Designing a Coordination Model for Open Systems

Thilo Kielmann

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

Abstract. Coordination models for closed concurrent systems like Linda and Gamma have been well established during the last few years. Closed systems typically are planned ahead and consist only of active components the behaviour of which is known in advance. In contrast, open systems are inherently heterogeneous and dynamically change their configuration over time. Models for coordinating software agents in open systems must therefore be able to cope with constantly changing configurations and new, unknown kinds of agents. In this paper, we identify the requirements of open systems with respect to coordination models and develop a coordination model suitable for these purposes.

1 Introduction

Open systems are systems in which new active entities (usually called "objects", "agents", or "actors") may dynamically join and later leave, i.e. evolving self–organizing systems of interacting intelligent agents [2, 10]. More precisely, open systems can be defined as being composed of software components which are *encapsulated* and *reactive* [31]. Components are called encapsulated if they have an interface that hides their implementation from clients; they are called to be reactive if their lifetime is longer than that of the atomic interactions (e.g. messages) which they execute. The fundamental property of open systems is their ability to cope with incremental adaptability, where encapsulation captures spatial incrementality by controlled propagation of local state changes and reactiveness enables temporal evolution by incrementally executing interactions.

A related important notion is the one of *open distributed systems*. It is defined in the upcoming ISO reference model of open distributed processing (RM–ODP) [15]. In the RM–ODP definition, *distributed systems* have to cope 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.

Programming of open systems is primarily concerned with coordinating concurrently operating active entities. Concurrent programming languages based on the concept of *generative communication* [12] initiated the research area of *coordination* [14]. Today, the interaction between active entities in open systems is typically investigated based on this notion [4, 7].

Coordination as the key concept for modelling concurrent systems involves managing the communication which is necessary due to the distributed nature

of a system, the expression of parallel and distributed algorithms, as well as all aspects of the composition of concurrent systems. We characterize coordination by the following notions: *agents* are active, self–contained entities performing *actions* on their own behalf. Actions are divided into two different classes: (1) *inter–agent actions* which perform the communication between different agents and hence are the subject of coordination models, and (2) *intra–agent actions* which are all actions belonging to a single agent like computations, low–level I/O operations or interactions with users. We call a collection (or a system of) interacting agents a *configuration*. Hence, *coordination* can be defined as managing the inter–agent activities of agents collected in a configuration.

The aim of this paper is to present a coordination model designed for the needs of open systems. For this purpose, we investigate the requirements imposed on coordination models by open (distributed) systems in Sect. 2 and emphasize the importance of generative communication in Sect. 3. Section 4 investigates related work before in Sect. 5 the design of our coordination model Objective Linda is presented some of the basic ideas of which have been introduced in another context in [19]. In Sect. 6 we finally illustrate Objective Linda's features by an example.

2 Coordination Models for Open Distributed Systems

As mentioned earlier, agents must be encapsulated and reactive in order to operate in open systems. Furthermore, coordination models which provide "the glue that binds separate activities into an ensemble" [14] have to reflect openness, too. Hence, coordination models for open systems have to satisfy the following requirements:

- At any time, agents must be allowed to enter or leave a configuration (**dynamics**). This implies that coordination laws must not rely on the existence of specific agents. The same holds for communication. Generative communication [12] by generating and consuming separate entities, usually contained in a specific computational space, is required in order to provide a suitable communication model (**generativeness**). On the other hand, it is also essential to protect a configuration from undesired interaction with agents outside (**encapsulation**).
  By the definition of open distributed systems, there is no overall compile time. Hence, it must be possible to program new agents during the runtime of an already existing system, i.e. specify a new agent's behaviour in a separate program (**decentralization**).
- Coordination models for use in heterogeneous environments must not rely on properties of specific hardware, programming languages or communication media like data types or their representations (**interoperability**).

Besides capturing the requirements of *what* to specify, the methods used *how* to model systems are essential in order to build large but still maintainable systems.

- A model suited for coordinating large systems must be as simple as possible. Hence, all agents should be modelled in a uniform way (**homogeneity**). For the purpose of large systems, it is vital to divide the overall configuration

into smaller subconfigurations. Hence, it must be possible to treat entire configurations like single agents at a more abstract coordination level (**hierarchical abstraction**). Inter–agent actions have to be cleanly separated from intra–agent actions in order to distinguish between the concerns of coordination on one hand and of computations on the other (**separation of concerns**).

Encapsulation and reactiveness as central requirements of agents in open systems directly leads to object–based modelling. Objects are by their very nature open interactive systems. They can not (completely) be described algorithmically because they interact while computing [32]. Hence, specification of object–based systems is inherently incomplete and hence reflects openness.

The RM–ODP model which conceptually provides the basis for commercially available systems uses object–based modelling too; also because of the principal object properties of encapsulation and reactiveness. RM–ODP focuses on interaction between objects based on the client/server architecture: "They (objects) embody ideas of services offered by an object to its environments, that is, to other objects." [15] In RM–ODP, coordination between objects takes place via centralized instances, so called *traders* [16], which are repositories of service type definitions, used to identify offered and requested services.

Presumably the most prominent commercial system for open, object–based systems is the Common Object Request Broker Architecture (CORBA) [25]. Its central component, the Object Request Broker (ORB) acts as a trader in the sense of RM–ODP. Like other traders, the ORB provides references to server objects which in case of dynamically changing configurations may quickly turn into void ("dangling") references causing problems in open configurations. Today, client/server architectures are seen as the current intermediate step on the way from mainframe–oriented to collaborative (peer–to–peer) computing [22]. Nevertheless, service–oriented communication is an important paradigm for open distributed systems [1] and must hence be captured by coordination models. But because client/server communication is restricted to the exchange of request/reply pairs, other communication forms like e.g. for group communication can not be modelled adequately. Hence, coordination models for open systems need to be more general in their applicability.

