Object-Oriented Parallel Programming with Objective Linda

Bernd Freisleben

Department of Electrical Engineering and Computer Science,

University of Siegen, Germany

Thilo Kielmann

Department of Mathematics and Computer Science,

Vrije Universiteit Amsterdam, The Netherlands

Running head:

Object-Oriented Parallel Programming with Objective Linda

Contact author:

Prof. Dr. Bernd Freisleben

Dept. of Electrical Engineering and Computer Science

University of Siegen

Hölderlinstr. 3

57068 Siegen

Germany

Phone: +49 271 740 3268

Email: freisleb@informatik.uni-siegen.de

Abstract

In this paper we present *Objective Linda*, a coordination model in which object-orientation is combined with uncoupled, generative communication in order to enable object-oriented parallel programming in networked computing resources. *Objective Linda* provides suitable abstractions for structuring large software systems, supports interoperability between different programming languages and parallel architectures and simplifies the development of parallel applications. Its use is illustrated by presenting programming examples, a prototype implementation is described, and measurements for evaluating the implementation efficiency and the performance of parallel applications are presented.

1 Introduction

The object-oriented programming paradigm [45] is commonly accepted as a powerful approach to master the complexity of sequential software development. Not surprisingly, in recent years several proposals have been made to utilize its many benefits in terms of software modularization, encapsulation and reusability in parallel programming, too [2, 19, 36]. The central notions of the traditional object model are objects, classes and inheritance: objects as the basic building blocks are defined as abstract data types which encapsulate their internal state through well-defined interfaces [45] and thus simply represent passive data containers, while classes are templates for representing sets of objects with similar functionality, and inheritance is a mechanism for incrementally introducing new functionality and supporting software reuse.

One possible approach to introduce parallelism into 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 [2]. In this way, the three basic programming paradigms used to exchange information between interconnected computing resources can be realized: *message passing* is achieved by giving the method invocations the semantics of messages being exchanged [15], *remote procedure calls* are modeled by treating method invocations as procedure calls and allowing objects being called to be "remotely" [1], and *(distributed) shared memory* programming is possible by adding monitor semantics to the objects in order to enable synchronization on the basis of the methods of the objects involved [8]. In any case, the objects are responsible for keeping their own states consistent, such that the operations concurrently executing on a given object must be somehow synchronized, which in turn leads to interferences between concurrency control mechanisms and notions of reuse based on inheritance. This problem, known as the *inheritance anomaly* [44], is the main obstacle prohibiting the widespread use of this style of object–oriented concurrent programming.

In our work, we follow the more promising approach of exploiting object–oriented technology in parallel systems by (a) keeping the idea of encapsulated entities as suitable

means for composing and refining predesigned plug–compatible software components [48] and (b) by unifying the notions of (passive) objects and processes by introducing *active objects*, each with its own thread of control and still being protected by their interfaces. This approach eliminates the consistency problems mentioned above, because there is exactly one thread operating in an active object. Of course, this is only true as long as the *coordination model* in charge of the communication between active objects is properly designed and does not introduce new consistency problems. Consequently, our work focusses on a well designed coordination model in which parallel computations are expressed by means of interacting active objects.

In general, a coordination model provides a framework in which the interaction of individual, active entities (called *agents*) can be expressed. In this framework, all aspects of the creation and destruction of agents, the communication among agents, their spatial distribution, as well as the synchronization and distribution of actions performed by agents over time are subsumed. Coordination models consist of the following components [16]: *coordination entities* (the agents which are to be coordinated), *coordination media* (the means enabling communication between agents), and *coordination laws* (the rules of how agents are coordinated making use of the given coordination media).

A natural way to realize a coordination model is to embody it in a *coordination language* [29], which ideally should be designed as an orthogonal combination of a coordination model (for the inter–agent actions) and a (sequential) computation model (for the intra–agent actions). The presumably most famous example of a coordination model is the *Linda* tuple space approach based on uncoupled generative communication [27], for which several linguistic embodiments like C–Linda or FORTRAN–Linda were developed, both for workstation networks and massively parallel architectures. Another prominent example are the *interaction abstract machines* [3] which have been realized with implementations of the *linear objects* language *LO* [4].

In this paper we present a coordination model called *Objective Linda*, which has been designed for the needs of object-oriented parallel programming. It rigorously combines

object orientation with the general Linda philosophy of uncoupled generative communication. Objective Linda has several features that make it unique among alternative proposals based on extensions of Linda: it has a language–independent object model to enable its use in conjunction with any existing object-oriented programming language, it provides an enhanced set of operations to express complex coordination requirements in a simple manner, and it supports hierarchical abstractions for structured application development. The paper describes Objective Linda's functionality and illustrates its use by programming examples. A prototypical implementation for the C++ language to be used on networks of workstations is presented, and the performance of applications written using Objective Linda is evaluated by making a comparison to equivalent realizations based on the PVM message passing library [25]. It will be shown that the runtimes of the Objective Linda programs are comparable to (or even better than) their PVM counterparts.

The paper is organized as follows. In Section 2, several related coordination models based on uncoupled generative communication are discussed. Section 3 presents the basic properties of Objective Linda as a suitable coordination model for object-oriented parallel programming. In Section 4, examples are given to illustrate the benefits of using Objective Linda for writing parallel applications. Section 5 describes our prototype implementation of Objective Linda. In Section 6, the efficiency of the implementation is measured and the runtime performance of two parallel applications is evaluated. Section 7 concludes the paper and outlines areas for future research.

2 Coordination Models

Well designed coordination models must allow to be orthogonally integrated into existing sequential programming languages without them being forced to modify their functionality [29]. In this way, it is possible to use the available sequential software technology, to minimize the amount of new programming constructs programmers have to learn for developing parallel applications, and to avoid the efforts required for designing completely

new parallel programming languages. Furthermore, a coordination model should provide suitable programming abstractions which may be implemented on top of various types of parallel computing systems in order to ensure portability.

In the following, we will briefly survey existing coordination models based on uncoupled generative communication, since these form the basis of the model we propose in this paper. Apart from the original Linda model, the presentation focuses on proposals for extending and combining Linda with object orientation and other techniques for achieving modularity in software design.

Linda

The Linda coordination model has been introduced to incorporate the idea of generative communication [27]. In Linda, processes (the coordination entities) communicate by putting tuples (ordered collections 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 forcing processes to wait until a suitable tuple to be read has been inserted into the tuple space. Furthermore, new processes can be invoked by putting active tuples into the tuple space (by the `eval` operation) for subsequent evaluation. Active tuples produce results in the form of passive tuples to which they are converted upon termination of their computations.

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.

Although several systems have been developed in the spirit of this philosophy, such as *Network Linda* [5], *Piranha* [13], and *Tuplex* [12], the classical Linda model has several

drawbacks: it is not homogeneous, because different entities (active and passive tuples, templates and a tuple space) must be considered, and it neither provides hierarchical abstractions nor encapsulation, because there is only one global tuple space.

Although not widely recognized, the existence of multiple tuple spaces has been envisioned since the early days of Linda [26]. The work in [28] brought this proposal to a wider audience. Here, tuple spaces are first–class entities of a new type ts which can only be created by a new operation tsc (tuple space create). Instances of this type form a hierarchy of nested tuples spaces. Additionally, tuple spaces may be accessed by given names whereas the complete hierarchy is visible to programmers using a syntax resembling UNIX filenames (e.g. /root/sub1/sub2). Tuple spaces are also subject to in, rd, and out operations as follows: computations of active tuples in a tuple space which is consumed by in are frozen and might be continued by out–ing the tuple space back to its enclosing tuple space. Furthermore, the rd operation allows to provide snapshots of running computations. These operations provide high–level abstractions to process scheduling and migration, but especially in the case of heterogeneous systems they are very difficult to implement.

Therefore, other attempts have been made to unify the concepts and/or provide abstractions to enrich the structural abilities of Linda. These are described in the following.

C++ Linda

C++ Linda [50] is a straightforward adaptation of Linda to the C++ programming language. It implements a global tuple space and deals with tuples consisting of simple data types, arrays, structures, and very restricted pointers. The eval operation directly specifies program binaries to be executed, such that the notion of processes on the operating system level is introduced into the language. Thus, the basic Linda model is simply embedded in C++ Linda without considering object-oriented programming concepts or providing language independence.

Eiffel Linda

Eiffel Linda [37] also implements a global tuple space. But different from C++ Linda, it represents tuples and templates as objects, making them first–class entities of the language. Tuple matching is based on the data representation (i.e. the implementation) of given objects, and not on types or interface predicates. Formals are represented by void references, which in turn introduces problems with formals for the basic data types. The eval operation, although not implemented, is supposed to make use of the inheritance mechanism of Eiffel. Here, every object to be eval–uated must be represented by a special heir of a common superclass which defines a method called evaluate, executed whenever an object of class h is subject to eval. Obviously, Eiffel Linda is not language independent.

Linda and Concurrent Smalltalk

The work presented in [43] goes even further with introducing objects into the Linda model. First, it treats tuples as objects, as Eiffel Linda does. Second, it eliminates asymmetries in the original Linda communication scheme. In order to make tuple space communication symmetrical, both senders and receivers put tuples into the tuple space, and they are distinguished only by their purposes as *sender tuples* or *receiver tuples*. Both kinds of tuples mutually check each other for matches (based on actual and formal values). This scheme frees receivers from the burden to completely specify the structure of tuples they wish to receive. For total symmetry, an additional operation called set is introduced which outs a tuple and blocks the sender until the tuple has been successfully matched against a receiver tuple. Finally, tuple spaces are represented by objects, too. This allows multiple tuple spaces to be introduced into a system. Consequently, two objects can communicate via a certain tuple space if and only if the tuple space is known to both objects. As a result, tuple spaces are also first–class entities which can be created or removed at runtime. Thus, Smalltalk Linda is tied to the general Smalltalk philosophy, and it does not add any substantially improved functionality compared to the original Linda operations.

The Object Space Approach

The *Object Space* approach [51] is aimed at combining generative communication with object orientation. It provides a global object space in which C++–Objects are deposited. The implementation is based on a class library for C++ which communicates over a local network; the eval operation directly creates new UNIX processes. The Object Space approach employs a specific *Object Space Language* (OSL) in which the operations on the object space (out, in, rd, eval) and the mechanisms for matching objects are expressed. A special preprocessor transforms OSL constructs to calls of the class library. Object matching is not directly based on the data items of the objects. Instead, it is the responsibility of the programmer to specify an *important* subset of these data items to be considered in the matching process. Matching itself is based on actuals and formals as known from the classical Linda model. Formals match any value of a given type, actual values may be matched using built–in matching functions which allow matching with one or several actual values or with a range of values. This matching functionality is an approach to deliberate object matching from object implementation. However, it still appears to be limited, because there are no means enabling the use of any abstractions; matching is just based on important subsets of given implementations.

Jada

Jada [18] ("Java Linda") has been targeted for use in World–Wide–Web (WWW) applications. Jada deals with multiple tuple spaces and with tuples consisting of Java objects. In order to allow objects of user–implemented classes to be part of tuples, they have to implement a special interface called *TupleItem*. This interface requires routines for dumping objects to byte streams and vice versa, and a matching predicate.

Due to the tuple notion, Jada tuples do not provide abstractions for the matching process (like e. g. within the *Object Space approach*); completely defined templates have to be provided by consuming processes. Unfortunately, Jada provides no precise definition of how matching of *TupleItems* is performed and whether objects can be treated as

encapsulated items or just as closely related to their implementation as with the other proposals discussed so far.

Multiple Tuple Spaces

The work reported in [38] is aimed at introducing modularity to Linda by means of *multiple tuple spaces* and is therefore called the MTS model. Deviating from [28], this approach has no global hierarchical naming scheme for tuple spaces because this property hampers the uncoupling of active agents by assumptions about the global system structure and by locking problems introduced by concurrent operations on complex tuple space hierarchies. Instead, tuple spaces are introduced as first–class entities which may be used like ordinary tuples as well as tuple elements. In order to use tuple spaces as shared communication media, they may be accessed by special handles, called names. These names are also first–class entities and introduce a flat, global namespace for tuple spaces which allows arbitrary reference structures between them. The MTS model predefines a special name called `context` which denotes the tuple space in which an active entity exists.

Active tuples are treated as in [28], where they are active while being stored in a tuple space and passive outside. Consequently, the problems with implementing these operations and with the matching of active tuples (due to the undecidable equality problem between two computations) [38] remain. Furthermore, the motivation for such operations is questionable: [38] mentions debugging purposes whereas [28] argues with forced termination of active tuples in lower levels of tuple space hierarchies and with load balancing issues. But none of these examples seems to provide a convincing motivation for explicitly starting and freezing activities at the programming language level.

The MTS approach specifies tuple spaces as abstract data types in order to coordinate concurrent accesses. This clearly is one of the main merits of this work. But consequently, abstract data types should also be used for every active and passive tuple in such a system which would introduce additional system structure and security for programmers.

