Object–Oriented Distributed Programming with Objective Linda*

Thilo Kielmann
University of Siegen
Dept. of Computer Science and Electrical Engineering
Hölderlinstr. 3
D–57068 Siegen, Germany
kielmann@informatik.uni-siegen.de

Abstract

In this paper we introduce the coordination model Objective Linda which has been designed to meet the needs of open distributed systems by rigorously combining object orientation with uncoupled communication. Hierarchical abstractions are provided for structuring large systems. It will be shown that using Objective Linda, interoperability can be achieved between different programming languages and heterogeneous system architectures. Its usefulness for open distributed systems will be illustrated by examples.

1 Introduction

Programming of open distributed systems is primarily concerned with coordinating concurrently operating active entities. In traditional concurrent programming, there are only three basic kinds of mechanisms and corresponding models [4]: shared variables, message passing, and remote procedure calls. However, parallel programming languages based on these concepts are not fully suitable to program open distributed architectures.

A fourth basic model called *generative communication* was introduced in [6]. Concurrent languages based on this concept initiated the research area of *coordination* [8]. Today, the interaction between active entities is typically investigated based on the notion of coordination, and by introducing a variety of non–conventional computing models, like for example the very influential work in [3].

Object–Orientation has been well established as an approach to the design and implementation of large application systems. The central notions exploited by object–oriented programming are objects, classes, and inheritance as means to structure applications and libraries of reusable software components. Hence, the way object–orientation is used in traditional sequential programming, objects are building blocks defined as abstract data types, which encapsulate their internal state through well-defined interfaces [15]. This traditional kind of objects simply represents passive data containers.

Because objects represent units of data encapsulation, the obvious idea of exploiting this property for purposes of concurrent programming has generated a lot of research work [1, 2, 9, 23]. The simplest way to achieve concurrent object–oriented programming is to add the notion of processes (usually in the form of lightweight processes, or "threads") to a given object–oriented language. In this way, all three classical models of concurrent programming can be realized [19]: Shared–variables programming can be done by adding monitor semantics to the objects. Message

*Proc. First International Workshop on High Speed Networks and Open Distributed Platforms, St. Petersburg (Russia), June 12 – 15, 1995 passing can be realized by giving method invocations the semantics of messages being exchanged. Finally, it is possible to model remote procedure calls by treating method invocations as procedure calls (as within most sequential object–oriented programming languages), but allowing objects being called to be "remote".

Independently from which one of these concurrency models is actually in use, it is in the responsibility of every object to keep its own state consistent. Therefore, it is necessary to control the operations concurrently executing on a given object. This problem is subject to a lot of ongoing research work. The fundamental problem is based on the interferences between concurrency control mechanisms and notions of reuse based on inheritance. This phenomenon is known as "inheritance anomaly" [14] and can be seen as the main obstacle prohibiting the wide use of this style of object–oriented concurrent programming.

By keeping the key idea of encapsulated entities, an alternative and much more attractive way of system modelling can be used. Here, *active objects* 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. This approach eliminates the consistency problems mentioned above because there is exactly one thread operating in an active object. Furthermore, the active–object approach enables generative communication to be included in object–oriented concurrent systems.

To conclude, active objects are well-suited to be used as active components in concurrent systems, because they combine the notions of encapsulated active entities with the power of object–oriented software development techniques.

In the following sections we concisely introduce the notions of coordination and of open distributed systems after which we identify properties of coordination models suitable for modelling them. Then, we investigate existing approaches to programming open distributed systems based on services and on generative communication. On the basis of this foundation we introduce the coordination model Objective Linda. We explain its principal properties and components after which we exemplify Objective Linda's usefulness for modelling and programming open distributed systems.

2 Coordination

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:

Agent Agents are active, self–contained entities performing actions on their own behalf.

Action Actions can be divided into two different classes:

1. *Inter–Agent actions.* These actions perform the communication between different agents. They are the subject of coordination models.
2. *Intra–Agent actions.* These are all actions belonging to a single agent. They perform computations as well as all communication of an agent outside the coordination model, like primitive I/O operations or interactions with users.

