The *Object Space* Approach: Decoupled Communication in C++ [1]

Andreas Polze

Institut für Informatik
Fachbereich Mathematik
Freie Universität Berlin
Takustr. 9
14195 Berlin, FR Germany
e-mail: polze@inf.fu-berlin.de

Abstract

We present the *Object Space* approach to distributed computation. *Object Space* allows for decoupled communication between program components by providing a shared data space of objects. This style of communication was inspired by Linda. The *Object Space* approach extends a sequential language (C++ in our case) with coordination and communication primitives. It integrates inheritance into associative addressing as known from Linda and facilitates passing of arbitrary objects between program components. Furthermore we introduce the notion of application-specific matching functions. A prototype for *Object Space* has been implemented in C++ under UNIX. A distributed phonebook and a scenario built around a time server and its clients serve as examples to demonstrate ideas and use of the concepts developed.

1. Introduction

In this paper we develop a model to integrate object-oriented programming paradigms like inheritance and data encapsulation with decoupled communication between components of a distributed application as known from Linda [Gelernter 85]. Within our model processes may exchange arbitrary objects. When matching one object against another inheritance will be taken into account by an associative addressing scheme. Furthermore the notion of application-specific *matching* functions allows access to more then one object within a single operation. This extends the coordination model of Linda [Gelernter 92].

We define the *Object Space Language* (*OSL*) to express communication and synchronization constructs and discuss how this language can be embedded into the sequential object-oriented language C++. Furthermore we briefly describe a prototype implementation of *Object Space* in C++ under UNIX. A distributed phonebook example and a distributed time service example show the use of ideas and concepts developed. The final section of the paper presents conclusions and an outline of future work.

[1] appeared in the Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS) USA'93, Santa Barbara, August 1993.

2. The *Object Space* Approach

Object Space supports communication and synchronization between components of a distributed application. All components (clients) have access to a shared associative data store known as the *Object Space*. Every client may write objects into the store which can subsequently be read by others. In order to read an object a component presents a template which is matched against the objects. If no matching object can be found within the *Object Space* then the read operation blocks. This way objects may be passed from one component to another. *Object Space* itself is implemented in a distributed fashion, employing *Object Space Manager* processes in different nodes of a network.

This style of communication is called decoupled. Neither does the sender of an object know its receiver nor vice versa. If there exist several components trying to read one object it is nondeterministic which of them will succeed. This style of decoupled communication is well suited for scalable distributed computing within a network of workstations. Decoupled communication over a tuple space has been introduced by Linda. *Object Space* extends the tuple space in several ways. It integrates coordination constructs (communication and synchronization) into the C++ language [2].

Object Space approach provides the usage of arbitrary C++ objects as messages. We define the *Object Space Language* (*OSL*) which gives a notion for describing *Object Space* operations. Furthermore we show how objects are represented within *Object Space*.

2.1. The *Object Space Language*

The *Object Space Language* (*OSL*) defines how communication and synchronization between program components via *Object Space* may be specified. Therefore it is a *coordination language* in the sense of [Gelernter 92] and has to be embedded into a sequential *computational language*, C++ in our case. We implemented the functionality of *OSL* in a C++ library (`libos++`). A preprocessor translates coordination constructs written in *OSL* into creation of C++ objects and calls to that library. Besides using *OSL* a programmer may call functions from `libos++` directly.

When matching an object against another we need some kind of structure information about them. Since C++ does not provide runtime type information we need a special mechanism to obtain this structure information. A metadata system [Pauw 92] could provide a solution. Certainly object-oriented approaches which consider classes as objects (e.g., Smalltalk) may take advantage from the fact, that class objects are accessible at runtime. So several problems do not appear when implementing decoupled communication on top of Smalltalk [Matsuoka 88].

When designing `libos++`, we have choosen to force the programmer to supply structure information by enumerating an object's data components using calls to special functions from `libos++`. To ensure identical representation of all instances of a class within *Object Space*, this structure information has to be placed into the special member function `description()` inside a C++ class definition. Besides an enumeration of all components of an object at runtime, we need a means to distinguish between those components being significant for associative addressing and those being insignificant. In correspondence to *actual* and *formal* data components as described below the special functions `actual()` and `formal()` serve this purpose. The introduction of *OSL* allows to hide these implementation-dependent details from the programmer. A preprocessor can automatically generate calls to the special functions mentioned above using information contained in OSL constructs. Here, we use *OSL* constructs to keep our examples short and understandable.