Bauhaus Linda

Bauhaus Linda [14] (or "Bauhaus" for short) is a proposal intended to simplify the Linda model while simultaneously adding expressive power. Simplicity is achieved by introducing a single basic data structure: the multiset. Multisets replace tuples as well as tuple spaces. Multisets may contain atomic data entities and other multisets, leading to a hierarchy of nested multisets. Furthermore, multisets replace templates, too. The operations in and rd are performed by specifying a multiset. These operations yield a multiset which includes the given one, such that they are based on multiset inclusion instead of pattern matching. Finally, Bauhaus makes no distinction between active and passive data entities. Passive data is handled as usual by out, in, and rd operations. Active data entities (processes) may be inserted into a multiset by out and removed from it by in, whereas rd produces a suspended copy (an image) of the activity. Thus, these operations do not alter the number of running processes in a system. Additionally, there is a move operation which allows an active entity to move from one multiset to another. The syntax of Bauhaus provides special symbols denoting sibling multisets (having the same "parent" multiset) and the parent multiset itself. These symbols allow processes to navigate within the coordination structure.

By reducing all necessary data structures to multisets, Bauhaus clearly provides a clean, elegant, and powerful coordination model. Unfortunately, the treatment of active data entities introduces inconsistencies. Because Bauhaus operations neither create nor destroy processes, the hosting programming language must be aware of processes in order to provide them as first–class entities to Bauhaus. But because rd provides a process image and move moves processes around in the coordination structure, Bauhaus itself must be conscious of processes, too. This resembles the very high–level but hard to implement abstractions on active tuples from [28].

3 Objective-Linda

In this section we present *Objective Linda* – a coordination model which is suitable for various kinds of concurrent systems like tightly–coupled parallel architectures on one hand and open distributed systems on the other [39]. The model is based on the following concepts: *object–orientation* is the primary design methodology yielding composable, self–contained entities which are protected by the interfaces of their corresponding abstract data types; *generative communication* provides a means to uncouple communication partners from each other in order to enable the coordination of dynamically changing configurations; *homogeneity* is provided by a simple, uniform and hence easy to understand programming model; *hierarchical abstractions* allow to have different views on configurations with different granularities of concurrently operating agents.

Objective Linda is to some degree inspired by the models reviewed in the previous section: Bauhaus Linda [14] unifies tuples and tuple spaces – Objective Linda extends this to a model with a unique kind of entities: objects. Objective Linda's objects serve as data units, as active agents, as object spaces, and as units of application structure. Therefore, the matching process for reading from object spaces can be based on predicates of the objects' abstract data types which significantly improves expressive power.

The object space approach [51] uncouples object matching performed by Linda's in and rd operations from the object implementation. So, objects lose their tuple character as ordered sequences of typed slots. Whereas [51] makes a first step by letting programmers specify important subsets of object implementations which become subject to matching, Objective Linda extends this idea in a consequent manner: 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 heterogeneous 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 parallel systems.

The work in [28, 38, 43] proposes hierarchies of nested tuples spaces by introducing tu-

ple spaces as first–class entities of the model. All of them introduce names (or references) to tuple spaces, whereas [38] 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 name space over the whole system. Consequently, Objective Linda provides hierarchies of object spaces in which active objects access object spaces by a generative matching mechanism. Thus, there are no global or shared objects in an Objective Linda configuration.

The work in [14, 28, 38] additionally treats activities (called "processes" or "control threads") as first–class entities and hence allows scheduling by explicitly starting and freezing those activities. In Objective Linda, objects are introduced by abstract data types. Operations on objects are therefore 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 features. In the following, the main concepts of Objective Linda are presented.

3.1 The Object Model

Objective Linda substitutes Linda's notion of *tuples* by *objects*, and correspondingly *tuple spaces* by *object spaces*. Therefore, a clear notion of *objects* has to be presented. This notion is called Objective Linda's *object model*.

Since the goal is to model possibly heterogeneous systems, a programming language–independent object model has to be defined. In Objective Linda, objects to be stored in object spaces are self–contained entities; their interface operations only affect their encapsulated object state. Objects do not share any kind of state (e. g. variables or other objects) with each other. The basic outline of an Objective Linda configuration is illustrated in Figure 1. In this figure, the active agents are illustrated as computers which

communicate via several object spaces in which objects (represented by the filled circles) are stored as opaque, encapsulated data items. Because there is no sharing between objects, there is also neither a sharing between agents nor between an agent and an object space. Consequently, whenever objects are transferred between agents and object spaces, they are either moved or copied. Hence, no object ever simultaneously belongs to more than one agent, to more than one object space, or to an agent *and* an object space. Object spaces are the only kind of entities being shared between agents. They can hence not be stored inside object spaces like *"ordinary objects"*. The treatment of multiple object spaces belongs to the notion of *dynamic composition* and will be discussed in Section 3.4.

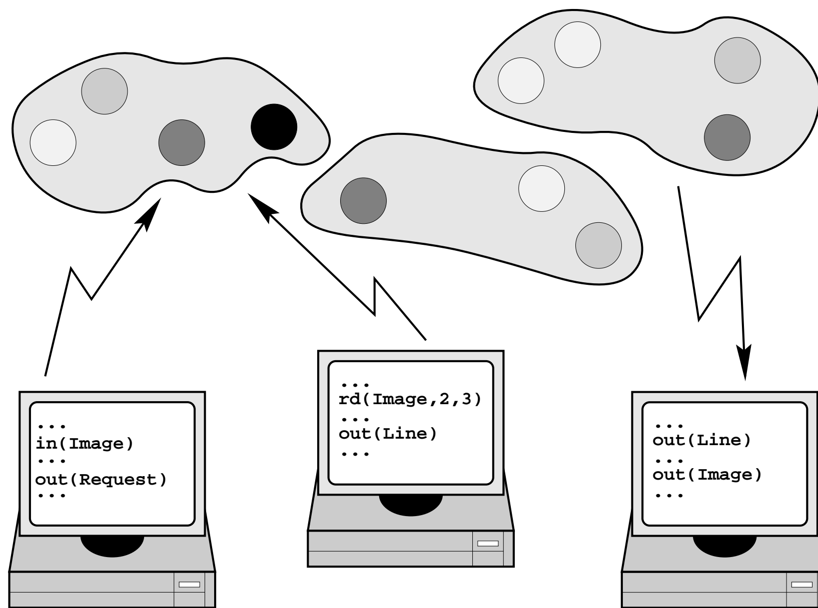

Figure 1: Configurations are composed of agents and object spaces.

The objects are instances of abstract data types which are described in a language–independent notation called *Object Interchange Language* (OIL). Actual programs are hence 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 being used in application programs are then derived from *OIL_Object* or its descendants.

OIL allows subtyping according to the "principle of substitutability" [65] such that an object of type S which is a subtype of T can be used whenever an object of type T is expected. OIL allows single as well as multiple subtyping, as long as this can be performed without conflicts in merging the supertypes' interface definitions.

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 it is of course necessary to identify identical types across multiple language bindings and possibly multiple implementations even within the same language. Naming schemes can in general not avoid name conflicts and consequently unintended name matches or mismatches [64]. Therefore, OIL types are identified by globally unique identifications. For this purpose, Objective Linda makes use of OSF DCE's [49] *Universal Unique ID's* (UUID's) which can easily be created from a host identification and a timestamp, hence without any coordination between multiple sites.

Besides the identification of types it is of course also necessary to transport possibly complex objects between agents operating in heterogeneous environments. For this purpose, Objective Linda defines a mechanism for migrating objects between distributed sites which is based on the objects' type interfaces. This mechanism is presented in Section 3.6.

3.2 Active and Passive Objects

Based on Objective Linda's notion of *objects*, it can now be shown how Objective Linda deals with these objects. Therefore, two kinds of objects have to be distinguished. On one hand, there are the *passive objects* which are stored inside object spaces. On the other hand, there are active objects which play the role of the active agents operating on the object spaces.

3.2.1 Object Matching

Objects being stored in an object space are simply passive data items. Therefore, they are called *passive objects*. While stored inside an object space, they cannot be used or manipulated, e. g. by calling routines defined in their interface. The only operation which can be performed on passive objects is their retrieval whenever an agent wishes to read or consume objects from an object space. The process of finding objects which are "appropriate" for a certain agent's request is called *object matching*. This is in analogy to Linda's *tuple matching* in charge of an in or rd operation.

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 a major improvement with respect to other Linda–like models: objects are treated based on their specification rather than on their implementation which allows to completely abstract from implementation (data representation) details.

Consequently, object matching is based on object *types* and *predicates* that are defined by type interfaces. A potential reader of an object has to provide a so–called *template object* to the in or rd operation. The template's type T denotes the type of object to be retrieved from an object space, whereas subtypes S of T will also satisfy the operation. While the type is the primary criterion for matching objects, further selection is performed by a predicate match (provided by the type OIL_Object) which takes an object of the same type as parameter and returns a boolean value determining whether a given object matches certain requirements. The operation match does so (and can only do so because of object encapsulation) by evaluating predicates defined in T's interface. Hence, whenever an in or rd operation is performed using a template object of type T, objects of type T (and its subtypes) are presented to the template's match predicate until this predicate returns true. Several variants of matching objects of a given type can be selected by presetting the encapsulated state of the template object before it is used in an in or rd operation.

Object matching is illustrated by the code fragment shown in Figure 2. It is taken

```
bool match (const OIL_Object &other){
    Fork &o = (Fork&)other;
    return (myseat == o.leftseat()) || (myseat == o.rightseat());
}
```

Figure 2: The predicate match of a C++ class Fork.

from the *Restaurant of Dining Philosophers* implementation presented in [39]. In this extension to the classical *Dining Philosophers* problem, philosophers first have to take a free seat before they try to grab the two forks adjacent to their seat. leftseat and rightseat hence identify forks during the matching process. A philosopher looking for suitable forks hence creates a template Fork object and sets the myseat attribute accordingly. When supplying this template object to a matching process in an object space, all available Fork objects are subsequently presented to the template's match predicate in order to find out whether they match the philosopher's needs.

The matching mechanism presented so far integrates the predicates used for matching directly into the OIL types. Alternatively, in order to provide more flexibility in how objects may be matched, those predicates could also be kept externally, e. g. by using so–called *function objects* [41]. This approach would allow to implement new matching predicates without changing the OIL type itself. But a major drawback of this approach would be the introduction of many new types (for the function objects) which would be difficult to maintain and to relate to the OIL types on which they operate. However, several variants of matching objects of a type are an exceptional case which can be covered by a few "hard wired" matching variants inside the single match predicate.

3.2.2 Object Evaluation

According to Linda's eval operation, the activation of an agent will be called 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 intended for active objects. As with `match`, behaviour can be individually parameterized by the object's state before it is evaluated.

In order to adapt to object–oriented concepts, the treatment of active objects differs from the way Linda deals with active tuples. 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 agents as it should be in an object–oriented coordination model. As a consequence, Objective Linda's `eval` operation gets 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.

Furthermore, by introducing a sophisticated type system for OIL objects, it hardly makes sense to have exactly one *"result"* per object evaluation of exactly the same type of the active object in charge. For example, in a producer/consumer setting a passive producer object does not make sense to an application program. Instead, objects of a type for data items being sought by consumers are required. Hence, Objective Linda refrains from automatically converting active objects into passive ones at the end of an object's evaluation.

3.3 Operations on Object Spaces

Besides adapting the Linda model to object-orientation, Objective Linda also provides an improved set of operations on object spaces in order to overcome deficiencies of the original Linda model. This concerns two features: first, the number of objects being transferred between agent and object space (or vice versa) and second, the blocking semantics of the operations.

3.3.1 Multisets of Objects

Linda's ability to retrieve only one object at a time from an object space is simple and elegant, but unfortunately too restrictive. The most important restriction is known from the literature as the *"Linda multiple* rd *problem"* [54]. This is due to the fact that the semantics of the rd operation cannot 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 Linda–like systems. The specification of 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, operations should be able to retrieve several objects atomically in order to cleanly introduce such a functionality. The work in [54] describing the York Linda kernel proposes a so–called copy-collect operation which atomically rd's *all* objects from an object space that match a given request. Objective Linda elaborates this idea introducing *multisets* of objects being returned by a rd operation and analogously by an in operation. These multisets are specified by two parameters, namely min and max, denoting a lower and an upper bound for the number of objects to be retrieved. Objective Linda's rd and in operations will hence complete successfully only if at least min matching objects could be found in an object space while they will return at most max objects even if the object space contained more. A special constant value denoting infinitely many objects may be used e. g. for the max parameter in order to retrieve all matching objects which are currently available in an object space.

With this mechanism, Objective Linda allows to atomically retrieve any number of matching objects from an object space, hence solving the *multiple* rd *problem* and additionally allowing to control the cardinality of multisets of objects being returned and hence the granularity of the communication between agents and object spaces. This cardinality control can be used for several purposes, e. g. when the object space is expected to contain much more matching objects than the consuming agent is interested in or is able to deal with. Finally, a major benefit of atomically retrieving several objects instead

of only one at a time is the resulting *bulkiness* of the operations [53] which contributes to implementation efficiency by packing several objects into a single transfer unit (e. g. message) which is crucial for application performance of parallel programs.

```
Seat *s = (Seat*)restaurant->in(*new Seat());
Fork *t = new Fork();
t->set_myseat(s->number());
Multiset *ms = restaurant->in(t,2,2);
```

Figure 3: Code fragment by which a philosopher gets a Seat and the corresponding Forks.

Analogous to rd, Objective Linda's in operation also retrieves multisets of objects. This is beneficial for runtime performance as with rd. Additionally, it helps avoiding deadlock situations with several agents trying to obtain more than one resource at a time. This is illustrated in Figure 3, also taken from the Dining Philosophers example. The code fragment shows how a philosopher gets a Seat the identification of which is then used by a template Fork object in order to atomically get the two corresponding Forks. By setting min = 2 and max = 2 as parameters to the in operation, exactly two Fork objects will be atomically removed from the object space (called restaurant) hence avoiding possible deadlock situations.

