Active Oberon Language Report

Patrik Reali
Institut für Computersysteme, ETH Zürich
reali@inf.ethz.ch

October 27, 2004

1 Introduction

Active Oberon is an extension of the original Oberon language [29, 30]. Its purpose is to introduce features into the language to express concurrency by means of *active objects*. This report assumes that you already know Oberon; only the extensions to it are described here.

The design of the language extension has been driven by the search for unification and symmetry. The changes are based on established concepts such as scopes and locality. The rationale behind Active Oberon is described in [10].

1.1 Acknowledgments

Many thanks to B. Kirk, A. Fischer, T. Frei, J. Gutknecht, D. Lightfoot, and P. Muller for reviewing this document and improving it with corrections and improvements, and to Vladimir Los for improving the barriers example.

1.2 History and Related Work

Programming language development at ETH Zurich has a long reaching tradition. The Oberon language is the latest descendant of the Algol, Pascal, and Modula family. Pascal [16] was conceived as a language to express small programs; its simplicity and leanness made it particularly well-suited for teaching programming. Modula [28] evolved from Pascal as a language for system programming, and benefited from the practical experience gained during the development of the Lilith [22] workstation and of the Medos [17] operating system. The need to support the programming-in-the-large paradigm was the motivation for Oberon [29]. The Ceres [6] and Chameleon [12] platforms, and the Oberon operating system [11] projects were developed in parallel with the language, and allowed to test and evaluate the language improvements.

Many experimental language extensions have been proposed for Oberon at, and outside of the ETH. Object Oberon [19], Oberon-2 [20], and Froderon [7] explored adding further object-oriented features to the language; Oberon-V [9] proposed additions for parallel operations on vector computers; Oberon-XSC [15] added mathematical features to support scientific computation; module embedding [24] was also proposed. Concurrency was first added to the operating system through a specialized system API in Concurrent Oberon [25] and

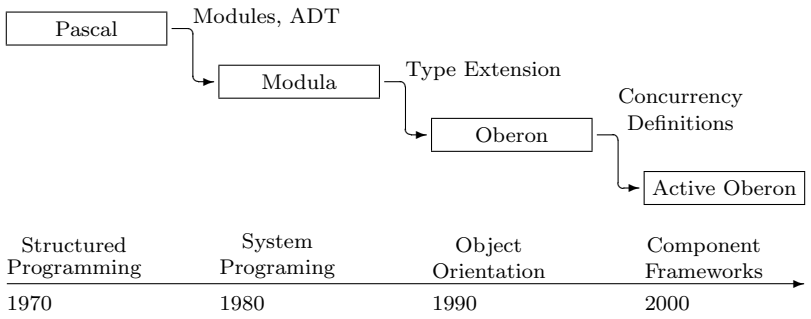

Figure 1: The Pascal Language Family Evolution

XOberon [2]; attempts to model concurrency in the language itself were also done by Radenski [23].

Active Oberon is the first exponent of a new generation of languages in this family. Our motivation is to support concurrency and component modeling in the language in a clean, seamless way.

1.3 Language Design

The design of Active Oberon was influenced by the experiences made with Object Oberon and Oberon-2. We follow the Object Oberon notation for declaring classes and methods, because we think it is more expressive than the one in Oberon-2: methods belong to the class scope and therefore they must be declared there; this way, other methods and fields belonging to the record scope can be accessed without explicit qualifier. Protection against concurrent access through the EXCLUSIVE modifier is easier to read if the methods are declared in the same scope. Active Oberon departs from the Object Oberon design, in that records are generalized to be both classes and records, instead of letting classes and records co-exist in the same system. Another important difference is the decision to let the compiler handle forward references. The syntax of Object Oberon and Oberon-2 was designed to simplify the compiler construction, whereas we chose to simplify the programmer's task by avoiding the needless redundancy of forward declarations and declarations, leaving it to the compiler to handle them.

Java [8] and C# [4] share some similarities with Active Oberon. They are both object-oriented languages stemming from the imperative language world, and the concurrency protection mechanism with object instance bound monitors is the same. On the other hand, they both put the accent on the object-orientation in such an extreme manner, that class methods and class fields seem just special cases of the instance fields and methods, because they belong to a class namespace. Furthermore, Java has no support at all for statically allocated structures: everything is dynamic, even user-defined constant arrays; for this reason, to perform at an acceptable speed, Java programs must rely on complicated and expensive compiler optimizations. All languages in the

Oberon family treat modules and classes as orthogonal concepts, each has its own scope; the module semantics is different from a class' semantics as shown in [26] (for a comparison, B. Meyer advocates the opposite [18]): modules group static components and related implementations, and provide a deployment and structuring primitive. In fact, Java and C# had to introduce concepts like packages, namespaces and assemblies, that de facto reintroduce modules with just another name. We think that static structures and modules still have a very valid reason to be part of a programming language.

