D: A Language Framework for Distributed Programming

Cristina Videira Lopes, Gregor Kiczales

Technical report SPL97-010, P9710047 Xerox Palo Alto Research Center. February 1997.

D: A LANGUAGE FRAMEWORK FOR DISTRIBUTED PROGRAMMING

CRISTINA VIDEIRA LOPES, GREGOR KICZALES

XEROX PALO ALTO RESEARCH CENTER*

We present an object-oriented language framework for distributed programming called D. D uses the aspect oriented programming approach to allow the code for the basic functionality of a distributed application to be written without having to explicitly deal with distribution and synchronization. Separate code deals with those issues.

The D language framework consists of: (i) Jcore, an object-oriented language used to express the basic functionality and the activity of the system; (ii) Cool, a language used to express coordination of threads; and (iii) Ridl, a language used to express remote access strategies. A special tool called an Aspect Weaver™ takes the programs written in the different languages and combines them together to produce an executable program with the specified distributed behavior. D builds on existing object-oriented languages, and aggressively adheres to syntactic separation of distribution concerns. D program texts are less tangled and therefore simpler and more reusable than their equivalents written in Java.

1. INTRODUCTION

One of the main problems with current distributed systems is the difficulty of developing the software. Existing general purpose programming languages, functional, procedural or object-oriented, provide little or no support for the special needs of distributed systems. Programmers must rely on analysis and design methodologies [9, 14, 30], design patterns [11] and programming guidelines to bridge the gap between design specifications and the actual code that they must write. This bridge implies a whole set of informal documentation that needs to be maintained along with the program.

One way of decreasing the complexity of the software is by using programming languages that capture more closely the concepts of the design space. Over the last 15 years, researchers have been proposing programming languages based on particular models of distributed and/or concurrent computation ([2-4, 7, 8, 13, 21, 29, 32, 34], just to mention a few). These models, and the consequent languages, have had a limited success in industry, either because they are too different from the most popular programming languages, or because they don't provide the flexibility the programmers need, or simply because they are not integrated with what already exists. In spite of their lack of linguistic support for distribution, the majority of real world distributed applications are written in C or C++, with direct access to some message passing library or RPC package and possibly also to a thread library. Nevertheless, some of the ideas in those many languages end up being adopted, one way or another, by mainstream programming. Java™ [12], for example, includes a number of features that have been proposed in other languages and systems.

* 3333 Coyote Hill Road, Palo Alto, CA 94304, USA. {lopes,gregor}@parc.xerox.com

As a consequence of using languages which lack appropriate support for distribution, the inherent complexity of distributed systems is severely magnified in the program texts themselves, making distributed applications extremely difficult to write and even more difficult to understand, debug and maintain. While very little can be done to decrease the natural complexity of distributed applications, there is space to improve the quality and reliability of the development and maintenance processes of such applications. The D framework fits in such space.

1.1. ASPECTS AND THE D LANGUAGE FRAMEWORK

Central to D is the concept of *aspect*. In a separate paper on aspect-oriented programming [18], we have compared *aspects* to *components* in the following way:

> "With respect to a system and its implementation using a generalized-procedure language [such as an object-oriented language], an issue that must be programmed is:
>
> A COMPONENT, if it can be cleanly encapsulated in a generalized procedure (i.e. object, method, procedure, API). By cleanly, we mean well-localized, and easily accessed and composed as necessary. Components tend to be units of a system's functional decomposition, such as image filters, bank accounts, GUI widgets, and service providers.
>
> An ASPECT, if it can not be cleanly encapsulated in a generalized procedure. Aspects tend not to be to units of the system's functional decomposition, but rather are properties that affect the performance or semantics of the components in systemic ways."

With respect to application objects written using the major object-oriented languages — Java, C++, Smalltalk, Eiffel —, concurrency control and access to remote objects are natural candidates for being aspects. The goal of identifying them as aspects is that we can abstract their interactions with the object-oriented program, provide aspect-specific languages and build tools around them that help the development and maintenance of distributed applications.

With this in mind, we have built a language framework which consists of: 1) Jcore, an object-oriented component language used to express the basic functionality and activity of the system; 2) Cool, an aspect language used to express coordination of threads; and 3) Ridl, an aspect language used to express remote access strategies. A special tool called an Aspect Weaver takes the programs written in the different languages and combines them together to produce an executable program with the specified distributed behavior.

1.2. WHAT IS NEW

Two principles distinguish D from other languages for distributed programming: (i) D builds on existing object-oriented languages; and (ii) D aggressively adheres to syntactic separation of concerns.

With respect to the first point, we believe that there are enough component languages, and in particular object-oriented languages, that are sufficiently powerful for implementing and composing object functionality. D concentrates on formalizing a number of programming practices that occur frequently in existing distributed software. The component language in D, Jcore, is a subset of Java[1]. Additionally, D defines two small, specicalized, aspect languages that can be ignored for non-distributed, sequential, applications.

With respect to the second point, syntactic separation of concerns, D enforces a formal separation between object functionality and other issues in distributed computation. The distribution issues dealt with in D are (i) coordination of threads (i.e. concurrency control) and (ii) data flow between different address spaces. Clean separation is achieved both by special language constructs (the aspect languages) and by textually separating the different aspect programs. So, for example, the data flow between address spaces is isolated in "remote interface" constructs, which use a specific interface aspect language and which are separated from the classes.

[1] D can easily be reimplemented in any other multithreaded object system/language (e.g. C++, Smalltalk).

By appropriately enforcing separation of concerns at the code level, the program texts become an effective part of the application's documentation, with all the benefits that that represents for purposes of program evolution. The aspect languages are also a formal means for checking the consistency of the aspect programs relatively independently from the classes. Finally, the particular notation used in the aspect languages, building on a huge body of previous work on distributed systems, is sufficiently high-level, so that it frees the programmer from the details of the implementation.

The remainder of this paper is organized as follows. Section 2 presents an example that is representative of a large number of distributed applications, and that will be used as a reference throughout the paper. Section 3 analyzes some distribution issues, and how their programming ends up getting tangled in the application classes. Section 4 describes the design and semantics of D. Section 5 contains the highlights of the implementation of the framework. Section 6 presents some results and analyzes the framework. Section 7 relates D to previous work. Finally, section 8 concludes the paper.

