Ada Binding to a Shared Object Layer

Johann Blieberger[1], Johann Klasek[1], and eva Kühn[2]

[1] Institute of Computer-Aided Automation, Technical University Vienna, Treitlstr. 1/1831, A-1040 Vienna, Austria ({blieb,jk}@auto.tuwien.ac.at)
[2] Institute of Computer Languages, Technical University Vienna, Argentinierstr. 8, A-1040 Vienna, Austria (eva@complang.tuwien.ac.at)

Abstract. CORSO, a coordination system for virtual shared memory, allows bindings to different programming languages. Currently C, C++, Java, VisualBasic, and Oracle's Developer2000 are supported. We implement an Ada binding to CORSO, thus opening the area of virtual shared memory to the Ada world. Our Ada CORSO binding enhances Ada with transaction-oriented, fault-tolerant, distributed objects in a straight-forward way without having to extend the Ada language.

1 A Layered Approach

In distributed and heterogeneous environments some technique is desirable to shield the attributes of distributed objects like location, replication, representation and persistency from the programmer. Different approaches exist and the relation between them points to some kind of orthogonality. The most common pattern seem to be the message passing versus virtual shared memory (VSM) paradigm.

VSM neither intends to replace nor to exclude message passing architectures like CORBA or DCOM. In contrast, VSM should be seen as an additional layer providing enhanced mechanisms to the programmer.

Specifically in Ada's case, a binding to a VSM increases functionality and facilitates developing distributed applications, despite the fact that a variety of Ada built-in features and annexes in the Ada standard are available. The following issues are to mention:

- Communication, data sharing, and persistence has not to be implemented by means of standard Ada but can be put under coordination of a VSM system where several other languages and system architectures are glued together.
- The symmetric property of a VSM covers the actual needs of an application well. Particular subtasks can be implemented by the best-suited language, e.g. core development in Ada for safety critical parts and GUI development using Java.

The remaining paper is organised as follows: Section 2 overviews shared object paradigms. Section 3 presents concepts of CORSO, a virtual shared object layer developed at the *Institute of Computer Languages* at the *Technical University Vienna* and now made available commercially by *Tecco Coordination*

Systems, Vienna. Technical aspects of CORSO are revealed in Section 4. Our Ada binding to CORSO is described in Section 5. Pros and cons of our binding can be found in Section 6 where we also compare our binding to other language bindings and conclude the paper.

There have been other implementations of VSM in Ada, namely of the Linda tuple space (cf. [5–7]). All these implementations are stand-alone Ada implementations which lack the multi-language support of CORSO. In addition, the features offered by Linda are only a subset of CORSO's functionality.

2 Shared Objects

For the communication and synchronisation of distributed systems there exist two paradigms: *Message Passing* versus a *Virtual Shared Memory.*

2.1 Message Passing

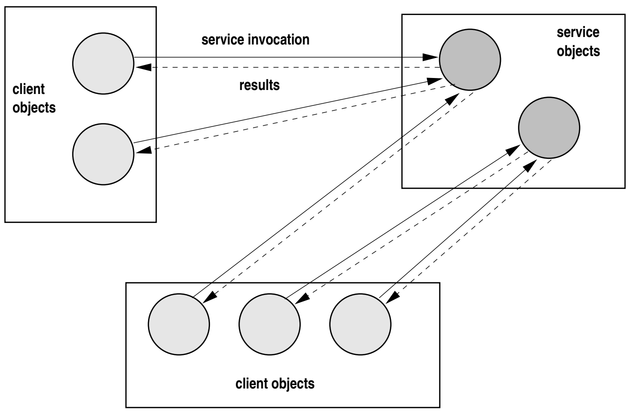

Fig. 1. Message Passing Paradigm

The classical and commercially wide-spread approach is the message passing paradigm. Processes communicate through the explicit (a-)synchronous sending and receiving of messages. The remote procedure call (RPC) is a two-way communication. The highest abstraction of the message passing paradigm are *distributed object* systems like CORBA and DCOM, where object methods can be invoked at remote sites which also results in a two-way communication pattern. Ada's Distributed Systems Annex (see [4]) also favours message passing. Application programs need to be aware of message sending or remote service invocation which means that e.g. fault tolerance has to be implemented explicitely into each distributed application. Adding and/or removing sites from the network of the distributed application has also to be considered explicitely

Moreover, such systems do not cache data fields locally which makes data field access expensive, because it requires an expensive remote procedure call.