3 The Role of Generative Communication

Generative communication as initially introduced in [12] is based on a shared data space, sometimes also called *blackboard*, in which data items can be stored ("generated") and later retrieved. This kind of communication inherently uncouples communicating agents: a potential reader of some data item does not have to take care about it (e.g. as with rendezvous mechanisms) until it really wants to read it. The reader even does not have to exist at the time of storing. The latter point directly leads to the other major advantage of generative communication: agents are able to communicate although they are anonymous to each other.

This uncoupled and anonymous communication style directly contributes to the design of coordination models for open systems: uncoupled communication enables to cope with dynamically changing configurations in which agents move

or temporarily disappear. Anonymous communication allows to communicate with unknown agents. Hence it allows communication with incomplete knowledge about the system configuration which is a crucial demand of open systems. Due to this fact, coordination models based on generative communication are superior to message passing or trader–based schemes because these both rely on knowledge about a receiver's or server's identification.

Based on this observation, the LAURA model [30] has been developed in order to introduce generative communication into the RM–ODP model. In LAURA, agents using and offering services share a so–called *service space*. Here, *offer* and *request* forms are matched by LAURA's service type system which replaces RM–ODP's trading function. This model introduces uncoupled and anonymous communication into RM–ODP, but it does not help to overcome the rather restrictive communication scheme of request/reply pairs. Hence, a general–purpose coordination model for open systems needs further improvements.

4 Object–based Generative Models

As we have seen so far, coordination models for open systems must be based on generative communication and on object–based modelling. A model combining these two features covers the requirements of *dynamics*, *encapsulation*, *separation of concerns*, (and of course *generativeness*) as listed in Sect. 2. The need for *hierarchical abstractions* directly implies the presence of multiple object spaces. In the following, we will therefore investigate how far existing coordination models can cover the given requirements. We focus on two properties: (1) The object model of a system which determines how object matching can be performed, and (2) the introduction of multiple data spaces.

4.1 Shared Object Spaces

In the original Linda model [12] by which generative communication was introduced, the shared data space contains tuples of basic data types like numbers and character strings (and is hence called *tuple space*). The most important feature of Linda is the associative way of consuming tuples from the tuple space. This is done by providing a template which *matches* certain tuples. The template itself has a tuple structure and hence determines arity, types of the elements and optionally constant values for the elements of a matching tuple. Elements specified by type *and* value are called *actuals* whereas typed placeholders (without a value) are called *formals*.

The obvious similarity between tuples and the record structure of object implementations has led to early adaptations of the Linda model to object–oriented languages [17, 23]. In these systems, language–level objects are stored in shared data spaces and are retrieved using template objects which specify actuals and formals. This straightforward adaptation allows to use the Linda model from within object–oriented languages. But because object matching is based on the objects' implementation, the encapsulation property which is essential for the notion of objects is ignored in these systems.

The work in [28] attempts to overcome this deficiency by letting programmers specify so–called *important sets* denoting those parts of object implementations

relevant to object matching. Although this approach abstracts from some irrelevant parts of implementations, matching is still based on it. Similarly, Bauhaus Linda [8] which is based on multisets as basic data structures, performs matching on multiset inclusion which also embodies the idea of matching based on relevant parts of the data items stored in the shared data space.

None of these models treats objects as completely encapsulated entities, hence they are not fully suited for open systems.

4.2 Multiple Data Spaces

The requirement of *hierarchical abstractions* (from Sect. 2) directly leads to the introduction of multiple data spaces arranged in nested hierarchies. Each data space can hence be treated as an agent as well as a configuration. Here, we discuss data spaces instead of object spaces because interesting approaches can also be found in models which are not based on objects.

Although there have been several proposals for introducing multiple data spaces to the Linda model, neither of them found broader acceptance. This might be due to the fact that generative communication with multiple data spaces implies a paradigmatical change. It can be seen to be in between the two extremes, denoted by message passing and the single–spaced Linda model [20]. In message–passing systems, senders have to know the name or address of the receiver. Hence, messages can only be sent to agents known by the sender. In Linda, producers of data entities only have to access the shared data space. It is up to the consumer to know which kinds of data entities are available. In systems with multiple data spaces, both variants must be combined: a producer of data items has to know the data spaces in order to put something inside whereas the consumer still has to know what kind of data is available. This setting degrades anonymity of communication and is most likely to be the main obstacle concerning general consensus about a suitable model with multiple data spaces.

One possible approach for introducing multiple data spaces is presented in [23]. Here, object spaces are simply first–class objects. They must hence be known by producers and consumers. This approach comes close to message passing and immediately leads to problems with dangling references if applied to open systems.

As an alternative to object–space references, the work in [13] uses a global naming scheme for tuple spaces (e.g. "*/root/sub1/sub2*") resembling Unix directory structures. This approach avoids dangling references but introduces problems with the global naming scheme in case of open systems in which name conflicts soon may arise. Additionally, problems may arise when some processes change the tuple space hierarchy (e.g. remove intermediate spaces like *sub1*) which may lead to invalid paths in use by other processes.