2. MOTIVATING EXAMPLE

Suppose we want to implement a book locator class which manages an association between books and their physical locations. The functionality of such objects consists of three services: (i) a `register` service that takes one book and one location, and registers the pair in some internal variables; (ii) an `unregister` service that takes one book and eliminates it from registry; and (iii) a `locate` service that takes a key string, searches the string fields of the books for possible matches with the key, and returns the location of the first book that matches it.[2]

Besides this basic functionality, book locators should be able to process several requests concurrently. With respect to this, book locators present a pattern of behavior that is quite common in concurrent systems: "read" accesses (in this case, the `locate` service) can be done concurrently while temporarily blocking all "write" accesses; "write" accesses (in this case, `register`, and `unregister`) should not be done concurrently and should block all other services.

The last piece of the specification is that book locators are network objects, i.e. they can be accessed remotely. Suppose, for example, that they are part of a much larger document management application, and that the complete book database is managed by some other server(s). In order to speed up the searches, book locators should cache information about the books, namely their titles, authors and isbn.

Figure 1 shows a possible implementation of this specification using Java and Java's Remote Method Invocation (RMI) facility [15]. For readability reasons, we've colored the code according to its role in the program: black applies to the basic functionality of the class, red applies to coordination code, and green applies to code for remote access.

3. CODE TANGLING

The first thing that shows up in the code in Figure 1 is the intertwining of colors. We call this the "code tangling" phenomenon. In this section we investigate the reasons it occurs, by going through the program and looking at specific pieces of tangling. What will emerge is that in each case, the root cause of the tangling is similar: the abstraction and composition mechanisms at the heart of OOP do a great job of capturing the functionality of this application, but they don't provide a "natural fit" for supporting the synchronization constraints and communication strategy we want to program.

We should point out that, although we're using Java as the illustrative language, all the observations made in this section apply to most other object-oriented language environments; in particular C++, Smalltalk, CLOS and Eiffel environments that support multithreading and RPC through the use of low-level synchronization primitives and an interface definition language (such as CORBA IDL).

[2] In a more realistic scenario, the locate service would, for example, return a list of pairs book/location corresponding to the books that match the given key. It could also have several forms of locating books, i.e. by ISDN, by title, etc., and it would certainly have more services. We're simplifying in order to keep the example simple and make it easier to focus on the issues at hand.

```
public interface Locator extends java.rmi.Remote
{
   void register (Book b, Location l)
        throws java.rmi.RemoteException,
               LocatorFull;
   void unregister (Book b)
        throws java.rmi.RemoteException;
   Location locate (String str)
        throws java.rmi.RemoteException,
               BookNotFound;
}
public interface Book extends Serializable {
  String get_title();
  String get_author();
  int get_isbn();
}
public interface Location extends Serializable {
  // nothing interesting
}

public class BookLocator
       extends UnicastRemoteObject
       implements Locator {
  // This is just one possible implementation.
  // books[i] is in locations[i]
  private Book     books[];
  private Location locations[];
  private int      nbooks = 0;
  protected int    activeReaders = 0;
  protected int    activeWriters = 0;
  // the constructor
  public BookLocator (int dbsize) {
    books = new Book[dbsize];
    locations = new Location[dbsize];
  }
  // the main activity
  public static void main(String args[]) {
    try {
       BookLocator obj = new BookLocator(17);
       Naming.rebind("BookLocator", obj);
    } catch (Exception e) {
       System.out.println("BookLocator err: "
                          + e.getMessage());
       e.printStackTrace();
    }
  }
  public void register (Book b, Location l)
  throws LocatorFull, RemoteException {
    beforeWrite();
    if (nbooks > books.length) {
      afterWrite();
      throw new LocatorFull();
    }
    else {
      // Just put it at the end
      books[nbooks] = b;
      locations[nbooks++] = l;
      afterWrite();
    }
  }
  public void unregister (Book b)
  throws RemoteException {
    Book abook = books[0]; int i = 0;
    beforeWrite();
    while (i < nbooks &&
           abook.get_isbn() != b.get_isbn())
      abook = books[++i];
    if (i == nbooks) {
      afterWrite();
      return;
    }
    // simply shift down the rest
    while (i < nbooks - 1) {
      books[i]= books[i+1];
      locations[i]= locations[++i];
    }
    --nbooks;
    afterWrite();
  }
```

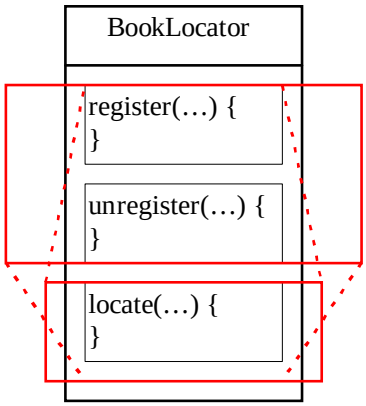
```
  public Location locate (String str)
  throws BookNotFound, RemoteException {
    Location l; Book abook = books[0];
    int i = 0; boolean found = false;

    synchronized (this) {
      while (activeWriters > 0)
        try { wait(); }
        catch (InterruptedException e) {}
      }
      ++activeReaders;
    }
    while (i < nbooks && found == false) {
      if (abook.get_title().compareTo (str) == 0 ||
          abook.get_author().compareTo (str) == 0)
        found = true;
      else abook = books[++i];
    }
    if (found == false) {
      synchronized (this) {
        --activeReaders;
        notifyAll();
      }
      throw new BookNotFound (str);
    }
    l = locations[i];
    synchronized (this) {
      --activeReaders;
      notifyAll();
    }
    return l;
  }

  protected synchronized void beforeWrite () {
    while (activeWriters > 0 ||
           activeReaders > 0)
      try { wait(); }
      catch (InterruptedException e) {}
    }
    ++activeWriters;
  }

  protected synchronized void afterWrite () {
    --activeWriters;
    notifyAll();
  }

}

public class BookImpl implements Book {
  // one possible implementation of Book
  String title, author;
  int isbn;
  Project owner;
  OCRImage firstpage;

  public BookImpl (String t, String a, int n) {
    title = t; author = a; isbn = n;
  }
  public String get_title() { return title; }
  public String get_author() { return author; }
  public int get_isbn() { return isbn; }
  private void writeObject(ObjectOutputStream s)
    throws NotSerializableException, IOException {
    s.writeObject(title);
    s.writeObject(author);
    s.writeInt(isbn);
  }
  private void readObject(ObjectInputStream s)
    throws ClassNotFoundException, IOException {
    title = (String)s.readObject();
    author = (String)s.readObject();
    isbn = s.readInt();
  }
}
```