Configuration We call a collection (or a system of) interacting agents a configuration.

Coordination Coordination is managing the inter–agent activities of agents collected in a configuration.

Coordination of agents can be expressed in terms of coordination models and languages. In the following, we try to clarify these two different notions. For coordination models we prefer the following intuitive definition: *"A coordination model is the glue that binds separate activities into an ensemble"* [8]. In other words, a coordination model provides a framework in which the interaction of individual agents can be expressed. This covers the aspects of creation and destruction of agents, communication among agents, spatial distribution of agents, as well as synchronization and distribution of actions over time.

A *coordination language* is *"the linguistic embodiment of a coordination model"* [8]. Thus, a coordination language should orthogonally combine two models: one for coordination (the inter–agent actions) and one for (sequential) computation (the intra–agent actions). The presumably most famous example of a coordination model is the Tuple Space in Linda which encountered several linguistic embodiments like C–Linda or FORTRAN–Linda, both on workstation networks and on massively parallel architectures.

3 Open distributed systems

Open distributed processing is a currently evolving field. It is characterized by the ISO standard on open distributed processing (ODP), the current draft of which can be found in [11]. It describes open distributed systems from different viewpoints. In the following, we will extract the parts of these descriptions relevant to programming models. Here, the objective of ODP is to "allow the benefits of distribution of information processing services to be realised in an environment of heterogeneous information technology resources and multiple organizational domains."

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 to some extent *mobility* of programs and data.

We can identify the requirements of open distributed systems on programming models by the notions of heterogeneity in the following senses:

1. Heterogeneity of hardware.

   Open distributed systems may consist of computers of various architectures, from different vendors, and with different data representations. Furthermore, there may be very different interconnection media and topologies. As a result, coordination models for open distributed systems cannot rely on specific data representations or communication structures.

2. Heterogeneity of software.

   Computers in open distributed systems of course run different operating systems. Also, programming languages in use may vary depending on the purposes of every system. Consequently, coordination models trying to integrate such systems cannot be bound to specific language interfaces or communication protocols.

3. Heterogeneity of configurations over time.

   Besides the different kinds of the involved systems, the most challenging property of open distributed systems is their dynamic nature. Examples are situations in which additional machines will be brought into the system, dial-up lines connect and disconnect, new machines replace older ones, new operating systems or communication protocols have to be integrated etc.

   Thus, agents in an open distributed system must be allowed to appear and disappear completely on their own behalf. This forbids coordination models to rely on specific (central) units or to make use of communication schemes based on static connections or specific identifiers (addresses).

4 Service–Oriented Models

We now investigate the traditional approach to programming of open distributed systems which focusses on providing and requesting services. Models following this approach are typically focussed around the notion of objects which provide the methods of their interfaces as services to other objects. Main objectives of such models are remoteness and service identification.

4.1 The ISO/IEC Reference Model of Open Distributed Processing

The upcoming ODP standard [11] uses objects as its basic modelling concept: "Object modelling was chosen because ... objects are useful for structuring and specification purposes, because they embody ideas of modularity and data abstraction. ... They embody ideas of services offered by an object to its environments, that is, to other objects."

This notion of objects originates in traditional passive objects and hence bears the problems of that approach as outline above. It focusses on encapsulation and abstract data types which are only weak concepts for modularization and system structuring purposes. Furthermore, it enforces a request–reply communication structure which directly reflects objects as providers of services defined by their interface specification. This service–oriented communication style seems to be appropriate for open distributed systems on first sight. It reflects applications with transaction–like operations, but it excludes different communication structures like for example group communications.

In the ODP model, services are identified using a so–called trading function [10]. 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. A subtype relation between interface types can be explicitly declared or derived from subtyping rules. 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. The trading function standard simply provides a framework which might be embodied by different services like the X.500 directory service or OMG's CORBA IDL [17].

