Active Objects in Practice: the Active Oberon Language

PATRIK REALI, Institut für Computersysteme, ETH Zürich

The Active Oberon Language provides in-language support for programming concurrent programs based on the active object model. This model was choosen because of its simplicity, intuitivity, and expressiveness.

This paper relates the design choices made and problem encountered during the language development, together with our first experiences with Active Oberon. It also presents a comprehensive catalogue of classical concurrent algorithms rewritten using Active Oberon.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs; D.1.3 [**Programming Techniques**]: Concurrent Programming

General Terms: Languages

Additional Key Words and Phrases: Oberon Language, Active Objects, Concurrent Programming

---

1. INTRODUCTION

This paper describes the Active Oberon language extensions to the Oberon language and the first experiences done with it. The extension introduces concurrency in the language by the mean of active objects, and an improved component modeling; it is the base for further language extensions introduced in the Oberon.Net project by Gutknecht [Gutknecht 2001a; 2001b]. The Active Oberon language was first presented by Gutknecht [Gutknecht 1997].

A compiler for Active Oberon is available [Reali 2000; 2003]. It is used to build the Aos system [Muller 2002], and all the software running on top of it; the compiler is obviously part of the Aos system. The Active Oberon Language is currently used for teaching programming for undergraduates at ETH-Zurich, and for developing software for research purposes.

This paper is organized as follows: Section 2 introduces the language extensions by presenting the concepts they are based on; Section 3 explains the extensions to complete Oberon's object oriented model; 4 explains the extensions for concurrency; 5 explains other additions made to round up the language; Appendix A shows a few classical synchronization algorithms implemented using Active Oberon; Appendix B shows some examples of active objects.

Author's address: P. Reali (reali@acm.org), Institut für Computersysteme, ETH Zentrum, CH-8092 Zürich, Switzerland.

I would like to thank M. Corti, A. Fischer, J. Gutknecht, B. Kirk, D. Lightfoot, and P.Muller for proofreading early drafts of this document.

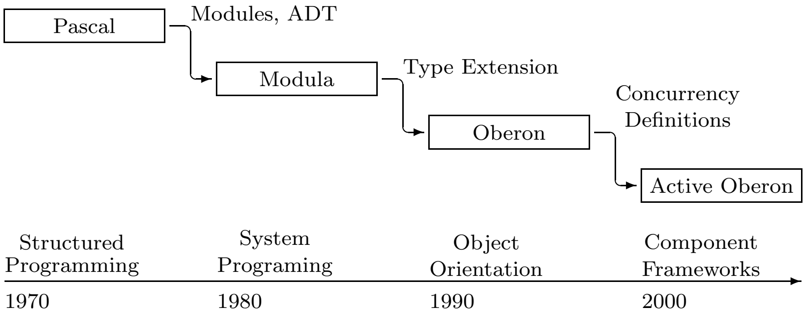

Fig. 1. The Pascal Language Family Evolution

2. CONCEPTS

2.1 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 [Jensen and Wirth 1974] was conceived as a language to express small programs; its simplicity and leanness made it particularly well-suited for teaching programming. Modula [Wirth 1977] evolved from Pascal as a language for system programming, and benefited from the practical experience gained during the development of the Lilith workstation [Ohran 1984] and of the Medos operating system [Knudsen 1983]. The need to support the programming-in-the-large paradigm was the motivation for Oberon [Wirth 1988]. The Ceres [Eberle 1987] and Chameleon [Heeb and Pfister 1992] platforms, and the Oberon operating system [Gutknecht and Wirth 1992] 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 [Mössenböck et al. 1989], Oberon-2 [Mössenböck and Wirth 1991], and Froderon [Fröhlich 1997] explored adding further object-oriented features to the language; Oberon-V [Griesemer 1993] proposed additions for parallel operations on vector computers; Oberon-XSC [Januschke 1998] added mathematical features to support scientific computation; module embedding [Radenski 1998] was also proposed. Concurrency was first added to the operating system through a specialized system API in Concurrent Oberon [Sanders and Lalis 1994] and XOberon [Brega 1995]; attempts to model concurrency in the language itself were also done by Radenski [Radenski 1995].

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.

2.2 The Active Object Model

Active Oberon concurrency's model is based on active objects. Active Objects are objects that include an own thread of control. In other words, each object carries an own independent activity, which is executed concurrently to the other objects activities. The new concepts are integrated into the language by generalizing the class construct to include concurrency. Passive classes, as used in Oberon, are a special case of Active Oberon classes.

The support for concurrency requires the language to address atomicity, synchronization and activity. Atomicity ensures that several operations are executed without interruption, as if they were only a single instruction. Synchronization forces the activity to wait for a condition to be established. Activity is the code concurrently executed by a thread.