Figure 1. ***Possible implementation of book locators in Java.***

3.1. LOW-LEVEL SYNCHRONIZATION PRIMITIVES

The basic primitive for synchronization in Java is the keyword `synchronized`. It implements monitors which are associated with objects. `synchronized` is used both as a statement and as a method modifier. Design guidelines for concurrent programming in Java recommend using `synchronized` methods instead of `synchronized` statements whenever possible; this practice ensures that a thread wanting to execute an object always gets and holds the object's lock before accessing any of its instance variables.[3]

But this practice is not always possible or convenient. Consider, for example, the book locator class in Figure 1. It is impossible to implement the specified coordination behavior by simply synchronizing its public methods.[4] The problem is that the specification calls for two different synchronization schemes applied to different subsets of the methods. The figure below sketches the cross-cutting that is going on between the OOPL constructs (i.e. the methods) and the synchronization: `register` and `unregister` share one synchronization scheme (that affects `locate` too), while `locate` has its own synchronization sheme (that affect `register` and `unregister` too).



This situation occurs every time a class provides inspective methods (i.e., that don't modify any instance variables) that can be run concurrently by many threads, and state-changing methods that must get exclusive access to the object.

Another variation of this is when the class can be partitioned into *n* independent, non-conflicting subsets of methods. In this case, Lea ([19] §3.3) suggests two different designs that solve the problem: (i) refactoring the class into *n* other classes, each containing a set of mutually exclusive methods that are synchronized, and (ii) defining *n* auxiliary lock objects, each corresponding to a set of mutually exclusive methods, that must be held by the threads before executing the methods. Although these design patterns implement the specification, they introduce "noise" in the code. Pattern (i) distracts from the way the code actually implements the functionality; pattern (ii) forces the explicit handling of lock objects and of synchronized statements.

What these situations show is that the abstraction behind a simple on/off synchronization of the object on method boundaries is insufficient. That's why Java supports the more basic `synchronized` statement, with which the programmers can do just about anything, including a tangled mess of code. In Figure 1, the bits of red code intertwined in the methods illustrate this point. Would it not be highlighted in a different color, it would be hard to understand the actual synchronization scheme for this class.

3.2. INSTANCE VARIABLES AND SYNCHRONIZATION STATE

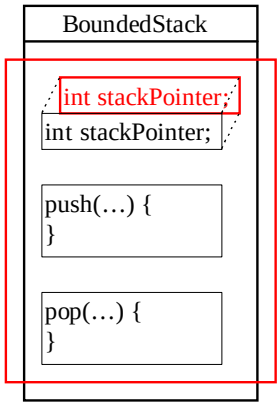
The design of concurrent systems usually involves identifying the states in which threads are suspended and the states in which they can proceed. The state space of the objects is usually very large, but only a small subset is important for purposes of action control. We call this the synchronization state. For example, in the book locator class of Figure 1 the arrays of books and locations are completely ignored for purposes of synchronization; only the instance variables activeReaders and activeWriters matter. In this particular case there is an obvious separation of the instance variables that hold

[3] In the on-line Java tutorial (http://www.javasoft.com/nav/read/Tutorial/java/threads/monitors.html) there is the following note: "**Note**: Generally, critical sections in Java programs are methods. You can mark smaller code segments as synchronized. However, this violates object-oriented paradigms and leads to confusing code that is difficult to debug and maintain. For the majority of your Java programming purposes, it's best to use synchronized only at the method level."

[4] At first, it seems that it would suffice to eliminate all the red code and to simply make `register` and `unregister` synchronized methods. On a more careful analysis, one concludes that that doesn't work. The absence of synchronization code in `locate` would make it ready for execution at any time, including when some other thread is executing one of the other two methods. That would violate the specification.

the synchronization state (activeReaders and activeWriters) from the ones that don't (books and locations), and we captured that by coloring activeReaders and activeWriters in red.

This separation, however, is not enforced by Java. As a consequence, programmers must make sure that suspensions and notifications happen at the right points in the code, and for the right values of the instance variables. For example, when implementing a bounded stack, we can use the stack pointer both for holding the ordinary stack state and the synchronization state, the latter being empty (stackPointer = 0), full (stackPointer = MAX) and middle (0 < stackPointer < MAX). This overloading of meaning is captured in the sketch on the left.



Defining and tracking the synchronization state is one of the most critical points of concurrent systems. Doing it in the objects' complete state space can be confusing and error-prone. It usually requires a number of additional coding rules (for example, the "state as objects" pattern [11]) in order to make it more intelligible.

3.3. INTERFACES

The intent of Java interfaces is to declare types with no implementations. But for the Remote Method Invocation facility, interfaces are also used to define the interaction with remote objects as well as the semantics of remote parameter passing. (See the interface declarations in Figure 1) A similar situation occurs with, for example, the CORBA IDL, which serves many different purposes.

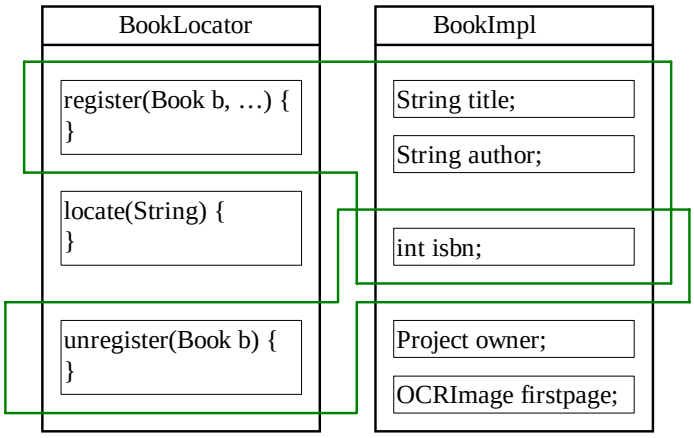
Using one construct for different purposes, by itself, is not necessarily bad. But a language type and an interface to a remote object are, in many important operational ways, two different things; a whole new set of issues arises from invoking remote services. The Java virtual machine knows that too, and it throws a number of exceptions on situations that are perfectly acceptable for local objects but unacceptable for remote objects. For example, all the arguments of calls to remote objects must themselves be instances of classes that implement either the Remote or Serializable interfaces. Further, those arguments that implement the Remote interface must have already been exported. However, both situations conform to the specifications with respect to the usage of interfaces-as-types.