Even if the same data fields have been used before by a client or by other clients at the same site, they must be fetched again. Because of the lack of replication and caching support, hierarchical client/server structures are built, although, they bear the disadvantage of a bottleneck w.r.t. performance and availability.

2.2 Virtual Shared Memory

Virtual shared memory offers a conceptually higher level of abstraction than message passing. It provides the vision of a common shared object space to which all participating, distributed and parallel executing processes have a consistent view. The *shared data objects* are used for communication between and synchronisation of parallel and distributed processes. VSM extends local memory to the memories of all sites where processes are running. This approach naturally hides heterogenous parts of the network from the programmer. Shared data objects relieve the application programmer from caching and replication issues. They allow the design of symmetric application architectures, thus avoiding the client/server bottleneck.

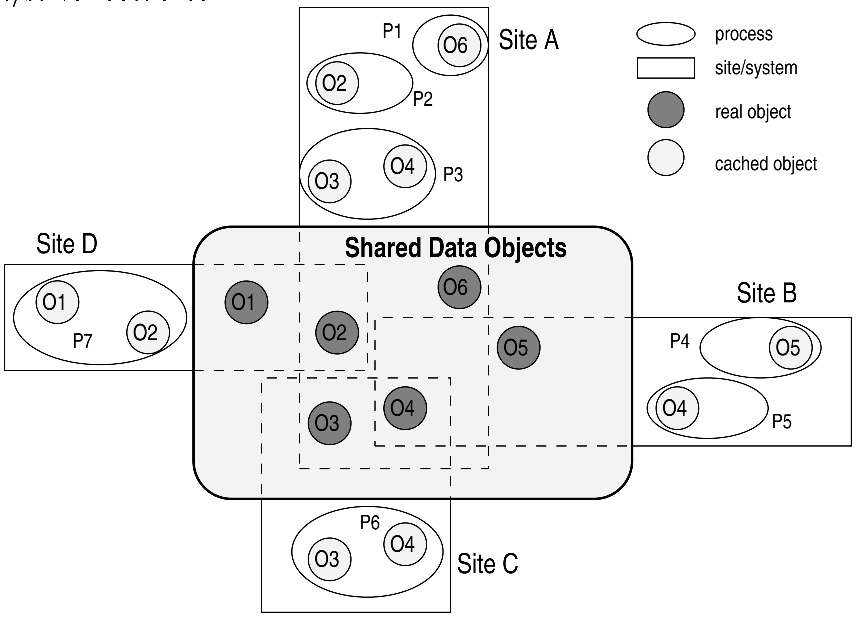

Fig. 2. Virtual Shared Memory Paradigm

Virtual shared memory is a new technology that goes beyond the possibilities offered by message passing. Its advantages concerning its conceptually higher abstraction of the underlying hardware, and its advantages concerning caching and replication have intensively been discussed in scientific literature (see e.g. [1–3]). An obvious tendency towards virtual shared memory replacing or accompanying client/server technology can be observed.

3 CORSO concepts

CORSO is a layered software component for the development of robust and parallel applications that supports the virtual shared memory paradigm. It has been developed at the *Institute of Computer Languages* at the *Technical University Vienna* and now commercially made available by *Tecco Coordination Systems, Vienna.* The granularity of sharing are data objects instead of entire memory pages, which makes CORSO suitable for all different kinds of distributed application scenarios. The shared objects serve for the communication and synchronisation between the parallel, concurrent and distributed processes.

CORSO is not only a pure memory model. It also supports complex coordination patterns on the shared objects through an advanced transaction and process model. Transactions serve to coordinate accesses to shared objects and to make shared objects persistent. Processes are used to reason about concurrency and parallelism and define the checkpoints for recovery.

CORSO meets the following requirements posed by the heterogeneity of distributed applications: It shields location, migration, replication, access, transaction, failure, representation, and persistency requirements from the programmer. Note that transaction transparency refers to transactions for the coordination of communication and synchronisation patterns that arise through concurrent accesses of multiple users and processes to the shared data objects. In contrast, in CORBA the notion of transactions refers to transactions at application level, where for example existing two-phase-commit components are controlled. CORSO transactions can also be used to control existing two-phase-commit components in that they control proxy objects reflecting the data and actions to be taken by the components. This way, CORSO transactions can be used to control database components in a highly flexible way.

CORSO supports object-oriented component programming through language bindings to C, C++, Java, Visual Basic, Oracle's Developer 2000, and Ada. The concept of component reuse can be fully exploited.