Finally, Objective Linda's out operation also takes a multiset of objects as a parameter in order to atomically store all objects contained in it; out'ing of several objects atomically rather than one–by–one does not have semantic advantages. Nevertheless, Objective Linda introduces this feature a) for keeping the operation set consistent, b) because this feature has shown to be convenient to use, and c) because it adds the runtime improvements of bulky operations. It should be noted that although out takes a multiset of objects as a parameter, it only stores the objects contained in it inside the object space. Hence multisets are only used as containers for transporting objects between agents and object spaces.

3.3.2 Operation Timeouts

In addition to moving multisets of objects, Objective Linda also improves the blocking semantics of its operations compared to the original Linda model. Linda had 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), and for reading an object (rd) assume unrestricted access to the data space as a whole and may hence block infinitely in case of object spaces in a distributed system. Here, access to an object space may e. g. fail due to communication problems.

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 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 such systems: operation failure of inp and rdp should indicate "no such object could be found (at the moment)". This change reflects the fact that synchronization based on the absence of a certain object is impossible in Linda–like systems, because even with centralized schemes, race conditions may lead to unexpected results.

Infinitely blocking operations due to communication problems 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.

The introduction of a timeout parameter is also beneficial for use in parallel applications. Here, explicit timeout values may e. g. be used for allowing a (manager) process to repeatedly consume all currently available results and to display them while worker processes are still operative. With Objective Linda, the manager's in operation by which it consumes results may e. g. be parameterized to return every n seconds all currently

available results, or alternatively up to max results. Such a setting exploits bulky transmissions while the manager is still able to operate because it neither has to wait until some result becomes available (like with using Linda's in) nor does it force busy–waiting loops (like with using Linda's inp).

Another possible use of timeout values in parallel applications may be a parallel search algorithm that has to deliver the best currently available result within a given time. An application might be a parallel chess program in which the computation has to be completed with tightly fixed time bounds in order to comply to the rules of chess tournaments. Further applications of timeout values in parallel computations can typically be recognized from situations in which flexible behaviour is necessary and busy–waiting schemes cannot be applied due to limited computing resources.

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.

With the introduction of multisets and timeouts, Objective Linda provides a minimal set of powerful operations in order to keep the model as simple as possible. At the same time, it is powerful enough to express original Linda's operations as well as to reflect additional requirements as discussed above. Table I summarizes Objective Linda's operations which are related to communication and synchronization between several agents. The table shows a small set of important parameter values for min, max, and timeout which illustrates the possible spectrum of Objective Linda's operations. It should be noted that out does not have the min and max parameters because the multiset of objects to be stored is created explicitly before the invocation of out.

3.4 Dynamic Composition

Configurations in Objective Linda consist of active as well as passive objects and of

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

	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

object spaces. The way in which active objects and object spaces are grouped together describes how a configuration is composed from active objects and object spaces. Dynamic composition describes the dynamic changes of a configuration during the runtime of a system. Besides the operations presented so far, Objective Linda also provides some concepts and corresponding operations for managing dynamic composition aspects. These are mainly related to the creation of agents and object spaces and to the attachment of agents to existing object spaces.

Active objects have, from the moment of their activation on, access to a particular object space called their context. This is the object space on which the corresponding eval operation has been performed. eval is one of Objective Linda's operations on object spaces which are related to dynamic composition. It operates analogously to out with the exception that objects are not passively stored in the corresponding object space but are being activated with the object space as their context.

In addition to the context, every agent can dynamically create new object spaces which are initially private to their creator. Unfortunately, the context and additional private object spaces do not suffice for expressing all relevant coordination problems. Therefore, Objective Linda provides a mechanism for allowing agents to attach to other, already existing object spaces.

The capability of attaching to existing object spaces implies the necessity of dealing with multiple object spaces inside the coordination model. Although there have been several proposals for introducing multiple data spaces to the Linda model in the literature, neither of them found broader acceptance. This might be due to the fact that generative communication with multiple data spaces implies a change of the communication paradigm. It can be seen to be in between the two extremes denoted by message passing and the single–spaced Linda model [40]. 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 the main obstacle with respect to general consensus about a suitable model with multiple data spaces.

Unlike the proposals discussed in Section 2, Objective Linda relies on its generative communication properties for managing multiple object spaces. It inherits the concept of a *context* object space from [38] but replaces the *name* "context" by an attribute of every active object with this name. Additional object spaces may be created by any agent which accesses them by its object attribute variables, hence in a basically anonymous way, because no object space ever gets assigned to any kind of (global) identification.

The mechanism used by agents for attaching to existing object spaces other than their *context* is based on a distinct concept called *object space logical*. Such *logicals* combine an object of any subtype of *OIL_Object* with the identification of an object space. *Logicals* are, like regular passive objects, stored inside object spaces. Analogous to the operations in and rd, the selection of an object space to attach to is based on object matching.

```
Object_Space *restaurant;
Restaurant_Sign *rs = new Restaurant_Sign();
rs->set_name("The Philo and The Fox");
restaurant = context->attach(rs);
```

Figure 4: Code fragment by which a philosopher attaches to a restaurant.

Objective Linda hence provides an operation called attach which takes a template object as a parameter and returns (when successful) a reference to the attached–to object space which is locally useful in the address space of the attaching agent. Figure 4 shows the code fragment by which a philosopher attaches to an object space denoting a restaurant which is identified via (logical) objects of type Restaurant_Sign.

A *logical* object is created by a dedicated operation expose that takes as parameter

```
Object_Space *rest = new Object_Space();
Restaurant_Sign *rs = new Restaurant_Sign();
rs->set_name("The Philo and The Fox");
context->expose(rs,rest);

// now serve some philosophers

rs = new Restaurant_Sign();
rs->set_name("The Philo and The Fox");
context->hide(rs,rest);
```

Figure 5: Code fragments by which a waiter exposes and later hides its restaurant.

an object with properties suitable for the application's needs in order to identify a given object space. Additionally, expose takes a (local) reference to the object space to be exposed and combines it with the object to a *logical* which is then stored inside another object space. The inverse operation to expose is called hide; hide takes as parameters a template object matching the *logical* and a reference to the exposed object space. hide removes the *logical* object from the object space if it matches the template object *and* if it refers to the same object space. This mechanism hence prevents agents from accidentally hiding the wrong object space. Figure 5 shows code fragments by which a waiter agent exposes and later hides its restaurant object space.

The *logical* mechanism can be viewed from two perspectives. The user's perspective is that object spaces can be combined with and identified via any kind of object, hence by any abstraction suitable to an application. There are no centralized naming or identification schemes. Conflicts resulting from multiple *logical* objects that denote different object spaces but match the same attach request may only occur local to a single object space in which they are stored.

The implementation's perspective is related to the identification of object spaces. The expose operation takes a locally useful reference to an object space from the exposing

agent, whereas attach returns a reference locally useful to the attaching agent. Both references have to be equivalent by denoting the same object space. But they may be different by belonging to different address spaces in a distributed system, by being part of bindings to different programming languages, etc. Hence, the management of *logical* objects is substantially more than simply storing pointers to object spaces inside objects. In fact, object spaces are identified by UUIDs to and from which local references are converted.

Finally, it should be noted that *logical* objects are *"invisible"* for the operations in and rd because *logicals* are not intended to be exchanged between agents. This separation hence avoids unintended matches. Furthermore, it prevents identifications to object spaces from being directly passed between agents. With this mechanism, Objective Linda forces agents to attach to new object spaces only under control of Objective Linda's runtime system. This basic feature allows to integrate access restriction policies for object spaces in cases in which such a feature may seem appropriate.

3.5 Summary of Objective Linda's Operations on Object Spaces

In the following, all operations on object spaces discussed are summarized. The set of operations hence constitutes the interface of a class Object_Space denoted in OIL syntax. A formal description of the semantics of Objective Linda's operations can be found in [32, 39].

out(m : Multiset, timeout : FLOAT) : BOOLEAN

> Tries to move the objects contained in m into the object space. Returns true if the operation completed successfully. Returns false, if the operation could not be completed within timeout seconds. If out returns true, the caller is no longer allowed to access the multiset m and all objects included inside m. Otherwise, m remains intact.

in(o : OIL_Object, min : INTEGER, max : INTEGER, timeout : FLOAT) : Multiset

Tries to remove multiple objects $O_1 \ldots O_n$ matching the template object o from the object space. 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, the resulting multiset is empty. The template object o will be consumed by the object space in any case. It is hence no longer available to the caller.

rd(o : OIL_Object, min : INTEGER, max : INTEGER, timeout : FLOAT) : Multiset

Tries to return clones of multiple objects $O_1 \ldots O_n$ matching the template object o. 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, the resulting multiset is empty. The template object o will be consumed by the object space in any case. It is hence no longer available to the caller.

eval(o : OIL_Object, timeout : FLOAT) : BOOLEAN

Tries to move the object o into the object space and starts its activity. Returns true if the operation completed successfully. Returns false, if the operation could not be completed within timeout seconds. If eval returns true, the caller is no longer allowed to access the object o. Otherwise, o remains intact.

expose(o : OIL_Object, s : Object_Space, timeout : FLOAT) : BOOLEAN

Tries to move the OIL_Object o into the object space. If this could be performed successfully, o will expose the object space s. Returns true, if the operation could be completed successfully, returns false, if the operation could not be completed within timeout seconds. If expose returns true, the caller is no longer allowed to access the object o. Otherwise, o remains intact.

hide(o : OIL_Object, s : Object_Space, timeout : FLOAT) : BOOLEAN

Tries to remove an OIL_Object from the object space which matches o and to which s had been assigned. Returns true, if the operation could be completed successfully, returns false, if the operation could not be completed within timeout seconds. The template object o will be consumed by the object space in any case. It is hence no longer available to the caller.

attach(o : OIL_Object, timeout : FLOAT) : Object_Space

Tries to get attached to an object space for which an OIL_Object 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, the result has a VOID value. The template object o will be consumed by the object space in any case. It is hence no longer available to the caller.

infinite_matches : INTEGER

This value will be interpreted as infinite number of matching objects when provided as min or max parameter to in and rd.

infinite_time : FLOAT

This value will be interpreted as infinite delay when provided as timeout parameter to out, in, rd, eval, expose, hide, and attach.

3.6 Object Migration

Objective Linda's operations require to migrate objects between multiple agents. Except for specialized shared–memory multiprocessor machines [34], object migration always implies the transfer of object state between distributed address spaces. This is a common problem with systems providing distributed, object–based computing.

One example is CORBA's *Object Externalization Service* [35]. Because CORBA objects lack the notion of properly constructed abstract data types, the externalization service defines a binary format for representing objects. This binary format has to be defined per object type and directly reflects its implementation instead of the client interface which would have been specified by a corresponding abstract data type. Hence, CORBA's object externalization service can in general not be used for externalization and later internalization of objects of the same type but with different implementation (languages) without explicit coding of conversion routines to the externalization format. This format in fact denotes an unspecified, second type description and is closely related to a certain implementation instead of the abstract type interface, as it should be for encapsulated objects.

The same objections hold against object–oriented programming languages whose runtime systems implement *externalization* or *serialization* mechanisms. Among such languages are Eiffel [46] and Java [7]. These languages allow their runtime systems to investigate the state of objects, to transform the state to a stream of basic data types, and to reversely reconstruct objects from such streams. Again, rather the object implementations than the interfaces are used. Hence, these externalization mechanisms provide separate, incompatible solutions, making interoperability between heterogeneous platforms impossible.

As a special case, implementations of Objective Linda might restrict themselves to a single programming language platform. Then, the above mentioned platform–specific mechanisms might be used. But in general, a more powerful mechanism is necessary. Therefore, Objective Linda relies on a technique known as *object imaging* [63]. The idea is to create at the target site of a communication objects which are not identical but *equivalent* to the original ones. When dealing with Objective Linda's typed objects, this equivalence relation is obviously defined via the type interfaces. Hence, two objects of the same type are equivalent whenever all observing interface routines return identical values when applied to both objects. This notion of equivalence provides the conceptual basis

for object migration in Objective Linda.

On one hand, type interfaces are the only common denominators between heterogeneous implementations. On the other hand, primitive or *basic* data types are the only ones that can be directly transferred and (if necessary) converted between different platforms. So, the problem is to provide transformations (1) from an object O_1 to a stream of basic data objects that can be transferred, and (2) back from these basics to another object O_2 which is equivalent to O_1 with respect to the observer operations of their common abstract data type. We call such streams of basic data objects "Imaging–Streams".

The transformation of an object with a given state to a corresponding stream of basic data objects is generally possible for every type and every object state, because it can be created in worst case from a history of operation calls that would have to be maintained for this purpose. But normally, this transformation should be possible by interpreting the object's current state which can be assumed to be relatively simple because well–designed types should represent one and only one abstraction [45]. So, complex objects typically consist of several "smaller" objects, each of which can be transformed to an *Imaging–Stream* of its own. The reverse transformation can simply be realized by interpreting the *Imaging–Stream* while continuously calling the respective operations.