3.4. PARAMETER PASSING IN REMOTE CALLS

Remote parameter passing in a object system poses some difficult choices. On the one hand, we would like to be able to use the same semantics as local parameters by passing them by global reference. On the other hand, that practice may hurt the performance of the applications, since it multiplies the number of remote invocations. Therefore, most object systems support an alternative pass-by-copy semantics that copies the argument to the remote site.

Java supports these two semantics, and it does so on a type basis: instances of classes that implement the Remote interface are always passed by global reference; instances of classes that implement the Serializable interface are always passed by (deep) copy.

Consider, for example, the serializable Book interface and the BookImpl class in Figure 1. The default serialization of book objects copies all of its instance variables, including `firstPage` and `owner`. Since the book locator works only on specific parts of the books, we have redefined the writeObject and readObject methods of BookImpl to marshal only the title, author and isbn. In doing so, we also fix the way BookImpl instances are passed for all other possible services.



In practice, however, remote parameter passing and class implementations cross-cut each other. Different remote services have different needs with respect to the amount of copying, if any, to be done.

The sketch on the left captures this point for the book locator example. The `unregister` service accesses only the `isbn` field of books, therefore we shouldn't need to copy any other field. Furthermore, other servers may use books for different purposes, caching, for example, the `owner` field. Choosing the semantics of remote parameter passing on a type basis rules out the possibility of these fine tunings.

Of course, there are many ways of dealing with this. We can change the interface to `unregister` so that, instead of a Book, it takes an integer (the ISBN). In general, that's not a good solution, since it may end up by imposing awkward interfaces to the classes. We can also have several implementations of the Book interface, each one defining a different copying scheme. But this makes the code bigger and less clear, since there will be several classes that don't do much except redefining the serialization methods; also the client code must do the appropriate class casts before passing books on remote calls.

3.5. SUMMARY

Object-oriented language mechanisms, namely class definitions, inheritance and method invocations, are powerful tools for modeling application objects and the interaction among them. However, as we have shown, synchronization and remote access are issues that tend to cut across the implementation of the application objects.

The source of the code tangling problem is that, by having only the definitional and composition mechanisms of OOP, programming synchronization and remote access becomes a complicated exercise on weaving these aspects with the functionality of the objects.

4. THE LANGUAGES OF D

Aspect-oriented programming aims at improving the quality of the software by decreasing the level of code tangling. That is also the goal of D. This section describes the design of this language framework.

D consists of: one object-oriented component language used to express the basic functionality and the activity of the system (Jcore); an aspect language used to express remote access strategies (Ridl); and an aspect language used to express coordination of threads (Cool). The two aspect languages can be seen as meta-languages specially designed for implementing aspect programs that control the object-oriented program itself.

In order to illustrate the language framework right up front, we present, in Figure 2, the source code of the book locator example described in section 2, this time using D.

```
public class BookLocator
{
  // This is just one possible implementation.
  // books[i] is in locations[i]
  private Book     books[];
  private Location locations[];
  private int      nbooks = 0;

  // the constructor
  public BookLocator (int dbsize) {
    books = new Book[dbsize];
    locations = new Location[dbsize];
  }
  public void register (Book b, Location l)
  throws LocatorFull {
    if (nbooks > books.length)
      throw new LocatorFull();
    else {
      // Just put it at the end
      books[nbooks] = b;
      locations[nbooks++] = l;
    }
  }
  public void unregister (Book b) {
    Book abook = books[0]; int i = 0;
    while (i < nbooks &&
           abook.get_isbn() != b.get_isbn())
      abook = books[++i];
    if (i == nbooks) return;
    // simply shift down the rest
    while (i < nbooks - 1) {
      books[i]= books[i+1];
      locations[i]= locations[++i];
    }
    --nbooks;
  }
  public Location locate (String str)
  throws BookNotFound {
    Book abook = books[0];
    int i = 0; boolean found = false;

    while (i < nbooks && found == false) {
      if (abook.get_title().compareTo (str)==0 ||
          abook.get_author().compareTo (str)==0)
        found = true;
      else abook = books[++i];
    }
    if (found == false)
      throw new BookNotFound (str);
    return locations[i];
  }
  private static void main (String args[]) {
    try {
       BookLocator obj = new BookLocator(17);
       Naming.rebind("BookLocator", obj);
    } catch (Exception e) {
       System.out.println("BookLocator err: "
                          + e.getMessage());
       e.printStackTrace();
    }
  }
}

public class Book {
  // one possible implementation of Book
  String title, author;
  int isbn;
  Project owner;
  OCRImage firstpage;

  public BookImpl (String t, String a, int n) {
    title = t; author = a; isbn = n;
  }
  public String get_title() { return title; }
  public String get_author() { return author; }
  public int get_isbn() { return isbn; }
}
```

```
coordinator Blcoord : BookLocator {
  selfexclusive{register, unregister};
  mutexclusive{register, unregister, locate};
}

remote BookLocator {
  void register (Book, Location);
  void unregister (Book: copy isbn);
  Location locate (String);

  default:
    Book: copy (*.String, isbn);
}
```

Figure 2. ***Implementation of book locators using the D framework. The black code implements the functionality of book locators; the red and green code (the aspect programs) define the concurrency control and the remote access strategy, respectively, for the BookLocator class. Note that the aspect programs refer to the functionality program by referring to its classes and methods.***

4.1. THE COMPONENT LANGUAGE: JCORE

One of the advantages of D is that its design is mostly independent of the details of the core OO language. While Jcore is now a subset of Java, D could easily be adapted to work with C++ or Smalltalk. We assume that the reader is moderately familiar with Java, and therefore we won't say much about Jcore. We simply enumerate what was removed from Java to get Jcore, and why:

- *For purposes of semantic strengthening*: the keyword `synchronized`; and the methods wait, notify and notifyAll. This is to enforce that Jcore classes don't do explicit synchronization.

- *For control of effort*: interfaces and overloading of methods.

(See class definitions in Figure 2)

4.2. THE COORDINATION ASPECT LANGUAGE: COOL

Cool is a very simple language that provides constructs for dealing with mutual exclusion of threads, synchronization state, guarded suspension and notification. Coordination programs consist of a set of coordinators (see `coordinator` in Figure 2, in red) which are associated with Jcore classes on a name basis. A coordinator can coordinate more than one class at the same time. Coordinators can inspect the Jcore objects' state, but they cannot modify it.