The AWAIT statement was proposed and investigated by Brinch Hansen [3] who showed its conceptual simplicity and elegance, but also thought it would be impossible to implement it efficiently. We repropose it in Active Oberon, with the conviction that it is a real improvement compared with signals and semaphores, because of the unification and clarity it brings; it becomes particularly obvious in an object-oriented programming style, where signals and semaphores are completely inappropriate because of their unstructured use, as they can be inserted arbitrarily in a program. Pieter Muller's thesis [21] proves, that with the appropriate restrictions, AWAIT can be implemented efficiently. The language Ada 95 [14] has also a construct called *barriers*, which is semantically very similar to Active Oberon's AWAIT, although with a coarser procedure-width granularity.

Concurrent Oberon was a first attempt to provide concurrency in an Oberon system. It was done through a specialized API defining a Thread type and functions to create, stop and resume execution. Protection was achieved through a single global system lock. There was no provision for a synchronization primitive. This model is too weak to support Active Oberon, because locks and synchronization are tightly bound (when synchronization is performed, locks are released), and the locking mechanism is too coarse; a single lock would make a multiprocessor-based system—where many threads run at the same time—useless.

2 Object Oriented Extensions

2.1 Pointer to Anonymous Record Types

```
TYPE
  (* examples of pointer to record types *)

  (* pointer to named record type *)
  Tree = POINTER TO TreeDesc;
  TreeDesc = RECORD  key: INTEGER;  l, r: Node  END;

  (* pointer to anonymous record types *)
  Node = POINTER TO RECORD  key: INTEGER;  next: Node  END;
  DataNode = POINTER TO RECORD  (Node)  data: Data  END;
  DataTree = POINTER TO RECORD (Tree)  data: Data  END;
```

The types `Node` and `DataNode` are *pointers to anonymous record* types; `Tree` is a *pointer to named record* type.

These types can only be dynamically instantiated (with NEW), no static instance is possible; this is enforced by the fact that the record type is anonymous and it is not possible to declare a variable having that type.

Both RECORD and POINTER TO RECORD types are allowed as base types of pointer to anonymous record type; a record type cannot extend a pointer to an anonymous record, thus preserving the property of allowing only dynamic instances.

2.2 Object Types

```
TYPE
  (* object types *)
  DataObj = OBJECT  VAR data: Data;  l, r: DataObj  END DataObj;
```

The type DataObj is an *object type*.

The type OBJECT has a syntax different from the POINTER TO RECORD type; it must match the [DeclSeq] Body production instead of the FieldList production. This implies that procedures can be declared inside an object type. We call them methods or type-bound procedures.

Only an object type can extend another object type. Object types must be dynamically instantiated with NEW, subject to the rules imposed by initializers (Section 2.4).

2.3 Type-bound Procedures

```
TYPE
  Coordinate = RECORD x, y: LONGINT END;
  VisualObject = OBJECT
    VAR  next: VisualObject;

    PROCEDURE Draw*;    (*draw this object*)
    BEGIN HALT(99); (*force extensions to override this method*)
    END Draw;
  END VisualObject;

  Point = OBJECT (VisualObject)
    VAR  pos: Coordinate;

    PROCEDURE Draw*;    (*override Draw method declared in VisualObject*)
    BEGIN  MyGraph.Dot(pos.x, pos.y)
    END Draw;
  END Point;

  Line = OBJECT (VisualObject)
    VAR pos1, pos2: Coordinate;

    PROCEDURE Draw*;
    BEGIN  MyGraph.Line(pos1.x, pos1.y, pos2.x, pos2.y)
    END Draw;
  END Line;

VAR
  objectRoot: VisualObject;

PROCEDURE DrawObjects*;
VAR  p: GraphObject;
BEGIN
  (* draw all the objects in the list *)
  p := objectRoot;
  WHILE p # NIL DO  p.Draw;  p := p.next  END;
END DrawObjects;
```

Procedures declared inside an object are called *type-bound procedures* or *methods*. Methods are associated with an instance of the type and operate on it; inside a method implementation, if another method is visible using the Oberon scope rules, it can be accessed without qualification.

A method can overwrite another method of the same name inherited from the base type of the record, but it must have the same signature. The visibility flag is part of the signature.

> **Note:** *We decided to declare the methods in a object scope, because they belong to the record scope and can only be accessed through record instances. This simplifies the method declaration (no receiver, as in Oberon-2 [20]) and the access to the fields and methods following the well-known Oberon scoping rules. We were aware that cyclic references would be a problem (i.e. whenever two object types mutually refer to each other), but considered the conceptual elegance of the language more important. The solution to the problem is a relaxation of the visibility rules that allows forward references of symbols declared later in the source text (section 4.1). An alternative would have been to extend the forward declarations to describe whole types (as in Object Oberon [19]). We discarded this solution because of the unnecessary redundancy it introduces in the code, and we delegate the problem to the compiler instead.*

Given an object instance o of type T with type-bound procedures P and Q, $o.P$ is the call to the method. Inside a method of T another method of T can be called with Q. A method P can call the method it overrides with the notation $P\uparrow$. This is a supercall and can only be made inside a method.

2.4 Initializers

A method tagged with `&` is an *object initializer*. This method is automatically called when an instance of the object is created. An object type may have at most one initializer. If present, it is always public, and can be called explicitly, like a method; if absent, the initializer of the base type is inherited. An initializer can have a signature differing from the inherited initializer from the base object type, in which case it must have a different name too.

