Modula-2 and Oberon

Niklaus Wirth

Paper submitted to HOPL-3, June 2005, revised March, May and June 2006

Abstract

This is an account of the development of the languages Modula-2 and Oberon. Together with their ancestors ALGOL 60 and Pascal they form a family called Algol-like languages. Pascal (1970) reflected the ideas of Structured Programming, Modula-2 (1979) added those of modular system design, and Oberon (1988) catered to the object-oriented style. Thus they mirror the essential programming paradigms of the past decades. Here the major language properties are outlined, followed by an account of the respective implementation efforts. The conditions and the environments are elucidated, in which the languages were created. We point out that simplicity of design was the most essential, guiding principle. Clarity of concepts, economy of features, efficiency and reliability of implementations were its consequences.

1. Background

In the middle of the 1970s, the computing scene evolved around large computers. Programmers predominantly used time-shared "main frames" remotely via low-bandwidth (1200 b/s) lines and simple ("dumb") terminals displaying 24 lines of up to 80 characters. Accordingly, interactivity was severely limited, and program development and testing was a time-consuming process. Yet, the power of computers – albeit tiny in comparison with modern devices – had grown considerably over the decade. Therefore the complexity of tasks, and thus that of programs had grown likewise. The notion of parallel processes had become a concern and made programming even more difficult. The limit of our intellectual capability seemed reached, and a noteworthy conference in 1968 gave birth to the term *software crisis* [1] (see p.120).

Small wonder, then, that hopes rested on the advent of better tools. They were seen in new programming languages, symbolic debuggers, and team management. Dijkstra put the emphasis on better education. Already in the mid 1960s he had outlined his discipline of *Structured Programming* [3], and the language Pascal followed his ideas and represented an incarnation of a *Structured Language* [2]. But the dominating languages were FORTRAN in scientific circles and COBOL in business data processing. IBM's *PL/1* was slowly gaining acceptance. It tried to unite the disparate worlds of scientific and business applications. Some further, "esoteric" languages were popular in academia, for example *Lisp* and its extensions, which dominated the AI culture with its list processing facilities.

However, none of the available languages were truly suitable for handling the ever growing complexity of computing tasks. FORTRAN and COBOL lacked a pervasive concept of data types like that of Pascal; and Pascal lacked a facility for piecewise

compilation, and thus of program libraries. PL/1 offered everything to a certain degree. Therefore it was bulky and hard to master. The fact remained that none of the available languages was truly satisfactory.

I was fortunate to be able to spend a sabbatical year at the new Xerox Research Laboratory in Palo Alto during this time. There, on the personal workstation *Alto*, I encountered the language *Mesa*, which appeared to be the appropriate language for programming large systems. It stemmed from Pascal [2, 38], and hence had adopted a strictly static typing scheme. But it also allowed to develop parts of a system – called modules – independently, and to bind them through the linking loader into a consistent whole. This alone was nothing new. It was new, however, for strongly typed languages, guaranteeing type consistency between the linked modules. Therefore, compilation of modules was not called independent, but *separate compilation*. We will return to this topic later.

As Pascal had been Mesa's ancestor, Mesa served as Modula-2's guideline. Mesa had not only adopted Pascal's style and concepts, but also most of its facilities, to which were added many more, as they all seemed either necessary or convenient for system programming. The result was a large language, difficult to fully master, and more so to implement. Making a virtue out of necessity, I simplified Mesa, retaining what seemed essential, and preserving the appearance of Pascal. The guiding idea was to construct a genuine successor of Pascal meeting the requirements of system engineering, yet also to satisfy my teacher's urge to present a systematic, consistent, appealing, and teachable framework for professional programming.

In later years, I was often asked, whether indeed I had designed Pascal and Modula-2 as languages for teaching. The answer is "Yes, but not only". I wanted to teach *programming* rather than a language. A language, however, is needed to express programs. Thus, the language must be an appropriate tool, both for formulating programs and for expressing the basic concepts. It must be supportive, rather than a burden! But I also hasten to add that Pascal and Modula-2 were not intended to remain confined to the academic classroom. They were expected to be useful in practice. Further comments can be found in [44].

To be accurate, I had designed and implemented the predecessor language Modula [7 - 9] in 1975. It had not been conceived as a general-purpose language, but rather as a small, custom-tailored language for experimenting with concurrent processes and primitives for their synchronization. Its features were essentially confined to this topic, such as process, signal, and monitor [6]. The monitor, representing critical regions with mutual exclusion, mutated into modules in Modula-2. Modula-2 was planned to be a full-scale language for implementing the entire software for the planned personal workstation *Lilith* [12, 19]. This had to include device drivers and storage allocator, as well as applications like document preparation and e-mail systems.

As it turned out later, Modula-2 was rather too complex. The result of an effort at simplification ten years later was Oberon.

2. The Language Modula-2

The defining report of Modula-2 appeared in 1979, a textbook in 1982 [13]. A tutorial was published, following a growing popularity of the language, in [17, 18]. In planning Modula-2, I saw it as a new version of Pascal, updated to the requirements of the time, and I seized the opportunity to correct various mistakes in Pascal's design, such as, for example, the syntactic anomaly of the dangling "else", the incomplete specification of procedure parameters, and others. Apart from relatively minor corrections and additions the primary innovation was that of modules.

2.1. Modules

ALGOL had introduced the important notions of limited scopes of identifiers and of the temporary existence of objects. The limited visibility of an identifier and the limited lifetime of an object (variable, procedure), however, were tightly coupled: All existing objects were visible, and one that was not visible, did not exist (was not allocated). This tight coupling was an obstacle in some situations. We refer to the well-known function to generate the next pseudo-random number, where the last one must be stored to compute the next, while remaining invisible. ALGOL's designers noticed this, and quickly remedied the shortcoming by introducing the *own* property, an unlucky idea. An appropriate example is a procedure for generating pseudo-random numbers (c1, c2, c3 stand for constants):

```
real procedure random;
begin own real x;
    x := (c1*x + c2) mod c3; random := x
end
```

Here *x* is invisible outside the procedure. However, its computed value is retained and available the next time the procedure is called. Hence *x* cannot be allocated on a stack like ordinary local variables. The inadequacy of the own concept becomes apparent, if one considers how an initial value should be given to *x*.