CORSO is an alternative technology for implementing distributed systems, although, it can also be used to enhance existing client/server technologies.

4 Technical Aspects of CORSO

4.1 CORSO Architecture

A distributed CORSO application consists of local software systems (LSYS) on which application processes are running. These application processes can use the functionality offered by CORSO and are termed CORSO processes. They are written in one of the programming languages, to which CORSO supports language bindings. A programming language "L" extended by CORSO features is termed "L&Co" which stands for "L plus Coordination". Currently, C&Co, C++&Co, Java&Co, VisualBasic&Co and Developer2000&Co are supported. The language bindings come in form of CORSO language libraries/classes. We are adding Ada&Co to the above list.

4.2 Interface Definition Language (IDL)

CORSO objects can be shared between different language paradigms across heterogeneous platforms that may differ in their representation of data. To make sure that data are understood correctly at each site, internally an interface definition language (IDL) is used. The Coordination Kernel automatically interprets and converts all information it receives from other Coordination Kernels. If a LSYS connects to a Coordination Kernel from a remote site that uses other data type formats, it also automatically converts these data. Application programs are thus portable across all platforms.

Depending on the capabilities of the host language, the employment of the IDL is more or less visible for the programmer. For example, in Java&Co un-/marshalling is done completely automatically so that the existence of the IDL is not evident at all, whereas in C&Co, the marshalling and unmarshalling is done via format commands. The IDL supports data terms of the following basic types:

- Integer: 32-bit integer value.
- Character: 8-bit byte value.
- String: 0-terminated character string.
- Raw: character string that may contain also the 0-character.
- Object identification (OID): unique reference to an object in the VSM. Pointers in local memory are generalised to OIDs.
- Structure: data structure that optionally may be given a name and that is composed of n members. Each member is an IDL-term.
- Stream: data type that is in particular useful for communication purposes. Internally it is a structure consisting of 2 components. The first component is termed head and the second one is termed tail. The head is of any IDL type and contains the user data. A single OID in the tail part acts as link to the next element in the stream. Streams can be conceptually seen as infinite coordination structures that are useful for communicating objects from a producer to a consumer in an obvious "UNIX pipe"-like way.

4.3 Object Sharing

The IDL is used to access and manipulate CORSO objects. Since these objects are usually cached—which is a big performance gain over simple two-way communication—the transaction concept ensures consistency with global virtual shared memory. Arbitrary communication patterns can be defined by selecting the proper coordination data structures composed of shared and nested objects. Different caching strategies are provided which may be selected on a per-object basis depending on the actual access frequency. For example if an object which is rarely written has been (partially) changed, it is automatically propagated to the cache of all current readers.

A variant of CORSO transactions are subtransactions which allow nesting of transactions. Every distributed task can be put under transaction control according to the ACID properties (atomicity, consistency, isolation, durability). Instead of dealing with a big single transaction which may fail or not, one can subdivide it into several subtransactions. The transaction scenario can be spread over several sites, where so called transaction dependent processes are responsible for handling such remote subtransactions. A successful subtransaction is able to provide its results independently of the success of all enclosing transactions. As a consequence a normal rollback of an enclosing transaction is no longer feasible. To guarantee atomicity of the transaction the effects of subtransactions must be compensated at least in a semantic manner. For this purpose special compensate actions can be defined.

Another aspect of transactions is synchronising CORSO processes. CORSO styled interprocess communication is based on transactions upon shared objects in a symmetric kind. CORSO stream objects can be used to implement a communication with properties similar to queues or pipes. Write operations under transaction control in conjunction with synchronous reads are the main constructs to obtain this kind of communication (comp. Section 5.4). CORSO processes can also be used to simulate a remote procedure call: a shared object incorporates input data, a method identification for the remote site, and the result from the method invocation.

5 Ada Binding to CORSO

The primary task in implementing a binding to CORSO is to build a LSYS in Ada. This task can be subdivided into

1. Declaring Ada-conforming IDL data types as can be found in Section 4.2.
2. Implementing marshalling and unmarshalling operations for these types[*].
3. Providing an Ada interface to CORSO processes and transactions.

Task 1 is done by defining an abstract tagged CORSO base type, from which the IDL types are derived.

Task 2 is done by implementing read and write attributes for all IDL types. Currently we do not support input and output attributes because the CORSO IDL types are elementary types only which do not profit from Ada's higher representation issues of input and output attributes. Details can be found in Section 5.1.

We implement Task 3 via a "thin binding" which is based on C&Co and is generated semi-automatically with help of the `c2ada`-tool developed by Intermetrics. This however is not directly presented to the programmer.