4.2.1. MUTUAL EXCLUSION OF THREADS

The most basic synchronization need of object-oriented programs is defining which methods can be executed concurrently by different threads and which methods cannot. Part of this issue was discussed in section §3.1. Based on this observation, Cool provides two constructs for addressing this need, the `selfexclusive` and `mutexclusive` method sets. In each coordinator there is at most one `selfexclusive` set, but there can be many `mutexclusive` sets. Each method included in the `selfexclusive` set is executed by at most one thread at a time. Methods in the same `mutexclusive` set mutually exclude each other, meaning that they are never executed concurrently.[5] These two simple, but powerful, constructs reduce all the red code in Figure 1 (about 35 lines) to two lines of code in the aspect program in Figure 2. What those two lines mean is that `locate` can be executed concurrently by several threads – because it is not in the `selfexclusive` set – and that the execution of any of the three methods by one thread blocks the execution of the others by other threads – because they are in one `mutexclusive` set.

4.2.2. SYNCHRONIZATION STATE AND GUARDED SUSPENSION

Besides controlling the exclusion rights, coordinators also manage the object's synchronization state. Synchronization state is captured by condition variables (`cond`) which can be declared in the coordinator. Synchronization state changes can occur only on entry and on exit of the methods of Jcore objects, and this is captured by the `on_entry` and `on_exit` clauses. Guarded suspension is captured by the `requires` clause. The `requires` and `on_entry` constructs are executed atomically. Notifications are issued automatically on changes of the values of condition variables.[6]

Programming the synchronization state consists of defining a number of condition variables and then defining requirements and state modifications for each of the methods of the Jcore classes. The example below shows the implementation of the synchronization scheme of a concurrent bounded stack. Threads are suspended when wanting to push elements in stacks that are full or wanting to pop elements from stacks that are empty.

---

[5] Methods in `mutexclusive` sets are not self-exclusive, unless they also belong to the `selfexclusive` set.

[6] The language implementation can easily optimize the number of notifications, and only issue them if the new value is actually being used as a waiting condition in some requires clause.

```
coordinator BoundedStackCoord : BoundedStack {
  selfexclusive{pop, push};
  mutexclusive{pop, push};

  cond boolean full = false;
  cond boolean empty = true;

  push : requires !full;
    on_exit {
      if (sp == MAX) full = true;
      if (sp == 1) empty = false;
    }
  pop : requires !empty;
    on_exit {
      if (sp == 0) empty = true;
      if (sp == MAX - 1) full = false;
    }
}
```

As this example shows, the coordinator can access the instance variables of the object being coordinated, in this case sp. However, it is illegal to modify them; assignments can be performed only on the coordinator's own variables. This ensures that the coordination program does not interfere with the program that is being coordinated other than through the protocol of coordination itself.

An earlier version of Cool was described in [24]. Other than what was described here, Cool allows for coordination of multiple classes.

4.3. THE REMOTE INTERFACE ASPECT LANGUAGE: RIDL

Remote interface programs define how and what data is sent when there are method invocations between different execution spaces. The basic construct is the keyword `remote`. Each remote interface declares the service signatures, including the parameter passing modes for each of them. This is very much like any other interface definition language. The novelty of Ridl is with respect to the degree of control it provides over the copying semantics. When an argument is passed by copy, an additional copying directive can be specified, defining the only parts of the object graph that should be copied.

In the remote interface in Figure 2, the `unregister` service specifies that only the isbn part of the Book parameter should be copied. Other than that, and as the default clause states, the default copying for books is all their fields of type String as well as their isbn part, leaving out all the other parts; for location objects, it's only their fields of type String.

As another example, consider a possible remote interface to a Printer:

```
remote Printer {
  // called from users
  Status print(Document:
               copy (*.int, *.String, documentRepository));
  Status cancel(int);
  Document[] get_printQueue();
  // called from document repositories
  Status transfer(Document: copy *.int, *.String, data);

  default:
    Document: copy (*.String, *.int)
}
```

In this case, printers are invoked both from user software (editors, etc.) and from the document repositories where the documents are stored. On a user `print` request, only the document identifiers (Strings and integers) and the `documentRepository` field is passed; assuming document repositories are also servers with a remote interface, only its global

reference is passed. The actual transference of the document to the printer is done by the document repository by invoking the `transfer` service. Because the different services have different needs with respect to documents, Document arguments are copied differently depending on the service.

Copying directives are a graph language inspired by traversal directives in the work of Lieberherr et al. [20]. The paper [23] contains a description of an earlier version of the remote interface language and the kinds of copying schemes that the programmer can define with it.

5. THE ASPECT WEAVER

D is implemented as a preprocessor that takes the Jcore component program and the aspect programs and produces a plain Java program. We call this tool the aspect weaver; The D weaver works by intertwining the lines of code of the Jcore program with new lines of code that implement the aspect programs.

The aspect weaver of D, unlike some other AOP weavers, is very simple. This is because the join points between Jcore and the two aspect languages are simply the message sends and receives. That is, Cool and Ridl affect Jcore only on method invocations: Cool controls the beginning and end of method executions; Ridl controls the sending and receiving of RMI messages. Therefore, the weaving is done on method entrances and exits (including `return` and `throw`), and a lot of code is generated for purposes of handling the copying directives of Ridl.

Rather than making an extensive description of the translation algorithms in the weaver, we present the most important ones by showing an example of the weaver's output code. Appendix A shows the result of weaving the programs shown in Figure 2. Different colors show how the different aspects spread out over the resulting program.

Note that there are many ways of implementing in Java the semantics of the aspect programs. In the current implementation of the weaver, we made a number of decisions, all of them favoring simplicity over any other thing; the goal was to have a proof-of-concept prototype in a short period of time. Left out for now, are the many possible optimizations that should certainly produce more efficient code.

