# Do the Fish Really Need Remote Control? A Proposal for Self-Active Objects in Oberon

Jürg Gutknecht, ETH Zürich

**Abstract**

Based on the language Oberon we propose a unified framework for concurrent, object-oriented programming. Inspired by simulation, the idea is to regard objects as processes in contrast with the more common approach treating processes as objects. More concretely, our framework extends the original Oberon language by four new concepts: (a) Object-centered access protection, (b) object-local activity control, (c) system-guarded assertions and (d) preemptive priority scheduling. (a) and (b) are expressed syntactically by upgraded record types, (c) by a passivation/activation mechanism and (d) by a priority option. None of the conventional facilities like semaphores, locks, critical regions, signals, channels, rendez-vous, forks etc. are primitive constructs in our framework. Currently, an implementation of a compiler, a corresponding runtime kernel and a non-blocking local server exists for native Intel architectures.

Keywords: Object Oriented Programming, Active Objects, Concurrency, Multiprogramming, Oberon.

## 1 Introduction

The technology of object-oriented software construction is a manifestation of the remarkable level that has been reached in this area in terms of abstraction, modularity, extensibility and reusability. However, at least in practically available systems, the conceptual combination of objects with processes has been neglected. Merely re-acting to messages, objects are typically passive and remote controlled by concept. This is regrettable in several respects, primarily in view of the desire for a high degree of coherence in programs on one hand and for most possible self-containedness of objects on the other hand. This latter aspect is of particular importance in connection with portable end-user objects that are now available, for example, via Internet in the form of "plug-ins" and "applets".

## 2 Processes and Objects

From a conventional point of view, the basic ingredients of a program are *processes* and *objects*, coupled by the obvious *operating-on* relation. Correspondingly, software research has pursued the conceptual development of processes and objects separately and has paid little attention to their integration or even their unification. In an object-oriented environment, multiprocess functionality is commonly provided in the form of a separate hierarchy of process classes, thereby emphasizing separation rather than unification. Smalltalk and Java [1] are prominent representatives of this kind. Due to its non-symmetric rules of inheritance among the two kinds of objects (active and passive), Synchronous C++ [2] is a further example of a separating architecture.

A different and somehow dual approach is taken by simulation programming. Rather than treating processes as objects, simulation programming treats objects as processes. Simulated objects often show a hybrid nature expressing itself in a continuous alternation of active and passive phases. For example, a simulated device takes an active role when operating and a passive role when being serviced, and a customer in a shopping center can be considered active when browsing around the shops and passive while waiting in a queue for check-out.

The two views, conventional and simulation, are substantially different. The essential point is that, in the conventional case, objects are basically passive in the sense that they are completely controlled by remote processes via message passing in contrast to the case of simulation, where objects are self-controlling their activity in principle or, in short, are *self-active*. We prefer the simulation view because it is more general (passive objects can be regarded as active objects with an empty active phase) and because it leads to more coherent mappings from given specifications to programs.

A trivial but entertaining example is the well-known animation of an underwater world (often used as a screen-saver) that mimics a random population of fish with some greedy sharks among them. Most naturally, the life-story of every single fish is defined intrinsically by some type-specific control-program that operates on its internal state: Ordinary fish move peacefully unless they are caught by a shark. Should this case occur, they change to a skeleton and sink down to the ground. Sharks are harmless unless they are hungry. When they are, they snatch any fish crossing their path coincidentally. We shall take up this example again in Section 4 when we present a sketch of a corresponding Oberon program.

## 3 A Unified Framework

Inspired by simulation programming in general and by the Simula language family [3] (the very origin of object-oriented programming) in particular, we have experimented with modelling self-active objects on the Oberon language and environment [4], [5]. The result is a unified framework for concurrent, object-oriented programming. It can briefly be summarized in terms of four new concepts on the level of individual objects that, together, constitute the core of our framework: (a) Protected access, (b) local activity control, (c) guarded assertions and (d) preemptive priority scheduling. (a) and (b) support the *competitive* aspects of concurrency, (b), (c) and (d) its *collaborative* aspects.

These concepts are integrated smoothly into the Oberon language by an upgrade of record types and the addition of one keyword and one built-in procedure.

We now present our unified framework in three steps: (a) Introduction of the new language constructs, (b) explanation of the new object concepts and (c) illustration of their combination by two comprehensive program examples from different areas.