The work in [18] also introduces a global hierarchy of nested tuple spaces but refuses the global, static naming scheme. Instead, every active object knows tuple spaces by names which omit the hierarchy information (like *my_space* and *your_space*) and especially a relative name *context* denoting the next "outer" tuple space. This helps avoiding problems of [13] with the static global space hierarchy. Unfortunately, names for tuple spaces are still globally visible and hence may lead to naming conflicts, especially in open systems.

5 A Coordination Model for Open Systems

We will now introduce our coordination model Objective Linda which has been designed in order to be suited for open systems. We start with its language–independent object model, present its set of operations on object spaces which directly reflects openness, and complete by outlining how multiple object spaces can be handled cleanly in open systems.

5.1 Objective Linda's Object Model

Since the goal is to model open systems, a language–independent object model is necessary. In Objective Linda, objects to be stored in object spaces are self–contained entities; their interface operations only affect the encapsulated object state. The objects are instances of abstract data types which are described in a language–independent notation, called *Object Interchange Language* (OIL). Actual programs may hence be written in conventional object–oriented languages to which a binding of the OIL types (e.g. to language–level classes) can be declared. In OIL, all types have a common ancestor called *OIL_object* which defines the basic operations needed by all types. Types needed in application programs are then derived from *OIL_object* or its descendants. OIL allows subtyping such that an object of type S which is a subtype of T can be used whenever an object of T is expected. More precisely, OIL types are not related in a subtype relation but follow the *matching relation* [5] which slightly weakens the strong subtyping notion in order to allow operations which take parameters of *the–same* type (e.g. the famous *is_equal* function [9]). In order to avoid confusion with object matching in object spaces, we still call them "subtypes". OIL allows single as well as multiple subtyping and parameterized types. It is subject to the actual language binding to map these mechanisms to available language constructs. Because abstract data types can conceptually be used even in programming languages without objects at all (e.g. *C* or *Pascal*), it only depends on the programming language in use how simple and elegant OIL types and their relations are mapped onto the language.

In mixed–language environments of open distributed systems it is of course necessary to identify identical types across multiple language bindings. Naming schemes can in general not avoid name conflicts and consequently unintended name matches or mismatches [30]. Therefore, OIL types are identified by globally unique identifications making use of OSF DCE's [27] *Universally Unique ID's* (UUID's) which can easily be created from a host identification and a timestamp. Besides the identification of identical types it is of course also necessary to transport objects between agents operating in heterogeneous environments. For this purpose, techniques known as *object externalization* or *object imaging* [29] can be used. However, this issue belongs to the technical realization of the Objective Linda model and is hence beyond the scope of this paper.

Object Matching in Objective Linda. Objective Linda's object model treats objects as encapsulated entities which can only be accessed via their interface routines defined by the corresponding type. Hence, operations belonging to the coordination model must be based on the type interfaces, too. This fact is not

only a restriction but also a major improvement with respect to other models: objects are treated based on their specification, rather than their implementation, which allows to completely abstract from implementation (data representation) details.

Consequently, object matching in Objective Linda is based on object types and the predicates defined by type interfaces. A potential reader has to specify the type of object it wishes to obtain from an object space and additionally a predicate (from the interface of the type) which selects the objects of a given type matching the specific request. Because OIL's subtype relations provide types which can be used as replacements for their supertypes, object matching will also consider objects of subtypes of the requested type.

As a toy example, consider two types: *linked_list* which has a subtype *double_linked_list* where type *linked_list* provides a predicate *nbr_items* giving the number of items in the list. A read request for an object of type *linked_list* for which the predicate *nbr_items = 3* holds will match any object of type *linked_list* containing three items which might in fact be a *double_linked_list*.

This kind of object matching ideally requires to have types and predicates as first–class entities in the programming language in use. Unfortunately, this is typically not the case with existing languages. Since the goal of this work is to model open systems which integrate existing tools and languages, it is not feasible to simply design yet another language and express the coordination model within it. Instead, the matching mechanism based on types and predicates must be expressed in existing languages in which only objects are available as first–class entities and hence as parameters to a matching operation.

Denoting the type of an object to be read from an object space is relatively simple because by passing an object as a parameter to a routine, its type can be taken as the desired one. Passing a predicate is a bit more difficult. One approach is to use so–called *function objects* [21] which realize specific predicates by implementing routines with predefined names (e.g. *eval*) operating on the types they have been tailored for.

Alternatively, the matching predicates can be directly integrated into the types on which they operate. Therefore, the type *OIL_object* provides a predicate *match* which takes an object of the same type as parameter and returns a boolean value deciding whether a given object matches certain requirements. This approach statically encodes object matching into the types themselves. Several variants of matching a type can be selected by presetting the encapsulated state of the object provided to a matching operation, which we call a *template object* in the following. In cases where a high degree of flexibility in matching predicates is needed, it is still possible to have specially tailored function objects as part of the template object's state. By this approach, matching flexibility is combined with a minimal amount of object types needed, because function objects are only introduced where necessary.