6. SOME RESULTS AND ANALYSIS OF THE FRAMEWORK

The benefits of using a programming language over another cannot be measured precisely. They depend, first of all, on the applications, what is of particular importance in them, and also on social variables. With this in mind, we tried, nevertheless, to compare D with Java with respect to the code tangling. We used a number of small applets and classes, mostly from Doug Lea's class repository.[7] Table 1 shows the results. (The classes are available in http://www.parc.xerox.com/aop/d). The quantitative raw material is lines of code, but we used that measure to infer the ratio of "aspectual bloat" of the Java classes with respect to D aspects as follows:

$$aspectual\ bloat = \frac{\#\ lines\ in\ Java - \#\ lines\ in\ JCore}{\#\ lines\ in\ Cool\ and\ Ridl}$$

[7] Lea used these examples in his book[19]. http://gee.cs.oswego.edu/dl/cpj/classes

Classes	Java	D			bloat
		Jcore	Cool	Ridl	
BoundedCounterV1	30	13	11	-	1.5
LockSplitShape	60	44	5	-	3.2
BoundedBufferVC	62	41	6	-	3.5
BoundedBufferVH	65	41	6	-	4.0
PassThroughShape	73	44	5	-	5.8
BoundedBufferVTO	76	25	8	-	6.4
BookLocator + Book	152	74	4	7	7.1

Table 1. ***Comparison between Java source and the corresponding D source code for a number of classes. BoundedBufferVC and BoundedBufferVH, in Java, are two different implementations for the same behavior; they correspond to only one D implementation. The same situation occurs for PassThroughShape and LockSplitShape.***

The aspectual bloat ratio is a rough measure of how much the intentions of the aspect program end up being redundantly coded in the Java classes. The bigger this ratio is, the more unnecessarily complicated the Java class is, and the better it is to use D. We note that there is a tendency to have higher tangling as the classes get bigger. These numbers show the result of our own "laboratory experiments" with respect to D, and we find them very encouraging. More and bigger applications will be necessary before we make any generalized conclusions.

We can also analyze D on a qualitative basis. The D source code is more clear about what's going on with respect to concurrency control and remote access; the textual separation of aspect programs makes it easier to identify the pieces of source code that deal with those issues. A possible counter-argument here is fragmentation. One could say that textual separation leads to code that is more fragmented and therefore more difficult for programmers to track. We think that fragmentation is not necessarily bad; on the contrary, as object-oriented languages have shown, it can be beneficial, especially if the fragments correspond more directly to pieces of design specifications.

Besides the textual separation, the D aspect languages are specific to the issues they handle. An alternative would be, for example, to use Java to program synchronization and remote access, modeling these in separate classes from the Jcore classes. This has been the approach taken by the reflection community. Following that approach, programmers can do anything in the meta-programs. While this is powerful, it is dangerous. We chose these specialized aspect languages because they help the aspect programmer express their intent and guarantee that the aspect programs will not overstep their bounds.

One last remark has to do with the inheritance anomaly, usually associated with object-oriented concurrent programming. (See Matsuoka and Yonezawa [27] for a detailed analysis of this issue) Reflective approaches have been shown to solve this anomaly [26]. Being a form of meta-level programming, D programs don't show the inheritance anomaly either. The aspect weaver automatically generates the methods that need specialization.

7. RELATED WORK

Approaches to concurrent and distributed programming can be roughly grouped into three categories:

- Define distribution facilities on top of the existing definitional mechanisms of a language. This includes all language environments that provide low-level constructs that the programmer can use to manually weave the concurrency control and remote access aspects. The level of abstraction of those constructs varies. In any case, this approach

causes code tangling, due to the cross-cutting nature of the aspects with respect to the abstraction and composition mechanisms of the languages.

Examples of this are several C++ environments [5, 31], the Eiffel environment [17], Smalltalk [33], CORBA implementations. Java [12] is half way between this approach and the next one.

- Extend the sequential, non-distributed, language with syntactically identifiable constructs for handling the aspects, and use a preprocessor to translate those constructs. This approach has the advantage of using well-known languages as the basis. However, the effect of the constructs are localized in particular points, and tangling is still likely to occur.

  Charm++ [16] and DOWL [1] are two examples. Concurrency annotations [22] is half-way between this approach and the next one.

- AOP-like solutions. These approaches provide a different abstraction and composition mechanism for the aspects and the components. They vary in factors such as how different the aspect languages are from the component language and how many new composition mechanisms they need. They have the nice properties of AOP with respect to a clean separation of concerns at the code level.

  Some examples follow. New languages: ABCL [26], Sina [3], Dragoon [6]; making concurrency control independent from the functionality language: synchronizers [10]; using the meta-level: [25, 28].

D enforces separation of concerns, and is implemented as a preprocessor, very much like Dragoon. One of the design principles of D was that it can be applied to existing object-oriented languages. In this respect, D has a similar philosophy as Frølund's synchronizers. In D, this is achieved by making the aspect languages use abstractions of OOP – i.e. class, method, and field names – without ever relying on how a particular OOP language implements them. For that reason, D also relates to approaches that use reflection and the meta-level.

8. CONCLUSIONS

We have presented D, an object-oriented language framework for distributed programming. D uses the aspect oriented programming approach to allow the code for the basic functionality of a distributed application to be written without having to explicitly deal with distribution and synchronization. Separate code deals with those issues.

The motivation for D comes for the observation of the code tangling phenomenon, which we described in Section 3. The abstraction and composition mechanisms at the heart of OOP do a great job of capturing the functionality of the applications, but they don't provide a "natural fit" for supporting the synchronization constraints and communication strategies. The root cause of the tangling is this lack of appropriate abstraction and composition mechanisms for the specific needs of the aspects.

Based on those observations, we designed D as being one object-oriented component language (Jcore) and two aspect languages (Cool and Ridl). Jcore is a subset of Java, and it is used to express the basic functionality and the activity of the system; Cool is used to express coordination of threads; and Ridl is used to express remote access strategies. A special tool called an Aspect Weaver™ takes the programs written in the different languages and combines them together to produce an executable program with the specified distributed behavior.

We have shown some results comparing programs written in D with programs written in plain Java. We see D as having three advantages over all the other approaches to distributed programming: (i) it has the syntactic and semantic separation, that keywords have; (ii) it has the power to write sophisticated aspect programs that act on the Jcore program on a more global level, like the reflective approaches; and (iii) the core language is a widely used object-oriented language, and the aspect languages are mostly independent of it.

Acknowledgments:

To John Lamping for his time, patience and invaluable suggestions, and to Anurag Mendhekar for helping clear out some design issues of D. Thanks also to Karl Lieberherr, Mitch Wand, Will Clinger and Boaz Patt-Shamir from Northeastern University, who gave extensive feedback on the semantics of the aspect languages. Thanks also to the whole AOP group at PARC for their comments and support.