If an object type T has or inherits an initializer P with signature (p0: T0; .... pn: Tn), then the instantiation of a variable $o{:}T$ by `NEW` requires the parameters needed by the initializer: `NEW(o, p0, ..., pn)`. The initializer is executed atomically with NEW.

> **Note:** *The optional initializer allows the parameterization of a type. In particular, it is very useful when working with active objects, because the instance parameterization must be done before the activity is started. To be honest, we don't really like this notation, but it is the only one we could imagine that would fit into the language without changing it.*

```
TYPE
  Point = OBJECT (VisualObject)
    VAR  pos: Coordinate;

    PROCEDURE & InitPoint(x, y: LONGINT);
    BEGIN  pos.x := x;  pos.y := y
```

```
    END InitPoint;
  END Point;

  PROCEDURE NewPoint(): Point;
  VAR  p: Point;
  BEGIN  NEW(p, x, y);  (*calls NEW(p) and p.InitPoint(x, y) *)
    RETURN p
  END NewPoint;
```

2.5 SELF

The keyword SELF can be used in any method or any procedure local to a method of an object. It has the object type and the value of the current object instance the method is bound to. It is used to access the object whenever a reference to it is needed or to access a record field or method when shadowed by other symbols, i.e. fields that are hidden by a local variable with the same name.

```
TYPE
  ListNode = OBJECT
    VAR data: Data;  next: ListNode;

    PROCEDURE & InitNode (data: Data);
    BEGIN
      SELF.data := data;    (* initialize object data *)
      next := root; root := SELF   (* prepend node to list *)
    END InitNode;
  END ListNode;

VAR
  root: ListNode;
```

2.6 Delegate Procedure Types

```
TYPE
  MediaPlayer = OBJECT
    PROCEDURE Play; .... play a movie .... END Play;
    PROCEDURE Stop; .... stop movie .... END Stop;
  END MediaPlayer;

  ClickProc = PROCEDURE {DELEGATE};
  Button = OBJECT
    VAR
      onClick: ClickProc;
      caption: ARRAY 32 OF CHAR;

    PROCEDURE OnClick;
    BEGIN  onClick  END OnClick;

    PROCEDURE & Init(caption: ARRAY OF CHAR;  onClick: ClickProc);
    BEGIN  SELF.onClick := onClick; COPY(caption, SELF.caption)
    END Init;
  END Button;

PROCEDURE Init(p: MediaPlayer);
VAR  b0, b1, b2: Button;
BEGIN
  (* Reboot -> call system reboot function *)
  NEW(b0, "Reboot", System.Reboot);
```

```
  (* MediaPlayer UI: bind buttons with player instance *)
  NEW(b1, "Play", p.Play);
  NEW(b2, "Stop", p.Stop);
END Init;
```

Delegates are similar to procedure types; they are compatible to both methods and procedures, while procedure types are only compatible with procedures.

Delegate procedure types are annotated with the `DELEGATE` modifier. They can be assigned with procedures and methods. Given a variable with procedure type t , o an object instance and M a method bound to o, it is allowed to assign $o.M$ to t if the method and t have compatible signatures. The object self-reference is omitted from the procedure type signature. Whenever t is called, the assigned object o is implicitly passed as self-reference. Assignment and call of procedures remains compatible with the Oberon definition.

2.7 Definitions

A *Definition* is a syntactic contract [0]

defining a set of method signatures. A definition *D0* can be *refined* by a new definition *D1*, which will inherit all methods declared in *D0*. Definitions and their methods are globally visible. An object can implement one or more definitions, in which case it commits itself to give an implementation to all the methods declared in the definitions.

```
 DEFINITION
Runnable;
 PROCEDURE Start;
 PROCEDURE Stop;
 END Runnable;

DEFINITION Preemptable REFINES Runnable;
 PROCEDURE Resume;
PROCEDURE Suspend;
 END Preemptable;

 TYPE
 MyThread = OBJECT
IMPLEMENTS Runnable;
 PROCEDURE Start;
 BEGIN .... END Start;

PROCEDURE Stop;
 BEGIN .... END Stop;
 END MyThread;
```

The keyword IMPLEMENTS is used to specify the definitions implemented by an object type. An object type can implement multiple definitions.

Definitions can be thought to be additional properties that a class must have, but that are orthogonal to the object type hierarchy. A object's method can be invoked through the definition, in which case the run-time checks if the object instance implements the definition and then invokes the method; if a definition is not implemented by the object instance, a run-time exception occurs.

[0] [1] describes four levels of contracts: 1. syntactic contracts (type systems), 2. behavioral contracts (invariants, pre- and post-conditions), 3. synchronization contracts, 4. quality of service contracts

```
PROCEDURE Execute(o: OBJECT; timeout: LONGINT);
BEGIN
  Runnable(o).Start;
  Delay(timeout);
  Runnable(o).Stop;
END Execute;
```

3 Concurrency Support

3.1 Active Objects

The declaration of an object type may include a `StatBlock`, called the *object body*. The body is the object's activity, to be executed whenever an object instance is allocated after the initializer (if any) completed execution; the object body is annotated with the `ACTIVE` modifier. At allocation, a new process is allocated to execute the body concurrently; the object is called an *active object*.