We illustrate this approach by a simple type taken from a real–world example. Figure 1 shows the code of a type *BARRIER_SYNC* which realizes objects being exchanged in a parallel computation between worker processes and a barrier for purposes of synchronization which is achieved by presence or absence of *BARRIER_SYNC* objects. In order to allow multiple barriers in an object space, it is necessary that *BARRIER_SYNC* objects contain an identification of the barrier object in charge. Barrier objects are active and consume *BARRIER_SYNC* ob-

jects using template objects containing their own identification. So, the *match* predicate checks for identical values of *barrier_id* in order to match the right objects. In the example, we use a binding to the Eiffel programming language.

```
class BARRIER_SYNC  inherit OIL_OBJECT redefine match end
creation create
feature barrier_id : STRING;
        create ( id : STRING ) is
           do  barrier_id := id;  end;
        match ( candidate : like current ) : BOOLEAN is
           do  Result := candidate.barrier_id = barrier_id;  end;
end -- class BARRIER_SYNC
```

Fig. 1. Complete code of an Eiffel class used for barrier synchronization

Evaluating Active Objects. According to Linda's *eval* operation, we will call the activity of an agent the *evaluation* of an active object. In favour of a homogeneous model, passive as well as active objects are characterized by their OIL type. The mechanism used to specify this activity is similar to object matching: the type *OIL_object* provides an operation called *evaluate* whose behaviour is refined by every type the objects of which will become active. As with *match*, behaviour can be individually parameterized by the object's state before it is evaluated.

Since the specification of abstract data types focusses on the objects of given types themselves rather than on protocols of interaction between them, its expressiveness is quite limited with respect to specifying behaviour of active objects. Methodologies for such specifications are subject to ongoing research. There already exist promising approaches for specifying behaviour based on message exchange in client/server like settings [24]. Unfortunately, behaviour specification of active objects based on generative communication is still unexplored. Hence, the expressiveness of Objective Linda's type specifications is limited in this respect.

5.2 Operations on Object Spaces

Besides the adaptation of the Linda model to object orientation, we also have to consider the operations on object spaces with respect to their suitability to open systems. The operations in the original Linda model have been designed without consideration of openness. As a consequence, the blocking operations for putting an object into an object space (*out*), for consuming an object (*in*), reading an object (*rd*), and activating a new active object (*eval*) assume unrestricted access to the data space and may hence block infinitely in case of object spaces in open systems (which are realized by several independent systems). Here, access to an object space may fail due to temporarily disconnected operation of a mobile host, transmission (line) errors, or missing access permissions.

Furthermore, the semantics of the non–blocking versions of *in* and *rd* (*inp* and *rdp*) imply access to a data space as a whole: these operations are defined to immediately return indicating a failure when there is no object matching a

given request. This immediately introduces semantical problems in the presence of distributed or even open–implemented object spaces where parts of the object space simply may not be (temporarily) accessible. Hence, the semantics of such operations must be slightly modified for open systems: operation failure of *inp* and *rdp* should indicate "no such object could be found (in the moment)". This change reflects the fact that synchronization based on the absence of a certain object is impossible in open and hence possibly only partially available object spaces.

Infinitely blocking operations due to disconnected operation or missing access permissions is by no means a suitable behaviour. Instead, it is necessary to dynamically adapt the behaviour of an agent to the properties of its environment. This can preferably be done by introducing a *timeout* value which determines how long an operation should block before a failure will be reported. By adjusting this parameter, an agent can easily adapt its communication behaviour. A value indicating infinite delay leads to a blocking operation and can be used for object spaces which are known to behave like closed systems, e.g. object spaces which are local to the agent itself. A zero value yields a behaviour as *inp* and *rdp* with semantics as outlined above. All values in between can be used to adapt to different communication delays.

The *eval* operation for activating new agents also needs adaptation to open systems. In Linda, all elements of a tuple given as a parameter to *eval* are evaluated in parallel by new processes and each yields a single return value. After termination of all these processes, the tuple is converted into a passive one containing the processes' results. This operation views active processes as functions instead of encapsulated and reactive agents as they should be in a coordination model for open systems. As a consequence, the *eval* operation should get objects to be activated which are (like in Linda) invisible to *in* and *rd* operations and which simply disappear after termination. Hence, the behaviour of agents can only be observed by monitoring the passive objects produced and consumed by them.

Linda's ability to retrieve only one object at a time from an object space is simple and elegant, but unfortunately too restrictive. Just with these operations, it is, for example, impossible to implement the functionality of the trading function from RM–ODP which is able to provide a list of servers for a client's request. This is due to the fact that the semantics of the *rd* operation can not specify which object will be returned by several subsequent invocations; it might be e.g. the same object all the time. This impossibility is a consequence of uncoupled communication which is as such vital for open systems. Specifying any relation between the results of subsequent calls to *rd* would couple them. This would be similar to establishing a connection (or "session") between an object space and a requesting agent. Instead, an operation atomically retrieving several objects is necessary in order to cleanly introduce this functionality into the generative communication model.

Approaches to support reading of multiple objects have been reported in the literature. One approach, used in [13] and [18], treats tuple spaces as first–class entities and allows to produce snapshots of tuple spaces which can separately be investigated. Because these proposals can not selectively extract multiple objects, they are hardly applicable to larger systems in which snapshot sizes might soon become prohibitive apart from the fact that taking a snapshot of

an open–implemented tuple space is close to impossible. Another approach is presented in [6] and introduces a *collect* operation which atomically returns all tuples matching a given template in a certain tuple space. This approach allows to select multiple objects to be consumed, but in the case of a RM–ODP trader, unrestrictedly returning the complete list of service providers might still be too much (e.g. with respect to memory size) or at least too inefficient.