Appendix A. Output of the aspect weaver

```
public class BookLocator
       extends UnicastRemoteServer
       implements BookLocator_ridl {
  // This is just one possible implementation.
  // books[i] is in locations[i]
  private Book     books[];
  private Location locations[];
  private int      nbooks = 0;
  private BookLocator_cool coord;
  // the constructor
  public BookLocator (int dbsize) {
    books = new Book[dbsize];
    locations = new Location[dbsize];
    coord = new BookLocator_cool();
  }
  public void register (Book b, Location l)
  throws LocatorFull, RemoteException {
    coord.enter_register(this);
    if (nbooks > books.length)
      coord.exit_register(this);
      throw new LocatorFull();
    else {
      // Just put it at the end
      books[nbooks] = b;
      locations[nbooks++] = l;
      coord.exit_register(this);
    }
  }
  public void unregister (Book b) {
  throws RemoteException {
    coord.enter_unregister(this);
    Book abook = books[0]; int i = 0;
    while (i < nbooks &&
           abook.get_isbn() != b.get_isbn())
      abook = books[++i];
    if (i == nbooks) {
      coord.exit_unregister(this);
      return;
    }
    // simply shift down the rest
    while (i < nbooks - 1) {
      books[i]= books[i+1];
      locations[i]= locations[++i];
    }
    --nbooks;
    coord.exit_unregister(this);
  }
  public Location locate (String str) {
  throws BookNotFound, RemoteException {
    coord.enter_locate(this);
    Location ret;
    Book abook = books[0];
    int i = 0; boolean found = false;

    while (i < nbooks && found == false) {
      if (abook.get_title().compareTo (str)==0 ||
          abook.get_author().compareTo (str)==0)
        found = true;
      else abook = books[++i];
    }
    if (found == false) {
      coord.exit_locate(this);
      throw new BookNotFound (str);
    }
    ret = locations[i];
    coord.exit_locate(this);
    return ret;
  }
  public static void main(String args[]) {
    try {
       BookLocator obj = new BookLocator(17);
       Naming.rebind("BookLocator", obj);
    } catch (Exception e) {
       System.out.println("BookLocator err: "
                          + e.getMessage());
       e.printStackTrace();
    }
  }
}
```

```
public class Book implements Book_ridl {
  // one possible implementation of Book
  String title, author;
  int isbn;
  Project owner;
  OCRImage firstpage;
  public BookImpl (String t, String a, int n) {
    title = t; author = a; isbn = n;
  }
  public String get_title() { return title; }
  public String get_author() { return author; }
  public int get_isbn() { return isbn; }
  public String _the_title() {return title;}
  public String _the_author() {return author;}
  public int _the_isbn() {return isbn;}
  public Project _the_owner() {return owner;}
  public OCRImage _the_firstpage() {
    return firstpage;
  }
  public Book_ridl
        _BookToBookLocator_register_1() {
    return new BookLocator_register_1(this);
  }
  public Book_ridl
        _BookToBookLocator_unregister_1()  {
    return new _BookLocator_unregister_1(this);
  }
  public Book_ridl
        _BookLocator_register_1ToBook() {
    return null;
  }
  public Book_ridl
        _BookLocator_unregister_1ToBook() {
    return null;
  }
}
class _BookLocator_register_1
      implements Book_ridl {
  public Book o;
  public String get_title() {returns null;}
  public String get_author() {returns null;}
  public String get_isbn() {returns null;}
  public Book_ridl
        _BookToBookLocator_register_1() {
    return null;
  }
  public Book_ridl
        _BookToBookLocator_unregister_1()  {
    return null;
  }
  public Book_ridl
        _BookLocator_register_1ToBook() {
    return o;
  }
  public Book_ridl
        _BookLocator_unregister_1ToBook() {
    return null;
  }
  private void writeObject(ObjectOutputStream s)
  throws NotSerializableException, IOException {
    s.writeObject(o._the_title());
    s.writeObject(o._the_author());
    s.writeInt(o._the_isbn());
  }
  private void readObject(ObjectInputStream s)
  throws ClassNotFoundException, IOException {
    o = new BookImpl((String)s.readObject(),
                     (String)s.readObject(),
                     s.readInt());
  }
}
```

```
class _BookLocator_unregister_1
      implements Book_ridl {
  public Book o;
  public String get_title() {returns null;}
  public String get_author() {returns null;}
  public String get_isbn() {returns null;}
  public Book_ridl _BookToBookLocator_register_1(){return null; }
  public Book_ridl _BookToBookLocator_unregister_1() {return null; }
  public Book_ridl _BookLocator_register_1ToBook() {return  null; }
  public Book_ridl _BookLocator_unregister_1ToBook() {return o; }
  private void writeObject(ObjectOutputStream s)
          throws NotSerializableException, IOException {
    s.writeInt(o._the_isbn());
  }
  private void readObject(ObjectInputStream s)
          throws ClassNotFoundException, IOException {
    o = new BookImpl(null, null, s.readInt());
  }
}
class BookLocator_cool {
  private Lock register = new Lock (false);
  private Lock unregister= new Lock (false);
  private Lock locate = new Lock (false);
  public synchronized void enter_register (BookLocator host) {
    // this provides for re-entrance
    while (register.value && register.t != Thread.currentThread() ||
           unregister.value && unregister.t!= Thread.currentThread() ||
           locate.value && locate.t != Thread.currentThread()) {
      try { wait(); }
      catch (InterruptedException e) {}
    }
    register.value = true;
  }
  public synchronized void exit_register (BookLocator host) {
    register.value = false;
    notifyAll();
  }
  public synchronized void enter_unregister (BookLocator host) {
    while (register.value && register.t != Thread.currentThread() ||
           unregister.value && unregister.t!= Thread.currentThread() ||
           locate.value && locate.t != Thread.currentThread()) {
      try { wait(); }
      catch (InterruptedException e) {}
    }
    unregister.value = true;
  }
  public synchronized void exit_unregister (BookLocator host) {
    unregister.value = false;
    notifyAll();
  }
  public synchronized void enter_locate (BookLocator host) {
    while (register.value && register.t != Thread.currentThread() ||
           unregister.value && unregister.t != Thread.currentThread()){
    while (unregister.value || register.value) {
      try { wait(); }
      catch (InterruptedException e) {}
    }
    locate.value = true;
  }
  public synchronized void exit_locate (BookLocator host) {
    locate.value = false;
    notifyAll();
  }
}
public interface BookLocator_ridl extends java.rmi.Remote {
  void register (Book_ridl b, Location_ridl l)
       throws java.rmi.RemoteException, LocatorFull;
  void unregister (Book_ridl b) throws java.rmi.RemoteException;
  Location_ridl locate (String str)
       throws java.rmi.RemoteException, BookNotFound;
}
public interface Book_ridl extends Serializable {
  String get_title();
  String get_author();
  int get_isbn();
  public Book_ridl _BookToBookLocator_register_1();
  public Book_ridl _BookToBookLocator_unregister_1();
  public Book_ridl _BookLocator_register_1ToBook();
  public Book_ridl _BookLocator_unregister_1ToBook();
}
```

Some comments:

1 - The reason why this code is considerably longer than the one in Figure 1 (the original Java program) is that this code implements two copying schemes, while the original Java program implements only one.