Modula's solution was found in a second scoping structure, the *module*. In the first, the procedure (*block* in ALGOL), locally declared objects are allocated (on a stack) when control reaches the procedure, and de-allocated when the procedure terminates. With the second, the module, no allocation is associated; only visibility is affected. The module merely constitutes a wall around the local objects, through which only those objects are visible that are explicitly specified in an "export" or an "import" list. In other words, the wall makes every identifier declared within a module invisible outside, unless it occurs in the *export list*, and it makes every identifier declared in a surrounding module invisible inside, unless it occurs in the module's *import list*. This definition makes sense, if modules are considered as nestable, and it represents the concept of *information hiding* as first postulated by D.L.Parnas in 1972 [4].

Visibility being controlled by modules and existence by procedures, the example of the pseudo-random number generator now turns out as follows in Modula-2. Local variables of modules are allocated when the module is loaded, and remain allocated until the module is explicitly discarded.

```
module RandomNumbers;
    export random;
```

```
    var x: real;

    procedure random(): real;
    begin x := c1*x +c2) mod c3; return x
    end random;

begin x := c0  (*seed*)
end RandomNumbers
```

The notation for a module was chosen identical to the one of a monitor proposed by Hoare in 1974 [6], but is without connotation of mutual exclusion of concurrent processes (as in Modula [7]).

Modula-2's module can also be regarded as a representation of the concept of *abstract data type* postulated by B. Liskov in 1974 [5]. A module representing an abstract type exports the type, typically a record structured type, and the set of procedures and functions applicable to it. The type's structure remains invisible and inaccessible from the outside of the module. Such a type is called *opaque*. This makes it possible to postulate module invariants. Probably the most popular example is the stack. (In order to exhibit the principle, we refrain from providing the guards $s.n < N$ for *push* and $s.n > 0$ for *pop*).

```
module Stacks;
    export Stack, push, pop, init;

    type Stack =   record n: integer; (*0 ≤ n < N*)
                     a: array N of real
                   end ;

    procedure push(var s: Stack; x: real);
    begin s.a[s.n] := x; inc(s.n) end push;

    procedure pop(var s: Stack): real;
    begin dec(s.n); return s.a[s.n] end pop;

    procedure init(var s: Stack);
    begin s.n := 0 end init
end Stacks
```

Here, it would be desirable to parameterize the type definition. A stack's size and element type (here *N* and *real*) are obvious candidates for type parameters. The impossibility to do so makes the limitations of Modula-2's form of module for this purpose apparent. As an aside, we note that in object-oriented languages the concept of data type is merged with the module concept and is called a *class*. The fact remains that the two notions have different purposes, namely data structuring and information hiding, and they should not be confused, particularly so in languages used for teaching programming.

The basic idea behind Mesa's module concept was also information hiding, as communicated by Ch. Geschke, J. Morris and J. Mitchell in various discussions [10, 11]. But its emphasis was on decomposing very large systems into relatively large components, called modules. Hence, Mesa's modules were not nestable, but formed separate units of programs. Clearly, the key issue was to *interface*, to connect such

modules. However, it was enticing to unify the concepts of information hiding, nested modules, monitors, abstract data types, and Mesa system units into a single construct. In order to consider a (global) module as a program, we simply need to imagine a universe, into which global modules are exported and from which they are imported.

A slight distinction between inner, nested modules and global modules seemed nevertheless advisable from both the conceptual aspect and that of implementation. After all, global modules appear as the parts of a large system that are typically implemented by different people or teams. The key idea is that such teams design the interfaces of their parts together, and then can proceed with the implementations of the individual modules in relative isolation. To support this paradigm, Mesa's module texts were split in two parts: The *implementation part* corresponds to the conventional "program". The *definition part* is the summary of information about the exported objects, the module's *interface*, and hence replaces the export list.

If we consider the example of module *Stacks* and reformulate it under this aspect, its definition part is

```
definition Stacks;
    type Stack;
    procedure push(var s: Stack; x: real);
    procedure pop(var s: Stack): real;
    procedure init(var s: Stack);
end Stacks
```

This, in fact, is exactly the information a user (client) of module *Stacks* needs to know. He must not be aware of the actual representation of stacks, which implementers may change even without notifying clients. 25 years ago, this water tight and efficient way of type and version consistency checking put Mesa and Modula-2 way ahead of their successors, including the popular Java and C++.

Modula-2 allowed for two forms of specifying imports. In the simple form the module's identifier is included in the import list:

```
import M
```

In this case all objects exported by *M* become visible. For example, identifiers *push* and *pop* declared in *Stacks* are denoted by *Stacks.push* and *Stacks.pop* respectively. When using the second form

```
from M import x, P
```

the unqualified identifiers *x* and *P* denote the respective objects declared in *M*. This second form became most frequently used, but in retrospect proved to be rather misguided. First, it could lead to clashes, if the same identifier was exported by two different (imported) modules. And second it was not immediately visible in a program text, where an imported identifier was declared.

A further point perhaps worth mentioning in this connection is the handling of exported enumeration types. The desire to avoid long export lists led to the (exceptional) rule that the export of an enumeration type identifier implies the export of all constant identifiers of that type. As nice as this may sound for the abbreviation enthusiast, it also

has negative consequences, again in the form of identifier clashes. This occurs if two enumeration types are imported which happen to have at least one common constant identifier. Furthermore, identifiers may now appear that are neither locally declared, nor qualified by a module name, nor visible in an import list; an entirely undesirable situation in a structured language.