5.1 IDL Types and Marshalling

This section describes how the CORSO IDL types are implemented in the Ada binding. How this is done is shown by one example, the type `CoKeInt`. For the other types listed in Section 4.2 the binding is similar. Package `CoKe` shown in

[*] These operations are needed to convert IDL types to their "internal" representation.

Program 1 Package Coke

```
package CoKe is
  type Comm_type is (const, var, stream);
end Coke;
```

Program 1 gives the root of the Ada CoKe library tree and also contains the declaration of the communication types of CORSO objects. These are `const`, which means the object is written only once, but can be read arbitrarily often, `var`, which means it can be read and written arbitrarily often, and `stream`, which allows for communicating objects from a producer to a consumer in an obvious "UNIX pipe"-like way. Program 2 shows the declaration of type

Program 2 Package Coke.Base.Attribute

```
with Ada.Streams, CoKe.Base.Streams;
package CoKe.Base.Attribute is
  type CoKeBaseAttribute is abstract new CokeBase with private;
private
  type CoKeBaseAttribute is abstract new CokeBase with
    record
      Strm: CoKe.Base.Streams.CoKe_Stream_AD;
    end record;
  procedure Read(
      Stream: access Ada.Streams.Root_Stream_Type'CLASS;
      Item: out CoKeBaseAttribute);
  for CoKeBaseAttribute'Read use Read;
  procedure Write(
      Stream: access Ada.Streams.Root_Stream_Type'CLASS;
      Item: in CoKeBaseAttribute);
  for CoKeBaseAttribute'Write use Write;
end CoKe.Base.Attribute;
```

`CoKeBaseAttribute` which forms the base type of all IDL types. Note the procedures `Read` and `Write` in the private part. These and similar procedures defined for the other IDL types perform marshalling and unmarshalling, such that the user of the Ada binding to CORSO can do reading from and writing to the shared memory pool via Ada's read and write attributes in a fairly transparent way.

Package `CoKe.Base.Streams` contains the declaration of type `CoKe_Stream` and some internals of the Ada binding, which is not shown explicitely.

Program 3 shows the generic package for creating objects of type `CoKeInt`, the type used for handling integer types in CORSO. Our current binding does not support operations for IDL types. Thus they can only be used for communication purposes. A later release, however, will provide suitable operations for all IDL types.

Package `Shared` shown in Program 4 is used for building instances of objects being in the shared memory pool. Objects of type `shared` can be equipped with an OID. In contrast to the `Read` and `Write` operations in Program 2 the

Program 3 Package Coke.Base.Attribute.CoKeInt

```
generic
  type Int is range <>;
package CoKe.Base.Attribute.CoKeInt is
  type CoKeInt is new CoKeBaseAttribute with private;
  procedure Int_to_CoKeInt(from: Int; to: out CokeInt);
  function CoKeInt_to_Int(from: CokeInt) return Int;
private
  type CoKeInt is new CoKeBaseAttribute with
    record
      I: Int;
    end record;
  procedure Read(
      Stream: access Ada.Streams.Root_Stream_Type'CLASS;
      Item: out CoKeInt);
  for CoKeInt'Read use Read;
  procedure Write(
      Stream: access Ada.Streams.Root_Stream_Type'CLASS;
      Item: in CoKeInt);
  for CoKeInt'Write use Write;
end CoKe.Base.Attribute.CoKeInt;
```

Program 4 Package Shared

```
with Ada.Streams, CoKe.Base.Attribute, CoKe.Base.Attribute.CoKeOid;
generic
  type Base is new CoKe.Base.Attribute.CoKeBaseAttribute with private;
  Comm_type: CoKe.Comm_type;
package Shared is
  package Oid renames CoKe.Base.Attribute.CoKeOid;
  type Shared is new Base with private;
  procedure Set_Oid(
      Obj: in out Shared;
      the_Oid: Oid.CoKeOid := Oid.Create_Oid);
  function Get_Oid(Obj: Shared) return Oid.CoKeOid;
private
  type Shared is new Base with
    record
      the_Oid: Oid.CoKeOid;
    end record;
  procedure Read(
      Stream: access Ada.Streams.Root_Stream_Type'CLASS;
      Item: out Shared);
  for Shared'Read use Read;
  procedure Write(
      Stream: access Ada.Streams.Root_Stream_Type'CLASS;
      Item: in Shared);
  for Shared'Write use Write;
end Shared;
```

corresponding procedures in this package not only perform (un-)marshalling, but they also read/write the corresponding objects from/to the shared memory pool, where they are identified by their unique OID. The basic IDL type `Oid` is defined in package `CoKe.Base.Attribute.CoKeOid` which is not explicitely shown here.