4.2 LAURA

The LAURA model [22] introduces agents using and offering services which are exchanged in a *service space* shared by all agents. This space is an equivalent to Linda's tuple space and enables uncoupled communication in the context of exchanging services. Therefore, LAURA provides three operations for service providers and requestors. First, a provider can use *SERVE* to put a serve form into the service space. The operation blocks the provider until a matching request form is found. After completing a service request, the provider performs the *RESULT* operation which puts a result form into the service space.

A client, on the other hand, uses a *SERVICE* operation to put a service–request form into the service space. This operation blocks the client until the results are available. The realization of this operation introduces uncoupled communication to service exchange: Instead of a single service–request there will be a service–put form and a service–get form inserted in the service space. The service–put form will be matched against a serve form whereas the service–get will match the corresponding result. Both forms, service–put and service–get will internally be augmented by a unique request ID which is used to relate the result to the correct request. This way, provider and client are anonymous to each other, and can exchange services in a connectionless and hence uncoupled manner.

5 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. An approach to overcome the deficiencies of connection–based communication is LAURA which introduces uncoupled communication for implementing the service–oriented style. But none of those models provides suitable abstractions in order to structure large systems.

The requirements analysis for general–purpose coordination models performed so far exhibited the necessity of uncoupled communication in order to cope with open distributed systems. As the communication style should not be confined to RPC, we have to keep the models more open. Therefore, we can identify the following four basic elements:

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 furthermore allows programming without the intra–object concurrency problems of the passive object approach.

2. Generative Communication

   Generative Communication, known from the Linda coordination model, enables uncoupled communication with anonymous peers and hence introduces the possibility of agents which dynamically enter and leave running configurations.

3. Homogeneity

   A single homogeneous model should be introduced, which is suited to be applied in a range from fine–grain parallelism to world–wide open distributed 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 and enable concurrency inside composed active objects.

In the following, we introduce a coordination model called *Objective Linda* which is intended to be suitable for various kinds of concurrent systems and especially open distributed systems. Because it is designed around a rigorous combination of object orientation and the Linda coordination model, we have to recall the latter first:

The Linda coordination model has been introduced to incorporate the idea of generative communication [6]. In Linda, processes communicate by putting tuples (in the mathematical sense; consisting of 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 letting processes 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) which are in turn evaluated. Active tuples produce results in form of passive tuples to which they are converted on termination of their computation.

The main coordination law defines how tuples are selected to be read from the tuple space. The potential reader specifies a template for a tuple it wishes to obtain. The tuple space performs a matching operation in order to find an appropriate tuple. Both tuples and templates may consist of actual fields (values) and formal fields (placeholders for specific data types). A tuple matches a given template if the arities of both correspond and if each actual field matches one of the same type and value or a formal field of the corresponding type.

Of course, Objective Linda is to some degree inspired by other approaches to improve the pure Linda model: Bauhaus Linda [5] unifies tuples and tuple spaces by replacing both by multisets. Objective Linda extends this to a model with a unique kind of entities: Objects serve as data units, as active agents, as object spaces, and as units of application structure. So, the matching process for reading from object spaces can be based on predicates from the objects' abstract data types which significantly improves expressive power.

The object space approach [20] attempts to uncouple object matching performed by Linda's *in* and *rd* operations from the object implementation by letting programmers describe "important" parts of the implementations which become subject to matching. So, objects loose their tuple character as ordered sequences of typed slots. Objective Linda expands this idea consequently: Objects are matched based on predicates out of the interface of their abstract data types. This property completely separates implementation from specification and enables the latter to become the basis of object interaction in the presence of hererogeneous systems. Furthermore, it removes all implementation details from the realization of agent interaction and hence applies this idea from sequential programming to agent interaction in concurrent systems.