Whereas the notation for the module concept is a matter of language design, the paradigm of system development by teams influenced the implementation technique, the way modules are compiled and linked. Actually, the idea of compiling parts of a program, such as subroutines, independently was not new; it existed since the time of FORTRAN. However, strongly typed languages add a new aspect: Type compatibility of variables and operators must not only be guaranteed among statements within a module, but also, and in particular *between* modules. Hence, the term of *separate compilation* was used in contrast to *independent compilation* without consistency checks between modules. With the new technique the definition (interface) of a module is compiled first, thereby generating a *symbol file*. This file is inspected not only upon compilation of the module itself, but also each time a client module is compiled. The specification of a module name in the import list causes the compiler to load the respective symbol file, providing the necessary information about the imported objects. A most beneficial consequence is that the inter-module checking occurs at the time of compilation rather than each time a module is linked.

One might object that this method is too complicated, and that the same effect is achieved by simply providing the service module's definition (interface) in source form whenever a client is compiled. Indeed, this solution was adopted, for example in Turbo Pascal with its *include files*, and virtually all successors up to Java. But it misses the whole point. In system development, modules undergo changes, and they grow. In short, new *versions* emerge. This bears the danger of linking a client with wrong, old versions of servers -- with disastrous consequences. A linker must guarantee the correct versions are linked, namely the same as were referenced upon compilation. This is achieved by letting the compiler provide every symbol file and every object file with a *version key*, and to make compilers and linkers check the compatibility of versions by inspecting their keys. This idea went back to Mesa, and it quickly proved to be an immense benefit, and soon became indispensable.

2.2. Procedure Types

An uncontroversial, fairly straight-forward, and most essential innovation was the procedure type, also adopted from Mesa. In a restricted form it had been present also in Pascal, even ALGOL, namely in the form of parametric procedures. Hence, the concept needed only to be generalized, i.e. made applicable to parameters and variables. In respect to Pascal (and ALGOL), the mistake of incomplete parameter specification was amended, making the use of procedure types type-safe. This is an apparently minor, but in reality most essential point, because a type-consistency checking system is worthless if it contains loopholes.

2.3. The Type CARDINAL

The 1970s were the time of the 16-bit minicomputers. Their word length offered an address range from 0 to 2^{16}-1, and thus a memory size of 64K. Whereas around 1970, this was still considered adequate, it later became a severe limitation, as memories became larger and cheaper. Unfortunately, computer architects insisted on byte addressing, thus covering only 32K words.

In any case, address arithmetic required unsigned arithmetic. Consequently, signed as well as unsigned arithmetic was offered, which effectively differed only in the interpretation of the sign bit in comparisons and in multiplication and division. We therefore introduced the type CARDINAL (0 ... 2^{16}-1) to the set of basic data types, in addition to INTEGER (-2^{15} … 2^{15}-1). This necessity caused several surprises. As a subtle example, the following statement, using a CARDINAL variable *x*, became unacceptable.

x := N-1; **while** x ≥ 0 **do** S(x); x := x-1 **end**

The correct solution, avoiding a negative value of *x*, is of course

x := N; **while** x > 0 **do** x := x-1; S(x) **end**

Also, the definitions of division and remainder raised problems. Whereas mathematically correct definitions of quotient *q* and remainder *r* of integer division *x* by *y* satisfy the equation

$$q \times y + r = x, \quad 0 \le r < y$$

which yields, for example (7 **div** 3) = 2 and (-7 **div** 3) = -3, most available computers provided a division, where (-x) **div** y = -(x **div** y), that is, (-7 **div** 3) = -2.

2.4. Low-level Facilities

Facilities that make it possible to express situations which do not properly fit into the set of abstractions constituting the language, but rather mirror properties of the computer, are called *low-level facilities*. Although necessary at the time – for example to program device drivers and storage managers - I believe that they were introduced too light-heartedly, in the naive assumption that programmers would use them only sparingly and as a last resort. In particular, the concept of *type transfer function* was a major mistake. It allows the type identifier *T* to be used in expressions as a function identifier: The value of *T(x)* is equal to *x*, whereby x is interpreted as being of type *T*, i.e. *x* is *cast* into a *T*. This interpretation inherently depends on the underlying (binary) representation of data types. Therefore, every program making use of this facility is inherently implementation-dependent, a clear contradiction of the fundamental goal of high-level languages.

In the same category of easily misused features is the *variant record*, a feature inherited from Pascal (see [13], Chap. 20). The real stumbling block is the variant without tag field. The tag field's value is supposed to indicate the structure currently assumed by the record. If a tag is missing, no possibility exists to determine the current variant. It is exactly this lack which can be misused to access record fields with intentionally "wrong" types.

2.5. What was left out

C.A.R. Hoare used to remark that a language is indeed defined by the features it includes, but more so even by those which it excludes. My own guide-line was to omit features, whose correct semantics and best form were still unknown. Therefore it is worth while mentioning what was left out.

Concurrency was a hot topic, and still is. There was no clear favorite way to express and control concurrency, and hence no set of language constructs that clearly offered themselves for inclusion. One basic concept was seen in concurrent processes synchronized by *signals* (or conditions), and involving critical regions of mutual exclusion in the form of monitors [6]. Yet, it was decided that only the very basic notion of coroutines would be included in Modula-2, and that higher abstractions should be programmed as modules based on co-routines. This decision was even more plausible, because the primitives could well be classified as low-level facilities, and their realization encapsulated in a module (see [13], Chap. 30 and 31).

We also abandoned the belief that *interrupt handling* should be treated by the same mechanism as programmed process switching. Interrupts are typically subject to specific real-time conditions. Real-time response is impaired beyond acceptable limits, if interrupts are handled by very general, complicated switching and scheduling routines.

Exception Handling was widely considered as a must for any language suitable for system programming. The concept originated from the need to react in special ways to rare situations, such as arithmetic overflow, index values being beyond a declared range, access via nil-pointer, etc., generally conditions of “unforeseen” error. Then the concept was extended to let any condition be declarable as an exception requiring special treatment. What lent this trend some weight was the fact that the program handling the exception might lie in a procedure different from the one in which the exception occurred (was *raised*), or even in a different module. This precluded the programming of exception handling by conventional conditional statements. Nevertheless, we decided not to include exception handling (with the exception of the ultimate exception called *Halt*).