5.2 Transactions

Shared objects can be combined with transactions. This is again done using standard object-oriented features of Ada, i.e., using generics and tagged types (cf. Program 5). In general all transactions are put under control of the trans-

Program 5 Package Tx

```
with Ada.Streams, Shared, Transaction_Mgt;
generic
  with package Some_Shared is new Shared(<>);
package Tx is
  type Txed is new Some_Shared.Shared with private;
  procedure Set_Tx(
        Obj: in out Txed;
        the_Tx: Transaction_Mgt.Tx);
  function Get_Tx(Obj: Txed) return Transaction_Mgt.Tx;
private
  type Txed is new Some_Shared.Shared with
    record
        the_Tx: Transaction_Mgt.Tx;
    end record;
  procedure Read(
        Stream: access Ada.Streams.Root_Stream_Type'CLASS;
        Item: out Txed);
  for Txed'Read use Read;
  procedure Write(
        Stream: access Ada.Streams.Root_Stream_Type'CLASS;
        Item: in Txed);
  for Txed'Write use Write;
end Tx;
```

action manager in package `Transaction_Mgt`. This manager provides the basic types and functionality to control transactions. By means of the generic package `Tx` the transaction semantics can be attached to any shared object type. Accessing an object of this new type (using `Read` and `Write` operations) remains the same, except of adding transaction properties to it.

To establish a transaction at first a transaction object has to be aquired from the transaction manager using function `Create_TX` (as shown in Program 9 later). Procedure `Set_Tx` binds this newly opened transaction to the shared object. `Get_Tx` is provided as counterpart to procedure `Set_Tx` for completeness only.

5.3 The LSYS

In order to implement a CORSO application, one has to register the processes in the LSYS. Our binding allows this to be done via package `CoKe.Entries` shown in Program 6. All CORSO processes are in fact Ada procedures of the form

Program 6 Package CoKe.Entries

```
with CoKe.Base.Attribute;
package CoKe.Entries is
  type Proc_Ptr is access
    procedure(Param: CoKe.Base.Attribute.CoKeBaseAttribute'CLASS);
  type CoKeBaseAttribute_AD is
    access all CoKe.Base.Attribute.CoKeBaseAttribute'CLASS;
  procedure Add_CoKe_Entry(
        Name: string;
        Run: Proc_Ptr);
  procedure Start_CoKe_Entry(
        Name: string;
        Param: CoKeBaseAttribute_AD);
end CoKe.Entries;
```

`Proc_Ptr`. The parameter `Param` can be used to pass arbitrarily complex data structures using the IDL type `CoKeStruct`.

How these pieces fit together is shown in Section 5.4 where the well-known producer/consumer problem is solved via the Ada CORSO Binding.

5.4 The Producer/Consumer Example

The procedures `Producer` and `Consumer` are given in Program 9 and 10, respectively. The instance of `CoKeInt` used in Program 9 is depicted in Program 7, equipping this type with a transaction is shown in Program 8.

Program 7 Package My_SharedCoKeInt

```
with Coke, My_CoKeInt, Shared;
package My_SharedCoKeInt is new Shared(My_CoKeInt.CokeInt,CoKe.stream);
```

Program 8 Package My_TxedSharedCoKeInt

```
with My_SharedCoKeInt, Tx;
package My_TxedSharedCoKeInt is new Tx(My_SharedCoKeInt);
```

Via calls to `CoKe.Entries.Add_CoKe_Entry` these procedures are registered as CORSO processes. Before that, connection to the LSYS has been established by a call to `LSYSConnect`. The procedures (CORSO processes) are then executed by `CreateIndependentProcess` which is given the name-strings "producer" and "consumer", respectively. In addition to that an identical OID is passed to both processes, which identifies the shared object stream used for communication.