The work in [7, 12, 13] proposes hierarchies of nested tuples spaces by introducing tuple spaces as first–class entities of the model. All of them introduce names (or references) to tuple spaces, whereas [12] additionally introduces "relative" references by a special reference called the *context* which denotes the tuple space in which a given tuple space is located. Such tuple space hierarchies contribute to the introduction of hierarchical abstractions over configurations. Unfortunately, references to tuple spaces destroy the strict hierarchy by imposing a flat and global namespace over the whole concurrent system. Consequently, Objective Linda provides hierarchies of nested object spaces in which only the current object space and its direct context are accessible. So, there are no global or shared objects in an Objective Linda configuration.

The work in [5, 7, 12] additionally treats activities as first–class entities and hence allows scheduling by explicitly starting and freezing them. In Objective Linda, objects are introduced by abstract data types. Operations on objects are hence defined by their interface specifications which assume operation atomicity. This contradicts to freezing active operations because intermediate states are undefined. Because the motivations for introducing operations on active tuples in the work quoted above are not too convincing, Objective Linda has no such concepts.

5.1 The Coordination Model

We will now introduce the core concepts of Objective Linda. Here we unify Linda's notions of tuples and tuple spaces and replace both 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) which we unfortunately cannot present here due to space limitations. Actual programs may then be written in traditional (sequential) object–oriented languages to which language bindings of the 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.

Objective Linda computations are performed in hierarchies of objects containing other objects. Coordination between objects is *only* performed by producing and consuming other objects (in a generative manner). Objects are instances of OIL classes which define the sets of methods the objects can execute. 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. Active objects act also 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. Between these two parts, there is absolutely no information sharing. All objects either are part of a computation, or stored in an object space, but never both. Every active object knows exactly two object spaces in can operate with: Its own OS, called `self` and the OS it is directly included in, called its `context`. Finally, objects residing in the same object space are called *peer objects.*

The tuple space operations from the original Linda model inspired the core operations needed for object spaces. We show how they have to be adapted to fit into the object–oriented setting of Objective Linda by informally introducing the following operations of the abstract data type *object space*:

1. `out`. When an object o `out`'s another object $o2$ from its computational part into an OS, $o2$ is moved into the OS. As an effect of this operation, o has lost connection and access to $o2$.

2. `in`. When an object o `in`'s an object i from an OS, i is moved into the computational part of o. `in` arbitrary selects an object in the OS which matches the given requirements (see below). If there is no such object, the `in` operation blocks until there is one.

3. `rd`. When an object o `rd`'s another object r from an OS, it moves a clone of r into its computational part.

4. For completeness there are two predicative versions of `in` and `rd`, called `inp` and `rdp` which immediately deliver an object if the OS actually contains (at least) one matching the given requirements. If not, `inp` and `rdp` return without delivering an object instead of blocking.

5. `eval`. An `eval` operation is identical to `out` except that after `out`'ing, object e will become active and process a method m out of its class interface. When m terminates, object e in turn disappears from the OS it was in. (When method m has to deliver results, object e has to `out` result objects.) Termination and removal of active objects is a bit more complicated in the presence of nested active objects. Hence, an active object o is removed from its context OS only as soon as its own method has terminated and all active objects in o's `self`–OS have terminated and disappeared.

   In one point active objects differ from passive ones: They are invisible to `in` and `rd` operations. This is important to avoid interrupting running activities which would make no sense with respect to the (atomic) semantics of the methods being executed.

All operations on an object space must be executed in a serializable order [21] which means that if two or more operations are running at the same time, to each of them and to other operations, the final result looks as though all operations ran sequentially in some (system dependent) order. This is necessary in order to cope with concurrent operations which change the state of object spaces. This way, operation semantics of the abstract data type *object space* are preserved. Of course, operations on different object spaces may be performed concurrently without any restrictions.

Figure 1 shows as an example 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.

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. This way, 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.