We have implemented a distributed prototype for *Object Space* in C++ under UNIX which will be described below.

Figure 1 shows the definition of the *Object Space Language* using extended Backus-Naur notation.

[2] Two other approaches which combine object-oriented programming languages with a Linda-like communication style may be found in [Matsuoka 88] and [Jellinhaus 90].

OSL-Program	::=	{ object_space_op | local_computation }.
object_space_op	::=	object "`.`" op_spec.
op_spec	::=	**`rd`** | **`in`** | **`out`** | **`eval`** .
object	::=	[objID]"`(`" class ["`:`" base] "`,`" data_comp { "`,`" data_comp } "`)`".
data_comp	::=	formal | actual.
formal	::=	[type] "`?`" name.
actual	::=	[type] name ["`=`" value [matching_fct]].
matching_fct	::=	**`delta`**"`(`" diff "`)`" | **`or`** value { `or` value }.

figure 1: *Object Space Language*

A program in *OSL* is a sequence of *Object Space* operations and local sequential operations. The notion of *local_computation*s is defined by the syntactic rules of C++, *OSL*'s host language.

In *OSL* objects may appear without a name. It is useful to leave out the object's name if that object is used by *Object Space* operations only (e.g. for synchronization purposes).

objID is the identifier for an object. Its class is described by *class*. The class has to be defined as an ordinary C++ class somewhere. Furthermore the base class of an object's class may be specified by *base*. If *base* is omitted the `objsp_comm` base class is used as default.

In our implementation of the *Object Space* approach class names are mapped to unique integer classIDs by a distributed type service[3]. Inheritance information is expressed in a distributed fashion as a relation over classIDs. The associative addressing scheme takes this information into account when performing *Object Space* operations.

name denotes a data component with type

type ∈ `Type = {char, int, str, ...}`

of an object. The set `Type` for data components of objects in *Object Space* contains arbitrary C++ classes besides the C++ builtin types. Although the type of a data component is already defined in the declaration of a class, we allow explicit specification of that type in *OSL*. The explicitly specified type overrides a type known from the class declaration during matching in *Object Space*.

The value of a data component may be specified by use of *value*. The notation *?name* describes a *formal* component. Its value is ⊥. The type of a *formal* component but not its value is taken into account by the associative addressing scheme within *Object Space*.

We give an example for operations written in *OSL*. Let `intC` be a C++ class containing an integer component `c` and a component `s` of class `string` carrying the name of an object. Furthermore we assume that member function `show_s` prints component `s` of an `intC` object:

```
(intC, c=1, s="object_1").out; (1)
(intC, c=2, s="object_2").out;
(intC, c=3, s="object_3").out;

intC o;                        (2)
o(intC, c=1 or 3, ?s).rd;      (3)

o.show_s();                    (4)
```

This example demonstrates how three objects are written into *Object Space*. Afterwards one object is read. The *matching* function **`or`** causes the object written first or last to be read nondeterministically. So within our example either the string `object_1` or `object_3` is printed out.

[3] A much more complex technique to obtain runtime type information from C++ classes is described in [Pauw 92].

Now lets briefly discuss the C++ code corresponding to the *OSL* example above. The lines (1) to (4) are transformed as follows:

```
{ intC _o;
  _o.c = 1; _o.s="object_1";
  _o.actual(_o.c);
  _o.actual(_o.s);
  _o.out();
}                                   }(1)

intC o;                             }(2)

o.actual(o.c,mfct_or(1,3));
o.formal(o.s);                      }(3)
o.rd();

o.show_s();                         }(4)
```

Line (1) is translated into the creation of an anonymous object `_o` of class `intC`. Proper values are assigned to its data components and both the integer component and the string component are marked as *actual*s. They appear as arguments in calls to `actual()`. The object is written into *Object Space* by a call to `out()`.

The creation of object `o` in line (2) remains unchanged. But the read operation in line (3) has been transformed into a sequence of function calls marking components as *actual* and *formal*, respectively, and a call to `rd()`. Function `actual()` may receive a descriptor for a matching function (**`or`**, in our case) as a second argument. This argument has a default value as described below.

Finally, line (4) remains completely unchanged. The string component of object `o` is printed out.