For transforming a given object into a stream, every OIL type has to provide an operation called marshal. Analogously, an unmarshal operation has to be provided for constructing objects from streams. Application programmers only have to provide these operations which are automatically called by Objective Linda's runtime system whenever a corresponding object has to be transferred.

3.7 Comparison to Related Approaches

We will now summarize how Objective Linda's features compare to related approaches. The important features of the Objective Linda coordination model can be categorized within three groups. The first group concerns object orientation. Here, all Linda dialects for object–oriented languages (as discussed in Section 2) have notions of objects as the

entities to be dealt within object spaces. But only Jada and Objective Linda allow object matching based on types and interface predicates. Uniquely, Objective Linda defines its notions of objects and types independent of existing programming languages.

The second feature group concerns the power of the object–space operations. Here, only the York Linda kernel [53] and Objective Linda provide solutions to the Linda multiple rd problem. York Linda does so by its bulky operations which may atomically collect all tuples matching a given request. Objective Linda elaborates this concept and allows to specify the cardinality of multisets of objects to be retrieved by two parameters denoting a minimal and a maximal number of desired objects. Additionally, Objective Linda is the only model that introduces timeout values in order to customize the blocking of its operations. (In fact, there is a second model called Sonia [9] in which timeout values are introduced. But because Sonia is targeted at coordinating activities in organizations composed of humans and computers, it does not contribute to the area of object–oriented parallel computing. Hence, it is not discussed here any further.)

Finally, the third feature group is concerned with multiple tuple spaces and composition of configurations. Whereas most of the models discussed in Section 2 support multiple data spaces, Objective Linda is the only one in which generative communication is employed in order to achieve dynamic composition of configurations, hence seamlessly introducing the treatment of multiple object spaces into the model.

4 Programming Examples

In this section we present two programming examples in order to show how Objective Linda can be used to express coordination problems in parallel applications. Since Objective Linda allows to use higher–level abstractions for writing parallel programs in the Linda style, it is possible to provide *reusable object-oriented coordination patterns* [23, 24] which cover basic abstractions common to frequently used settings in which parallel processes have to interact in a coordinated manner. Developing parallel programs from these reusable basic building blocks would then simply require to parameterize the behaviour

of the patterns to the needs of the given problem and to compose the parallel application out of these patterns. This approach follows the idea of *software composition* [48], i.e. producing new software by composing it from already existing components which can simply be "plugged together".

In the following, we present two coordination patterns useful for performing many coarse-grained parallel tasks, i.e. tasks where the fraction between the time for interprocess communication and for computation is rather small. These are the *manager/worker* patterns and a parallel *divide-and-conquer* pattern. Code fragments for components instantiating these patterns are based on C++ notation, i.e. a C++ language binding is assumed to have been established through the OIL.

4.1 Manager and Worker Patterns

It is a very common situation in parallel programming to employ a specific manager process to divide a given problem into smaller tasks and to distribute these tasks among several worker processes. While workers repeatedly process such tasks and return the computed results to the manager process, the managerial task is much more complex. The manager not only has to operate on the application level by providing task units and later combining the received results to the overall result of the application. It also has to perform coordination–level tasks, such as assigning tasks to workers and terminating workers.

Although both levels of managerial tasks are independent of each other, they are typically intermixed in existing applications. They are hardly made explicit, but instead implicitly performed by the communication operations of manager and workers. Therefore, it is our motivation to provide clearly defined abstractions for both levels in the form of coordination patterns suitable for building reusable coordination components.

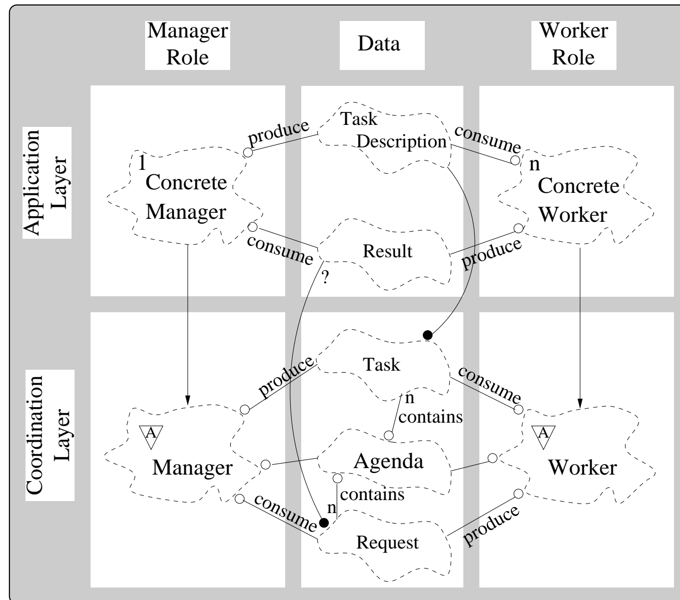

Figure 6: Booch Diagram for Manager/Worker Patterns

Structure and Organization

The intent of the manager and worker patterns is to decouple coordination–level issues such as task assignment strategies and worker termination from application–level issues such as task creation, task computation, and result combination. The Manager pattern is responsible for providing and assigning task units and for collecting results. The Worker pattern is responsible for acquiring and executing task units and for transmitting computed results to the manager.

The structure of the Manager and Worker patterns are illustrated by the Booch diagram [11] shown in Figure 6. In these diagrams, dashed clouds indicate classes; solid triangles marked with an "*A*" denote abstract classes; a solid undirected edge with a hollow circle at one end indicates a "*uses*" relation between two classes. Furthermore, directed edges indicate inheritance relationships between classes; and a solid circle illustrates a composition relationship between two classes. The figure is divided into two horizontal layers, one for coordination aspects and one for application aspects. Furthermore, the diagram is vertically divided into the manager role, the worker role, and the data exchanged between both, namely objects and object spaces.

The Agenda is the central shared data structure (i.e. the object space) through which the abstract components in the coordination layer, Manager and Workers, communicate. The Task and Request objects exchanged in this layer are primarily used as containers for their application–specific counterparts, Task Description and Result, which are transparently transported via the coordination layer. Concrete Manager and Concrete Worker are instantiated by inheriting from Manager and Worker, respectively, while providing suitable implementations for their application–specific, abstract methods.

Interactions between Manager and Worker

Although a manager/worker architecture seems to be straightforward at first glance, there is a broad spectrum of possible interaction protocols between Manager and Workers [21]. In the simplest case, the Manager just out's all Task objects while the Workers in them as long as there are still Task objects available. They then out Request objects containing the corresponding results back to the Agenda. This type of collaboration is only useful in situations in which (a) workers can be safely set up *after* the agenda has been filled (because their reasoning is based on the absence of objects and hence they cannot distinguish between "no more tasks" and "tasks not yet available"), (b) the entire set of tasks is known at program startup, e.g. there are no iterations with processing steps of the manager in between (also due to "reasoning on the absence of objects"), (c) the assignment of tasks to particular workers is not important (e.g. it will have no significant impact on performance), and (d) the entire set of tasks can be stored in the agenda without violating memory restrictions.

In order to deal with problems (a) and (b), Workers can also join a "*worker group*" maintained by the Manager at program startup and leave it after having received a "stop task". In order to deal with problems (c) and (d), workers can *request* tasks being assigned to them from the manager which e.g. allows to perform load–balancing issues.

Further refinements of manager/worker interactions may stem from the need to decouple the number of Result objects being created by the workers from the number of tasks

created by the manager, which might become necessary in order to overlap processing between manager and workers or for implementing adaptive schemes in which workers may act as "sub–managers" for parts of the tasks assigned to them.

Implementation

In the following, an implementation of Manager and Worker components written in the C++ language and using Objective Linda will be sketched. The presentation is restricted to the evaluate routines. These define the behaviour of the components once they are activated.

As shown in Figure 7, the Manager component first creates the Agenda object space and expose's it in its context. If implemented by heirs, setup_agenda() inserts task objects into the agenda, supporting the simplest case of manager/worker interaction.

Then, the Manager dispatches Request objects sent by the Worker's. Depending on the kind of request, the Manager keeps track of the size of its worker group, processes results, or assigns new tasks to workers. The Manager's implementation relies on two (abstract) routines which have to be implemented by heir implementations: get_next_task() returns an object denoting the next task to be assigned to a worker whereas process_result() takes a result object and performs application–specific postprocessing.

Figure 8 shows the Worker component. Its basic mode of operation (according to the scope of possible interaction schemes) is determined by the abstract routines do_register() and use_handshake(), the former determining whether the worker will join a worker group, the latter indicating whether the worker will explicitly request tasks to be assigned. The Worker's fundamental operation is to process tasks until it is requested to terminate. Its abstract routine do_the_work() has to be implemented by heirs in order to perform the "real", application–specific task. In "handshake" mode, the Worker explicitly requests new tasks and terminates as soon as it get the special *stop task*. In "non–handshake" mode, the Worker simply consumes and processes task objects until the Agenda has become empty.

```
public: void Manager::evaluate(void){
  OIL_Object *task_descriptor;  Request *r;  Task *t;
  int workers = 0;
  bool still_work_to_do = true;
  agenda = new Object_Space;
  context->expose(*new OIL_Object,agenda);
  setup_agenda();
  while ( still_work_to_do || (workers > 0) ){
    r = agenda->in(*new Request);
    switch (r->get_tag()){
      case JOIN_GROUP: workers++;  break;
      case LEAVE_GROUP: workers--;  break;
      case RESULT: still_work_to_do = process_result(r->get_result()); break;
      case TASK_REQUEST: still_work_to_do = process_result(r->get_result());
                         task_descriptor = get_next_task();
                         t = new Task(r->get_id());
                         if ( still_work_to_do )
                            t->assign_task(task_descriptor);
                         else  t->set_stop_task(true);
                         agenda->out(*t);  break;
  } delete r; } }
```

Figure 7: A C++ class Manager

Sample Usage: Visualization of the Mandelbrot Set

To illustrate the use of the manager and worker patterns, a program for visualizing the Mandelbrot set has been implemented [42]. The Mandelbrot set M is defined over the set of complex numbers c for each of which the function $f_c(z) = z^2 + c$ is investigated. z is a complex variable out of a series $z_0 = 0, z_1 = f_c(z_0), z_2 = f_c(z_1), \ldots$ This series either remains bounded, in which case c is in M, or it diverges away from 0 in which case it is not. It can be shown that once $|f_c^n(0)| > 2$ for a certain iteration n, the sequence diverges.

```
public: void Worker::evaluate(void){
  OIL_Object *result;  bool running;  Task *t;
  agenda = context.attach(new OIL_Object);
  if (do_register() || use_handshake()) // register with manager
     agenda->out(*new Request(JOIN_GROUP));
  result = NULL;  running = true;
  while (running){
    if (use_handshake()){ // request new task
      agenda->out(*new Request(TASK_REQUEST,my_worker_id(),result));
      t = agenda->in(*new Task(my_worker_id()));
      if (t->is_stop_task())
         running = false;
      else {  result = do_the_work(t->get_task());
              delete t; } }
    else { t = new Task(my_worker_id());
           t->match_valid_tasks_only(true);
           t = agenda->in(*t,0);
           if ( t == NULL )
              running = false;
           else { agenda->out(*new Request(RESULT,my_worker_id(),
                  do_the_work(t->get_task());
                  delete t; } }
    if (do_register()||use_handshake())
       agenda->out(*new Request(LEAVE_GROUP)); } }
```

Figure 8: A C++ class Worker

The Mandelbrot set can be visualized by assigning colours to points c depending on the number of iterations n the sequence f_c took until it diverged. A sequence will be aborted when a certain limit of iterations n_l has been reached. In this case, c is assumed to be in M and the corresponding pixel is typically coloured black.

Because the colour of every point in the complex plane can be computed independent of

each other, the Mandelbrot visualization constitutes an embarrassingly parallel problem. It can hence easily be approached by a manager/worker architecture. In the implemented solution, the manager divides the total image into small sub-images, denoted by Region objects. Workers repeatedly retrieve the Region objects for which they compute (and sf out) Image objects containing the corresponding images.

The implementation shown below assumes the existence of classes for Region and Image objects without presenting them explicitly. In Figure 9, the class Mandelbrot_Manager implements a manager refined for the application domain. Its get_next_task method successively returns Region objects while process_result displays the corresponding images. Both methods cumulate the image sizes in order to determine termination conditions. Figure 10 shows the class Mandelbrot_Worker that implements the corresponding, application–specific workers. It trivially implements do_register() and use_handshake(). Furthermore, do_the_work() invokes the numeric calculations which are not shown here. The runtime behaviour of this design is analyzed in Section 6.2.1.

4.2 Divide–and–Conquer Pattern

Another common situation in parallel programming arises when the given computation can be structured as a single recursive function that concurrently serves multiple, dynamically created requests, each computing results for different subsets of the problem data. This case is known as *tree computation* because such parallel algorithms dynamically form recursive task trees in order to solve a given problem. An important representative of tree computations is the class of *divide–and–conquer* algorithms [22].