There is also a demand for an *in* operation which atomically removes several objects from an object space. Applications for such an operation come from the field of synchronization problems. One example is atomic allocation of more than one resource at a time which can help avoiding deadlock situations. On the other hand, it is not necessary to put several objects atomically into an object space. After successful completion, a single *out* operation providing several objects can not be distinguished from a sequence of *out* operations providing one object each; the same holds for *eval*. Nevertheless, our *out* and *eval* operations take multiple objects to be stored and possibly activated. We introduce them in order to achieve a small and consistent set of powerful operations. Technically, Objective Linda's operations use objects of the type *Multiset* as a parameter for the objects to be stored (by *out* and *eval*) and as result type for the objects being matched (by *in* and *rd*). Here, a *Multiset* is a simple container type with the operations *put* and *get* and the predicate *nbr_items* denoting the number of items stored inside.

We can now introduce the set of operations on object spaces supported by Objective Linda. They are based on the following three design decisions. (1) Because objects in Objective Linda are self–contained entities and consequently contain no references to objects outside their encapsulated state, there is absolutely no sharing of data between them. Hence, the operations *out*, *eval* and *in* move objects between agents and objects spaces whereas *rd* returns clones of matching objects. (2) All operations dealing with multiple objects operate indivisibly: in case of failures the state of the object space in charge remains unchanged. (3) For reasons of "intellectual compatibility", we borrow the meanings of *in* and *out* from Linda which reflects the view of an agent operating on an object space; from the viewpoint of an abstract data type *object space*, the names of the *in* and *out* operations should have actually been exchanged. In the following, we will again use the syntax of an Eiffel language binding.

out (m : MULTISET ; timeout : REAL) : BOOLEAN
Tries to move the objects contained in m into the object space. Returns *true* if the operation could be completed successfully; returns *false* if the operation could not be completed within *timeout* seconds.

in (o : OIL_OBJECT ; min, max : INTEGER ; timeout : REAL) : MULTISET
Tries to remove multiple objects $o'_1 \ldots o'_n$ matching the template object o from the object space and returns a multiset containing them if at least *min* matching objects could be found within *timeout* seconds. In this case, the multiset contains at most *max* objects, even if the object space contained more. If *min* matching objects could not be found within *timeout* seconds, *Result.Void* is true.

rd (o : OIL_OBJECT ; min, max : INTEGER ; timeout : REAL) : MULTISET
Tries to return clones of multiple objects $o'_1 \ldots o'_n$ matching the template ob-

ject o and returns a multiset containing them if at least *min* matching objects could be found within *timeout* seconds. In this case, the multiset contains at most *max* objects, even if the object space contained more. If *min* matching objects could not be found within *timeout* seconds, *Result.Void* is true.

eval (m : MULTISET ; timeout : REAL) : BOOLEAN
Tries to move the objects contained in m into the object space and starts their activities. Returns *true* if the operation could be completed successfully; returns *false* if the operation could not be completed within *timeout* seconds.

infinite_matches : INTEGER
Returns a constant value which will be interpreted as infinite number of matching objects when provided as *min* or *max* parameter to *in* and *rd*.

infinite_time : REAL
Returns a constant value which will be interpreted as infinite delay when provided as *timeout* parameter to *out*, *in*, *rd*, and *eval*.

This set of operations is the minimal one needed for operations in open systems. At the same time, it is powerful enough to express original Linda's operations as well as to reflect all requirements of open systems listed above. Objective Linda provides a minimal set of powerful operations in order to keep the model as simple as possible; but nevertheless it is still possible to add convenience operations (e.g. for frequently used special cases) to an OIL language binding or directly to application programs in order to simplify the programmer's task. Table 1 shows how the behaviour of Objective Linda's operations can be adapted by their parameters in order to match different requirements.

	min	*max*	*timeout*	behaviour
out			0	immediately fail on errors
			t	wait t sec. before failing on errors
			infinite_time	Linda's *out*
in	0	0	any	empty operation
	0	1	0	Linda's *inp*
	1	1	*infinite_time*	Linda's *in*
	1	n	any	consume up to n matching obj.
	1	*infinite_matches*	any	consume all matching objects
	infinite_matches	any	t	sleep t seconds
rd	0	0	any	empty operation
	0	1	0	Linda's *rdp*
	1	1	*infinite_time*	Linda's *rd*
	1	n	any	read up to n matching objects
	1	*infinite_matches*	any	read all matching objects
	infinite_matches	any	t	sleep t seconds
eval			0	immediately fail on errors
			t	wait t sec. before failing on errors
			infinite_time	Linda's *eval*

Table 1. Behaviour of Objective Linda's operations on object spaces

5.3 Multiple Object Spaces in Objective Linda

Configurations in Objective Linda consist of two kinds of objects, namely object spaces and OIL objects, the latter may be active as well as passive ones. Active objects have, from the moment of their activation on, access to two object spaces: (1) their *context* which is the object space on which the corresponding *eval* operation has been performed, and (2) a newly created object space called *self* which is directly associated to the object. With this basic mechanism, hierarchies of nested object spaces can be built providing hierarchical abstractions for sub–configurations. Before we explain how objects can get attached to more than these two object spaces, we illustrate this hierarchy and its basic applications in Fig. 2(a). Here, rounded boxes denote object spaces, boxes show passive objects, circles denote active ones. The *self* object spaces are shown close to the corresponding active objects. The *context* object spaces are the additional object spaces to which arrows are drawn.

In this abstracted example, 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 which are then performed without further 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.

The restriction to exactly the *context* and *self* object spaces is not powerful enough in order to generally express coordination problems. There are four problems involved: (1) without implementing specialized routing agents, two agents with different *context* spaces could never communicate, (2) new agents would not be able to enter a given object space which contradicts openness, (3) mobile agents in open distributed systems could not move to other communication contexts, and (4) it should be possible to have more than one object space attached to an object in order to further structure computations.