2 - The green code shown here implements those copying directives. It does so by class conversions between the Book class and the two related classes that implement the two different copying strategies.

3 - The important links that are missing are the calls to the type conversions. Those calls happen in the stub and skeleton of the BookLocator class.

4 - With respect to synchronization, and as shown, the coordinator is implemented as a class. In this case, there is one instance of the coordinator class per instance of the BookLocator class.

Bibliography:

1. Achauer B., *Implementation of distributed Trellis*, in proc. *ECOOP*, pp. 103-117, Kaiserslautern, Germany, 1993.
2. Agha G. and Hewitt C., *Concurrent programming using Actors*, in *Object-Oriented Concurrent Programming* : 37-53, *Series in Computer Science*, Yonezawa A. and Tokoro M., Eds.: MIT Press, 1987.
3. Aksit M., Wakita K., et al., *Abstracting object interactions using composition filters*, in proc. *ECOOP'93 Workshop on Object-Based Distributed Programming*, pp. 152-184, 1993.
4. America P., *POOL-T: A parallel object-oriented language*, in *Object-Oriented Concurrent Programming* : 55-86, Yonezawa A. and Tokoro M., Eds.: MIT Press, 1987.
5. Assenmacher H., Breitbach T., et al., *PANDA -- Supporting distributed programming in C++*, in proc. *ECOOP*, pp. 361-383, Kaiserlautern, Germany, 1993.
6. Atkinson C., *Object-oriented Reuse, Concurrency and Distribution: an Ada-based approach*. Book published by ACM Press and Addison-Wesley, 1991.
7. Bal H. E., Tanenbaum A. S., et al., *Orca: A Language for Distributed Programming*, Vrije Universiteit 1987.
8. Black A., Hutchinson N., et al., *Distribution and abstract data types in Emerald*, in *IEEE Transactions on Software Engineering*, vol. SE-13(1): 65-76, 1987.
9. Booch G., *Object-oriented design with applications*. Book published by Benjamin/Cummings Publishing Company, 1991.
10. Frølund S. and Agha G., *A language framework for multi-object coordination*, in proc. *ECOOP*, pp. 346-360, Kaiserslautern, Germany, 1993.
11. Gamma E., Helm R., et al., *Design patterns -- elements of reusable object-oriented software*. Book published by Addison-Wesley, 1994.
12. Gosling J., Joy B., et al., *The Java™ Language Specification*. Book published by Addison-Wesley, 1996.
13. Hoare C., *Communicating Sequential Processes*, in *Communications of the ACM*, vol. 21(8): 666-677, 1978.
14. Jacobson I., Christerson M., et al., *Object-oriented software engineering - a use case driven approach*. Book published by Addison-Wesley, 1992.
15. Java, *Java Remote Method Invocation Specification, Revision 1.0*, JavaSoft 1996.
16. Kale L. and Krishnan S., *CHARM++: a protable concurrent object oriented system based on C++*, in proc. *OOPSLA*, pp. 91-108, Washington, DC, 1993.
17. Karaorman M. and Bruno J., *Introducing concurrency to a sequential language*, in *Communications of the ACM*, vol. 36(9): 103-116, 1993.
18. Kiczales G., Lamping J., et al., *Aspect-Oriented Programming*, Xerox PARC, Palo Alto, CA, Submitted to OOPSLA'97 Available upon request, 1997.
19. Lea D., *Concurrent Programming in Java™: design principles and patterns*. Book published by Addison-Wesley, 1996.
20. Lieberherr K., *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns*. Book published by PWS, 1996.
21. Liskov B., Curtis D., et al., *Implementation of Argus*, in *11th ACM Symposium on Operating Systems Principles* : 111-122. Austin, Texas U.S.A.: SIGOPS and ACM and MCC, 1987.
22. Löhr K.-P., *Concurrency annotations for reusable software*, in *Communications of the ACM*, vol. 36(9): 90-101,

1993.

23. Lopes C. V., *Adaptive parameter passing*, in proc. *2nd International Symposium on Object Technologies for Advanced Software*, pp. 118-136, Kanazawa, Japan, 1996.

24. Lopes C. V. and Lieberherr K., *Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications*, in proc. *European Conference on Object-Oriented Programming*, pp. 81-99, Bologna, Italy, 1994.

25. Masuhara H., Matsuoka S., et al., *Object-oriented concurrent reflective languages can be implemented efficiently*, in proc. *OOPSLA*, pp. 127-144, Vancouver, Canada, 1992.

26. Matsuoka S., Taura K., et al., *Highly efficient and encapsulated reuse of synchronizarion code in concurrent object-oriented languages*, in proc. *OOPSLA*, pp. 109-126, Washington, DC, 1993.

27. Matsuoka S. and Yonezawa A., *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages*, in *Research Directions in Concurrent Object-Oriented Programming* : 107-150, Agha G., Wegner P., et al., Eds.: MIT press, 1993.

28. McAffer J., *A Meta-level architecture for prototyping distributed object systems*, PhD Thesis, Department of Information Science, University of Tokyo, Tokyo, 1995.

29. Milner R., Parrow J., et al., *A Calculus of Mobile Processes, I, II*, in *Information and Computation*, vol. 100(1): 1-40;41-77, 1992.

30. Rumbaugh J., Blaha M., et al., *Object-oriented modeling and design*. Book published by Prentice Hall, 1991.

31. Sousa P., Sequeira M., et al., *Distribution and Persistence in the IK platform: overview and evaluation*, in *Usenix Computing Systems*, vol. 6(4): 391-424, 1993.

32. Yokote Y., *The Apertos Reflective Operating System: the concept and its implementation*, in proc. *OOPSLA*, pp. 414-434, Vancouver, Canada, 1992.

33. Yokote Y. and Tokoro M., *Concurrent programming in ConcurrentSmalltalk*, in *Object-oriented Concurrent Programming* : 129-158, *MIT Press Series in Computer Systems*, Yonezawa A. and Tokoro M., Eds.: MIT Press, 1987.

34. Yonezawa A., *ABCL: An Object-Oriented Concurrent System*. Book published by MIT Press, Cambridge, MA, 1990.