As in the manager/worker case, parallel implementations of divide–and–conquer algorithms typically intermix code of the algorithm itself with code needed for coordination purposes like the spawning of tasks and the synchronization with concurrently computed results. It is hence the motivation of the divide–and–conquer pattern to provide clearly defined abstractions for each of the two purposes, yielding reusable coordination components.

```
class Mandelbrot_Manager : public Manager{

private: int total_area, area_received, area_out;

protected: virtual OIL_Object *get_next_task(void)
  { Region* result;
    if ( area_out >= total_area ) return NULL;
    else { result = next_region();
           area_out += result->size(); return result;}}

virtual bool process_result(OIL_Object* img)
  { Image *image = (Image*)img; image->display();
    area_received += image->size();
    return area_received == total_area; }

public: Mandelbrot_Manager (int size_of_image)
  { total_area = size_of_image;
    area_received = 0;
    area_out = 0; } };
```

Figure 9: A C++ class Mandelbrot_Manager

Structure and Organization

The structure of the Divide–and–Conquer pattern is illustrated by the Booch diagram in Figure 11. In analogy to Figure 6, there is a horizontal layering between coordination and application purposes. Vertically, there are two distinct layers separating computation from the data being exchanged.

The Divide–and–Conquer class is the core component in this setting. It implements the interaction between multiple instances of itself. For synchronization with results computed by other instances, a class for Future objects (as they are known from parallel functional programming [31]) is employed. Application programmers simply have to implement

```
class Mandelbrot_Worker : public Worker{

protected: char *uuid;

public: virtual bool do_register(void) { return true; }

Mandelbrot_Worker() : Worker() { uuid = new UUID(); }

~Mandelbrot_Worker() { delete uuid; }

virtual bool use_handshake(void) { return true; }

virtual char *my_worker_id(void) { return uuid; }

virtual OIL_Object *do_the_work(OIL_Object* my_region)
  {  return image(my_region); } // compute image of region
};
```

Figure 10: A C++ class Mandelbrot_Worker

heir classes of Divide–and–Conquer that provide suitable realizations for its application–specific, abstract methods. Therefore, the two classes Task and Result represent the data the recursive function deals with.

Interactions between Divide–and–Conquer instances

The basic interaction scheme is that a Divide–and–Conquer object consumes an object describing its task from its context object space into which it finally stores its computed result. Consequently, a Divide–and–Conquer object dynamically creates a new object space in which it eval's a clone of itself and provides it with the corresponding subtask. Hence, the dynamically shaped task tree results in a tree of object spaces through which pairs of instances of the Divide–and–Conquer class communicate.

Figure 11: Booch Diagram for the Divide–and–Conquer Pattern

Whereas interaction with object spaces is subject to the abstract Divide–and–Conquer class, computing the application's recursive function belongs to the duties of the concrete Divide–and–Conquer class. Hence, the control flow has to pass through both abstraction layers once per recursion level. For example, the coordination–layer evaluate method first in's a task object before it calls the application–layer compute method. compute in turn has to invoke the coordination layer in order to perform a recursive call concurrently by another Divide–and–Conquer instance. For this purpose, the coordination layer provides a method called recurse that returns a Future object for synchronizing with the possibly remotely computed result of a subtask.

Finally, for achieving runtime efficiency the task tree has to be pruned efficiently in order to create (roughly) only as many active objects as the underlying computing platform can execute in parallel. As shown in [22], efficient pruning depends on how workload is divided by recursive calls. This is an application–dependent property and therefore has to be implemented in the application layer. For this purpose, a boolean predicate called do_local has to be implemented within the concrete Divide–and–Conquer class.

Implementation

According to the interaction scheme described so far, the implementation of an abstract Divide–and–Conquer class is straightforward. Its core elements are shown in Figure 12. It defines compute and do_local as abstract methods. Furthermore, evaluate consumes a task object and in turn out's the computed result. Finally, recurse checks whether the given subtask should be computed locally. In this case, it initializes the resulting Future object with a locally computed result. Otherwise, it creates a new object space and eval's a clone of itself.

Figure 13 shows the implementation of the class Future the objects of which are either assigned directly to a locally computed result or are initialized with an object space. The latter is used for synchronization with the concurrently computed result. Hence, the calling active object will wait for the result when it invokes the item method for the first time.

Sample Usage: Knapsack Packing

To illustrate the use of the Divide–and–Conquer pattern, a program for solving the *knapsack problem* has been implemented. The knapsack problem is defined as the problem of finding a set of items each with a weight w and a value v in order to maximize the total value while not exceeding a fixed weight limit. In the implemented divide–and–conquer algorithm, the problem for n items is recursively divided into two subproblems for $n - 1$ items, one with the missing item put into the knapsack and one without it. Whereas the first subproblem is handed over to another processor, the second one is recursively computed within the same node, yielding a dynamically shaped task tree. The pruning of the task tree is performed by keeping track of the actual recursion depth.

Figure 14 shows the methods do_local and compute of a class Packer that implements a concrete Divide–and–Conquer solver. The Packer class assumes the presence of classes Solution and Item which are not shown here. Solution basically enacts sets of Item's with given weight and value, and with the total nbr_items in the set as well as the number of

```
class Divide_and_Conquer : public OIL_Object {


public: virtual OIL_Object* compute(OIL_Object* task);
        virtual bool do_local(OIL_Object* task);


virtual void evaluate(void){
  OIL_Object* my_task = context->in(*new OIL_Object);
  context->out(*compute(my_task));
  delete my_task;
}


Future *recurse(OIL_Object *task){
  Future *result = new Future();
  if (do_local(task)) result->assign(compute(task));
  }  else {
    Object_Space *OS = new Object_Space();
    OS->out(*task);
    OS->eval(*(this->clone()));
    result->set_os(OS);
  }
  delete task;
  return result;
}};
```

Figure 12: A C++ class Divide_and_Conquer

already_fixed items in the solution along the current recursion tree. The compute method first checks whether including the currently considered item will exceed the total weight limit. If this is not the case, it spawns a new Packer object by invoking recurse. To find the solution with the best total value by omitting the current item from the set, the recursive call is made locally. Finally, both solutions are compared to each other before the better one is returned as the result. The runtime behaviour of this algorithm is analyzed in

```
class Future {

protected: Object_Space *os;
           OIL_Object *my_item;

public: Future(){ os = 0; my_item = 0; }

void set_os(Object_Space *o){ os = o; }

void assign(OIL_Object *i){ my_item = i; }

OIL_Object *item(void){
  if (my_item == 0)  my_item = os->in(*new OIL_Object);
  return my_item;
}};
```

Figure 13: A C++ class Future

Section 6.2.2.

5 Implementation Issues

In order to enable the evaluation of Objective Linda's concepts, a prototypical implementation for a cluster of networked workstations and a language binding to C++ has been carried out.

The requirements of supporting C++, multithreading and flexible configurations suggested that an implementation of Objective Linda has to rely directly on TCP/IP sockets as they are supported on the operating system level. Unfortunately, the programming interface to sockets is very complicated and hence error prone. This led to the decision to base Objective Linda's implementation on the ACE toolkit [55, 57]. ACE is a class library for C++ which provides a set of wrapper classes for portably accessing operating system

```
bool Packer::do_local(OIL_Object *task){
  return (task->already_fixed > parallel_depth);
}


OIL_Object* Packer::compute(OIL_Object* t){
  Solution* s = (Solution*) t;
  Solution *result,*solution_with,*solution_without;
  Future *f = 0;
  int current_item = s->already_fixed + 1;
  if (current_item < s->nbr_items){
    if ( s->weight + s->item(current_item)->weight <= s->weight_limit ){
      solution_with = s->clone();
      solution_with->include_item(current);
      solution_with->set_already_fixed(current);
      f = recurse(solution_with);
    }
    solution_without = s->clone();
    solution_without->set_already_fixed(current);
    solution_without = compute(solution_without);
    if ( f != 0 ){   // we have two possible results
      solution_with = f->item();
      if (solution_with->value() > solution_without->value()){
        result = solution_with; delete solution_without; }
      else { result = solution_without; delete solution_with; }
    } else { // we have only one possible result
      result = solution_without; }
  } else result = s;
  return result;
}
```

Figure 14: A C++ class Packer

features with a type–safe, object–oriented interface. Additionally, it provides a framework of design patterns for various aspects of communication between processes as well as between lightweight threads of control. Furthermore, the ACE toolkit is available for a wide spectrum of operating systems, covering various flavours of UNIX, Windows/NT, Windows/95, and even MVS, VxWorks (a UNIX variant for embedded systems), and other platforms.

The existing prototype implementation provides the full functionality of Objective Linda's operations on object spaces and its basic data types. In the following, the implementation design will be discussed and compared (where applicable) to existing implementations of the original Linda model. The discussion focuses on six major design decisions concerning configuration setup, object storing, timeout behaviour, load distribution, management of type hierarchies, and object transfer.

5.1 Configuration Setup

As mentioned earlier, configurations in Objective Linda consist of active objects that store and retrieve *passive objects* using *object spaces*. These three kinds of entities have to be mapped onto operating system–level processes running on different workstations that communicate across a local area network.

Such a process consists of a set of C++ classes that implement the OIL types in use, as well as a so–called *kernel* implementation and a startup mechanism. The active objects perform the application's computations whereas the kernel part of a process is responsible for the management of object spaces. Both parts can consist of several independent activities which are to be executed concurrently within each process. Consequently, Objective Linda processes consist of several concurrently operating, lightweight threads of control.

Computations are set up in a SPMD (*single program multiple data*) style [20]. Hence, all application processes run the same program binary. One of them will be forced by the startup mechanism to start the program execution by `eval`'uating the so–called *root object* of a certain OIL type specified at compile time by the application's programmer.

All other processes start up passively and wait to serve requests for eval operations of active objects.

All these processes are started by a dedicated startup program called olrun (in analogy to MPI's mpirun program [30]) that is concerned with selecting a group of workstations, with starting processes, and with the establishment of communication channels between the processes. Hence, Objective Linda's implementation is responsible for all aspects of program execution except the application's active and passive objects. Consequently, programmers are completely freed from concerns of control flow and can focus on the implementation of their application types.

It should be noted that this startup mechanism has been designed in order to simplify the start of several processes belonging to a parallel application. By simply exchanging the startup mechanism (and without further changes to the implementation), distributed applications can be built in which every process has its own active object at startup time.

5.2 Object Storing

An important goal of every platform for parallel programming is to reduce communication costs, since this factor significantly influences achievable application speed. In Linda–like models, communication costs stem from storing and retrieving data into and from data spaces. There is a tradeoff between centralized and decentralized data storing. When using a centralized data space S_c, every process simply has to send a unicast message to S_c whenever it wants to store or retrieve data. Although this is the simplest form of data space implementation, S_c immediately becomes a bottleneck, and the performance severely degrades, even with small–sized configurations. An example for this behaviour is the Glenda system [61].

Alternatively, data may be stored in a distributed fashion in which every node of a system contains its own data storage. With distributed storage, two alternatives are possible [62]. The first alternative is to replicate data objects over all nodes. Then, the out operation broadcasts the data objects to all nodes, whereas rd operations may

be performed locally without any communication. Besides the overhead of storing data objects on all nodes (while typically needed only by a few), this scheme has another performance drawback: since the in operation has to ensure that every object must be consumed at most once, costly protocols such as e. g. two–phase–commit [62] have to be implemented.

Consequently, it is a better option to store data objects locally where the corresponding out operation has been performed. in and rd then have to broadcast to all other nodes in order to retrieve data. With this scheme, there are still two options concerning the situation in which matching data objects cannot be found immediately. The first option is to simply discard unresolvable requests on the receiver nodes and to repeat requests after a certain timeout. This scheme tries to minimize communication but has the drawback that late–coming matching objects cannot be found immediately, but only after the next timeout. Hence, it is preferable to broadcast request cancelation as soon as the request has been satisfied. It should be noted that the non–replicated storage approach may lead to unnecessary data movement when requests are simultaneously satisfied by multiple nodes. In this case, the unused objects simply migrate to a new location.

Traditional implementations such as e. g. [60] try to achieve runtime efficiency by static compile time analyses. A first variant is to combine the two approaches of distributed storage, yielding a system with partial replication. Here, the set of processes is logically arranged in a grid where each process is assigned a coordinate (r, c). Processes broadcast out'ed objects to all processes within the same row r and broadcast in or rd requests to all processes within the same column c. Hence, every matching data object will be found on exactly one node per request.

Another variant is to classify data objects (Linda tuples) by their structure (which is called their type signature) and to statically assign storing nodes for every kind of signature in use [60]. This scheme can be implemented without any broadcasting, but due to the necessary compile–time analysis, it is (like the partial replication approach) confined to static, closed systems and hence not suitable for Objective Linda.

A better distributed–storage implementation can be achieved when exploiting the presence of multiple data spaces, as it is done in the York Linda kernels [52]. Here, tuple spaces are at runtime classified into two categories: local and remote ones. A local tuple space is only known on the node on which the corresponding tuples are stored. A remote tuple space resides on a special process called tuple space server (or a set of servers sharing work load). Remote tuple spaces are the ones known by more than one process. In this approach, communication costs are minimized by locally accessing local tuple spaces (without any communication), by unicast communication between application process and tuple space server, and finally by bulk movement of complete tuple spaces in case of classification changes between application and server processes.

The object storage implemented for Objective Linda has evolved from the York kernel's concepts. As outlined above, application processes of Objective Linda are multithreaded in order to reduce costs of inter–process communication between active objects. Consequently, application processes consist of two kinds of lightweight threads, one for active objects, and one for the *kernel* which implements the local parts of the object spaces and the communication with the other processes of a configuration.