In the following sections we use notation from *OSL* in our examples. Lets briefly summarize the rules for objects within *OSL*:

- Objects are declared by their first appearance. They may be denoted by a name.
- A component of an object may be referred to by its name qualified with the name of the object.
- Objects possess the four operations **`rd`**, **`in`**, **`out`** and **`eval`**.
- Since data components only are considered by the associative addressing scheme within *Object Space*, objects are described by enumeration of their data components.
- Data components are denoted by their name and type, an optional value and an optional *matching* function.
- A data component may be *formal*. The name of a *formal* is preceded by a question mark (?).

2.2. Associative Addressing

In contrast to *OSL*, objects have no idea about their identity once put into *Object Space*. Within *Object Space* an object is represented as the aggregate of class-specific type information and of its visible data components. Inheritance is expressed as a relation over the type information and is taken into account when performing *Object Space* operations. Therefore an object of a derived class may match an object of its base class.

Arbitrary C++ objects may be used as messages within *Object Space*. When using *Object Space* it is not necessary to convert logically coherent data structures into tuple structures as in Linda to transmit them from one component to another. The error prone conversion of data before and after each communication operation is avoided. During *Object Space* operations an object may be stored or retrieved as a whole only. So each attempt to access an object's data component must pass the access control mechanisms known from C++. Thus concepts of object-oriented programming like data encapsulation and inheritance are effective for communication between program components.

Read operations over *Object Space* carry a template as argument. A template is an object with (perhaps) *formal* data components. It describes a set M_{match} of matching objects. An element of this set is used to fill the *formal* data components of the template with values. In the language *OSL* templates may be written as objects.

Lets now consider under which circumstances an object matches a template. Templates may contain

matching functions. These boolean functions influence the construction of M_{match}. An object matches a template if:

— it has an identical number of data components and corresponding data components match.

— it contains more data components then the template and its class is derived from the template's class and corresponding data components match.

Two data components match if they are of the same type and the template's *matching* function returns true when applied on an object's data component or if one of these components is *formal*.

Now we can define a relation *match* which says whether data components match. For each data component of a template (m_i) and the corresponding component of an object (o_i) holds:

$$(m_i, o_i) \in \texttt{match} \longleftrightarrow \begin{cases} m_i = \perp \text{ or} \\ o_i = \perp \text{ or} \\ \texttt{match\_fct}(o_i, m_i) \end{cases}$$

Now we have to define the *matching* functions available within *Object Space* approach. While an object-oriented notation for these *matching* functions is used within *OSL* we use a functional notation here. Within the object-oriented notation a data component of an object stored in *Object Space* is an invisible argument to each of these functions. In the functional notation we write this argument first and separate it from the visible arguments by a semicolon.

An operation `equal` is used in the definition of each *matching* function. This operation is defined for all types used within *Object Space*. The following *matching* functions are defined:

Let `m` be a template's data component,
let m_i for $1 \le i \le n$ be alternative values of a template's data component,
let `o` be a component of an object stored within *Object Space*:

$$\textbf{or}(o; m_1, \ldots, m_n) = \begin{cases} \texttt{true}, \exists\, i\text{: o equal } m_i \\ \texttt{false} \text{ else} \end{cases}$$

$$\texttt{default}(o; m) = \begin{cases} \texttt{true} & : m \texttt{ equal } o \\ \texttt{false} & \text{else} \end{cases}$$

Let `m` be a numerical[4] data component of a template,
let `o` be a numerical component of an object stored within *Object Space*,
let `d` be a numerical value:

$$\textbf{delta(}o; m, d\textbf{)} = \begin{cases} \texttt{true} & : o - d \le m \le o + d \\ \texttt{false} & \text{else} \end{cases}$$

The *matching* function `default` is used by the associative addressing scheme if no *matching* function is explicitly given for a template's data component.

Four operations are defined for dealing with *Object Space*:

- **`out`** writes an object into the *Object Space*.
- **`in`** and **`rd`** carry a template as argument. Both operations retrieve a matching object from the *Object Space* and store its values in the template; they block until a matching object is found. **`in`** removes the object from *Object Space*. An object matches a template depending on its class and the values of its data components. If the template has less data components then a retrieved object (i.e., object's class is derived from template's class), extra components are silently discarded.
- **`eval`** creates a new UNIX process either locally or remotely. It carries the command line arguments for a remote process or the address and arguments of a function which has to be executed locally as parameters.

The *Object Space* approach has been implemented in C++ under UNIX. Within that implementation access to *Object Space* is available trough inheritance from two communication base classes: `objsp_comm` and `objsp_proc`.

[4] Lets assume that an ordering relation, addition and subtraction are defined for each numerical type.