Modula-2 features pointers and thereby implies dynamic storage allocation. Allocation of a variable x↑ is expressed by the standard procedure *Allocate(x)*, typically taking storage from a pool area (*heap*). A return of storage to the pool is signaled by the standard procedure *Deallocate(x)*. This was known to be a highly unsatisfactory solution, because it allows to return records (storage) that are still reachable from other, valid pointer variables, and therefore constitutes a rich source of disastrous errors.

The alternative is to postulate a *global storage management*, which retrieves unused storage automatically, that is, a *garbage collector*. We rejected this for several reasons.

1. I believed it was better if programmers would devise their own storage managers, thus obtaining the most *effective use* of storage for the case at hand. This was of great importance at the time, considering the small memory size, such as 64k bytes for the PDP-11, on which Modula-2 was first implemented.

2. Garbage collectors could activate themselves at *unpredictable* times, and hence preclude dependable real-time performance, which was considered an important domain of applications of Modula-2.
3. Garbage collectors must rely on incorruptible meta-data about all variables in use. Given the many loopholes for breaching the typing system, it was considered impossible to devise secure garbage collection with a reasonable effort. The flexibility of the language had become its own impediment. Even today, providing a garbage collector with an unsafe language is a sure guarantee for occasional crashes.

3. Implementations

Although a preliminary technical memorandum stating certain goals and concepts of the new language was written in 1977, the effective language design took place in 1978-79. Concurrently, a compiler implementation project was launched. The available machinery was a single DEC PDP-11 with a 64K-byte store. The single-pass strategy of our Pascal compilers could not be adopted; a multipass approach was unavoidable in view of the small memory. It had actually been the Mesa implementation at the Palo Alto Research Center (PARC) of Xerox which had proved possible what I had believed to be impracticable, namely to build a complete compiler operating on a small computer. The first Modula-2 compiler, written by K. van Le in 1977 consisted of 7 passes, each generating sequential output written onto the 2M-byte disk. This number was reduced in a second design by U. Ammann to 5 passes in 1979. The first pass, the scanner, generated a token string and a hash table of identifiers. The second pass (parser) performed syntax analysis, and the third pass handled the task of type checking. Passes 4 and 5 were devoted to code generation. This compiler was operational in early 1979.

In the meantime, a new Modula-2 compiler was designed in 1979-80 by L. Geissmann and Ch. Jacobi with the PDP-11 compiler as a guide, but taking advantage of the features of the new Lilith computer. *Lilith* was designed by the author and R. Ohran along the guide lines of the Xerox *Alto* [12, 19, 21]. It was based on the excellent Am2901 bit-slice processor of AMD, and it was microprogrammed. The new Modula-2 compiler consisted of only four passes, code generation being simplified due to the new architecture. Development took place on the PDP-11. Concurrently, the operating system *Medos* was implemented by S. Knudsen - a system essentially following in the footsteps of batch systems with program load and execute commands entered from the keyboard. At the same time, display and window software was designed by Ch. Jacobi. It served as the basis of the first application programs, such as a text editor - mostly used for program editing - featuring the well-known techniques of multiple windows, a cursor/mouse interface, and pop-up menus.

By 1981, Modula-2 was in daily use and quickly proved its worth as a powerful, efficiently implemented language. In December 1980, a pilot series of 20 Liliths, manufactured in Utah under the supervision of R. Ohran, had been delivered to ETH Zürich. Further software development proceeded with a more than 20-fold hardware power at our disposal. A genuine personal workstation environment had successfully been established.

During 1984, the author designed and implemented yet another compiler for Modula-2. It was felt that compilation could be handled more simply and more efficiently, if full use were made of the now available larger store which, by today's measures, was still very small. Lilith's 64K-word memory and its high code density allowed the realization of a single-pass compiler. This resulted in a dramatic reduction in disk operations. Indeed, compilation time of the compiler itself was reduced from some 4 minutes to a scant 45 seconds.

The new, much more compact compiler retained the partitioning of tasks. Instead of each task constituting a pass - with sequential input from, and output to disk - it constituted a module with a procedural interface. Common data structures, such as the symbol table, were defined in a data definition module imported by (almost) all other modules. These modules represented a scanner, a parser, a code generator, and a handler of symbol files. During all these re-implementations, the language remained practically unchanged. The only significant change was the deletion of export lists in the definition parts of modules. The compilation of imports and exports constituted a remarkable challenge under the objective of economy of linking data and under the absence of automatic storage management [24].

Over the years, it became clear that designers of control and data acquisition systems found Modula-2 particularly attractive. This was due to the existence of both low-level facilities to control interfaces, and of modules to encapsulate critical, device-specific parts. A Modula-2 compiler was offered by two British companies, but Modula-2 never experienced the same success as Pascal, and it never became as widely known. The primary reason was probably that Pascal had been riding on the back of the micro-computer wave invading homes and schools, reaching a class of people not infected by earlier programming languages and habits. Modula-2, on the other hand, was perceived as merely an upgrade on Pascal, hardly worth the effort of a language transition. The author, however, naively believed that everyone familiar with Pascal would happily welcome the additions and improvements of Modula-2.

Nevertheless, numerous implementation efforts proceeded at various universities for various computers [15, 16, 23]. Significant industrial projects had adopted Modula-2. Among them was the control system for a new line of the Paris Metro and the entire software for a new Russian satellite navigation system. User’s groups were established, and conferences held, with structured programming in general and Pascal and Modula-2 in particular, as their themes. A series of tri-annual Joint Modular Languages Conferences (JMLC) started in 1987 in Bled (Slovenia), followed by events in Loughborough (England, 1990), Ulm (Germany, 1994), Linz (Austria, 1997), Zürich (Switzerland, 2000), Klagenfurt (Austria, 2003) and Oxford (England, 2006). Unavoidably, suggestions for extensions began to appear in the literature [25]. Even a direct successor was designed, called Modula-3 [33], namely in cooperation between DEC’s Systems Research Center (SRC) and Olivetti’s Research Laboratory in Palo Alto. Both had adopted Modula-2 as their principal system implementation language.