The latter point can be handled by locally created object spaces which are then only known to their attached agents unless they make them available to others. But the problem of attaching to already existing object spaces needs further elaboration. This is due to the fact that object spaces are not part of agents but are accessed by references. This is necessary because object spaces must by their very nature be shared between agents. Consequently, object spaces must not be stored in object spaces; *Object_Space* cannot be a subtype of *OIL_object*.

Because object spaces must be accessed by reference, we have to minimize possible impacts of dangling references due to configuration changes in open systems. This is the reason for not introducing references to object spaces as subtypes of *OIL_objects*. Furthermore, it would hardly make sense to transfer language–level references between agents in separate address spaces, which is the typical case in open (distributed) systems.

This makes it necessary to introduce a construct which allows agents to attach to existing object spaces without using low–level references and which is based on the generative communication mechanism. Objective Linda therefore introduces a special subtype of *OIL_object* which is called *object space logical*. *Logicals* combine a reference to an object space with a logical identification such

that an object space can be found by matching properties of *logical* objects. These properties can of course be specialized by subtyping to application needs, like numerical values, keywords, network addresses, geographical locations, or even "Uniform Resource Locators" (URLs) known from the World–Wide–Web.

Agents willing to let others attach to object spaces they are already attached to simply create a *logical* object including the reference to the object space to be made available which also contains a convenient logical identification for that object space. This *logical* is then *out*'ed to an object space. There, other agents may *rd* this *logical* like any other object; but the reference contained in it is useless for the agent in most of the cases (with different address spaces). Instead, an agent a willing to attach to object space n must call a special operation called *attach* on the object space o in which the corresponding *logical* object for n is stored. This operation has two effects: (1) o verifies that n can be attached to (is reachable, allows attachment, etc.), and (2) returns a reference to n which is locally useful to a. The *attach* operation can be introduced as follows:

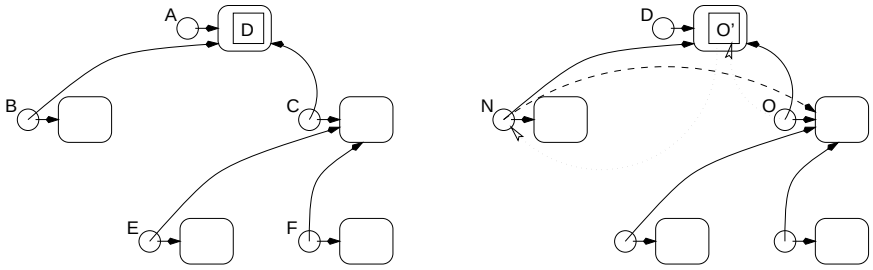
attach (o : OS_LOGICAL ; timeout : REAL) : OBJECT_SPACE
Tries to get attached to an object space for which an *OS_LOGICAL* matching o can be found in the current object space. Returns a valid reference to the newly attached object space, if a matching object space logical could be found within *timeout* seconds; otherwise *Result.Void* is true.

As we have seen so far, configurations in Objective Linda consist of hierarchies of object spaces. Because attachment to further object spaces can only be accomplished using information available inside a configuration, all object spaces form a connected graph with a common root. Navigation in a configuration is based on information relative to other object spaces which is accessed via logical identifications. Subtyping applied to *logical* objects allows several identification mechanisms for object spaces to coexist and hence diminishes the need for global naming schemes.

The attachment of new agents to running configurations can also be performed using the mechanism introduced so far. Therefore, every agent has by default a *context* space by which it may attach to further object spaces. These default object spaces must of course be valid object spaces of a given configuration. The provision of such default object spaces is implementation dependent and may be based on object–space servers in the local network of a site. A typical example might be the entry point to an information system; roughly equivalent to a home page in the WWW.

Figure 2(b) illustrates this. Here, agent O has put a *logical* O' for its *self* space into D which locally serves as default context. Then, a new agent N performs an *attach* operation on D and gets access to O's space, indicated by the dashed line.

With the addition of *logicals* and the *attach* operation, the coordination model for open systems is complete. To summarize, agents can get access in the following three ways: (1) attachment to *context* and *self* on creation, (2) creation of new locally associated object spaces, and (3) explicit execution of *attach* to already existing object spaces.



(a) Some nested configurations (b) Agent N attaching config. O

Fig. 2. Hierarchical abstractions for configurations

6 Example: The Restaurant of Dining Philosophers

We will now illustrate Objective Linda's capabilities for expressing coordination problems in open systems. Therefore, we present a solution to the problem of *The Restaurant of Dining Philosophers* taken from [10]. This is an extension of Dijkstra's classical problem to open systems. In the restaurant, there is the table with n seats around. Between each two seats, there is exactly one chopstick on the table. Philosophers sitting down need two chopsticks (the ones left and right to their seat) in order to eat rice from the bowl on the table. So far, this is the classical synchronization problem. Additionally, philosophers can enter the restaurant, wait for a free seat, and leave the table after eating.

