which are produced and consumed by them. Internally, they consist of the activity (the object's computational part), depicted by a circle, and of an associated object space which is separated by a dashed line. This line separates the computational from the coordination part and illustrates the absence of sharing between both parts.

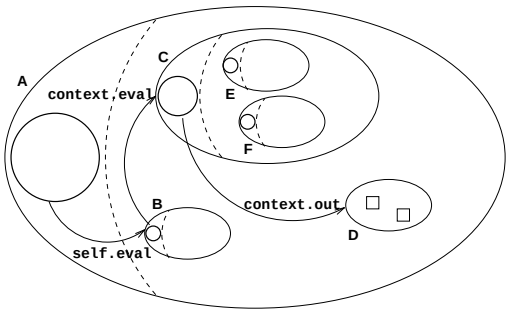

Figure 1: Some Abstract Nested Objects

In Figure 1, the active object A has created object B by performing `self.eval`. This is the way how a computation may be decomposed into several subtasks. B has created a peer object C by invoking `context.eval`. This way, new computations can be started without control of the invoker. C has created a peer D by `context.out` which is hence simply a passive object. By consuming D, B may receive results from its peer C. E and F are located in object space C. The latter one must have created at least one of them; the other one might have also been created by its peer.

3 Mobility in Objective Linda

Uncoupled generative communication enables to model mobile agents by allowing them to enter and leave configurations without loosing their capability to communicate with each other. We will now illustrate how Objective Linda exploits the generative communication style to model mobility itself.

For this purpose, we introduce *object space logicals* (or logicals, for short). These logicals are special objects which provide logical identifications for object spaces. Logicals are intended to be used as ordinary passive objects: Active objects willing to let others enter their object space `out` a logical into their `context`. Others may use such a logical to enter the other object space by executing the `join` operation introduced for this purpose. The basic idea behind this operation is that only the object space containing the logical is able to relate it to its source. One might compare logicals with keys while only the surrounding context object space has the corresponding locks where they fit into. In this sense, logicals are different from proxies, because they can not be used instead of an obejct space. They simply provide a clean and controlled way of entering an object space specified by certain ("logical") properties.

In order to fit for different purposes, there can be multiple classes for logicals which differ in the way they can be matched in charge of `in, read,` or `join` operations. Hence, different identification mechanisms can be realized. Examples may be numerical values, keywords, network addresses, geographical locations, or even "Uniform Resource Locators" (URLs) as they are used in the World–Wide–Web. Object space logicals are primarily passive objects. Additionally, they may be used for operations by which active objects can `join` object spaces.

Figure 2 illustrates how object mobility is realized in Objective Linda. We use the graphical notation from Figure 1. Here, all (rectangular) data objects are logicals. Objects A and B both have `out`'ed a logical for themselves into their common `context`. Additionally, A has `out`'ed a logical denoting its context into its `self` OS. This way, A allows its objects to leave object space A. In the moment t_1, object C is inside A. Here, it executes a `join` operation using a reader object matching the logical for A's context. Hence on t_2, C is inside A's context, which is the object space one level higher in the hierarchy. Here, C again executes a `join` operation, this time using the logical for B. Finally, at t_3, C is inside B.

The design for introducing object mobility is based on the assumption that object spaces provide the

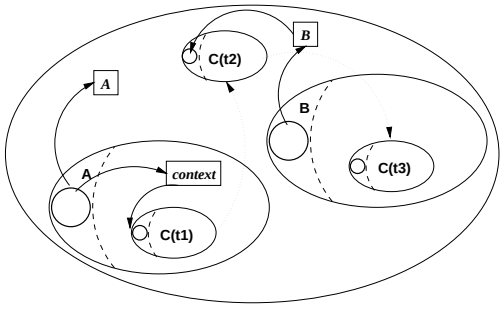

Figure 2: Object C moving from A to B

communication infrastructure which has to be protected against untrusted agents which occasionally enter an object space hierarchy. Consequently, object space logicals and the `join` operation have been designed such that only object spaces are capable of manipulating other objects. By design of logicals, the objects themselves have no capability of manipulating or accessing object spaces.

Besides keeping the encapsulation property of the object spaces, this design allows to keep the strict hierarchy of objects spaces, because objects are unable to reference or access other object spaces besides their `context` and `self` spaces. Furthermore, the way of object mobility introduced so far consequently employs the notion of generative communication between objects and object spaces in a configuration.

New agents willing to join running configurations are modelled in a similar way. In our model, agents have a default context OS on initialization of their activity. This OS might be implemented using some standardized network communication protocol. Possible implementations may be based on broadcast messages or on dedicated servers. Once initialized, this default context OS can be accessed in order to get a logical for any configuration an agent wishes to join, e.g. some given information system.

4 Conclusion

As we have outlined so far, programming active mobile agents in open distributed systems should preferably be based on a suitable coordination model which can be embodied with already existing (sequential) programming languages. Favourable properties of object–oriented programming and of generative communication have led to the design of the coordination model Objective Linda which allows to model communicating agents in the presence of moving communication partners as well as mobility itself.

Currently, there are two implementations of Objective Linda underway, both with language bindings to C++. One is based on a distributed shared–memory system and aims at implementing parallel algorithms. The other one is based on the MPI message passing interface standard and shall become a platform for modelling open distributed systems. Hence, we expect practical experiences in near future.

References

[1] Gul Agha, Peter Wegner, and Akinori Yonezawa (Eds.). *Research Directions in Concurrent Object–Oriented Programming.* MIT Press, Cambridge, Mass., 1993.

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

[3] David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):96–107, 1992.

[4] ISO/IEC JTC1/SC21/WG7. Reference Model of Open Distributed Processing. Draft International Standard ISO/IEC 10746–1 to 10746–4, Draft ITU–T Recommendation X.901 to X.904, May 1995.

[5] Thilo Kielmann. Object–Oriented Distributed Programming with Objective Linda. In *Proc. First International Workshop on High Speed Networks and Open Distributed Platforms*, St. Petersburg, Russia, 1995.