### Upgraded Record Types

Record types are used in Oberon as a model for object classes. A record type can be viewed as a production pattern for object instances. The two kinds of components of a record type are data variables and procedure variables. They are mapped at creation time to the new object's state variables and its (instance-specific) methods respectively.

New types can be derived from an existing base-type by extension of the set of components. Such derived types inherit the base-type's structure and therefore correspond to subclasses in object-oriented terminology.

Like modules, record types define a static scope. It is now natural to use this scope for the specification and implementation of local functionality, in particular for (a) *controlled access* to local data and (b) implementation of local activities, i.e. of *intrinsic behavior*. For this purpose, we upgrade Oberon record types by (a) optional procedure entries for controlled access and (b) an optional record body for the implementation of intrinsic behavior.

A further syntactical upgrade concerns an *option specification clause* for statement blocks, i.e. sequences of statements delimited by BEGIN ... END brackets. In original Oberon, statement blocks are scheduled unconditionally and uniformly as sub-routines of the (one and only) system process. In our new multi-process environment, however, different possible scheduling options exist as, for example, exclusive access, separate thread of control and special priority. Accordingly, the syntax now allows a list of runtime options and directives to be specified within a pair of curly braces immediately following the BEGIN keyword of a statement block.

Example:

```
TYPE
  Object = RECORD (Object0) (* derived from base-type Object0 *)
    VAR
      t: T; (* state variables *)
      h: H; (* instance-specific methods *)

    PROCEDURE P (t: T); (* procedure entry for controlled access *)
    BEGIN { optionlist } (* option specification clause *)
```

```
    ... (* statement sequence *)
    END P;
  BEGIN (* record body implementing intrinsic behavior *)
    { optionlist } (* option specification clause *)
    ... (* statement sequence *)
  END Object;
```

Remarks:

(1.) Procedure entry implementations are inherited in type-extensions unless they are overridden by a new implementation (under the same name and with the same parameter list). Within the scope of the new implementation, the original version can be identified by adding a "$\uparrow$" to its name. This notation is borrowed from Oberon-2 [6], where it is used for a similar purpose.

(2.) Within the scope of a record type, (unqualified) names of its variables and procedures always refer to components of the current object. The current object as a whole is denoted in the record scope by the new keyword SELF.

(3.) The section keyword VAR and the repetition of the record type's name after the keyword END are optionally allowed as a concession to syntactical conformity with the module construct.

Of course, such far-reaching conformity of object types and modules on the syntactical level should reflect itself on the semantic level. It is therefore justified to clarify the semantical relationship between (upgraded) record types and modules.

### Upgraded Record Types versus Modules

We first remember that, by definition, an Oberon module is a set of logically connected ingredients like type-declarations, variables, procedures and body. Partitioning this set in two, type-declarations and complement, the module can be regarded as a (possibly empty) collection of explicit type-declarations and an implicit (upgraded) record type that we call *module type.*

Module types are anonymous and cannot be referred to explicitly, that is they cannot be extended, nor can instances be created explicitly. The one and only instance of a module type is created automatically at loading time: The module itself. However, apart from the anonymity of their type, modules are like any other object and are therefore treated as such consequently by our unified framework. In particular, whenever we refer to object in general, modules are included. Modules as objects are beneficially used to model system-wide resources and services.

It is noteworthy that the presented upgrade of record types significantly increases the language's uniformity and "orthogonality", so that it appears to programmers and compiler-writers as a removed restriction much rather than as extension.

After this formal introduction of syntactic constructs, we are now prepared to focus on their use for the construction of the desired unified framework.

### Access Protection

Independent of any self-activity, an object is still an ob-ject and can therefore be used as a *resource*. Only in the simplest of all scenarios, if the object is passive and under exclusive possession of some specific process, no access control is needed. In all other cases, multiple processes may potentially compete for the use of the object, so that some kind of access protection is indispensible. For this reason, we introduce two protective options EXCLUSIVE and SHARED for procedure entries.

We postulate *mutual exclusion* among the processes requiring service from an object via any procedure entry marked EXCLUSIVE. This means that, for every single instance of an object type, one process at most is allowed to be active in the set of its procedures marked EXCLUSIVE at any time. In other words, every process entering this set *locks* the object for the time of its active presence.