If the `ACTIVE` modifier is not present, the body is executed synchronously; NEW returns only after the body has terminated execution.

The system holds an implicit reference to an active object as long as the activity has not terminated to prevent garbage collection of the object. The object can live longer than its activity.

```
TYPE
  (*define the object and its intended behavior*)
  Object = OBJECT

  BEGIN {ACTIVE}  (*object body*)
    ... do something ...
  END Object;

PROCEDURE CreateAndStartObject;
VAR  o: Object;
BEGIN
  ... NEW(o); ...
END CreateAndStartObject;
```

The active object activity terminates whenever the body execution terminates. As long as the body executes, the object is kept alive (i.e. cannot be garbage collected). After that, the object becomes a passive one, and will be collected according to the usual rules.

3.2 Protection

A *Statement Block* is a sequence of statements delimited by `BEGIN` and `END`. It can be used anywhere like a simple statement. It is most useful when used with the `EXCLUSIVE` modifier to create a critical region to protect the statements against concurrent execution.

```
PROCEDURE P;
VAR x, y, z: LONGINT;
BEGIN
  x := 0;
  BEGIN
    y := 1
  END;
  z := 3
END P;
```

An object can be viewed as a resource and various activities may potentially compete for using the object or for exclusive access to the facilities it provides; in such cases some kind of access protection is essential. Our protection model is an instance-based monitor.

```
(* Procedures Set and Reset are mutually exclusive*)
TYPE
  MyContainer = OBJECT
    VAR  x, y: LONGINT;    (* Invariant:  y = f(x) *)

    PROCEDURE Set(x: LONGINT);
    BEGIN {EXCLUSIVE}    (* changes to both x and y are atomic *)
      SELF.x := x;  y := f(x)
    END Set;

    PROCEDURE Reset;
    BEGIN
      ...
      BEGIN {EXCLUSIVE}    (* changes to both x and y are atomic *)
        x := x0;  y := y0;
      END;
      ....
    END Reset;
  END MyContainer;
```

Every object instance is protected and the protection granularity is any statement block inside the object's method, ranging from a single statement to a whole method. A statement block can be protected against concurrent access by annotating it with the modifier `EXCLUSIVE`. Upon entering an exclusive block, an activity is preempted as long as another activity stands in an exclusive block of the same object instance.

An activity cannot take an object's lock more than once, re-entrancy is not allowed.

Every module is considered to be an object type with a *singleton instance*, thus its procedures can also be protected. The scope of protection is the whole module, like in a monitor [13].

> **Note:** *Only the EXCLUSIVE locks are implemented: SHARED locks (as in [10]) can be implemented using EXCLUSIVE locks and are thus not a basic concept. They are very seldom used and don't justify the added complexity in the language and its implementation. See Appendix B.1 for a implementation of SHARED locks.*

> **Note:** *Lock re-entrancy is not supported because it is conceptually unclear (see [27, section 5.9]) and expensive to handle correctly; it is not really needed, since it is possible to design the program to work without using it. Re-entrant locks can be implemented using non re-entrant locks (see Appendix B.3)*

3.3 Synchronization

```
TYPE
  Synchronizer = OBJECT
    awake: BOOLEAN

    PROCEDURE Wait;
```

```
    BEGIN {EXCLUSIVE} AWAIT(awake); awake := FALSE
    END Wait;

    PROCEDURE WakeUp;
    BEGIN {EXCLUSIVE} awake := TRUE
    END WakeUp;
  END Synchronizer;
```

The built-in procedure `AWAIT` is used to synchronize an activity with a state of the system. `AWAIT` can take any boolean *condition*; the activity is allowed to continue execution only when *condition* is true. While the condition is not established, the activity remains *suspended*; if inside a protected block, the lock on the protected object is released, as long as the activity remains suspended (to allow other activities to change the state of the object and thus establish the condition); the activity is restarted only if the lock can be taken.

The system is responsible for evaluating the conditions and for resuming suspended activities. The conditions inside an object instance are re-evaluated whenever some activity leaves a protected block inside the same object instance. This implies that changing the state of an object outside a protected block won't have the conditions re-evaluated.

When several activities compete for the same object lock, the activities whose conditions are true are scheduled before those that only want to enter a protected region.

Appendix B.6 shows the synchronization inside a shared buffer.

> **Note:** *Synchronization depends on the object state, e.g waiting for some data to be available or the state to change. The object is used as container for the data and every access is done through protected methods or blocks. We assume that every time a protected block is accessed, the state of the object has changed; thus the condition is only re-evaluated at that time. This implies that changing the state of an object without protecting it won't have the condition re-evaluated. To force re-evaluation of the conditions in one object, an empty protected method can be called or protected block entered.*

4 Other Language Extensions

This section describes a few minor changes made to better integrate the extensions into the language.

4.1 Declaration sequence and forward references

In Active Oberon, the definition scope of a symbol ranges over the whole block containing it. This implies that a symbol can be used before being declared, and that names are unique inside a scope.