The question a language designer usually faces is, shall concurrency be added through a library (thus providing an optional implementation) or embedded into the language? A library is a collection of useful abstractions built using the language itself, and is at first sight a comfortable place to add things that don't belong to the language, but can be constructed using it. Many language designers made this choice, including [Sanders and Lalis 1994] for Oberon. This can be an approach for experimenting with concurrency, but on the long run is a dead end: concurrency is a metaphor that cannot be mapped well to an existing non-concurrent model, but rather requires a new model that can be provided only changing the language model.

Providing atomicity through a library (usually with a lock and unlock call) is not satisfactory: atomicity is to be used in a structured way, by wrapping a block of instructions, as for every lock there must be an unlock; a library call will never be able to enforce this property. The library approach provides an imperative view, possibly wrapped in object-oriented cloths, but with insufficient abstraction. This is obviously not desirable.

As an example, Java tries to partially solve this problem by embedding atomicity in the language with synchronized methods and synchronized statement blocks (here synchronized in fact means a critical section, i.e. atomicity), but leaving synchronization and activity in the libraries. Libraries are supposed to be a collection of useful additions to the language, but the language should be independent of the libraries, and even able to exist with a different set of libraries. This choice makes matters even worse: why should the language provide atomiticy, if it doesn't define (know about) concurrency? For the sake of consistency, concurrency should be addressed either in the language or in the libraries, but not in both. C# instead is completely concurrency agnostic; atomicity, synchronization, and activities are provided by the libraries.

Believing now that the language model must support concurrency, we think that the active object model is the most suitable for our language.

The model must have the following properties:

(1) simple to understand (concurrency is already hard enough)
(2) atomicity must be a structured construct.
(3) objects define the atomicity's scope, because objects are the unit of reasoning.
(4) synchronization must be structured.
(5) the condition for synchronization must be explicit.
(6) activities must be bound to objects.

2.2.1 *Atomicity.* A statement block can require atomic execution, thus atomicity must encompass a statement block. This enforces the structured property of atomicity. The granularity can range from one single statement to a whole procedure.

Because every atomic region in an object will modify the state of the object it is in, it makes sense to group all the atomic regions in the same instance in monitor-like way.

2.2.2 *Synchronization.* Objects are our first-class citizens, and since the whole model deals with their state, it makes sense to model synchronization on the object state itself. In this way, both above properties are fulfilled: the program specifies on which object state it is waiting for; using a single statement (instead of a pair like wait and notify), the construct is automatically structured, and the condition explicit.

2.2.3 *Activity.* Every object class can define its own activity in the class body. When an object is instantiated, the body is automatically executed in a concurrent thread.

2.3 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 [Gosling et al. 1996] and C# [Microsoft 2001] 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 [Szyperski 1992] [1]:

---

[1] as a comparison, B. Meyer advocates exactly the opposite [Meyer 1997]

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 in [Brinch Hansen 1972] 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 [Muller 2002] proves, that with the appropriate restrictions, AWAIT can be implemented efficiently. The language Ada 95 [International Organization for Standardization 1995] 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.

2.4 General Remarks

Components are the a technique for coping with programming-in-the-large problems that reach across subsystems boundaries. Programming languages have undergone many changes to address the evolution in the scope of the problem to be solved. Structured programming tried to make the implementation of algorithms easier; Abstract Data Structures addressed the combination of data structures in a first step toward programming-in-the-large; then came object orientation, which improved the way to model systems; now we have component frameworks to combine software systems together.

This shows that, where components are used, the implementation of the internals can be done using object orientation, and the implementation of the single objects and deployment units is done using the concepts of structured programming and typing. Since each paradigm comes with additional costs for the augmented flexibility, the choice of the implementation technique is driven by the trade-off between efficiency and generality. In theory, nothing speaks against the use of components everywhere and this would make sense because it is the most generic approach and can be considered a generalization of the other paradigms; only the practical impact on program efficiency can be used as an argument for using the other techniques.

For this reason, we think that all these paradigms have their place and should coexist in a multi-purpose language, as the language may be used to address problems of different size and scope.

The step taken in Oberon.Net [Gutknecht 2001a] to completely discard subclassing and to offer only definitions is very courageous because it only relies on the most general concept in the language, and all others are shown to derive from it (although by removing them, a comparison is not possible anymore). The cost of using only interface calls instead of method calls should be investigated; we think this will not limit the applications written with Oberon.Net in any way, as the language will not be used for low-level time-critical system implementation (the .Net platform is already implemented). On the other hand, Active Oberon has to address this problem, as the whole Aos kernel is implemented using it, and efficiency is an issue there.