We extend the problem a bit and come to the following modelling: there are two kinds of agents in the system: a waiter who runs the restaurant with the name *The Philo and the Fork* and philosophers who come in and eat. Arriving philosophers try to enter the restaurant (*attach* to its object space), wait for a free seat and then try to atomically grab the two chopsticks next to their seat. After eating, they put the (now dirty) chopsticks back on the table, stand up, and leave the restaurant. The waiter has to do all the work in the restaurant; he starts his job by opening the door (putting a *logical* for his restaurant into his *context* space which is of a special subtype of *OS_LOGICAL* denoting restaurants). Then he sets up the table by putting chairs and chopsticks into his *self* object space. Until it is time to close the restaurant, the waiter looks for dirty chopsticks which he cleans and puts back on the table. In order not to forget closing time in case there are no philosophers producing dirty sticks, the *in* operation for the dirty sticks waits at most five minutes before the waiter can check again the wall clock. When closing time has arrived, the waiter closes the door (consuming the *logical* from the *context*), so that newly arriving philosophers can not get in any more. Then, the waiter waits until the last philosopher has gone (by waiting for n free seats) and can finally go home (terminate).

Below, we show our solution consisting of four Eiffel classes, namely *WAITER*, *PHILO*, *CHOPSTICK*, and *SEAT*. As it can be seen, Objective Linda provides a rather natural modelling to the problem. Philosophers simply have to know

the name of the restaurant in order to get in. They then take any free seat and the two chopsticks next to the seat without having to know anything about seats, sticks, or even the number of them which are available. Also, there is no need to introduce artificial synchronization objects besides the seats and the chopsticks themselves. The waiter is the only one who knows how many seats (and sticks) are at the table; he can open and close the restaurant whenever he wants. The passive objects in our example are of types *SEAT* and *CHOPSTICK*. Seats simply contain a numerical identification and inherit the *match* routine from *OIL_OBJECT* which by default matches any object of the type. Chopsticks have more features. They may be clean or dirty, and they have a position between the seats on their left and right side. Furthermore, *myseat* is used to parameterize *match*. This *match* routine is also an example how matching can be parameterized by simple state changes of the template object before it is used for matching.

```
class PHILO  inherit OIL_OBJECT redefine evaluate end
creation make
feature evaluate is
   local r     : OBJECT_SPACE;
         l     : RESTAURANT_LOGICAL;
         s     : SEAT;
         m     : MULTISET;
         c1,c2 : CHOPSTICK;
         ok    : BOOLEAN;
   do !!l.create("the philo and the fork");
      r := context.attach(l,600);             -- wait at most 10 minutes for
                                              -- restaurant opening
      if not r.Void then                      -- entered the restaurant
         !!s.make;
         m := r.in(s,1,1,r.infinite_time);    -- wait for a free seat
         s := m.get;
         !!c1.make; c1.set_myseat(s.number);
         m := r.in(c1,2,2,r.infinite_time);   -- wait for 2 chopsticks
         c1 := m.get; c2 := m.get;
         eat;
         c1.mark_dirty; c2.mark_dirty;        -- sticks are dirty after eating
         m.put(c1); m.put(c2); m.put(s);      -- put dirty sticks back on table
         ok := r.out(m,0);                    -- and stand up
      end
   end                                        -- go home
end -- class PHILO

class CHOPSTICK  inherit OIL_OBJECT redefine match, make end
creation make
feature is_dirty, match_if_dirty    : BOOLEAN;
        leftseat, rightseat, myseat : INTEGER;
        make is do is_dirty := false; match_if_dirty := false; end;
        mark_clean is do is_dirty := false; end;
        mark_dirty is do is_dirty := true; end;
        set_leftseat ( s : INTEGER ) is do leftseat := s; end;
        set_rightseat ( s : INTEGER ) is do rightseat := s; end;
        set_myseat ( s : INTEGER ) is do myseat := s; end;
        set_match_dirty is do match_if_dirty := true; end;
        match ( candidate : like current ) : BOOLEAN is
           do if match_if_dirty
                 then Result := candidate.is_dirty;
                 else Result := not candidate.is_dirty and
                                ( myseat = candidate.leftseat or
                                  myseat = candidate.rightseat );
              end;
        end;
end -- class CHOPSTICK
```

```
class SEAT  inherit OIL_OBJECT
creation make, create
feature number : INTEGER;
        create ( n : INTEGER ) is do number := n; end;
end -- class SEAT

class WAITER  inherit OIL_OBJECT redefine evaluate end
creation make
feature evaluate is
   local l  : RESTAURANT_LOGICAL;
         i  : INTEGER;
         n  : INTEGER is 42;                        -- nbr of seats at the table
         s  : SEAT;
         c  : CHOPSTICK;
         m  : MULTISET;
         ok : BOOLEAN;
   do !!l.create("the philo and the fork");
      l.set_space(self); !!m.make; m.put(l);
      if context.out(m,0) then                      -- managed to open the door
         !!m.make;
         from i := 1; until i > n loop
            !!s.create(i);m.put(s);                 -- get seats
            !!c.make; c.set_leftseat(i);            -- get sticks
            if i < n then c.set_rightseat(i+1);
                     else c.set_rightseat(1); end;
            m.put(c);
         end;
      ok := self.out(m,0);                          -- put seats and sticks to table
      from until closing_time                       -- serve philosophers:
         loop clean_sticks; end;
      !!l.create("the philo and the fork");
      m := context.in(l,1,1,                        -- close the door
                      context.infinite_time);
      clean_sticks;                                 -- in case a philo dropped in
      !!s.create;                                   -- wait until the last guest
      m := self.in(s,n,n,self.infinite_time);       -- has gone
   end
end                                                 -- go home
clean_sticks is
local c    : CHOPSTICK;
      m,m2 : MULTISET;
do !!c.make;                                        -- replace dirty sticks by
   c.set_match_dirty;                               -- clean ones
   m := self.in(c,1,n,300);                         -- sleep at most 5 minutes
                                                    -- before rechecking wall clock
                                                    -- if it's closing time
   if not m.Void then
      from !!m2.create;                             -- clean sticks if we found some
      until m.nbr_items = 0 loop
         c := m.get;
         c.mark_clean;
         m2.put(c);                                 -- collect cleaned sticks in m2
      end;
      if m2.nbr_items > 0
         then ok := self.out(m2,0); end             -- put them back on the table
   end
end
end -- class WAITER
```