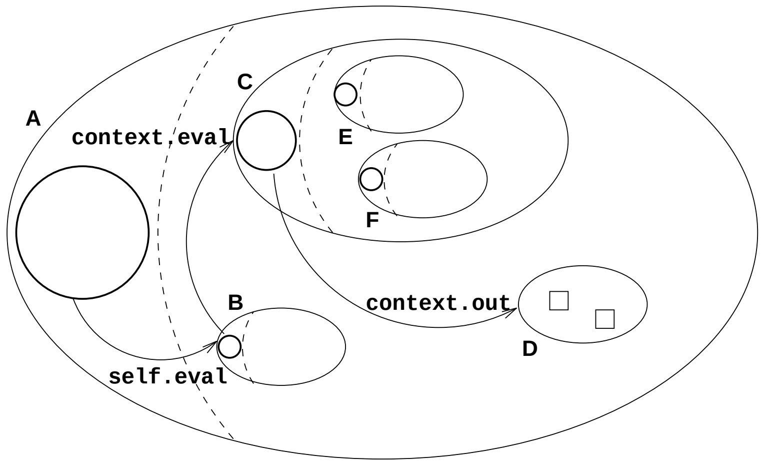

Figure 1: Some abstract nested objects

5.2 Object Matching and Method Evaluation

The conceptual framework for `in`, `rd`, and `eval` operations needs further elaboration. This is necessary because the corresponding operations of the traditional Linda model have to be adapted to our object–oriented framework. Furthermore, the way `in`, `rd`, and `eval` are modeled significantly influences the expressive power of Objective Linda.

Matching in the traditional Linda model is based on the data representation, in fact the implementation, of tuples. There is no other mechanism for reasoning about tuples than just interpreting tuple arity, types, and values of the tuple elements. The only degree of freedom is introduced by formal fields which allow to match every value of a given type. But this is not suitable for matching objects which are instances of given abstract data types. Because objects may only be accessed based on their class interface, matching must also be performed that way. This forbids the examination of implementation aspects and hence enables interoperability in heterogeneous systems because objects are only examined by using their abstract interface. Object matching based on abstract data types additionally enables arbitrary predicates to be used for this purpose which is much more flexible than just matching one or all values of a type.

Unfortunately, usual object–oriented languages do not allow to have types or predicates as parameters to a given routine like `in`. The only possible kind of parameters are objects. So we have to pass types and predicates by passing objects. Passing a type is simple because when we pass an object its actual type can be taken as the desired one. Passing different predicates which parameterize the matching process can be achieved in several more or less elegant ways in object–oriented systems.

In Objective Linda, we have to specify the type of object we wish to receive, so we have to pass an object of the corresponding class to the `in` and `rd` operations anyway. So it is an obvious idea to pass the matching predicates inside this object, too. This resembles a concept introduced in [13] where `in` and `rd` operations are performed by putting a so–called *reader object* into the object space. In our case, a reader object would be an object of the type to be matched and the state of which contains all necessary information to parameterize the function (e.g. called) `match` which becomes a mandatory routine of all classes of objects to be included in object spaces.

This approach has the advantage of concentrating the information concerning a given class and especially concerning the matching of objects of this class inside the class itself. Object encapsulation is still kept because all objects (even of the same class) have to use the class interface to reason on properties of a given object.

With the solution developed so far, it is simple to realize the `eval` operation of the OS, too: Every class gets another mandatory routine called `evaluate` which is executed when the corresponding object is put into an OS by `eval`. As with `match`, the necessary parameterization has to be performed by other operations on the object before it is put into the OS.

5.3 Mobile Objects

The core coordination model introduced so far directly adapts the traditional Linda operations to our object–oriented setting of hierarchically structured object spaces. Open distributed systems require to additionally model agents eventually entering or leaving configurations. Especially agents occasionally wishing to enter a running system require to model activities which are initiated outside the system (as opposed to new active agents created by the `eval` operation). Furthermore, mobile agents are a requirement of open distributed processing [11]. Objective Linda models both requirements by a special operation which allows agents to move from one object space to another.