4.2 HUGEINT

The 64-bit signed integer type HUGEINT has been added to the language. It fits into the numeric type hierarchy as follows:

LONGREAL ⊇ REAL ⊇ HUGEINT ⊇ LONGINT ⊇ INTEGER ⊇ SHORTINT

Name	*Argument Type*	*Result Type*	*Function*
SHORT(x)	HUGEINT	LONGINT	identity (truncation possible)
LONG(x)	LONGINT	HUGEINT	identity
ENTIERH(x)	real type	HUGEINT	largest integer not greater than x

Table 1: New Type Conversion Procedures

Name	*Function*
PUT8(adr: LONGINT; x: SHORTINT) PUT16(adr: LONGINT; x: INTEGER) PUT32(adr: LONGINT; x: LONGINT) PUT64(adr: LONGINT; x: HUGEINT)	Mem[adr] := x
GET8(adr: LONGINT): SHORTINT GET16(adr: LONGINT): INTEGER GET32(adr: LONGINT): LONGINT GET64(adr: LONGINT): HUGEINT	RETURN Mem[adr]
PORTIN(port: LONGINT; x: AnyType) PORTOUT(port: LONGINT; x: AnyType)	x := IOPort(port) IOPort(port) := x
CLI STI	disable interrupts enable interrupts
	PUTREG/GETREG constants
EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP	32-bit register
AX, BX, CX, DX, SI, DI	16-bit register
AL, AH, BL, BH, CL, CH, DL, DH	8-bit register

Table 2: IA32 Version Additions to SYSTEM

Table 1 shows the new conversion functions added for HUGEINT.

No new constant definition is needed; constants are typed according to their value.

4.3 Untraced Pointers

Untraced pointers are pointers that are not traversed by the garbage collector. A structure or object referenced only through an untraced pointer may be collected at any time.

Untraced pointers are defined using the UNTRACED modifier.

```
TYPE  Untraced = POINTER {UNTRACED} TO T;
```

4.4 IA32 Specific Additions

The functions in Table 2 have been added to the Intel IA32 version of the compiler.

The `PUTx` and `GETx` have been added for security sake, to cope with untyped constants.

4.5 Miscellaneous

A few Oberon-2 extensions have been adopted in Active Oberon:

- ASSERT
- FOR

- read-only export
- dynamic arrays

Pointer variables are automatically initialized to NIL.

References

[1] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, July 1999.

[2] R. Brega. Real-time kernel for the Power-PC architecture. Master's thesis, Institut für Robotik, ETH Zürich, 1995.

[3] P. Brinch Hansen. Structured multiprogramming. *Communications of the ACM*, 15(7):574–578, July 1972. Reprinted in The Search for Simplicity, IEEE Computer Society Press, 1996.

[4] Microsoft Corporation. *Microsoft C# Language Specifications*. Microsoft Press, 2001.

[5] Edsger W. Dijkstra. The structure of the THE-multiprogramming system. *Communications of the ACM*, 11(5):341–346, May 1968.

[6] H. Eberle. *Development and Analysis of a Workstation Computer*. Dissertation 8431, ETH Zürich, 1987.

[7] P. Fröhlich. Projekt Froderon: Zur weiteren Entwicklung der Programmiersprache Oberon-2. Master's thesis, Fachhochschule München, 1997.

[8] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. The Java Series. Addison-Wesley, 1st edition, 1996.

[9] R. Griesemer. *A Programming Language for Vector Computers*. Dissertation 10277, ETH Zürich, 1993.

[10] J. Gutknecht. Do the fish really need remote control? A proposal for self-active objects in Oberon. In *Proc. of Joint Modular Languages Conference (JMLC). LNCS 1024*, Linz, Austria, March 1997. Springer Verlag.

[11] J. Gutknecht and N. Wirth. *Project Oberon - The Design of an Operating System and Compiler*. Addison-Wesley, 1992.

[12] B. Heeb and C. Pfister. Chameleon: A workstation of a different colour. In *Field-Programmable Gate Arrays: Architectures and Tools for Rapid Prototyping. Second International Workshop on Field Programmable Logic and Applications*, pages 152–161, August 1992.

[13] C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–557, October 1974. Erratum in *Communications of the ACM*, Vol. 18, No. 2 (February), p. 95, 1975. This paper contains one of the first solutions to the Dining Philosophers problem.

[14] International Organization for Standardization. *ISO/IEC 8652:1995: Information technology — Programming languages — Ada*. International Organization for Standardization, Geneva, Switzerland, 1995.

[15] P. Januschke. *Oberon-XSC - Eine Programmiersprache und Arithmetikbibliothek für das Wissenschaftliche Rechnen*. PhD thesis, Universität Karlsruhe, 1998.

[16] K. Jensen and N. Wirth. *PASCAL - User Manual and Report*, volume 18 of *Lecture Notes in Computer Science*. Springer, 1974.

[17] S. E. Knudsen. *Medos-2: A Modula-2 oriented operating system for the personal computer Lilith*. Diss no. 7346, ETH Zürich, 1983.