The design objectives of the Active Oberon's extension were to generalize the language in a way that a) the new concepts would fit into the language, and b) the old concepts would become special cases of the new ones. Generalization is a very useful tool, because the language is enriched with more expressive tools, whereas the complexity of the language does not grow much: an old concept is substituted by a new one that is richer and more powerful. The same process is well known in other sciences like physics, where sometimes a new law unifies a couple of old ones. The new, unique law precisely describes the physical world and removes the old ones, now superseded. Even if the new law is more complex than the old laws, the simplification due to the use of only one law instead of many has a beneficial effect. Computer science often is confronted with the desire to add as many features as possible without trying to generalize the currently available ones, which raises the complexity of many systems to an intolerable level. Generalization permits to add features without increasing complexity and thus allows to create bigger and more powerful systems.

3. OBJECT ORIENTED EXTENSIONS

3.1 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.

3.2 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 3.4).

3.3 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 [Mössenböck*

> *and Wirth 1991]) 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 5.1). An alternative would have been to extend the forward declarations to describe whole types (as in Object Oberon [Mössenböck et al. 1989]). 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 P in the context of o. Inside any method of T, another method can be called with Q (no specification of the self object is required). A method P can call the method it overrides in its superclass with the notation $P\uparrow$. Supercalls are legal only inside a method.

3.4 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;
```

3.5 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;
```

3.6 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;
```

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

3.7 Definitions

A *Definition* is a syntactic contract[2] 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.

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

4. CONCURRENCY SUPPORT

4.1 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.

---

[2] [Beugnard et al. 1999] 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

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.

4.2 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 a special *singleton instance* object, thus its procedures can also be protected. The scope of protection is the whole module, like in a monitor [Hoare 1974].

> **Note:** *Only the EXCLUSIVE locks are implemented: SHARED locks (as in [Gutknecht 1997]) 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 A.1 for an implementation of SHARED locks.*

> **Note:** *Lock re-entrancy is not supported because it is conceptually unclear (see [Szyperski 1998, 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 A.3)*

4.3 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.

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 I. New Type Conversion Procedures

Appendix A.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.*

5. OTHER LANGUAGE EXTENSIONS

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

5.1 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.

5.2 HUGEINT

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

$$\text{LONGREAL} \supseteq \text{REAL} \supseteq \text{HUGEINT} \supseteq \text{LONGINT} \supseteq \text{INTEGER} \supseteq \text{SHORTINT}$$

Table I shows the new conversion functions added for HUGEINT.

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

5.3 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;
```

5.4 IA32 Specific Additions

The functions in Table II 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.

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)	x := IOPort(port)
PORTOUT(port: LONGINT; x: AnyType)	IOPort(port) := x
CLI	disable interrupts
STI	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 II. IA32 Version Additions to SYSTEM

5.5 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.

6. INTEROPERABILITY ADDITIONS

Supporting interoperability with other languages requires the ability to access and use their features in Oberon. These extensions are not defined in the language, but are implemented in the compiler ad-hoc. We use the modifier syntax to enable these features in the compiler whenever needed.

The `OVERLOADING` modifier enables overloading in a module. Procedures with the same name can be declared in a block, as long as their respective signatures differs. Procedure calls are resolved according to the parameters[3].

---

[3] This concession does not imply that we agree with overloading. We consider overloading a bad feature to avoid, and are considering an alternative implementation avoiding overloading along the lines of [Meyer 1999]

A. SYNCHRONIZATION EXAMPLES

A.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.

A.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.

A.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`.

Care must be taken, whenever such an object also contains a synchonization primitive (e.g. wait) whose semantic requires to automatically release the lock, and to take it again when the thread is resumed. Because another thread may be resumed prior to the current one, and thus take the lock, the reentrancy counter must be saved in the current thread state. This is easily achieved using a variable local to the procedure causing the synchronization (the stack is private to each thread).

```
PROCEDURE Wait*;
VAR count: LONGINT;
BEGIN
  count := SELF.count;
  .. release lock ..
  .. suspend thread ..
  .. acquire lock ..
  SELF.count := count;
END Wait;
```

A.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 [Dijkstra 1968]. 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.

A.5 Barrier

```
MODULE Barriers;  (** prk/pjm 12.6.97  **)
(*
  A barrier is used to synchronize N activities together.
*)
  TYPE
    Barrier = OBJECT
      VAR n, N: LONGINT;

      PROCEDURE Enter*;
        VAR i: LONGINT;
      BEGIN {EXCLUSIVE}
        i := n DIV N;
        INC(n);
        AWAIT (i < n DIV N)
      END Enter;

      PROCEDURE & Init (nofProcs: LONGINT);
      BEGIN
        N := nofProcs; n := 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;
```

A.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.