The option SHARED refers to a weaker form of access protection. Procedure entries with this option can be shared by an arbitrary number of processes as long as no process is active in an EXCLUSIVE entry. The SHARED option is typically used for *read-only* access.

Procedure entries neither specified as EXCLUSIVE nor as SHARED are unprotected and can be entered unconditionally. They are assumed to implement their own access control.

Example:

```
TYPE
  Object = RECORD
    VAR t: T;

    PROCEDURE P (t: T);
    BEGIN { EXCLUSIVE } ... (* mutually exclusive access*)
    END P;

    PROCEDURE Q (): T;
    BEGIN { SHARED } ... (* shared access *)
    END Q;

    PROCEDURE R (): T;
    BEGIN ... (* unprotected *)
    END R;

  END Object;
```

For any given instance $x$ of type *Object*, multiple processes are allowed to enter x.Q simultaneously. However, if any process enters x.P, mutual exclusion is required and no other process is allowed to enter either x.P or x.Q as long as the first process is still active in x.P. Any process may enter x.R at all times.

**Local Activity Control**

An object is rarely a pure resource that is used by remote processes only. Typically, the object itself develops some kind of *intrinsic behavior*. We already know that the intrinsic behavior of an object is expressed by its body part, running as a separate and local *thread of control*. In detail, several aspects have to be considered carefully.

A first question arises in connection with initialization. A special mechanism is required because an object must normally be initialized consistently by external parameters before its local process can start. For this purpose, an *initializer* can be distinguished among the procedure entries in the object type by an "&" tag. The way of parameter specification at creation time depends on the kind of the object's type. In the cases of pointer-based types and static types, the actual parameters of the initializer are simply added to the NEW statement's parameter list and to the type specification respectively. For module objects no parameters are allowed.

In the case of extending types several bodies along the base-type chain may exist. By definition, the *total behavior* of an object is the *parallel composition* of these bodies. Objects without a body along their base-type chain or objects whose bodies have been terminated are invariantly passive.

The creation of a general object is thus an atomic three-step action:

Create object = { allocate memory block; call initializer and pass parameters; create and start a separate process for each body in the base-type chain }.

In concluding this section we note that the above introduced protective options EXCLUSIVE and SHARED for procedure entries are applicable in principle to object bodies as well. However, most local processes should not block their object permanently and, therefore, run in unprotected mode. Unprotected local processes access their object's shared data via the official protected interface - as any remote process does.

Example:

```
TYPE
  Object0 = POINTER TO ObjDesc0;
  ObjDesc0 = RECORD (* base-type *)
  BEGIN
    (* local thread of control,
    implementing base part of intrinsic behavior,
    running concurrently with local threads in extending types,
    unprotected *)
    ...
  END ObjDesc0;

  Object = POINTER TO ObjDesc;
  ObjDesc = RECORD (ObjDesc0) (* derived from base-type ObjDesc0 *)
```

```
    VAR
      t: T; (* state variables *)
      h: H; (* instance-specific methods *)

    PROCEDURE P (t: T); (* procedure entry for controlled access *)
    BEGIN { EXCLUSIVE } ... (* statement sequence *)
    END P;

    PROCEDURE& Init (t: T; h: H); (* tagged as initializer *)
    BEGIN ... (* statement sequence, automatically called at creation time *) ...
    END Init;

  BEGIN
    (* local thread of control,
    implementing extended part of intrinsic behavior,
    running concurrently with local threads in base type,
    unprotected *)
    ...
  END ObjDesc;

VAR x: Object;
BEGIN ... NEW(x, myT, myH); ... (* creation of pointer-based object *)
END

VAR X: ObjDesc(myT, myH); (* creation of static object *)
```

### Guarded Assertions

Assertions are an important concept in programming. They are typically used at critical points in a program as a safeguard against erroneous continuation. Let us imagine the guard as a universal demon behind the scenes. In a single-process environment where assertions in a given program state are invariantly valid or invalid, the guard simply either lets through or aborts. In a multiprogram, however, the validity of a given assertion may be changed by partner processes and the guard's task is getting more intricate. Instead of just aborting a process in case of an invalid assertion, the guard should rather suspend this process, watch the assertion and resume the process at a later time when the assertion is valid.

We have adapted the concept of assertions to active objects. In this context, suspending an object's process amounts to *passivating* the object, that is to diminishing or to freeze its activity. The built-in operation PASSIVATE serves exactly this purpose. More precisely, the effect of calling PASSIVATE with condition parameter c within object (or module) scope S is

PASSIVATE(c) =
{ suspend current process; unlock S; await c; lock S; resume process }.