Furthermore, Objective Linda distinguishes between *local* and *remote* object spaces like the York kernel does. In Objective Linda, every object space has its owner process in the address space of which it resides. If an object space is local, then only its owner process is attached to it. (No threads of other processes *"know about"* this object space.) If threads of more than one process are attached to an object space, the object space is called to be remote. During the runtime of an Objective Linda application, object spaces may change state from local to remote whenever new processes attach. The object store of an object space physically resides in the address space of its owner process. Hence, every object space is implemented in a centralized manner, while each object space may have its own server process. Ownership is directly coupled to the creation of the object space; the process on which an object space is created implicitly becomes its owner. With this scheme, the implementation of object space operations becomes

rather simple. Only threads local to the owner process access an object space. They do so under mutual exclusion – very similar to the monitor concept known from operating system technology [62]. Threads from other processes access (remote) object spaces in a way resembling remote procedure calls: They connect to the owner process which in turn starts a new proxy thread in charge of processing this request. The proxy thread now operates like any other local thread. After completing its task, the proxy thread returns its results back to the remote process and terminates.

With this storage scheme, communication between processes is tailored to application needs: Local operations (between threads on one process) are performed locally, and other requests are exchanged only by unicast communication. Hence, it is up to the programmer to make use of several object spaces allowing process sets operating on disjoint data to operate without interfering with each other. Furthermore, the *thread–per–request* model [58] of implementing object space servers supports a maximal degree of concurrency because every application thread may instantaneously access all object spaces. Even multiple threads of a single process may simultaneously access the same remote object space without interfering with each other.

It should be finally noted that every Objective Linda configuration provides its own *root* object space. There is nothing special about it besides the fact that this is the *context* object space assigned to the *root object* of a parallel computation. In realizations of Objective Linda for distributed applications in which every process involved runs its own *root object*, the root object space becomes necessary in order to initiate communication between all processes of a system.

5.3 Timeout Behaviour

The implementation of timeout values for local object spaces is rather simple. Whenever an operation is invoked, it computes the absolute time T_a until it returns by adding the current time and the timeout parameter. Then, the operation itself is performed. If the operation is unsuccessful, the current thread sleeps until it is woken up by another thread

or until the absolute time T_a is reached. As long as T_a has not been reached, the thread retries the operation every time it is woken up. Otherwise, it returns unsuccessfully.

This mode of operation introduces some additional delay Δt because of two reasons: (a) when the operation is finally woken up due to timeout, it needs some time for cleaning up and returning. And (b) when the object–space lookup itself takes longer than the given timeout value, this cannot be (easily) recognized. But both delays should be very small and hence negligible.

The implementation of timeout behaviour for remote object spaces poses the fundamental question whether the timeout should be measured at the client side or at the server side. The client side measurement comes closer to the intended semantics, because it eliminates communication delays. Server side measurement simplifies the implementation because the communication parts can be implemented without consideration of possible timeout errors. Furthermore, server side timeouts increase scheduling fairness, because all (local as well as remote) requests remain active within the object space operations for the same amount of time.

This view is clearly biased by the anticipated application area of parallel computing. If the focus of the implementation would be closer to distributed systems, the modeling of communication failures would be more important, forcing timeout measurements to be performed at the client side. But because the existing implementation is aimed at parallel rather than distributed computing, scheduling fairness is valued higher than fault tolerance. Consequently, all operation timeouts of Objective Linda operations are measured homogeneously at the server sides, namely at the owner processes of the object spaces.

5.4 Load Distribution

As shown in [10], the implementation of the `eval` operation significantly influences application speed. SCA's traditional implementations [33] are based on a fixed set of processes each executing at most one active tuple at a time. Unfortunately, such a static approach

is not quite suitable for workstation clusters in which dynamically changing configurations must be supported. Nevertheless, one concept from these implementations is worth to adopt: Objective Linda also relies on *eval servers*, a set of processes waiting to execute active objects to be *eval*uated. Using this concept, costly process creation and destruction can be avoided at application runtime.

Systems like Glenda and the York kernel are implemented on top of PVM [25] and map eval onto PVM's spawn operation, hence creating new processes at runtime. Although this approach supports dynamically changing sets of processes, it needs spawning new (heavy weight) processes for every eval operation.

Because the Objective Linda implementation supports multithreading, evaluation of active objects can be performed based on creating only lightweight threads and is hence better suited for finer grained units of execution. In order to support this, several *worker* processes serving eval requests have to be provided.

The current implementation relies on a static set of processes, but plans for the near future include to provide an interface to a resource management system [6] which will enable Objective Linda applications to dynamically change the sets of processes during runtime whenever the application or other users in a workstation cluster require to do so. It is expected that such a coupling will further increase runtime performance of Objective Linda by exploiting adaptive load distribution schemes.

Besides the mechanism responsible for managing the processes involved, it is also crucial for application performance how to map eval requests to worker processes. This problem incurs two contradicting goals: On one hand, new threads should be created on the local machine in order to enforce data locality, hence minimizing communication across the network. On the other hand, different machines should serve eval requests in order to achieve sharing of workload.

Because it is impossible to decide automatically in advance whether an active object to be created should (in the programmer's mind) communicate only locally (e. g. for very fine–grained operations) or whether it is intended to share real workload, Objective

Linda's implementation realizes a workload distribution scheme which relies on locality or remoteness of the object space in which the new active object is to be created. The decision can be taken in a very simple manner: if the object space in charge is marked to be local to the process executing `eval`, the newly created active objects will run within the same process. If the object space is marked to be remote, meaning that operations will need network communication anyway, the newly created active object will run within different processes. Currently, these processes are simply determined using a round–robin strategy which statistically distributes workload quite evenly across all nodes. But in the course of interfacing with resource management systems, the problem of finding the best–suited processes will be solved by a resource manager, too. In order to get workload distributed initially, the *root object space* is marked as being remote during its initialization.

5.5 Management of Type Hierarchies

Objective Linda's object model is defined using the independent OIL language the concepts of which have to be expressed by the means available in the hosting programming language. Unfortunately, C++ is except for its popularity and implementation efficiency not too well suited for expressing concepts of object orientation.

Generally, OIL types can be approximated by a disciplined use of C++ classes. But the language lacks means to deal with types (or classes) as first–class values. Any implementation of the Objective Linda model needs three kinds of type–related information, namely functions from objects to their type identification and vice versa, and predicates representing subtype relations. A function from a given object to its type is needed for implementing object matching in order to find out which type of objects should be matched. The reverse function is needed whenever objects are transferred from one process to another and new objects have to be created from a description generated by the source process (see Sect. 5.6). Finally, subtype relations have to be maintained in order to implement subtype search in cases in which objects of subtypes of a type to be matched have to be found in an object space.

Every OIL object can be characterized by a pair (T, S) where T denotes its OIL type specifying the object's interface, and S denotes the set of supertypes, from which T is a subtype of. (It is sufficient to denote the supertypes of which T is a direct subtype, because the transitive closure can be computed automatically.) In order to identify types across systems with different language bindings (and to avoid accidental type matches or mismatches), types have to be identified in a globally unique manner. This can be easily done by employing *Universal Unique IDs* (UUIDs), as defined within OSF's DCE [49]. UUID's have an internationally standardized format and consist of date, time, and host identification, allowing any computer in the world to create globally unique identifications without any interactions with others. Consequently, Objective Linda employs UUIDs in order to identify the OIL types T. Accordingly, the sets of supertypes S are represented as sets of UUIDs.

The current implementation requires every class that implements subtypes of OIL_Object to provide class members for T and S. These are declared as static class members, hence requiring storage only once per class which can be considered to be negligible. Using T, a mapping from objects to their respective OIL type follows immediately.

Reverse mappings, from types or classes to objects, must be constructed separately. This is achieved by a process–global registry at which every class used by a process registers its type id together with a function that returns an initialized object of its corresponding class. This registration can be performed by instantiating one static object per class which can be done almost automatically by a declaration in the classes' type interfaces. The C++ compiler preprocessor is used in order to transform declarations in the form of

DEFINE_OIL_TYPE(type name, UUID, supertype UUIDs), and

IMPLEMENT_OIL_TYPE(type name, UUID, supertype UUIDs)

into all necessary code. Furthermore, part of the implementation is a utility program called olclassgen that automatically generates skeleton files for the implementation of OIL types in which (among other parts) most of the details concerned with the management of type hierarchies is already filled in.

This automatic class registration is also used by the implementation in order to construct a graph data structure representing the subtype relations between all OIL types used in a program. To summarize, the existing implementation allows to deal with explicit type information provided on a per–class basis without the necessity to maintain static type tables by programmers.

5.6 Object Transfer

Objective Linda's implementation also relies on the York kernel in another respect: the bulkiness of object transfer between processes. The York kernel provides two operations called collect and copy_collect for retrieving *all* tuples matching a certain request template. All these tuples are then transferred in one bulk message to the requesting process, resulting in increased communication performance because of the saved message startup times of separate transmissions.

It should be noted that the York implementation might also omit bulk transmission whenever the amount of data to be transferred might become prohibitive, e. g. with respect to memory limitations. In this case, the tuples are only logically moved and can be transferred one–by–one. Although this is performed transparently, it sacrifices increased performance for the sake of feasibility.

Objective Linda parameterizes its in and rd operations with the min and max parameters, denoting the minimum and respectively the maximum number of objects to be read atomically by one operation. All objects (up to max) matching a certain request are then transferred in one "bulky" message. Here, Objective Linda builds on York's bulk transfer and adapts it to the granularity needed by applications. So, unnecessary overheads can be avoided when some but not all objects matching a certain request are needed. In our implementation, we assume that the number of matching objects to be transferred at once is under control of the application programmer and will hence always fit into given memory restrictions except for cases of fatal errors.

As mentioned earlier, objects are converted to streams consisting of basic data types

only. Such streams can directly be transferred between processes by using the IOStream classes from the ACE toolkit. Together with the implemented thread–per–request model, the use of IOStreams avoids the necessity of implementing a separate message passing layer inside Objective Linda's runtime system.

5.7 Basic Components

A schematic view on a running Objective Linda configuration can be seen in Figure 15. The bottom of the figure shows some worker processes. In each worker process, the *main thread* (the initial thread of the process) is used for event dispatching in a so–called *reactor loop*, using ACE's *Reactor* pattern [56]. Occurring events are signals, as well as command data from the olrun process, or connection requests from other worker processes.

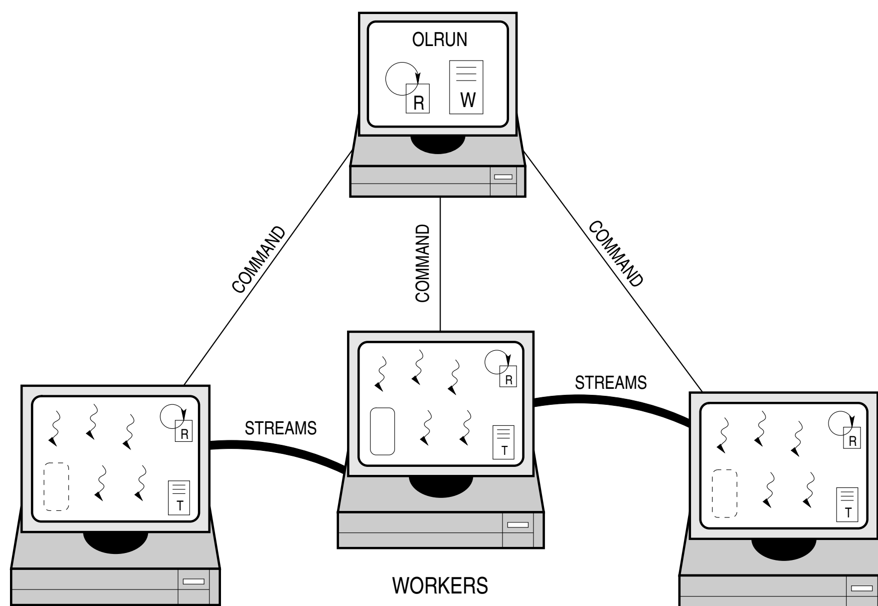

Figure 15: A running Objective Linda configuration.

Additionally, each worker process contains a static type registry T in which the typing information as described in Section 5.5 is stored. Besides some threads evaluating active objects, worker processes contain objects of the class Object_Space, drawn as rounded boxes. In the figure, the object spaces drawn with dashed lines indicate *proxy object spaces* representing remote object spaces to threads of the local process. These proxy object

spaces (of class OS_Proxy) are responsible for the communication with the corresponding owner process. OS_Proxy is implemented as a subclass of Object_Space and can hence be transparently used by application threads.

The communication between worker processes is performed in a RPC–like fashion. Hence, connections between worker processes are set up dynamically according to the communication patterns of the application threads. These communication channels directly rely on ACE's stream classes. They hence provide a reliable transport layer suitable for exchanging all kinds of basic data types. The communication streams between worker processes are set up passively and actively by using ACE's *Acceptor* and *Connector* patterns, respectively [59].

All worker processes are statically connected with the olrun process via so–called command connections. These command connections also rely on ACE's stream classes. The olrun process is of a simpler structure compared to the workers. There is just one thread operating in a reactor loop which maintains a worker registry W. The command interface is used for starting the *root object*, for signaling its termination, for assigning idle worker processes to eval requests in the case of remote object spaces, and for the joint termination of all processes.

An integral part of any Objective Linda implementation is its binding of the OIL concepts to the underlying object–oriented programming language. In the case of the existing prototype implementation, this is C++. Such a language binding consists of the set of entities that express Objective Linda's concepts in terms of the programming language. In the case of C++, the language binding is performed by a set of class interfaces.