[18] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[19] H. Mössenböck, J. Templ, and R. Griesemer. Object Oberon: An object-oriented extension of Oberon. Technical Report 1989TR-109, Department of Computer Science, ETH Zürich, June 1989.

[20] H. Mössenböck and N. Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–195, 1991.

[21] P.J. Muller. *The Active Object System – Design and Multiprocessor Implementation*. PhD thesis, ETH Zürich, 2002.

[22] R. Ohran. *Lilith: A Workstation Computer for Modula-2*. Dissertation 7646, ETH Zürich, 1984.

[23] A. Radenski. Introducing objects and concurrency to an imperative programming language. *Information Sciences, an International Journal*, 87(1-3):107–122, 1995.

[24] A. Radenski. Module embedding. *Software - Concepts and Tools*, 19(3):122–129, 1998.

[25] B. A. Sanders and S. Lalis. Adding concurrency to the Oberon system. In *Proceedings of Programming Languages and System Architectures*, Lecture Notes in Computer Science (LNCS) 782. Springer Verlag, March 1994.

[26] C. Szyperski. Import is not inheritance – why we need both: modules and classes. In O. Lehrmann Madsen, editor, *Proceedings, ECOOP 92*, number 615 in Lecture Notes in Computer Science, pages 19–32. Springer-Verlag, 1992.

[27] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[28] N. Wirth. MODULA : A language for modular multiprogramming. *Software Practice and Experience*, 7:3–35, 1977.

[29] N. Wirth. The programming language Oberon. *Software Practice and Experience*, 18(7):671–690, July 1988.

[30] N. Wirth and M. Reiser. *Programming in Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley, 1992.

A Active Oberon Syntax

```
Module      =  MODULE ident ';' [ImportList] {Definition} {DeclSeq} Body ident '.'.
ImportList  =  IMPORT ident [':=' ident] {',' ident [':=' ident ]} ';'.
Definition  =  DEFINITION ident [REFINES Qualident] {PROCEDURE ident [FormalPars] ';'} END ident.
DeclSeq     =  CONST {ConstDecl ';'} | TYPE {TypeDecl ';'} | VAR {VarDecl ';'} | {ProcDecl ';'}.
ConstDecl   =  IdentDef '=' ConstExpr.
TypeDecl    =  IdentDef '=' Type.
VarDecl     =  IdentList ':'  Type.
ProcDecl    =  PROCEDURE ProcHead ';' {DeclSeq} Body ident.
ProcHead    =  [SysFlag] ['*' | '&'] IdentDef [FormalPars].
SysFlag     =  '[' ident ']'.
FormalPars  =  '(' [FPSection {';' FPSection}] ')' [':'  Qualident].
FPSection   =  [VAR] ident {',' ident} ':'  Type.
Type        =  Qualident
            |  ARRAY [SysFlag] [ConstExpr {',' ConstExpr}] OF Type
            |  RECORD [SysFlag] ['(' Qualident ')'] [FieldList] END
            |  POINTER [SysFlag] TO Type
            |  OBJECT [[SysFlag] ['(' Qualident ')'] [IMPLEMENTS Qualident] {DeclSec} Body]
            |  PROCEDURE [SysFlag] [FormalPars].
FieldDecl   =  [IdentList ':'  Type].
FieldList   =  FieldDecl {';' FieldDecl}.
Body        =  StatBlock | END.
StatBlock   =  BEGIN ['{'IdentList'}'] [StatSeq] END.
StatSeq     =  Statement {';' Statement}.
Statement   =  [Designator ':=' Expr
            |  Designator ['(' ExprList')']
            |  IF Expr THEN StatSeq {ELSIF Expr THEN StatSeq}[ELSE StatSeq] END
            |  CASE Expr DO Case {'|' Case} [ELSE StatSeq] END
            |  WHILE Expr DO StatSeq END
            |  REPEAT StatSeq UNTIL Expr
            |  FOR ident ':=' Expr TO Expr [BY ConstExpr] DO StatSeq END
            |  LOOP StatSeq END
            |  WITH Qualident ':'  Qualident DO StatSeq END
            |  EXIT
            |  RETURN [Expr]
            |  AWAIT '(' Expr ')'
            |  StatBlock
            ].
Case        =  [CaseLabels { ',' CaseLabels } ':'  StatSeq].
CaseLabels  =  ConstExpr ['..'  ConstExpr].
ConstExpr   =  Expr.
Expr        =  SimpleExpr [Relation SimpleExpr].
SimpleExpr  =  Term {MulOp Term}.
Term        =  ['+'|'-'] Factor {AddOp Factor}.
Factor      =  Designator['(' ExprList')'] | number | character | string
            |  NIL | Set | '('Expr')'|' 'Factor.
Set         =  '{' [Element {',' Element}] '}'.
Element     =  Expr ['..'  Expr].
Relation    =  '=' | '#' | '<' | '<=' | '>' | '>=' | IN | IS.
MulOp       =  '*' | DIV | MOD | '/' | '&' .
AddOp       =  '+' | '-' | OR .
Designator  =  Qualident { '.'  ident | '['ExprList']' | '^' | '(' Qualident ')' }.
ExprList    =  Expr {',' Expr}.
IdentList   =  IdentDef {',' IdentDef}.
Qualident   =  [ident '.']  ident.
IdentDef    =  ident ['*'|'-'].
```