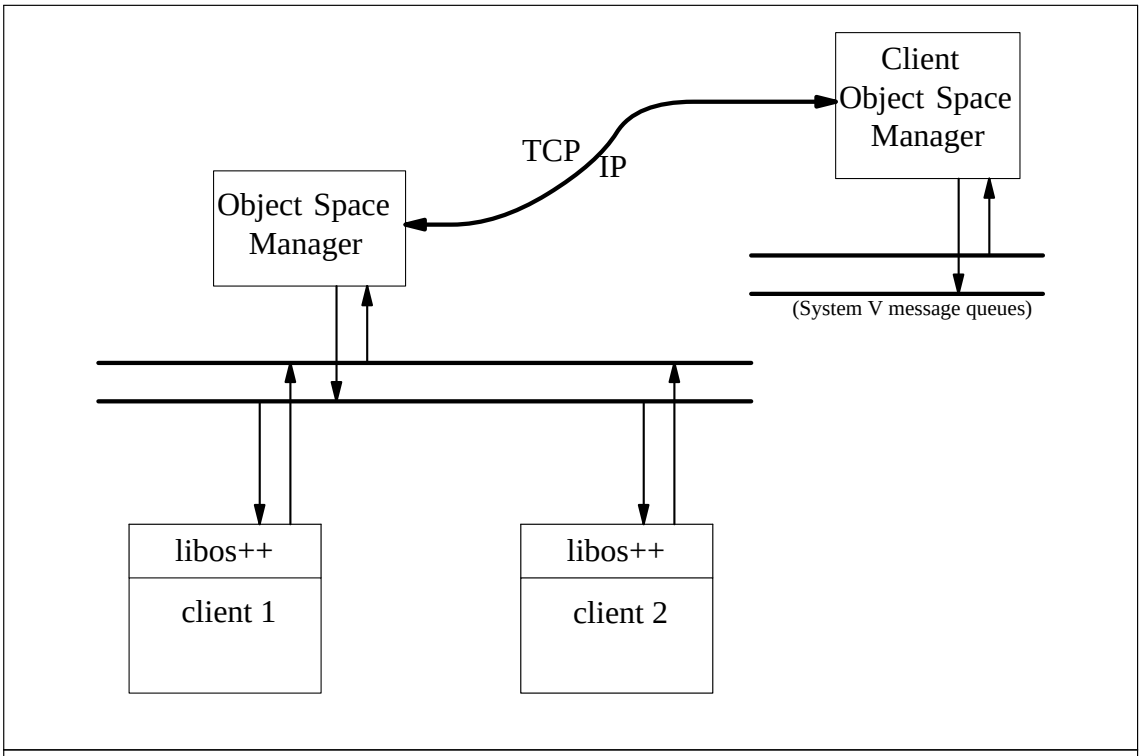

figure 2: *distributed Object Space Manager processes*

The operations **`out`**, **`in`** and **`rd`** are implemented within `objsp_comm` class. Furthermore this class provides access to the distributed type service. Client classes, which want to use *Object Space* have to supply their class name to a constructor from `objsp_comm` class. Versions of functions `actual()` and `formal()` as used in code generated from *OSL* constructs are accessible trough class `objsp_comm`, too. These functions are overloaded and allow to enumerate data components of several types.

Operation **`eval`** is implemented by class `objsp_proc`. An instance of that class may initiate process creation, either locally or remotely. Afterwards it contains a unique identifier for the newly created process, composed from the process' nodename and its UNIX process identifier.

3. Implementation Issues

The *Object Space* approach supports the idea of a distributed shared memory. Our prototype implementation in C++ is based on interprocess communication mechanisms available within the UNIX operating system.

A scenario around *Object Space* includes several processes each of them storing objects into and retrieving them from *Object Space*. Some kind of storage medium is needed to keep the data which make up an object available. In our implementation these storage media are provided by a separate UNIX process, the *Object Space Manager*. It uses virtual memory to store the objects. Furthermore this process performs matching between templates and objects. In addition to the *Object Space Manager* several *Client Object Space Manager* processes may exist. These processes do not store any object at all. They just forward operation requests to the *Object Space Manager* process using a special protocol on top of TCP/IP. Thus crossing machine boundaries is possible.