6 Performance Evaluation

In this section, the efficiency of the presented prototype implementation and the runtime behaviour of parallel applications programmed using Objective Linda is evaluated. In order to do so, we present performance measurements for the main operations of Objective Linda and for two sample applications. In both cases, a comparison to an equivalent

implementation using C++ and the *Parallel Virtual Machine* (PVM) [25] as the message passing library is made.

6.1 Runtime Measurements of Objective Linda's Operations

The performance measurements presented below were performed on a pool of Digital Alpha workstations running Digital UNIX in version 4.0, connected by switch–based high–speed networks, partly using FDDI (DEC 3000 Model 300 X), partly using ATM (AlphaStation 200 4/166). PVM in version 3.3.11 has been used for the comparison. The runtimes presented below are average values of multiple runs in each of which every operation has been invoked 1000 times by every thread.

In order to measure the runtimes of Objective Linda's operations, a distinction between accesses to local and to remote object spaces must be made. Of further interest is the runtime behaviour in the case of multiple threads that concurrently access the same object space. The performance evaluation presented here is restricted to the data–moving operations out, in, and rd, because these operations are the most frequently called ones. Their performance hence dominates the communication fraction of application runtime behaviour. All tests have been performed with multisets of one object of type OIL_Object containing no information besides its type. Additionally, OIL_Object's match operation is empty and always returns the value true. With these restrictions, the runtime measurements directly reflect the costs of the object–space operations without consideration of further complexity added by more elaborate object types.

Table II shows the runtimes of the out, in, and rd operations while comparing local and remote object spaces with message passing using the PVM system. Local object spaces reside in the same address space (the same process) as the caller of the operations. Their runtime consists of the costs for acquiring and releasing the mutex variable of the object space and for moving the objects into or from the object space's data containers. The out operation is the fastest among the set of measured operations. This is because out simply stores objects. The in operation retrieves objects from an object space. It

therefore has to call the match operation of the given type. This causes the additional runtime costs compared to the out operation. The rd operation is slightly slower than in. This is because rd has to perform the same tasks as in, but additionally rd has to clone the objects it found in order to keep the object space unchanged.

As shown in the table, contention effects of multiple threads concurrently accessing the same object space increase the operation runtime. This increase is significant and almost linear with the number of concurrent threads. Nevertheless, the access times of local object spaces are very fast compared to remote object spaces. This significant difference justifies the introduction of local object spaces for tightly coupled threads of control.

Remote object spaces are located in different processes. Their operations hence include the connection setup to the remote process, the spawning of a proxy thread, and the costs of the corresponding local operation performed by the proxy thread. On remote object spaces, differences between the three operations almost disappear. This is because runtime costs are dominated by connection setup and spawning of new threads. In an optimized implementation, these costs might be significantly reduced e. g. by reusing already established connections and by using a pool of proxy threads.

For the PVM library, the runtime of message sending has been measured by *ping–pong tests* [47]. Here, a client process sends a message to a server process which on reception directly returns the message to its sender. The runtime is measured at the client side. Ping–pong tests cover the complete operation runtime including the local operations for sending and receiving as well as the network transmission. Additionally, ping–pong runtimes are directly comparable to Objective Linda's operations on remote object spaces which consist of request/reply pairs. Measurements with multiple concurrent (single–threaded) PVM clients have been performed with a shared server process.

As shown in Table II, Objective Linda's operations take between two to five times as much time as a PVM ping–pong test. Although this is a significant overhead, both operation types are still within the same order of magnitude. As mentioned above, an optimized implementation of Objective Linda might perform much closer to the PVM

operations. The impact of these differences on the performance of parallel applications will be studied in the next subsection.

Table II: Operation runtime (msecs) on object spaces compared to PVM.

	local			remote			
threads	out	in	rd	out	in	rd	PVM
1	0.17	0.25	0.26	10.3	10.1	10.0	4.2
2	0.29	0.42	0.50	18.7	18.5	19.2	5.4
3	0.36	0.57	0.72	27.3	27.5	28.1	6.9
4	0.43	0.71	0.92	35.9	35.1	35.9	10.2
5	0.69	0.86	1.13	44.2	47.1	45.2	10.9

6.2 Runtime Measurements of Parallel Applications

The parallel runtime behaviour can hardly be reflected by a coordination model, because such behaviour is not related to the communication (data) itself but to dynamic properties like communication frequencies. One exception is the introduction of bulky operations that allow to store and retrieve several objects at a time and consequently reduce parts of the communication overhead by minimizing the number of necessary communication operations [21, 52].

Such communication properties belong to the parallel algorithm itself and will hence influence any implementation regardless of the underlying communication platform or coordination model. The implementation of identical parallel algorithms based on different communication platforms consequently allows to compare their runtime performance with each other. Such comparisons will be provided for the example applications discussed below. In particular, Objective Linda's runtime behaviour will be compared to PVM, version 3.3.11. The computing platform is the same as the one used for measuring the runtimes of the operations. Since all workstations used for the measurements are of equal computing power within an interval of 3%, all machines will be treated as of equal computing power in order to simplify the following analyses. The first application studied is the parallel visualization of the Mandelbrot set – an example of a manager/worker computation. The second application is a parallel algorithm for the knapsack problem – it represents a divide-and-conquer computation. In both cases, the coordination patterns described earlier have been used as the basis for writing the code.

6.2.1 Mandelbrot Set

A parallel visualization program for the Mandelbrot set has been implemented with Objective Linda following the design outlined in Section 4.1. Analogously, a PVM implementation relies on the same manager/worker scheme. With PVM, the manager object must additionally implement a worker pool because it has to identify workers in order to communicate with them. Besides the additional implementation of a worker pool and

the exchange of communication operations, both versions of the Mandelbrot program use the same program code and have been compiled with the same compiler using the same compiler options. Of course, runtime measurements have been performed with the same machines, using an identical data set. The image size was 1000 × 1000 pixels.

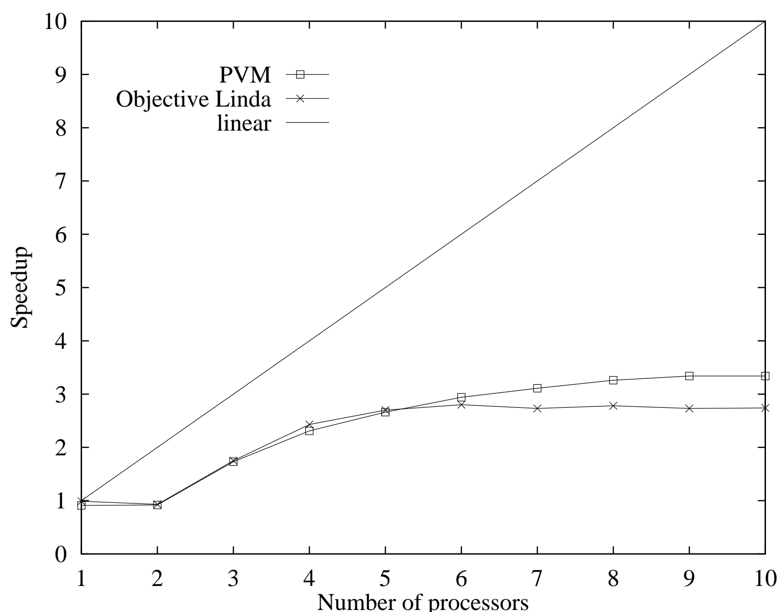

Figure 16: Application speedup of the Mandelbrot program.

Figure 16 shows the speedup of parallel runs with both platforms, compared to a sequential program which has also been derived from the original source code. Measurements have been performed for one up to ten processors. With one processor, the manager and one worker both run on the same machine. With two processors and more, one workstation runs the manager while each other runs a worker object.

The most obvious result is that Objective Linda and PVM yield almost identical results. Both platforms achieve a maximal speedup of about three when using ten machines. This rather poor speedup is due to the large amount of image data that has to be transferred from the workers to the manager. Hence, although embarrassingly parallel, the visualization of the Mandelbrot set achieves its maximal speedup already with relatively few processors.

The Mandelbrot algorithm was chosen in order to implement an embarrassingly par-

allel problem while simultaneously exploiting task parallelism. Its large amount of image data allows to validate Objective Linda's approach to object migration: Although Objective Linda's *imaging streams* use a rather "verbose" data format, they yield almost identical results for up to six processors compared to PVM. Only with more processors, Objective Linda's communication overhead causes measurably slower application behaviour. This result has to be evaluated under consideration of the fact that Objective Linda's prototype implementation has not at all been optimized. Currently, every object is transferred using an ASCII representation of all data items which was introduced in order to avoid problems with conversions between different binary formats.

6.2.2 Knapsack Problem

A parallel solver for the knapsack problem has been implemented with Objective Linda following the design outlined in Section 4.2. A PVM implementation has been derived from the Objective Linda source code. Here, new tasks are dynamically spawned as heavy–weight UNIX processes. Task and subtask communicate via their task id's they become aware of while spawning the new task.

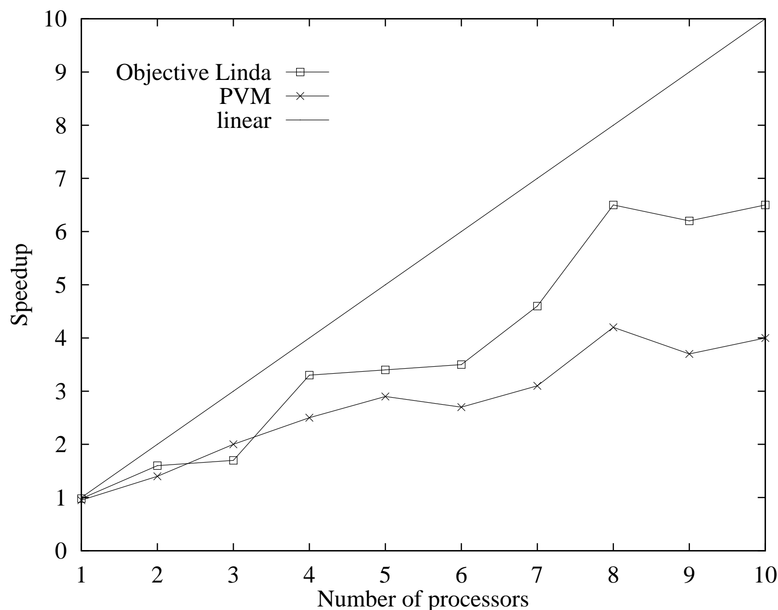

Figure 17: Application speedup of the knapsack program.

Figure 17 shows the speedup of parallel runs with both platforms, compared to a sequential program which has also been derived from the original source code. Measurements have been performed for one up to ten processors. Communication latency plays only a minor role concerning application speed of the knapsack programs, because the problem needs exponential computation effort while only a few data items have to be transmitted. Instead, dynamic task assignment dominates the achievable speedup values. Figure 17 shows that Objective Linda yields significantly better runtime performance than PVM. This is because PVM fails to efficiently distribute dynamically created processes among the available machines. A similar effect has already been reported in [10], where SCA Linda also performed better than PVM because of better task assignment strategies.

Both implementations of the knapsack algorithm relied on the task assignment facilities of the respective platforms. A parallel knapsack algorithm based on PVM could probably achieve speedup values comparable to Objective Linda if task assignment would be performed by the application itself e. g. in form of a worker pool to which tasks may be assigned.

7 Conclusions

In this paper we have presented Objective Linda, a coordination model based on the general principles of the original Linda approach, developed in order to enable object-oriented parallel programming in networked computing resources. The proposed model incorporates (a) object-orientation as the primary design methodology yielding composable, self–contained entities which are protected by the interfaces of their corresponding abstract data types, (b) generative communication as a means to uncouple communication partners from each other in order to enable the coordination of dynamically changing configurations, (c) homogeneity through its simple, uniform and easy to understand programming model, and (d) hierarchical abstractions to allow to have different views on configurations with different granularities of concurrently operating agents.

The major benefits of Objective Linda are:

- Its object model is thoroughly defined and based on abstract types and a corresponding subtyping relation. This language–independent object model allows the interaction of objects and systems implemented in different programming languages.
- Matching of passive objects and evaluation of active objects is based on object types and interface operations only.
- Operations on object spaces are enhanced by dealing with multisets of objects and by the introduction of a timeout parameter for behaviour adaptation.
- Dynamic composition of configurations is based on object matching and generative communication. This leads to a model of multiple object spaces allowing to build hierarchical abstractions of related configurations.
- Migration of objects between distributed platforms is based on object types only.

It was shown that Objective Linda could orthogonally be used in conjunction with any existing object-oriented programming language, such that the development of parallel applications becomes a straightforward matter. Objective Linda's notion of active objects and dynamically configurable object spaces allowed the construction of generally useful higher-level coordination patterns and components which could be easily adapted to the desired functionality of a given application scenario. A prototypical implementation for the C++ language to be used on clusters of workstations was discussed, and its runtime efficiency was measured. Furthermore, performance measurements of two example applications were performed and compared to equivalent implementations using the PVM message passing library. The Objective Linda programs showed runtime performance comparable (or even better) to their PVM counterparts.