The difficult part to implement in the run-time system is *await c* that, of course, must run concurrently with the partner processes. A straightforward solution for the frequent case of c *local* to S is checking c as a side-effect of unlocking S.

Note that this implementation is efficient in terms of context-switches, because no context-switch is needed for the evaluation of the condition. Also useful are global *stable conditions* that are not falsified by peer processes. Such conditions can easily be watched by some periodic system process. A separate report [7] in detail discusses an implementation of self-active objects for the Intel architecture.

### Preemptive Priority Scheduling

Regarded on a somewhat lower level of abstraction, the topic of guarded assertions is nothing but the topic of synchronous process scheduling in a different clothing, tailored to our framework of self-active objects. The purpose of this section is a brief discussion of the complementary topic of *asynchronous scheduling* that, within our framework, is based on (a) a priority scheme and (b) time-slicing.

There is only one syntactical feature for the support of implicit scheduling, namely an optional *priority* option: PRIORITY(level). Possible priority levels are non-negative integers, where 0 is lowest priority. When a priority option is omitted, priority level 0 is ssumed.

The essential rule guarantees that no process of any given priority is allowed to progress as long as any process of a higher priority is able to progress (i.e. active and not blocked by some lock of mutual exclusion). This rule implies potential preemption of any process of a priority level lower than the maximum. In particular note this corollary: A precondition for an object to be reactivated in a certain scope is the absence of any higher priority objects that are active in the same scope.

Time-slicing is a processor-sharing scheduling strategy that concedes a time-slice of a certain length to each process before suspending it again in favor of another process. While the use of explicit assertions is far preferable to asynchronous preemption in most cases, time slicing can sometimes be justified to allow a group of independent (and perhaps never-ending) processes to progress simultaneously. In such situations, the option TIMESHARED should be included in the body of an object type as a hint to the runtime system. Note that the implementations of preemptive priority scheduling and time-slicing are very similar. The only difference is that, after suspension of a time-sliced process at the end of a slice, the set of candidates for resumption may include processes on the same priority level.

## 4 Series of Program Examples

### Selective Lister: A Case of Structure Conflict

This first example is a member of a class of problems that have been used to show the benefits of coroutines for the design of seemingly ordinary uniprograms (one process only). The task is the generation of several lists from one input stream of typed items, so that each item goes to one or more lists, according to its type, and each page of each list is headed by a title.

More concretely, the following program produces two lists *List1* and *List2*, where all items of type 1 and type 2 go to *List1* and *List2* respectively and all other items go to both lists. Lists are active objects. They use a one-element buffer for structural decoupling and resolution of the structure conflict. Their type is roughly defined as

```
TYPE
  List = POINTER TO ListDesc;
  ListDesc = RECORD
    VAR buf, x: Item; line: INTEGER;

    PROCEDURE Enter (x: Item);
    BEGIN { EXCLUSIVE } PASSIVATE(buf = NIL); buf := x
    END Enter;

    PROCEDURE Get (VAR x: Item);
    BEGIN { EXCLUSIVE } PASSIVATE(buf # NIL); x := buf; buf := NIL
    END Get;

    PROCEDURE& Init (listName: ARRAY OF CHAR);
    BEGIN buf := NIL
    END Init;

  BEGIN Get(x);
    WHILE type of x # 0 DO PrintTitle;
      line := 0;
      REPEAT PrintItem(x); INC(line); Get(x)
      UNTIL (type of x = 0) OR (line = MaxLine);
      ShowPage
    END
  END ListDesc;
```

The relevant part of the list generator then looks like this:

```
NEW(L1, "List1"); NEW(L2, "List2");
REPEAT Read(x);
  IF type of x = 1 THEN L1.Enter(x)
    ELSIF type of x = 2 THEN L2.Enter(x)
    ELSE L1.Enter(x); L2.Enter(x)
  END
UNTIL type of x = 0
```

### Underwater World: A Real-Time Animation

Our second example is the animation of an underwater world, referred to in the title of this article and at the beginning. It is a caricature of a real-time program. Participating objects are (ordinary) fish and sharks. Both kinds of object basically move uniformly and straight with a certain probability of changing direction a little every once in a while. When hungry, sharks catch any fish that happens to cross their way. If they do not succeed, they finally starve. Starved or caught fishes sink down to the ground as skeletons.