Program 9 Procedure Producer

```
with CoKe.Base.Attribute.CoKeOid, CoKe.Base.Streams, My_SharedCoKeInt,
My_TxedSharedCoKeInt, Transaction_Mgt;
use CoKe.Base.Attribute.CoKeOid, CoKe.Base.Streams, My_SharedCoKeInt,
My_TxedSharedCoKeInt, Transaction_Mgt;

procedure Producer(Param: CoKe.Base.Attribute.CoKeBaseAttribute'CLASS)
is
  data: My_TxedSharedCoKeInt.Txed;
  CoKe_Strm: aliased CoKe_Stream;
  Oid: CoKeOid := CoKeOid(Param);
begin
  Set_Oid(Shared(data), Oid);        -- attach Oid we have got from Param to data
  for i in 1..3 loop
    Int_to_CoKeInt(from => i, to => data);
    declare
      topTx: Tx := Create_TX;                     -- create and open a transaction
    begin
      Set_Tx(data,topTx);                            -- attach transaction to data
      loop
        Txed'WRITE(CoKe_Strm'access, data);               -- write data to VSM
        exit when Commit(topTx);                    -- exit if everything is okay
        Cancel(topTx);                -- cancel transaction if something went wrong
      end loop;
    end;
  end loop;
  Process_Commit;                                         -- exit in commit state
end Producer;
```

Program 10 Procedure Consumer

```
with CoKe.Base.Attribute.CoKeOid, CoKe.Base.Streams, My_SharedCoKeInt,
Text_IO;
use CoKe.Base.Attribute.CoKeOid, CoKe.Base.Streams, My_SharedCoKeInt;

procedure Consumer(Param: CoKe.Base.Attribute.CoKeBaseAttribute'CLASS)
is
  data: shared;
  CoKe_Strm: aliased CoKe_Stream;
  Oid: CoKeOid := CoKeOid(Param);
begin
  Set_Oid(Shared(data), Oid);        -- attach Oid we have got from Param to data
  for i in 1..3 loop
    Shared'READ(CoKe_Strm'access, data);                -- read data from VSM
    Text_IO.Put_Line(integer'IMAGE(CoKeInt_to_Int(data)));
  end loop;
  Process_Commit;                                         -- exit in commit state
end Consumer;
```

The calls to `CreateIndependentProcess` can even be done from programs not written in Ada. Note that the two procedures `Producer` and `Consumer` may reside on different sites in the network, too. Procedure `Process_Commit` serves as indication for the transaction manager how to proceed with the transaction.

6 Discussion and Conclusion

There is only one disadvantage of our Ada CORSO binding, namely that Ada's tasks cannot be used directly as CORSO processes. Nevertheless tasks can be used freely within a CORSO process. By the way, Java&Co suffers from the same problem.

We have found that Ada's read and write attributes are very useful to incorporate access to objects in a shared memory pool.

Transactions have not been considered in the Ada Reference Manual until now. Our Ada CORSO binding enhances Ada with transaction-oriented, fault-tolerant, distributed objects in a straight-forward way without having to extend the Ada language.

Compared to other language bindings, we see that Ada's generics and tagged types provide an excellent means to correctly model the interdependency between IDL types, the shared object paradigm, and transactions. For example Java&Co lacks multiple inheritance and thus cannot model this interdependency in an accurate way. C++&Co heavily uses multiple inheritance for this purpose but is less readable than Ada (at least for an Ada person).

The CORSO layer supports a comprehensive programming model for distributed, parallel and concurrent programming. It supports the virtual shared memory approach and in addition an advanced transaction/process model. Providing an Ada binding to CORSO opens the area of virtual shared memory to the Ada world.

References

1. D. E. Bakken, *Supporting fault-tolerant parallel programming in LINDA*, Ph.D. thesis, University of Arizona, Department of Computer Science, 1994.
2. M. R. Eskicioglu, *A comprehensive bibliography of distributed shared memory*, Tech. Report TR96-17, University of Alberta, Edmonton, Canada, 1996.
3. eva Kühn and Georg Nozicka, *Post-client/server coordination tools*, Proc. of the Second Asian Computer Science Conference on Coordination Technology for Collaborative Applications, LNCS, Springer-Verlag, 1997.
4. ISO/IEC 8652, *Ada reference manual*, 1995.
5. Y. Kermarrec and L. Pautet, *Ada-Linda: A powerful paradigm for programming distributed Ada applications*, Proceedings of the TRI-Ada'94 conference (Baltimore, Maryland), 1994, pp. 438–445.
6. Kristina Lundqvist and Göran Wall, *Using object oriented methods in Ada 95 to implement Linda*, Proceedings of Ada-Europe'96 (Montreux, Switzerland), Springer-Verlag, 1996, pp. 211–222.
7. Göran Wall and Kristina Lundqvist, *Shared packages through Linda*, Proceedings of Ada-Europe'96 (Montreux, Switzerland), Springer-Verlag, 1996, pp. 223–234.