Client processes communicate with a local *Object Space Manager* process. They use message queues[5] as communication mechanism. The library `libos++` transforms calls to *Object Space* operations into a proper sequence of `msgsnd()` and `msgrcv()` system calls on these message

[5] Message queues have been introduced with UNIX System V but actually they are available with nearly every UNIX System.

queues. Client programs may access the operations from `libos++` trough inheritance from classes `objsp_comm` and `objsp_proc`. An *Object Space Manager* handles two message queues, a read queue and a write queue. Each client process contacts the read queue of a local *Object Space Manager* to initiate operations on *Object Space*. Results of operations are returned on the corresponding write queue.

The operation **`out`** as available to client programs from the communication base class `objsp_comm` initiates a write-operation on the read-queue of a local *Object Space Manager*. It transmits an *actual* object and doesn't return a result. On the other hand the operations **`rd`** and **`in`**, called on client side, write a template object onto the read-queue of a local *Object Space Manager* and block until it returns a matching object. The matching object is returned on the corresponding write-queue. Both read- and write-queue are multiplexed, thus allowing arbitrary many clients to communicate with the same *Object Space Manager* at once.

Matching between templates and *actual* objects is performed by the *Object Space Manager*. It implements matching strategies which correspond to *matching* functions available in the language *OSL*.

Object Space operations written in the language *OSL* are translated into C++ by a preprocessor. Currently this preprocessor is implemented by an *awk*-script. The C++ code generated by the preprocessor relies heavily on functionality available in `libos++`.

4. A Distributed Phonebook

Lets consider the following scenario: A small company's phonebook arranges selected information from private notebooks of the employees. Both, the phonebook and the notebooks are implemented in a distributed fashion, using *Object Space* as an object repository.

The phonebook may be accessed by an public phonebook agent, which retrieves an object of class `phonebook_entry` or one of its subclasses from *Object Space*. But in any case it obtains the `phonebook_entry` part of the object only.

A private notebook may be implemented by defining a class `notebook_entry` which inherits from `phonebook_entry` and extends it by additional data components. One of these component may be a password which is enforced to be never a *formal* (i.e. having no value). Then, these objects are accessible through *Object Space* read operations only if the password is known by a user. So a certain degree of privacy is guaranteed by the design of the `notebook_entry` class.

Lets consider several agents which communicate via *Object Space* and allow users to read phonebook entries and to store or remove notebook entries. These agents may contain the following lines of code:

```
class phonebook_entry:
          public objsp_comm {

public:
     char* name;
     char* number;

     ...
};

class notebook_entry:
          public phonebook_entry {

private:
     char* password;
public:

     ...
     // confidential information
     ...
};
```

We have shown some data components of the classes. Certainly they need several member functions, too. Note the inheritance relation between `phonebook_entry` and `objsp_comm` which means that instances from class `phonebook_entry` may access *Object Space*.

Now we describe some functionality of the agents using a syntax combined from C++ and *OSL*.

```
void store_notebook_entry(char* na,
          char* no, char* pwd, ...) {

     (notebook_entry:
      phonebook_entry,
      char* name=na,
      char* number=no,
      char* password=pwd,
      ...).out;
}

char* get_number( char* na ) {
     phonebook_entry phoneb;

     phoneb(phonebook_entry,
          char* name=na,
          char* ?number).rd;

     return phoneb.number;
}
```

One may notice that function `get_number` blocks if a user asks for a completely unknown name. To avoid this problem the design should be improved by using *Object Space* as communication means rather then as an object repository. So the objects should be stored by a special agent process which may answer ‘‘number unknown’’.

Nevertheless the distributed phonebook example shows how objects as data capsules are transmitted from one process to another. No explicit conversion into a tuple structure is needed. We take advantage from inheritance as implemented in the associative addressing scheme of *Object Space* when arranging the phonebook from several notebooks. The decoupled communication style allows for startup and shutdown of an arbitrary number of phonebook and notebook agents accessing the *Object Space*.

5. Using the right Abstractions

Whereas the previous section explained the role of inheritance in context of *Object Space* we now show how objects may be used to define a communication protocol for components of a distributed application. Our scenario consists of a *time server* which may answer requests from several *time client*s. Both share a protocol defined by class `timprot`. A *time client* is an instance of class `timc` which is derived from `timprot`. Similarly the *time server* is represented by an instance of class `tims` which is derived from `timprot`, too. Here we show classes `timprot` and `timc`. A sample `main` program implements the entire *time client*.