The principal problem in introducing the new operation is to seamlessly embed it into the model while not violating its properties, namely encapsulation of object spaces and their strict hierarchy. For this reason, it must not be allowed to give active objects direct access to object spaces even in the form of handles as it is the case for tuple spaces in [12]. Such handles to object spaces would immediately destroy the object–space hierarchy, because an agent might “keep in mind” a set of handles while “travelling” and hence become able to arbitrarily move around in a configuration ignoring each and every hierarchical structure. Finally, the possiblity of entering a given object space must be under the control of the object space itself in order to keep its encapsulation property.

These considerations led to the introduction of a new concept: *object space logicals* (or logicals, for short). These logicals are special OIL 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 new operation introduced in this section. 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 knows the corresponding locks where they fit into. In order to fit for different purposes, there will be multiple classes for logicals which differ in the way their `match` routine is implemented. Hence, different identification mechanisms can be realized. Examples may be numerical values, strings, network addresses, 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 the following operations:

1. `join (l : OS_LOGICAL)`. When an object o wants to join another object space, it calls the `join` operation on its current `context` OS and specifies the desired target OS by `l`. `l` is moved into `context` (like a reader object used by an `in` operation). `join` arbitrarily selects a logical m it contains which matches `l`. o then joins the object space which is denoted by m. As long as there is no matching logical denoting an object space for which the movement operation succeeds, the `join` operation blocks.

2. `enter (l : OS_LOGICAL)`. `enter` operates exactly like `join`. Additionally, the matching logical m is consumed from the `context` OS.

3. Additionally, there are predicative versions of `join` and `enter`, called `joinp` and `enterp` which either move object o to an object space or alternatively return immediately, indicating the failure. In the latter case, the operations have no effects besides failure indication.

Figure 2 shows how the concept of object mobility can be used to model new agents willing to join running configurations in an open distributed system. In our model, agents have a default context OS on initialization of their activity, in the example called "World". 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.

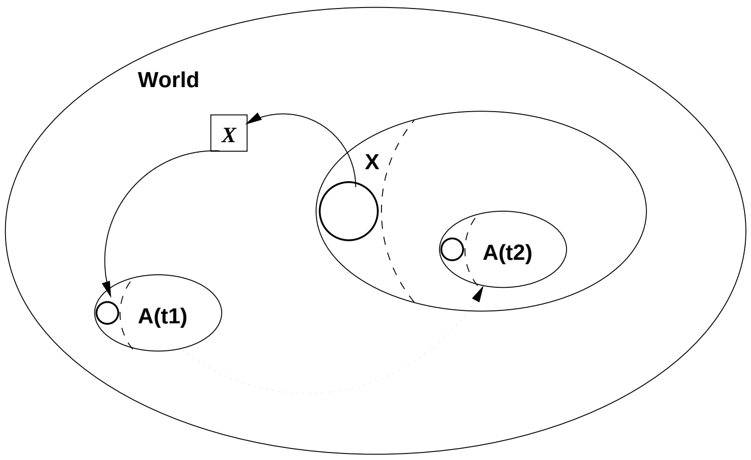

Figure 2: A new agent A entering configuration X

6 Examples

We will now provide two examples for the use of Objective Linda in open distributed systems. First, we present an improved scheme for providing services. The second example outlines the use of Objective Linda for retrieving information in the WWW.

6.1 Providing Services

The software architectures for open distributed processing, as introduced in Section 4, are based on providing and requesting services. Of course, this communication style can be modelled with Objective Linda, too. In the following, we introduce two classes, namely *SERVICE_REQUEST* and *SERVICE_RESULT*. Clients put service requests into an object space from which servers may take them in order to compute results which they return to the object space. Due to Objective Linda's uncoupled communication style, we additionally have to relate results to their corresponding requests. For this purpose, LAURA [22] introduces unique identifiers into the system internal communication. In Objective Linda, we do not need to implement this on the system level. We can simply make use of unique identifications in the classes built for this purpose. On creating a *SERVICE_REQUEST* object, a unique identifier is created for and stored in it. The server may take this and put it into the *SERVICE_RESULT* object it creates. The `match` routine of this class then simply compares the identifiers. A possible realization of such identifiers might be the "universal unique ID's" from OSF DCE [18] which globally unique combine host ids with timestamps.