7 Conclusion

The example of the restaurant of dining philosophers illustrated how well Objective Linda is suited to model coordination problems in open systems. This is primarily due to Objective Linda's three main contributions: (1) its object model which allows to build open systems from heterogeneous components which are

modelled as encapsulated and reactive entities communicating in an uncoupled manner, (2) its set of object–space operations which directly reflect the requirements of openness, and (3) its model of multiple object spaces which on one hand provides hierarchical abstractions from configurations and on the other hand allows agents to simultaneously communicate via several object spaces.

We are currently experimenting with a prototypical class library implementing Objective Linda for clusters of workstations which is based on the PVM [11] package. First results are encouraging, so we are working on the evaluation of Objective Linda's concepts in a wider range of applications.

Acknowledgements

I am really grateful to Paolo Ciancarini, Bernd Freisleben, and Guido Wirtz for many encouragements and helpful discussions. Without them, my work would not be like it is.

References

1. Richard M. Adler. Distributed Coordination Models for Client/Server Computing. *IEEE Computer*, 28(4):14–22, 1995.
2. Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* M. I. T. Press, Cambridge, Massachusetts, 1986.
3. Gul Agha, Peter Wegner, and Akinori Yonezawa, editors. *Research Directions in Concurrent Object–Oriented Programming.* MIT Press, Cambridge, Mass., 1993.
4. J. M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction Abstract Machines. In Agha et al. [3], pages 257–280.
5. Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A Type–Safe Polymorphic Object–Oriented Language. In Olthoff [26], pages 27–51.
6. Paul Butcher, Alan Wood, and Martin Atkins. Global Synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.
7. Christian J. Callsen and Gul Agha. Open Heterogeneous Computing in ActorSpace. *Journal of Parallel and Distributed Computing*, 21:289–300, 1994.
8. Nicholas Carriero, David Gelernter, and Lenore Zuck. Bauhaus Linda. In P. Ciancarini, O. Nierstrasz, and A. Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, number 924 in Lecture Notes in Computer Science. Springer, 1995.
9. Giuseppe Castagna. Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.
10. Paolo Ciancarini. Coordination Languages for Open System Design. In *Proc. of IEEE Intern. Conference on Computer Languages*, New Orleans, 1990.
11. G. A. Geist, A. L. Beguelin, J. J. Dongarra, W. Jiang, R. J. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.
12. David Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
13. 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.

14. David Gelernter and Nicholas Carriero. Coordination Languages and their Significance. *Communications of the ACM*, 35(2):96–107, 1992.
15. 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.
16. 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.
17. Robert Jellinghaus. Eiffel Linda: an Object–Oriented Linda Dialect. *SIGPLAN Notices*, 25(12):70–84, 1990.
18. 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.
19. 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.
20. Oliver Krone and Marc Aguilar. Bridging the Gap: A Generic Distributed Coordination Model for Massively Parallel Systems. In *Proc. of SIPAR Workshop on Parallel and Distributed Systems*, pages 109–112, Biel–Bienne, Switzerland, 1995.
21. Thomas Kühne. Parameterization versus Inheritance. In Christine Mingins and Bertrand Meyer, editors, *Technology of Object-Oriented Languages and Systems: TOOLS 15*, pages 235–245, Melbourne, Australia, 1994. Prentice Hall.
22. Ted G. Lewis. Where is Client/Server Software Headed? *IEEE Computer*, 28(4):49–55, 1995.
23. Satoshi Matsuoka and Satoru Kawai. Using Tuple Space Communication in Distributed Object-Oriented Languages. In *ACM Conference Proceedings, Object Oriented Programming Systems, Languages and Applications, San Diego California*, pages 276–284, 1988.
24. Oscar Nierstrasz. Regular Types for Active Objects. In O. Nierstrasz and D. Tsichritzis, editors, *Object-Oriented Software Composition*, chapter 4, pages 99–121. Prentice Hall, 1995.
25. Object Management Group. The Common Object Request Broker: Architecture and Specification. OMG Document Number 93.12.43, 1993.
26. Walter Olthoff, editor. *Proc. ECOOP'95*, number 952 in Lecture Notes in Computer Science, Århus, Denmark, 1995. Springer.
27. Open Software Foundation. Introduction to OSF DCE. Open Software Foundation, Cambridge, USA, 1992.
28. 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.
29. Satish R. Thatté. Object Imaging. In Olthoff [26], pages 52–76.
30. Robert Tolksdorf. *Coordination in Open Distributed Systems.* PhD dissertation, Technical University of Berlin, Berlin, Germany, 1994.
31. Peter Wegner. Tradeoffs between Reasoning and Modeling. In Agha et al. [3], pages 22–41.
32. Peter Wegner. Interactive Foundations of Object–Based Programming. *IEEE Computer*, 28(10), 1995.

This article was processed using the LaTeX macro package with LLNCS style