There are several areas for future research. First, the existing implementation should be optimized in order to support large applications. Possible optimizations are related to a more compact format for the transmission of objects, the reuse of connections between processes, and the introduction of a thread pool in order to reduce the overheads of frequent thread creation. Another area for optimization could be an improved scheme for

mutual exclusion of operations on object spaces, allowing more than one thread at a time to access a certain object space.

Based on such an improved implementation, the Objective Linda model should be evaluated with real–world applications. Such an evaluation should answer questions like whether Objective Linda meets its requirements for dynamically changing configurations and whether the proposed set of operations proves to be useful in real applications.

Analogously, the application of Objective Linda to (wide area) distributed systems should be evaluated with example applications of larger size. It appears to be very interesting to see whether Objective Linda can be successfully employed in this application area.

Related to the area of (open) distributed systems, Objective Linda's support for integrating heterogeneous environments could be evaluated by the introduction of bindings to further programming languages. Of special interest seem to be bindings to Java and Smalltalk. A Java binding would open the Internet world whereas a Smalltalk binding could allow the verification of Objective Linda's concepts in the area of pure object–oriented languages.

Finally, the idea of developing reusable coordination patterns for parallel computing based on Objective Linda should be explored further. Examples for such patterns might be synchronizers, sequencers, iterators, barriers, and other kinds of active objects with reusable behaviour. The construction of a "toolkit" of reusable coordination components should be the long–term goal of this research.

Acknowledgements

We would like to thank Henri Bal and Paolo Ciancarini for their very useful and thorough comments on draft versions of the paper. Tom Holvoet beneficially influenced Objective Linda's mechanism for dynamic composition. We also appreciate Antony Rowstron's help giving shape to our prototype implementation.

References

[1] B. Achauer. The DOWL Distributed Object–Oriented Language. *Commun. ACM*, 36(9):48–55, 1993.

[2] G. Agha, P. Wegner, and A. Yonezawa, editors. *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 Agha et al. [2], pages 257–280.

[4] J.-M. Andreoli and R. Pareschi. Communication as Fair Distribution of Knowledge. In *Proc. of OOPSLA '91*, pages 212–229, Phoenix, Az., 1991.

[5] M. Arango, D. Berndt, N. Carriero, and D. Gilmore. Adventures with Network Linda. *Supercomputer Review*, 10(3):42–46, 1990.

[6] O. Arndt, B. Freisleben, T. Kielmann, and F. Thilo. Dynamic Load Distribution with the Winner System. In *Proc. Workshop Anwendungsbezogene Lastverteilung (ALV'98)*, pages 77 – 88, Munich, Germany, 1998. Published as Technical Report TUM-I9806, SFB 342/01/98 A, Technische Universität München.

[7] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[8] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A Language for Parallel Programming of Distributed Systems. *IEEE Trans. Softw. Eng.*, 18(3):190–205, 1992.

[9] M. Banville. Sonia: an Adaptation of Linda for Coordination of Activities in Organizations. In Ciancarini and Hankin [17], pages 57–74. Proc. COORDINATION'96.

[10] R. Baraglia, D. Laforenza, and R. Perego. Programming a Workstation Cluster with PVM and Linda: a Qualitative and Quantitative Comparison. In *Proc. of the AICA'93 International Section: Parallel and Distributed Architectures and Algorithms*, pages 101–114, 1993.

[11] G. Booch. *Object Oriented Design with Applications.* Benjamin/Cummings, 1991.

[12] B. D. Bowen. Tuplex: A C–Linda Implementation for SUN, Next and Mac. *Sun–World*, pages 30–34, 1992.

[13] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive Parallelism and Piranha. *IEEE Computer*, 28(1):40–49, 1995.

[14] N. Carriero, D. Gelernter, and L. 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.

[15] D. T. Chang. CORAL: A Concurrent Object–Oriented System for Constructing and Executing Sequential, Parallel, and Distributed Applications. In G. Agha, C. Hewitt, P. Wegner, and A. Yonezawa, editors, *Proc. of the ECOOP–OOPSLA Workshop on Object–Based Concurrent Programming*, pages 26–30, Ottawa, Canada, 1990. Published as OOPS Messenger2(2), 1991.

[16] P. Ciancarini. Coordination Models and Languages as Software Integrators. *ACM Comput. Surv.*, 28(2):300–302, 1996.

[17] P. Ciancarini and C. Hankin, editors. *Coordination Languages and Models*, number 1061 in Lecture Notes in Computer Science, Cesena, Italy, 1996. Springer. Proc. COORDINATION'96.

[18] P. Ciancarini and D. Rossi. Jada: Coordination and Communication for Java Agents. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems Workshop, MOS'96*, number 1222 in Lecture Notes in Computer Science, pages 213–226, Linz, Austria, 1996. Springer.

[19] J. Dongarra, R. Pozo, and D. Walker. ScaLAPACK++: An object–oriented linear algebra library for scalable systems. In *Proc. Scalable Parallel Libraries Conference*, pages 216–223. IEEE, 1993.

[20] I. Foster. *Designing and Building Parallel Programs.* Addison–Wesley, 1995.

[21] B. Freisleben, D. Hartmann, and T. Kielmann. Parallel Raytracing: A Case Study on Partitioning and Scheduling on Workstation Clusters. In R. H. Sprague Jr., editor, *Proc. of the Thirtieth Annual Hawaii International Conference on System Sciences*, volume 1, pages 596–605, Wailea, Hawai'i, USA, 1997. IEEE.

[22] B. Freisleben and T. Kielmann. Automated Transformation of Sequential Divide–and–Conquer Algorithms into Parallel Programs. *Computers and Artificial Intelligence*, 14(6):579–596, 1995.

[23] B. Freisleben and T. Kielmann. Coordination Patterns for Parallel Computing. In D. Garlan and D. L. Métayer, editors, *Coordination Languages and Models*, number 1282 in Lecture Notes in Computer Science, pages 414–417, Berlin, Germany, 1997. Springer. Proc. COORDINATION'97.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object–Oriented Software.* Addison Wesley, 1994.

[25] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A Users Guide and Tutorial for Network Parallel Computing.* MIT Press, 1994.

[26] D. Gelernter. *An Integrated Microcomputer Network for Experiments in Distributed Programming.* PhD dissertation, State University of New York at Stony Brook, 1982.

[27] D. Gelernter. Generative Communication in Linda. *ACM Trans. Prog. Lang. Syst.*, 7(1):80–112, 1985.

[28] D. 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.

[29] D. Gelernter and N. Carriero. Coordination Languages and their Significance. *Commun. ACM*, 35(2):96–107, 1992.

[30] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message–Passing Interface.* MIT Press, 1994.

[31] R. H. Halstead Jr. Parallel Symbolic Computing. *IEEE Computer*, 19(8):35–43, 1986.

[32] T. Holvoet and T. Kielmann. Behaviour Specification of Parallel Active Objects. *Parallel Computing*, 1998. To appear in special issue on coordination languages and systems.

[33] S. Hupfer, D. Kaminsky, N. Carriero, and D. Gelernter. Coordination Applications of Linda. In J. P. Banâtre and D. L. Métayer, editors, *Research Directions in High–Level Parallel Programming Languages*, number 574 in Lecture Notes in Computer Science, pages 187–194. Springer, 1991.

[34] K. Hwang. *Advanced Computer Architecture.* McGraw-Hill Book Co., 1993.

[35] International Business Machines Corporation and SunSoft, Inc. Object Externalization Service. OMG TC Document Number 94.6.21, 1994.

[36] Y. Ishikawa, editor. *Proc. Scientific computing in object-oriented parallel environments: first International Conference, ISCOPE 97, Marina del Rey, California*, number 1343 in Lecture Notes in Computer Science. Springer, 1997.

[37] R. Jellinghaus. Eiffel Linda: an Object–Oriented Linda Dialect. *SIGPLAN Notices*, 25(12):70–84, 1990.

[38] K. 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.

[39] T. Kielmann. *Objective Linda: A Coordination Model for Object–Oriented Parallel Programming.* PhD dissertation, Dept. of Electrical Engineering and Computer Science, University of Siegen, Germany, 1997. Published by Shaker Verlag, Aachen, Germany.

[40] O. Krone and M. 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.

[41] T. Kühne. Parameterization versus Inheritance. In C. Mingins and B. Meyer, editors, *Technology of Object–Oriented Languages and Systems: TOOLS 15*, pages 235–245, Melbourne, Australia, 1994. Prentice Hall.

[42] B. B. Mandelbrot. *The Fractal Geometry of Nature.* W. H. Freeman and Co., San Francisco, CA, 1982.

[43] S. Matsuoka and S. 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.

[44] S. Matsuoka and A. Yonezawa. Analysis of Inheritance Anomaly in Object–Oriented Concurrent Programming Languages. In Agha et al. [2], pages 107–150.

[45] B. Meyer. *Object–Oriented Software Construction.* Prentice Hall, New York, 1988.

[46] B. Meyer. *Eiffel the Language.* Prentice Hall, 1992.

[47] N. Natawut and L. M. Ni. Performance Evaluation of some MPI Implementations on Workstation Clusters. In *Proc. Scalable Parallel Libraries Conference (SPLC94)*, pages 98–108, 1994.

[48] O. Nierstrasz and D. Tsichritzis, editors. *Object–Oriented Software Composition.* Prentice Hall, 1995.

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

[50] H. Pedersen, D. Gehrke, M. G. Jensen, J. N. Larsen, and L. R. Skyum. *Guides to C++Linda*. Aalborg University, Dept. of Mathematics and Computer Science, Inst. for Electronic Systems, Fredrik Bajers Vej 7E, DK–9220 Aalborg Ø, Denmark, 1990.

[51] A. 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.

[52] A. Rowstron. *Bulk Primitives in Linda Run–Time Systems*. PhD thesis, Department of Computer Science, University of York, UK, 1997.

[53] A. Rowstron and A. Wood. An Efficient Distributed Tuple Space Implementation for Networks of Workstations. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Proc. EuroPar'96*, number 1123 in Lecture Notes in Computer Science, pages 510–513. Springer, 1996.

[54] A. Rowstron and A. Wood. Solving the Linda Multiple rd Problem. In Ciancarini and Hankin [17], pages 357–367. Proc. COORDINATION'96.

[55] D. C. Schmidt. The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software. In *Proceedings of the* 12th *Annual Sun Users Group Conference*, pages 214–225, San Francisco, CA, 1994. SUG.

[56] D. C. Schmidt. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In J. O. Coplien and D. C. Schmidt, editors, *Pattern Languages of Program Design*, pages 529–545. Addison-Wesley, Reading, MA, 1995.

[57] D. C. Schmidt. A Family of Design Patterns for Application–Level Gateways. *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, 2(1):15–30, 1996.

[58] D. C. Schmidt. A Family of Design Patterns For Flexibly Configuring Network Services in Distributed Systems. In *International Conference on Configurable Distributed Systems*, pages 124–135, 1996.

[59] D. C. Schmidt. Acceptor and Connector: Design Patterns for Initializing Communication Services. In R. Martin, F. Buschmann, and D. Riehle, editors, *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1997. To appear.

[60] Scientific Computing Associates. *Linda: User's Guide and Reference Manual*, 1995.

[61] R. Seyfarth, J. Bickham, and S. Arumugham. *Glenda Installation and Use*. University of Southern Mississippi, Hattiesburg, MS, 1994.

[62] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[63] S. R. Thatté. Object Imaging. In W. Olthoff, editor, *Proc. ECOOP'95*, number 952 in Lecture Notes in Computer Science, pages 52–76, Århus, Denmark, 1995. Springer.

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

[65] P. Wegner and S. B. Zdonik. Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. In S. Gjessing and K. Nygaard, editors, *Proc. ECOOP'88*, number 322 in Lecture Notes in Computer Science, pages 55–77, Oslo, Norway, 1988. Springer.

List of Figures

1 Configurations are composed of agents and object spaces. 14

2 The predicate match of a C++ class Fork. 17

3 Code fragment by which a philosopher gets a Seat and the corresponding Forks. 20

4 Code fragment by which a philosopher attaches to a restaurant. 25

5 Code fragments by which a waiter exposes and later hides its restaurant. . 26

6 Booch Diagram for Manager/Worker Patterns 34

7 A C++ class Manager 37

8 A C++ class Worker 38

9 A C++ class Mandelbrot_Manager 40

10 A C++ class Mandelbrot_Worker 41

11 Booch Diagram for the Divide–and–Conquer Pattern 42

12 A C++ class Divide_and_Conquer 44

13 A C++ class Future 45

14 A C++ class Packer 46

15 A running Objective Linda configuration. 57

16 Application speedup of the Mandelbrot program. 64

17 Application speedup of the knapsack program. 65

Biographies:

Bernd Freisleben is currently a Professor for Computer Science in the Department of Electrical Engineering and Computer Science at the University of Siegen, Germany. He received his Master's degree in computer science from Pennsylvania State University, USA, in 1981, and his Ph.D. degree in computer science from Darmstadt University of Technology, Germany, in 1985. His research interests include parallel programming, distributed systems, computer networks, and soft computing.

Thilo Kielmann is currently a post–doctorate researcher in the Department of Mathematics and Computer Science at the Vrije Universiteit Amsterdam, The Netherlands. He received his diploma degree in computer science from Darmstadt University of Technology, Germany, in 1992, and his Ph.D. degree in computer science from the University of Siegen, Germany, in 1997. His research interests include parallel programming, object–oriented systems, computer networks, and wide–area distributed computing environments.