B Synchronization Examples

B.1 Readers and Writers

```
MODULE ReaderWriter;

TYPE
    RW = OBJECT
       (* n = 0, empty *)
       (* n < 0, n Writers  *)
       (* n > 0, n Readers *)
       VAR n: LONGINT;

      PROCEDURE EnterReader*;
      BEGIN {EXCLUSIVE}
          AWAIT(n >= 0); INC(n)
      END EnterReader;

      PROCEDURE ExitReader*;
      BEGIN {EXCLUSIVE}
          DEC(n)
      END ExitReader;

      PROCEDURE EnterWriter*;
      BEGIN {EXCLUSIVE}
          AWAIT(n = 0); DEC(n)
      END EnterWriter;

      PROCEDURE ExitWriter*;
      BEGIN {EXCLUSIVE}
          INC(n)
      END ExitWriter;

      PROCEDURE & Init;
      BEGIN n := 0
      END Init;
    END RW;

END ReaderWriter.
```

The Readers - Writers paradigm regulates the data access in a critical section. Either a single Writer (activity changing the state of the object) or many Readers (activities that don't change the state of the object) are allowed to enter the critical section at a given time.

B.2 Signals

```
TYPE
  Signal* = OBJECT
    VAR
        in: LONGINT;    (*next ticket to assign*)
        out: LONGINT;    (*next ticket to service*)
        (* entries with (out <= ticket < in) must wait *)

    PROCEDURE Wait*;
    VAR ticket: LONGINT;
    BEGIN {EXCLUSIVE}
        ticket := in; INC(in); AWAIT(ticket - out < 0)
    END Wait;

    PROCEDURE Notify*;
    BEGIN {EXCLUSIVE}
        IF out # in THEN INC(out) END
    END Notify;

    PROCEDURE NotifyAll*;
    BEGIN {EXCLUSIVE}
        out := in
    END NotifyAll;

    PROCEDURE & Init;
    BEGIN  in := 0; out := 0
    END Init;
  END Signal;
```

`Signal` implements signaling primitives in Active Oberon, similar to those of Java and Modula-2. It uses a slightly modified ticket-algorithm. Like in some stores, every customer receives a numbered ticket, to ensure that the customers are serviced in order of arrival. This algorithm handles the wrap-around of `in` and `out` indexes too.

B.3 Re-entrant Locks

```
ReentrantLock* = OBJECT
  VAR
    lockedBy: PTR;
    depth: LONGINT;

  PROCEDURE Lock*;
  VAR  me: PTR;
  BEGIN {EXCLUSIVE}
    me := AosActive.CurrentThread();
    AWAIT((lockedBy = NIL) OR (lockedBy = me));
    lockedBy := me;
    INC(depth)
  END Lock;

  PROCEDURE Unlock*;
  BEGIN {EXCLUSIVE}
    DEC(depth);
    IF depth = 0 THEN lockedBy := NIL END
  END Unlock;

END ReentrantLock;
```

The `ReentrantLock` Object allows to re-lock an object by its owner more than once. Clients of this object must explicitly use `Lock` and `Unlock` instead of tagging their protected regions with `EXCLUSIVE`.

B.4 Binary and Generic Semaphores

```
MODULE Semaphores;

TYPE
  Sem* = OBJECT  (* Binary Semaphore *)
      VAR  taken: BOOLEAN

      PROCEDURE P*;  (*enter semaphore*)
      BEGIN {EXCLUSIVE}
          AWAIT(~taken); taken := TRUE
      END P;

      PROCEDURE V*;  (*leave semaphore*)
      BEGIN {EXCLUSIVE}
          taken := FALSE
      END V;

      PROCEDURE & Init;
      BEGIN  taken := FALSE
      END Init;
  END Sem;

  GSem* = OBJECT  (* Generic Semaphore *)
      VAR  slots: LONGINT;

      PROCEDURE P*;
      BEGIN {EXCLUSIVE}
          AWAIT(slots > 0); DEC(slots)
      END P;

      PROCEDURE V*;
      BEGIN {EXCLUSIVE}
        INC(slots)
      END V;

      PROCEDURE & Init(n: LONGINT);
      BEGIN slots := n
      END Init;
  END GSem;

END Semaphores.
```

The well-known synchronization primitive by Dijkstra [5]. Note that the ability to implement semaphores shows that the Active Oberon model is also a synchronization primitive and is powerful enough to support protection and synchronization of concurrent processes.