On one hand, module *UnderwaterWorld* provides two types *Fish* and *Shark*. They essentially represent moving objects with some generic behavior determining their life-story. On the other hand, the module stands for the entire population of fish. Together with the functionality for finding a victim, it is therefore a complex resource object itself that is readily used by sharks.

Notice that types *Fish* and *Shark* are derived from a purely passive base type *Object* that implements general objects in a two-dimensional space together with functionality for (a) moving to a new position within a given (incremental) amount of time and (b) deciding about vicinity to a given area. We emphasize that any "active" reuse of a type whose designer has not taken specific precaution would not be possible in combination with a separating framework based on some root class of active objects as, for example, *Threads* in Java [1].

We should also point out the formal similarity of this real-time example and typical simulation examples. Both use a *Hold* statement for the description of an invariant phase. However, while simulation typically refers to some notion of *virtual* time, *Hold* in the current case refers to *real* time and is provided by the base type *Kernel.Object*.

```
TYPE
  (* passive base object *)
  Object = POINTER TO ObjDesc;
  ObjDesc = RECORD (Kernel.ObjDesc)
    VAR prev, next: Object; X, Y: INTEGER;

  PROCEDURE Move (dT: REAL; dX, dY: INTEGER);
    BEGIN { EXCLUSIVE } Hold(dT); X := X + dX; Y := Y + dY
    END Move;

    PROCEDURE IsClose (X0, Y0: INTEGER): BOOLEAN;
    BEGIN { SHARED } (* decide about vicinity and return result *)
    END IsClose;

    PROCEDURE& Init (X0, Y0: INTEGER);
    BEGIN X := X0; Y := Y0
    END Init

  END ObjDesc;

  (* active objects *)
  FishDesc = RECORD (ObjDesc)
    VAR dX, dY: INTEGER;
  BEGIN
    LOOP
      (* calculate next step dX, dY in current direction *)
      Move(dT, dX, dY);
      IF next = NIL (* caught *)THEN EXIT END
```

```
    END;
    LOOP
      (* calculate next step dX, dY in sinking direction *)
      Move(dT, dX, dY)
      IF Y <= 0 THEN EXIT END
    END
  END FishDesc;

  SharkDesc = RECORD(ObjDesc)
    VAR dX, dY: INTEGER; obj: Object;
  BEGIN
    LOOP
      (* calculate next step dX, dY in current direction *)
      Move(dT, dX, dY);
      IF (* hungry *)THEN FindVictim(X, Y, obj);
        IF obj IS Fish THEN (* at it *)
          ELSIF (* starved *)THEN EXIT
        END
      END
    END;
    LOOP
      (* calculate next step dX, dY in sinking direction *)
      Move(dT, dX, dY);
      IF Y <= 0 THEN EXIT END
    END
  END SharkDesc;

(* fish population *)
VAR pop: Object;

PROCEDURE Include (pop, obj: Object);
BEGIN { EXCLUSIVE }
  obj.prev := pop; obj.next := pop.next;
  pop.next.prev := obj; pop.next := obj
END Include;

PROCEDURE Exclude (obj: Object);
BEGIN { EXCLUSIVE }
  obj.next.prev := obj.prev; obj.prev.next := obj.next;
  obj.prev := NIL; obj.next := NIL
END Exclude;

PROCEDURE FindVictim (X, Y: INTEGER; VAR obj: Object);
BEGIN { EXCLUSIVE } obj := pop.next;
  WHILE (obj IS Fish) & ~obj.IsClose(X, Y) DO obj := obj.next END;
  IF obj IS Fish THEN Exclude(obj) END
END FindVictim;
```

## 5 Conclusion and Outlook

The framework presented, that is the set of new concepts together with the corresponding language constructs, shows several attractive properties, distinguishing it from existing and more pragmatic solutions. Most important, the framework is (a) unified, (b) complete and (c) minimal. (a) and (b) can be paraphrased as providing sufficient expressive power for a natural formulation of every possible object-oriented program, every possible process-oriented program (multiprogram, concurrent program) and every combination thereof. (c) means that none of the concepts can be removed without sacrifying property (b).