B. ACTIVE OBJECT EXAMPLES

B.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.
```

B.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.

C. 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 [‘*‘|’-’].
```

REFERENCES

Beugnard, A., Jézéquel, J.-M., Plouzeau, N., and Watkins, D. 1999. Making Components Contract Aware. *Computer 32,* 7 (July), 38–45.

Brega, R. 1995. Real-Time Kernel for the Power-PC Architecture. M.S. thesis, Institut für Robotik, ETH Zürich.

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

Dijkstra, E. W. 1968. The Structure of the THE-Multiprogramming System. *Communications of the ACM 11,* 5 (May), 341–346.

Eberle, H. 1987. Development and Analysis of a Workstation Computer. Ph.D. thesis, ETH Zürich.

Frey, T. 2003. A window manager for bluebottle. to be published.

Fröhlich, P. 1997. Projekt Froderon: Zur weiteren Entwicklung der Programmiersprache Oberon-2. M.S. thesis, Fachhochschule München.

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

Griesemer, R. 1993. A Programming Language for Vector Computers. Ph.D. thesis, ETH Zürich.

Gutknecht, J. 1997. 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.* Springer Verlag, Linz, Austria.

Gutknecht, J. 2001a. Active Oberon for .NET. http://www.oberon.ethz.ch/oberon.net/whitepaper/.

Gutknecht, J. 2001b. Active Oberon for .NET: An Exercise in Object Model Mapping. In *BABEL'01 First International Workshop on Multi-Language Infrastructure and Interoperatibility*, N. Benton and A. Kennedy, Eds. ENTCS, vol. 59.1. Elsevier.

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

Heeb, B. and Pfister, C. 1992. 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.* 152–161.

Hoare, C. A. R. 1974. Monitors: An Operating System Structuring Concept. *Communications of the ACM 17,* 10 (Oct.), 549–557. 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.

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

Januschke, P. 1998. Oberon-XSC- Eine Programmiersprache und Arithmetikbibliothek für das Wissenschaftliche Rechnen. Ph.D. thesis, Universität Karlsruhe.

Jensen, K. and Wirth, N. 1974. *PASCAL - User Manual and Report.* Lecture Notes in Computer Science, vol. 18. Springer.

Knudsen, S. E. 1983. Medos-2: A Modula-2 Oriented Operating System for the Personal Computer Lilith. Ph.D. thesis, ETH Zürich.

Meyer, B. 1997. *Object-Oriented Software Construction*, 2nd ed. Prentice Hall.

Meyer, B. 1999. Overloading vs. Object Technology. *Journal of Object-Oriented Programming*, 3–7.

Microsoft. 2001. *Microsoft C# Language Specifications.* Microsoft Press.

Mössenböck, H., Templ, J., and Griesemer, R. 1989. Object Oberon: An Object-Oriented Extension of Oberon. Technical Report 1989TR-109, Department of Computer Science, ETH Zürich. June.

Mössenböck, H. and Wirth, N. 1991. The Programming Language Oberon-2. *Structured Programming 12,* 4, 179–195.

Muller, P. 2000. A Multiprocessor Kernel for Active Object-based Systems. In *Proceedings of JMLC*, J. Gutknecht and W. Weck, Eds. LNCS, vol. 1897. Springer, Zurich, Switzerland, 263–277.

Muller, P. 2002. The Active Object System – Design and Multiprocessor Implementation. Ph.D. thesis, ETH Zürich.

Ohran, R. 1984. Lilith: A Workstation Computer for Modula-2. Ph.D. thesis, ETH Zürich.

Radenski, A. 1995. Introducing Objects and Concurrency to an Imperative Programming Language. *Information Sciences, an International Journal 87,* 1-3, 107–122.

Radenski, A. 1998. Module Embedding. *Software - Concepts and Tools 19,* 3, 122–129.

Reali, P. 2000. Structuring a Compiler with Active Objects. In *Proceedings of JMLC*, J. Gutknecht and W. Weck, Eds. LNCS, vol. 1897. Springer, Zurich, Switzerland, 250–262.

Reali, P. 2003. Using Oberon's Active Objects for Language Interoperability and Compilation. Ph.D. thesis, Institut für Computersysteme, ETH Zürich.

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

Szyperski, C. 1992. Import is not Inheritance – Why we need both: Modules and Classes. In *Proceedings, ECOOP 92*, O. Lehrmann Madsen, Ed. Number 615 in Lecture Notes in Computer Science. Springer-Verlag, 19–32.

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

Wirth, N. 1977. MODULA : A Language for Modular Multiprogramming. *Software Practice and Experience 7*, 3–35.

Wirth, N. 1988. The Programming Language Oberon. *Software Practice and Experience 18,* 7 (July), 671–690.