The code fragments of clients and servers may look as outlined in Figure 3 which uses Eiffel [16] notation. The communication shown is furthermore improved comparing to the RPC style of classical service–oriented systems: Here, servers are not forced to process a request by being called. Instead, they are free to process it or to reject it. This is simply due to generative communication.

Clients	Servers
`req : SERVICE_REQUEST;`	`req : SERVICE_REQUEST;`
`res : SERVICE_RESULT;`	`res : SERVICE_RESULT;`
`id : UUID;`	
`!!req.make;`	`!!req.make;`
`!!id.make;`	
`req.put_uuid(id);`	
`context.out(req);`	`context.in(req);`
	`if req.is_valid`
	`then -- compute service`
	`  !!res.make;`
`!!res.make;`	`  res.put_uuid(req.get_uuid);`
`context.in(res);`	`  context.out(res);`
	`else -- reject request`
	`   context.out(req);`

Figure 3: Providing Services with Objective Linda

6.2 Information Retrieval in the WWW

As another example for the usefulness of Objective Linda for open distributed systems, we now try to outline how Objective Linda can contribute to improve the capabilities for finding information in the World–Wide–Web.

In general, the WWW can be seen as a flat space of objects of various (document) types with innumerable references between them. Basically, a document can only be found by its *Uniform Resource Locator* (URL) which a potential reader has to know or to guess. With the growing size of the net, *finding* information has become more and more difficult. Consequently, several sites started to maintain lists of interesting URLs and special search engines. But these lists and search engines locally gather information which is always endangered to become out of date. The core problem here is that information is only available via URLs and not by its content. In the following, we propose three steps which make use of Objective Linda in order to improve the WWW with respect to retrieving information stored in it.

The first step would be to use object spaces as search engines. Therefore, each Web server (willing to provide the retrieval service) would additionally implement an object space in which its documents are represented by objects. Because this object space would directly be implemented on top of the server's file system, this would be an example of a persistent object space. Access to this object space could then be provided by network communication. Match operations based on document contents like keywords, full text search, or similiar mechanisms for image or audio data could facilitate searching on a given server. This search facility obviously superseeds mechanisms based on dedicated data collections because now searching could be performed based on the documents themselves which removes problems with out-of-date information.

The second step would be to organize Web servers in object space hierarchies. So far, users still had to know *where* to search. Representing the servers hierarchically alleviates this problem, too. There could be several subhierarchies providing different views on the information. Views could be geographically, organizationally, or oriented on topics. For example, Web servers of Siegen University should provide their information in object spaces of German sites, of educational institutions, and the server of the CS department should also be found in an object space related to computer science.

On first sight, such an organization contradicts to Objective Linda's property of non–intersecting object spaces which would prevent servers being represented in more than one object space. Instead, Web servers could create active objects as "proxy agents" which could move inside the object space hierarchy into the object spaces in which servers wish to be "classified". Such a scheme would again be superior to others because there is no centralized information repository. Instead, information providers themselves would be responsible for and able to make their information accessible in ways suitable for them. Furthermore, a client's information search could also be performed by an active agent object which moves around in the object space hierarchy.

As a last step, the WWW servers themselves could be organized as object space hierarchies in order to simplify and to improve the information retrieval tasks performed by them.

Conclusion

In this paper we introduced the coordination model Objective Linda. Therefore, we discussed the notion of coordination in the context of open distributed systems which are characterized by heterogeneous components as well as dynamically changing configurations. We identified active objects and generative communication as necessary elements of coordination models for open distributed systems whereas homogeneity and hierarchical abstractions have to be considered as vital requirements for building really large systems. We then introduced Objective Linda which consequently realizes these properties by providing a coordination model in which communication is performed generatively in hierarchies of nested homogeneous objects.