Our unified framework suggests a programming style that formally resembles simulation programming. However, it is more general thanks to the institution of guarded assertions that allows any kind of genuine concurrency, including parallelism modelled on a multiprocessor architecture.

With the condition-based guarded assertions, the usual self-administration of peer processes (typically implemented by mutual signalling) is replaced in our framework by a systemwide scheduler. A similar (but less rigorous) step has been taken by Ada95 with guarded entries [8]. We believe that the trend to delegate responsibilities from individual participants to the operating system is equally beneficial in the area of multiprocessing as it was in the area of memory management and garbage collection.

Interestingly, the delegation of responsibilities to the operating system has the very desirable side effect of a move of lower-level mechanisms to behind the scenes. In fact, none of the low-level institutions like semaphores and forks traditionally used for multiprogramming are part of the unified framework in their raw form but are, possibly, used as implementation tools behind the scenes. Nevertheless, the framework is expressive enough to support constructs like *locks* etc. for fine-grained access protection on demand.

It is perhaps desirable to briefly compare our Active Oberon framework with Java's thread system. We already mentioned that Java is a representative of the "separating" culture that roots "active" classes in a base class called *Threads*. This is different from our approach in the sense that Java subordinates concurrency to inheritance by concept, while we clearly aim at a decoupling of "orthogonal" combination of objects and threads from inheritance. Java class designers need to specifically design "runnable" objects as such and, in particular, they cannot reuse passive classes by inheritance. However, this problem could be healed practically by preventively deriving every potentially "runnable" Java class from class Threads.

One significant difference still remains. Unlike our object bodies, Java threads are not composable along subclass chains, because they are represented in the form of a method. Parallel composition of behavior is sometimes structurally desirable, as the following archetypal example of objects driven by a *consumer-producer* scheme shows. Assuming the existence of some generic base type producing a buffered stream of tokens, we can construct any specific processor of such a stream simply by composing this base type with a subtype that contributes to the object's total behavior by an application-specific consumer process.

Two potential advantages of Active Oberon over Java are its assertion-based concept of synchronous process scheduling and the SHARED option for controlled access that essentially represents a built-in version of the *readers-and-writers* paradigm.

We have implemented our framework on the base of a version of the OP2 Oberon compiler and on the native Oberon kernel for the Intel architecture. The current implementation of the runtime kernel is primarily designed and used for a versatile and non-blocking local Oberon server that has been in successful use since July 1996. A next and more refined version of a runtime system (in terms of stack management and scheduling strategy) that is in addition able to make beneficial use of multiple physical processors is currently under development.

## Acknowledgement

A successful implementation is always a final legitimation of a design. Rarely, however, is an implementation so demanding and its quality of such paramount importance as in the case of a programming framework. In this sense, my special respect and thanks go to the implementation team consisting of Andreas Disteli (runtime kernel and server) and Patrick Reali (compiler). Their expertise and continuous feedback substantially contributed to the advanced state of the project. The implementors themselves could not have succeeded without the sound and professional basis that was laid earlier by Pieter Muller (native Intel kernel) and the development team of the Oberon OP2 compiler. I no less greatfully acknowledge their work. My best thanks may further reach the audience of an early talk that I gave on the subject at DEC SRC, Palo Alto, and that resulted in a stimulating discussion and in some most valuable suggestions. Last but not least, I thank Martin Reiser and the referees for their apt comments.

## References

[1] J. Gosling, F. Yellin, The Java Team,
The Java Application Programming Interface, Volume 1, Addison-Wesley, 1996.

[2] A. Divin, G. Caal, C. Petitpierre,
Active Objects: A Paradigm for Communications and Event Driven Systems, Proceedings GLOBECOM'94.

[3] K. Nygaard, O.-J. Dahl, Simula 67,
History of Programming Languages (R.W. Wexelblat, ed.), Addison-Wesley, 1981.

[4] N. Wirth, The Programming Language Oberon,
Software – Practice and Experience, 18(7), 671-690.

[5] N. Wirth, J. Gutknecht, Project Oberon,
Addison-Wesley, 1992.

[6] H. Mössenböck, N. Wirth, The Programming Language Oberon-2, Structured Programming, 12, 179-95.

[7] A. Disteli, P. Reali, Combining Oberon with Active Objects, Proceedings of the JMLC, Linz, Austria, 1997.

[8] J.G.P. Barnes, Programming in Ada,
Addison-Wesley, 1994.