Closer to home, the Modula/Lilith project had a significant impact on our own future research and teaching activities. With 20 personal workstations available in late 1980, featuring a genuine high-level language, a very fast compiler, an excellent text editor, a high resolution display, a mouse, an Ethernet connecting the workstations, a laser

printer, and a central file server, we had in 1981 the first modern computing environment outside America. This gave us the opportunity to develop modern software for future computers fully 5 years ahead of the first such system commercially available, the Apple Macintosh, which was a scaled-down version of the Alto of 10 years before. Of particular value were our projects in modern document preparation and font design [20, 22].

4. From Modula to Oberon

Like my sabbatical year at Xerox in 1976/77 had inspired me to design the personal workstation Lilith in conjunction with Modula-2, my second stay in 1984/85 provided the necessary inspiration and motivation for the language and the operating system Oberon [29, 30]. Xerox PARC's Cedar system for its extremely powerful Dorado computer was based on the windows concept developed also at PARC for Smalltalk.

The Cedar system [14] was – to this author's knowledge – the first operating system that featured a mode of operation completely different from the then conventional batch processing mode. In a batch system, a permanent loop accepts command lines from a standard input source. Each command causes the loading, execution, and release of a program. Cedar, in contrast, allowed many programs to remain allocated at the same time. It did not imply (storage) release after execution. Switching the processor from one program to another occurred through the invocation of a program's commands, typically represented by *buttons* or *icons* in *windows* (called *viewers*) belonging to the program. This scheme had become feasible through the advent of large main stores (up to several hundred kilobytes), high-resolution displays (up to 1000 by 800 pixels), and powerful processors (with clock rate up to 25 Megahertz).

Because I was supposed to teach the main course on system software and operating system design after my return from the sabbatical leave, the encounter with the novel Cedar experiment appeared as a lucky coincidence. But how could I possibly teach with a good conscience the subject without truly understanding it? Thus the idea was born to gain first-hand experience by designing such a modern operating system on my own with Cedar as the primary source of ideas.

Fortunately, my colleague Jürg Gutknecht concurrently spent a sabbatical semester at PARC in the summer of 1985. We both were intrigued by this new style of working with a computer, but at the same time also appalled by the system's complexity and lack of a clear, conceptual basis. This lack was probably due to the merging of several, innovative ideas, and partly also due to a contagious enthusiasm of the pioneers, encouraged by an apparently unbounded future reservoir of hardware resources.

But how could the two of us possibly undertake and successfully complete such a large task? Were we not victims of an exuberant overestimation of our capabilities, made possible only by a naïve ignorance of the subject? We felt the strong urge to try and risk. We believed that a turnaround from the world-wide trend to more and more unmanageable complexity was long overdue. We felt that the topic was worthy of academic pursuit, and that ultimately teachers, students, and practitioners of computing would benefit from it.

We both felt challenged to mold the new concepts embodied by Cedar into a scheme that was clearly defined and therefore easy to teach and understand. Concentration on the essentials, omission of “nice-to-have” features, and careful planning before coding, were no well-meant guidelines heard in a classroom, but an absolute necessity considering the size of the task. The influence of the Xerox Lab was thus – the reader will excuse some oversimplification – twofold: We were inspired by *what* could be done, and shown *how not* to do it. The essential, conceptual ingredients of our intentions are summarized as follows:

1. Clear separation of the notion of *program* into the two independent notions of (1) the *module* as the unit of compilable text and of code and data to be loaded into store (and discarded from it), and of (2) the *procedure* as the unit of action invoked by a *command*.

2. The elimination of the concept of command lines written from the keyboard into a special command viewer. Actions would now be invoked by mouse button clicking (middle button = command button) on the *command name* appearing in *any* arbitrary text in *any* viewer. The form of a command name, *M.P*, *P* denoting the procedure, and *M* the module of which *P* is a part, would allow for a simple search in the lists of loaded modules and *M*’s commands.

3. The core of execution being a tight loop located at the bottom of the system. In this loop the common sources of input (keyboard, mouse, net) are continuously sampled. Any input forming a command causes the dispatch of control to the appropriate procedure (by an *upcall*), if needed after the prior loading of the entire module containing it (*load on demand*). Note that this scheme excludes the preemption of program execution at arbitrary points.

4. Storage retrieval by a single, global *garbage collector*. This is possible only under the presence of a watertight, preferably static, *type checking* concept. De-allocation of entire modules (code, global variables) occurs only through commands explicitly issued by the user.

5. Postulation of a simple syntax for (command) texts, paired with an *input scanner* parsing this syntax.

These five items describe the essence of our transition from batch mode operation to a modern, interactive multi-viewer operating environment, manifest by the transition from the Modula-2 to the Oberon world [30, 35]. The clearly postulated conceptual basis made it possible for two programmers (J. Gutknecht and me) alone to implement the entire system, including the compiler and text processing machinery, in our spare time during only two years (1977-79). The tiny size of this team had a major influence on the conceptual consistency and integrity of the resulting system, and certainly also on its economy.

We emphasize that the mentioned aspects concern the *system* rather than the *language* Oberon. A language is an abstraction, a formal notation; notions such as command line, tight control loop, and garbage collector do not and must not occur in a language definition, because they concern the implementation only. Therefore, let us now turn

our attention to the language proper. Like for the system, our intention was also for the language to strive for conceptual economy, to simplify Modula-2 where possible.

As a consequence, our strategy was first to decide what should be omitted from Modula-2, and thereafter to decide which additions were necessary.