In Objective Linda, uncoupled communication, encapsulated objects, a language–independent description of object classes, and the possibility to implement object spaces differently in order to suit the needs of various concurrent systems by inheriting from the general class for object spaces all together contribute to programming open distributed systems with powerful and flexible abstractions.

References

[1] Gul Agha, Carl Hewitt, Peter Wegner, and Akinori Yonezawa, editors. *Proc. of the ECOOP–OOPSLA Workshop on Object–Based Concurrent Programming*, Ottawa, Canada, 1990. Published as OOPS Messenger2(2), 1991.

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

[3] J. M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction Abstract Machines. In *Research Directions in Concurrent Object–Oriented Programming* [2], pages 257–280.

[4] G. Andrews. Synchronizing Resources. *ACM Transactions on Programming Languages and Systems*, 3(4):405–430, 1981.

[5] Nicholas Carriero, David Gelernter, Susanne Hupfer, and Lenore Zuck. Bauhaus–Linda. In *Proc. of ECOOP'94 Workshop on Languages and Models for Coordination*, Bologna, Italy, 1994.

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

[7] David Gelernter. Multiple Tuple Spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE'89, Parallel Architectures and Languages Europe*, number 366 in Lecture Notes in Computer Science, pages 20–27, Eindhoven, The Netherlands, 1989. Springer.

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

[9] R. Guerraoui, O. Nierstrasz, and M. Riveill, editors. *Proceedings of the ECOOP '93 Workshop on Object–Based Distributed Programming*, number 791 in Lecture Notes in Computer Science, Kaiserslautern, Germany, 1993. Springer.

[10] ISO/IEC JTC1/SC21/WG7. Information Technology – Open Distributed Processing – ODP Trading Function. Draft ISO/IEC Standard 13235, Draft ITU–T Recommendation X.9tr, July 1994.

[11] ISO/IEC JTC1/SC21/WG7. Reference Model of Open Distributed Processing. Draft ISO/IEC Standard 10746–1 to 10746–4, Draft ITU–T Recommendation X.901 to X.904, July 1994.

[12] Keld K. Jensen. *Towards a Multiple Tuple Space Model.* PhD dissertation, Aalborg University, Dept. of Mathematics and Computer Science, Inst. for Electronic Systems, Fredrik Bajers Vej 7E, DK-9220 Aalborg Ø, Denmark, 1994.

[13] Satoshi Matsuoka and Satoru Kawai. Using Tuple Space Communication in Distributed Object-Oriented Languages. In *ACM Conference Procedings, Object Oriented Programming Systems, Languages and Applications, San Diego California*, pages 276–284, 1988.

[14] Satoshi Matsuoka and Akinori Yonezawa. Analysis of Inheritance Anomaly in Object–Oriented Concurrent Programming Languages. In *Research Directions in Concurrent Object–Oriented Programming* [2], pages 107–150.

[15] Bertrand Meyer. *Object–oriented Software Construction.* Prentice Hall, New York, 1988.

[16] Bertrand Meyer. *Eiffel the Language.* Prentice Hall, 1992.

[17] Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG Document Number 93.12.43, 1993.

[18] Open Software Foundation. Introduction to OSF DCE. Open Software Foundation, Cambridge, USA, 1992.

[19] Michael Papathomas. Concurrency in Object–Oriented Programming Languages. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 2, pages 31–68. Prentice Hall, 1995.

[20] Andreas Polze. The Object Space Approach: Decoupled Communication in C++. In *Proc. of Technology of Object–Oriented Languages and Systems (TOOLS) USA'93*, Santa Barbara, 1993. Prentice Hall.

[21] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice Hall, 1995.

[22] Robert Tolksdorf. *Coordination in Open Distributed Systems.* PhD dissertation, Technical University of Berlin, Berlin, Germany, 1994.

[23] Akinori Yonezawa and Mario Tokoro, editors. *Object–Oriented Concurrent Programming.* The MIT Press, Cambridge, Mass., 1987.