The C++ code for class `timprot` looks like this:

```
enum { REQUEST, ANSWER } operT;

class timprot: public objsp_comm {

protected:
     char * timestr;
     operT  op;
     int    magic;

public:

     // (1)
     timprot():objsp_comm("timprot") {
          timestr=NULL;
          op=REQUEST;
          magic=getpid();
     }

     // (2)
     virtual void description() {
          significant(timestr);
          significant(op);
          significant(magic);
     }
};
```

The `timprot` communication protocol used by *time server* and *time client*s includes a string representing the time, an operation specifier, and a magic number, which allows to distinguish between several `timprot` objects used by different *time client*s.

Line (1) defines the constructor of class `timeprot`. It provides class’ `objsp_comm`

constructor with a string argument describing the class name. Line (2) shows the special function `description()` which is used during *Object Space* operations to enumerate an object's components. We leave out a detailed discussion of function `significant()` which belongs to `libos++`, too.

Now, we present class `timc`, representing the client side of our scenario:

```
class timc: public timprot {

     // (3)
     void ask_server() {
          timprot tim_obj;

          tim_obj(timprot,
                timestr,
                int op=REQUEST,
                magic).out;

          tim_obj(timprot,
                ?timestr,
                int op=ANSWER,
                magic).in;
     }

public:

     timc() { ask_server();}
     char* show() { return timestr;}
};
```

Function `ask_server()` in line (3) implements the whole functionality of a *time client*. One should notice that C++ allows access to `timprot`'s `protected` data components within a member function of class `timprot` or one of its descendants only. Thus the two *OSL* constructs used to write a request to the *time server* and to read its answer are allowed to appear within member function `ask_server()`. So design of class `timc` and its base ensures consistency of `timprot` objects by implementing data capsules.

Finally, we show the `main` program of a very simple *time client*.

```
main() {
     timc client_obj;

     printf("the time is %s\n",
            client_obj.show());
}
```

Our small client/server scenario shows how *Object Space* supports the usage of proper abstractions when designing a distributed application. The C++ concepts of data encapsulation and access control are extended to the field of communication.

6. Conclusion

We have presented the *Object Space* approach to distributed computation. It provides decoupled communication to C++ programs thus integrating Linda-style communication with object-oriented programming paradigms. Besides application-specific *matching* functions the associative addressing scheme takes inheritance relations into account. This allows for efficient implementation of distributed algorithms and integrates object-oriented programming paradigms like data encapsulation and inheritance with the decoupled communication style.

Actually a prototype implementation of the *Object Space* approach employs a couple of distributed *Object Space Manager* processes. These processes communicate via TCP/IP. A client program may communicate with a local *Object Space Manager* process via UNIX message queues.

We have defined the *Object Space Language* (*OSL*) thus giving a means to specify operations over *Object Space*. A preprocessor allows for embedding of *OSL* into C++. Two classes from `libos++` provide an interface to *Object Space* on C++' language level. So *Object Space* may be used by deriving subclasses from these classes.

Currently we use the *Object Space* approach to implement a distributed parallel `make` program [Polze 93a] and to develop a mechanism for automatic generation of distributed prototype implementations from LOTOS specifications [Polze 93b].

References

[Gelernter 92] D.Gelernter, N.Carriero
Coordination Languages and their Significance
Communications of the ACM, Vol.35, No.2, Feb. 1992.

[Gelernter 91] D.Gelernter, N.Carriero
New Optimization Strategies for the Linda Pre-Compiler
Technical Report 91-13, Edinburgh Parallel Computing Centre, Greg Wilson (Editor).

[Gelernter 85] D.Gelernter
Generative communication in Linda
ACM Transactions on Programming Languages and Systems, 7(1):80-112, 1985.

[Jellinhaus 90] R.Jellinhaus
Eiffel Linda: An Object-Oriented Linda Dialect
ACM Sigplan Notices, Vol.25, No.3, December 1990.

[Matsuoka 88] S.Matsuoka, S.Kawai
Using Tuple Space Communication in Distributed Object-Oriented Languages
Proceedings of OOPSLA'88.

[Pauw 92] P.-A.Pauw, R.Werring, A.Jansen
An operational metadata system for C++
Proceedings of TOOLS USA'92.

[Polze 93a] A.Polze
Using the Object Space: A Distributed Parallel make
to appear in Proceedings of IEEE Workshop on Future Trends of Distributed Computing Systems, Lisbon, September 1993.

[Polze 93b] A.Polze, A.Vogel
Generierung verteilter Prototypimplementationen aus LOTOS-Spezifikationen nach dem Object Space-Ansatz
Report B-93-2, Freie Universität Berlin, Institut für Informatik, 1993.