B.5 Barrier

```
MODULE Barriers;
(*
  A barrier is used to synchronize N activities together.
*)
  TYPE
    Barrier = OBJECT
      VAR in, out, N: LONGINT;

      PROCEDURE Enter*;
        VAR i: LONGINT;
      BEGIN {EXCLUSIVE}
        INC(in);
        AWAIT (in >= N);
        INC(out);
        IF (out = N) THEN in := 0; out := 0 END;
      END Enter;

      PROCEDURE & Init (nofProcs: LONGINT);
      BEGIN
        N := nofProcs; in := 0; out := 0;
      END Init;

    END Barrier;
END Barriers.
```

Barriers are used to synchronize activities together. If activities are defined as

$$P_i = Phase_{i,0}; Phase_{i,1}; .....Phase_{i,n}$$

then the barrier is used to ensure that all activities will complete $Phase - i, j$ before starting $Phase_{i,j+1}$. One thread of execution would look like this:

```
  FOR j := 0 TO N DO
    Phase(i, j); barrier.Enter
  END;
```

The barrier resets the *in* counter after the condition to avoid an overflow. This is possible because activities competing to reacquire the lock in the AWAIT instruction have priority over the activities competing to acquire the same lock for the EXCLUSIVE statement block.

B.6 Bounded Buffer

```
MODULE Buffers;

CONST
  BufLen = 256;

TYPE
  (* Buffer- First-in first-out buffer *)

  Buffer* = OBJECT
    VAR
      data: ARRAY BufLen OF INTEGER;
      in, out: LONGINT;

    (* Put - insert element into the buffer *)

    PROCEDURE Put* (i: INTEGER);
    BEGIN {EXCLUSIVE}
      AWAIT ((in + 1) MOD BufLen # out);  (*AWAIT ~full *)
      data[in] := i;
      in := (in + 1) MOD BufLen
    END Put;

    (* Get - get element from the buffer *)

    PROCEDURE Get* (VAR i: INTEGER);
    BEGIN {EXCLUSIVE}
      AWAIT (in # out);  (*AWAIT ~empty *)
      i := data[out];
      out := (out + 1) MOD BufLen
    END Get;

    PROCEDURE & Init;
    BEGIN
      in := 0; out := 0;
    END Init;

  END Buffer;

END Buffers.
```

Buffer implements a bounded circular buffer. The methods *Put* and *Get* are protected against concurrent access; they also check that a buffer slot , resp. data, is available, otherwise the activity is suspended until the until the slot or data become available.

C Active Object Examples

C.1 Dining Philosophers

```
MODULE Philo;

IMPORT  Semaphores;

CONST
    NofPhilo = 5;  (* number of philosophers *)

VAR
    fork: ARRAY NofPhilo OF Semaphores.Sem;
    i: LONGINT;

TYPE
    Philosopher = OBJECT
      VAR
        first, second: LONGINT;
        (* forks used by this philosopher *)

      PROCEDURE & Init(id: LONGINT);
      BEGIN
          IF id # NofPhilo-1 THEN
              first := id; second := (id+1)
          ELSE
              first := 0; second := NofPhilo-1
          END
      END Init;

    BEGIN {ACTIVE}
        LOOP
            .... Think....
            fork[first].P; fork[second].P;
            .... Eat ....
            fork[first.V; fork[second].V
        END
    END Philosopher;

VAR
  philo: ARRAY NofPhilo OF Philosopher;

BEGIN
    FOR i := 0 TO NofPhilo DO
      NEW(fork[i]);
      NEW(philo[i]);
    END;
END Philo.
```

C.2 Sieve of Eratosthenes

```
MODULE Eratosthenes; (* prk 13.09.00 *)

IMPORT Out, Buffers;

CONST
  N = 2000;
  Terminate = -1;    (* sentinel *)

TYPE
  Sieve = OBJECT (Buffers.Buffer)

    VAR prime, n: INTEGER; next: Sieve;

    PROCEDURE & Init;
    BEGIN
      Init^;    (*call Buffer's (superclass) initializer *)
      prime := 0; next := NIL
    END Init;

  BEGIN {ACTIVE}
    LOOP
      Get(n);
      IF n = Terminate THEN
        (* terminate execution *)
        IF next # NIL THEN next.Put (n) END;
        EXIT
      ELSIF prime = 0 THEN
        (* first number is always a prime number *)
        Out.Int(n, 0); Out.String(" is prime"); Out.Ln;
        prime := n;
        NEW (next)
      ELSIF (n MOD prime) # 0 THEN
        (* pass to the next sieve if not a multiple of prime *)
        next.Put (n)
      END
    END
  END Sieve;

  PROCEDURE Start*;
  VAR s: Sieve;  i: INTEGER;
  BEGIN
    NEW(s);
    FOR i := 2 TO N-1 DO s.Put (i) END;
    s.Put(Terminate)    (* use sentinel to indicate completion*)
  END Start;

END Eratosthenes.
```

`Eratosthenes` implements the sieve algorithm for finding prime numbers. Every sieve is an active object that passes the all the received values that are not a multiple of the first received value to the next sieve. The synchronization between sieves is encapsulated in the buffer.

D Version Log

15 Mar 2003	Barrier example improved
4 Apr 2002	Minor corrections
14 Mar 2002	Definition's syntax; some small corrections
11 Jan 2002	Small corrections, Statement Block moved to Protection
20 Dec 2001	History
9 Aug 2001	Small corrections and Delegate modifier added
19 Apr 2001	Object Types instead of dynamic record, many small improvements
25 Mar 2001	Revision
20 Feb 2001	Delegates added