5. The Language Oberon

The programming language Oberon was the result of a concentrated effort to increase the power of Modula-2 and simultaneously to reduce its complexity. Oberon is the last member of a family of "ALGOL-like" languages. It started with ALGOL 60, followed by ALGOL-W, Pascal, Modula-2, and ended with Oberon [27, 28]. By "ALGOL-like" is meant the procedural paradigm, a rigorously defined syntax, traditional mathematical notation for expressions (without esoteric ++, ==, /= symbols), block structure providing scopes of identifiers and the concept of locality, the availability of recursion for procedures and functions, and a strict, static data typing scheme.

The principal guideline was to concentrate on features that are basic and essential and to omit ephemeral issues. This was certainly sensible in view of the very limited manpower available. But it was also driven by the recognition of the cancerous growth of complexity in languages that had recently emerged, such as C, C++ and Ada. They appeared as even less suitable for teaching than for engineering in industry. Even Modula-2 now appeared as overly bulky, containing features that we had rarely used, and whose elimination would not cause a sacrifice. To try to crystallize the essential - not only the convenient and conventional - features into a small language seemed like a worth-while (academic) exercise [28, 43].

5.1. Features omitted from Oberon

A large number of standard data types not only complicates compilers, but also makes it more difficult to teach and master a language. Hence, data types were a primary target of our simplification zeal.

An undisputed candidate for elimination was Modula's *variant record*. Introduced with the laudable intent of providing flexibility in data structuring, it ended up mostly being misused to breach the typing concept. The feature allows to interpret a data record in various ways according to various overlaid field templates, where one of them is marked as valid by the current value of a *tag field*. The true sin was that this tag could be omitted. For more details, the reader is referred to [13], Chap. 20.

Enumeration types would appear to be an attractive and innocent enough concept to be retained. However, a problem appeared in connection with import and export: Would the export of an enumeration type automatically also export the constants' identifiers, which would have to be prefixed with the module's name? Or could, as in Modula-2, these constant identifiers be used unqualified? The first option was unattractive, because it produced unduly long names for constants, and the second because identifiers would appear without any declaration. As a consequence, the enumeration feature was dropped.

Subrange types were also eliminated. Experience had shown that they were used almost exclusively for indexing arrays. Hence, range checks were necessary for indexing rather

than for assignment to a variable of subrange type. Lower array bounds were fixed to 0, making index checks more efficient and subrange types even less useful.

Set types had proved to be of limited usefulness in Pascal and Modula-2. Sets implemented as bit strings of the length of a "word" were rarely used, even though union and intersection could be computed by a single logical operation. In Oberon, we replaced general set types by the single, predefined type **set**, with elements 0 – 31.

After lengthy discussions, it was decided (in 1988) to merge the definition text of a module with its implementation text. This may have been a mistake from the pedagogical point of view. The definitions should clearly be designed first as contracts between its designer and the module's clients. Instead, now all exported identifiers were simply to be marked in their declaration (by an asterisk). The advantages of this solution were that a separate definition text became superfluous, and that the compiler was relieved of consistency checking (of procedure signatures) between the two texts. An influential argument for the merger was that a separate definition text could automatically be generated from the module text.

The *qualified import* option of Modula-2 was dropped. Now every occurrence of an imported identifier must be preceded by its defining module's name. This actually turned out to be of great benefit when reading programs. The import list now contains module names only. This we believe to be a good example for the art of simplification: A simplified version of Mesa's module machinery was further simplified without compromising the essential ideas behind the facility: information hiding and type safe, separate compilation.

The number of *low-level facilities* was sharply reduced, and in particular type transfer functions were eliminated. The few remaining low-level functions were encapsulated in a pseudo-module, whose name would appear in the prominently visible import list of every module making use of such low-level facilities.

By eliminating all potentially unsafe facilities, the most essential step was finally made to obtain a truly high-level language. Watertight type checking, also across modules, strict index checking at run-time, nil-pointer checking, and the safe type extension concept let the programmer rely on the language rules alone. There is no longer a need to know about the underlying computer, or how the language is translated and data are represented. The old goal, that a language must be defined without mentioning an executing mechanism, had finally been reached. Clean abstraction from machines and genuine portability had become a reality. Apart from this, absolute type safety is - an often ignored truth - also the undisputable prerequisite for an underlying automatic storage management (garbage collector).

One feature must be mentioned that in hindsight should have been added: finalization. It implies the automatic execution of a specified routine when a module is unloaded or a record (object) is collected. Inclusion of finalization had been discussed, but its cost and implementation effort had been judged too high relative to its benefit. Evidently, its importance had been underestimated, particularly that of a module being unloaded.

5.2. New features introduced in Oberon

Effectively there are only two features introduced in Oberon: Type extension and type inclusion. This is surprising, considering the large number of eliminations.

The concept of *type inclusion* binds all arithmetic types together. Every one of them defines a range of values that variables of said type can assume. Oberon features 5 arithmetic types:

longreal ⊇ real ⊇ longint ⊇ integer ⊇ shortint

The concept implies that values of the included type can be assigned to variables of the including type. Hence, given

var i: integer; k: longint; x: real

assignments *k* := *i* and *x* := *i* are legal, whereas *i* := *k* and *k* := *x* are not. In hindsight, the fairly large number of arithmetic types looks like a mistake. The two types *integer* and *real* might have been sufficient. The decision was taken in view of the high importance of storage economy at the time, and because the target processor featured instruction sets for all five types. Of course, the language definition did not forbid implementations to treat *integer, longint*, and *shortint*, or *real* and *longreal* as the same.

The vastly more important new feature was *type extension* [26, 39]. Together with procedure-typed fields of record variables, it constitutes the technical foundation of the object-oriented programming style. The concept is better known under the anthropomorphic term *inheritance*. Consider a record type (class) *Window* (T0) with coordinates *x, y,* width *w* and height *h*. It would be declared as

T0 = **record** x, y, w, h: integer **end**

T0 may serve as the basis of an *extension* (subclass) *TextWindow* (T1), declared as

T1 = **record** (T0) t: Text **end**

implying that *T1* retain (inherit) all properties, (*x, y, w* and *h*) from its *base type T0*, and in addition feature a text field *t*. It also implies, that all *T1*s are also *T0*s, thereby allowing to form heterogeneous data structures. For example, the elements of a tree may be defined as of type *T0*. However, individually assigned elements may be of any type that is an extension of *T0*, such as a *T1*.

The only new operation required is the *type test*. Given a variable *v* of type *T0*, the Boolean expression *v* ***is*** *T* is used to determine the effective, current type assigned to *v*. This is a run-time test.

Type extension alone, in addition to procedure-types, is necessary for programming in object-oriented style. An *object* type (class) is declared as a record containing procedure-typed fields, also called *methods*. For example:

```
type viewer = pointer to record x, y, w, h: integer;
                  move: procedure (v: viewer; dx, dy: integer);
                  draw: procedure (v: viewer; mode: integer);
              end
```

The operation to draw a certain viewer *v* is then expressed by the call *v.draw(v, 0).* The first *v* serves to qualify the method *draw* as belonging to the type *viewer*, the second *v* designates the specific object to be drawn.

This reveals that object-oriented programming is effectively a style based on (inheriting) conventional, procedural programming. Surprisingly, most people did not consider Oberon as supporting object-orientation, simply because it did not use the new terminology. In 1990, H. Mössenböck spearheaded an effort to amend this apparent shortcoming and to implement a slight extension called Oberon-2 [34]. In Oberon-2 methods, i.e. procedures bound to a record type, were clearly marked as belonging to a record, and they implied a special parameter designating the object to which the method was to be applied. As a consequence, such methods were considered as constants and therefore required the additional override feature for subclasses.

6. Implementations

The first ideas leading to Oberon were drafted in 1985, and the language was fully defined in early 1986 in close cooperation with J. Gutknecht. The Report was only 16 pages long! [28]

The first compiler was programmed by this author, deriving it from the single-pass Modula-2 compiler. It was written in (a subset of) Modula-2 for Lilith with the clear intention to translate it into Oberon, and it generated code for our Ceres workstation, equipped with the first commercial 32-bit microprocessor NS32032 of the *National Semiconductor* Corporation. The compiled code was downloaded over a 2400 b/s serial data line. As was expected, the compiler was considerably simpler than its Modula-2 counterpart, although code generation for the NS processor was more complex than for Lilith's byte-code.

With the porting of the compiler completed, the development of the operating environment could begin. This system, (regrettably) also called Oberon, consisted of a file system, a display management system for windows (called *viewers*), a text system using multiple fonts, and backup to diskettes [30]. The entire system was programmed by J. Gutknecht and the author as a spare time activity over more than two years. This development process from scratch is described in [31]. The system was released in 1989, where after a larger number of developers became involved to generate applications. These included a network based on a low-cost RS-485 connection operating at 230 Kb/s [32], a laser printer, color displays (black and white was still the standard at the time), a laser printer, a mail and a file server, and more sophisticated document and graphics editors.

With the availability of a large number of Ceres workstations, Oberon was introduced as the language for introductory courses at ETH Zürich in 1990 [35, 36, 42], and also for courses in system software and compiler design. Having ourselves designed and implemented a complete system down to the last details, we were in a good position to teach software design. For many years, it had been our goal to publish a textbook not only sketching abstract principles, but rather showing concrete examples. Our efforts resulted in a single book containing the complete source text of this compact, yet real, useful, and convenient system. It was published in 1992 [37].

Following the earlier suggestion of C.A.R. Hoare to write texts describing master sample programs to be studied and followed by students, we had published a text on widely useful algorithms and data structures, and now extended the idea to an entire operating system. Hoare had claimed that every other branch of engineering is taught by its underlying theoretical framework and by textbooks featuring concrete, practical examples. However, interest in our demanding text remained disappointingly small. This may be explained in part by the custom in Computer Science of learning to write programs before reading any. Consequently, literature containing programs worth reading is rather scarce. A second reason for the low interest is that languages and operating systems are no longer popular topics of research. Also among leading educational institutions prevails the widespread feeling that the current commercial systems and languages are the end of the topic and here to stay. Their enormity is taken as evidence that there is no chance for small research groups to contribute; arguing is considered beside the point and providing an alternative without chance of any acceptance in practice.

Nevertheless we believe that the Oberon project was worth the effort, and that its educational aspect was considerable. It may still serve as an example of how to strive for simplicity and perspicuity in complex situations. Gigantic commercial systems are highly inappropriate to study principles and their economic realization. However, Oberon should not be considered as merely “a teaching language”. While it is suitable for teaching, because it allows starting with a subset without mentioning the rest, it is powerful enough to be used for large engineering projects.

During the years 1990 – 1995, Oberon received much attention, not the least because of our efforts to port it to the majority of commercial platforms. Compilers (code generators) were developed for the Intel xx86, the Motorola 680x0 (M. Franz), the Sun Sparc (J. Templ), the MIPS (R. Crelier) and the IBM Power (M. Brandis) processors [40]. The remarkable result of this concerted effort was that Oberon became a *truly portable platform*, allowing programs developed on one processor to compile and run on any other processor without adaptation.

Let us conclude this report with a peculiar story. The author wrote yet another code generator, not for a different processor, but rather for the same NS32000. This may seem strange and needs further explanation.

The NS processor had been chosen for Ceres, because of its HLL-oriented instruction set, like that of Lilith. Among other features, it contained a large number of addressing modes, among which was the *external mode*. It corresponded to what was needed to address variables and call procedures of imported modules, and it allowed a fast linking process through the use of link tables. This sophisticated scheme made it possible to quickly load and link modules without any modification of the code. Only a simple link table had to be constructed, again similar to the case of Lilith.

The implementers of Oberon for other platforms had no such feature available. Nevertheless, they managed to find an acceptable solution. At the end, it turned out less complicated than feared, and I started to wonder, how an analogous scheme used in the processor of National Semiconductor would perform. To find the answer, I wrote a

code generator using regular branch instructions (BSR) in place of the sophisticated external calls (CXP), and I developed a linking loader adapted to the new scheme.

The new linker turned out to be not much more complicated, and hardly any slower. But execution speed of the new programs was considerably (up to 50%) higher. Such a factor was totally unexpected. It is explained by the development of the NS processor over various versions and many years. In place of the 32032 in 1985 we used the 32532 in 1988 and the 32GX32 in 1990, which had the same instruction set, but were internally very different. The new versions were internally organized rather like RISC architectures, with the effect that simple, frequent instructions would execute very fast, while complex, rarely used instructions, such as our external calls, would perform poorly. Simple operations had become extremely fast (due to rising clock rates), whereas memory accesses remained relatively slow. On the other hand, memory capacity had grown tremendously. The relative importance of speed and code size had been changed. Hence, the old goal of high code density had become almost irrelevant.

The same phenomenon caused us to abandon the use of other “high-level” instructions, such as index bound checks and multiply-adds for computing matrix indices. This is a striking example of how hardware technology can influence software design considerations very profoundly.

7. Conclusions and Reflections

My long term goal had been to demonstrate that a systematic design using a supportive language leads to lean, efficient, and economic software, requiring a fraction of the resources that is usually claimed. This goal has been reached successfully. I firmly believe, out of many experiences over many years, that a structured language is instrumental in achieving a structured design. In addition, it was demonstrated that a clean, compact design of an entire software system can be described and explained in a single book [37]. The entire Oberon System, including its compiler, text editor and window system occupied less than 200K bytes of main memory, and compiled itself in less than 40 seconds on a computer with a clock frequency of 25 MHz.

In the current year 2007, however, such figures seem to have little significance. When the capacity of main memory is measured in hundreds of megabytes, and disk space is available in dozens of gigabytes, 200K bytes do not count. When clock frequencies are of the order of gigahertz, the speed of compilation is irrelevant. Or, expressed the other way round, in order that a computer user will recognize a process as being slow, the software must indeed be lousy. The incredible advances in hardware technology have exerted a profound influence on software development. Whereas they allowed systems to reach phenomenal performance, their influence on the discipline of programming have been rather detrimental as a whole. They have permitted software quality and performance to drop disastrously, because poor performance could easily be hidden behind faster hardware. In teaching, the notions of economizing memory space and processor cycles have become a thing apart. In fact, programming is hardly considered as a serious topic; it can be learnt by osmosis or, worse, by relying on extant program “libraries”.

This stands in stark contrast to the times of ALGOL and FORTRAN. Languages were to be precisely defined, their unambiguity to be proven; they were to be the foundation of a logical, consistent framework for proving programs correct, and not only syntactically well-formed. Such an ambitious goal can be reached, only if the framework is sufficiently small and simple. By contrast, modern languages are constantly growing. Their size and complexity is simply beyond anything that might serve as a logical foundation. In fact, they elude human grasp. Manuals have reached dimensions that effectively discourage any search for enlightenment. As a consequence, programming is not learnt from rules and logical derivations, but rather by trial and error. The glory of interactivity helps.

The world at large seems to tolerate this development. Given the staggering hardware power, one can usually afford to be wasteful in space and time. The boundaries will be hit in other dimensions: usability and reliability. The enormity of current commercial systems limits understanding and fosters mistakes, leading to product unreliability. Signs that the limits of tolerance are being reached, have begun to appear. Over the past few years I heard of a growing number of companies that had adopted Oberon as their exclusive programming tool. Their common characteristic was the small size and a small number of trusting and faithful clients, requesting software of high quality, reliability and ease of use. Creating such software requires that its designers understand their products thoroughly. Naturally, this understanding must include the underlying operating system and the libraries, on which the designs rest and rely. but the perpetual complexification of commercial software has made such understanding impossible. These companies have found Oberon to be the viable alternative.

Not surprisingly, these companies consist of small teams of expert programmers having the competence to make courageous decisions and enjoying the trust and confidence of a limited group of satisfied customers. It is neither surprising that small systems like Oberon are finding acceptance primarily in the field of embedded systems for data acquisition and real-time control. Here, not only economy is a foremost concern, but even more so are reliability and robustness.

Still, these clients and applications are the exception. The market favors languages of commercial origin, regardless of their technical merits or defects. The market’s inertia is enormous, as it is driven by a multitude of vicious circles which reinforce themselves. Hence, the value and role of creating new programming languages in research is a legitimate question, and it must be posed.

New ideas for improving the discipline of programming stem from practice. They are to be expressed in a notation, eventually forming a concrete language, which is to be implemented and tested in the field. Insights thus gained find their ways into new versions of widely used, commercial languages, slowly, very slowly over decades. It is fair to claim that Pascal, Modula-2, and Oberon have been successful in making such contributions over time.

The most essential of their messages, however, is expressed in the heading of the Oberon Report: “Make it as simple as possible …”. This advice has not yet been widely understood [41]. It seems that currently commercial interests point in another direction. Time will tell.

Acknowledgements

The design, implementation and development of Modula-2 and Oberon constituted a long-term team effort. I feel much indebted to everybody, who contributed to this effort, be it as a member of a team, or as a user providing valuable feedback. I had made it a principle to have a language implemented and tested *before* publication. Hence the work of my associates was of paramount importance and is gratefully acknowledged. In fact, the implementers (in the case of Oberon myself) were our first guinea pigs and sources of valuable feedback.

Foremost my thanks go to the Federal Institute of Technology (ETH) in Zürich for its continuous, tacit support and trust in our research. All over three decades, four to five assistants (Ph.D. students) acted as my associates as full-time employees of the school. The projects provided topics for their dissertations and trained them to become excellent software engineers. I rarely had to seek funding, nor write innumerable proposals and justifications. Therefore I could concentrate on teaching and research. I felt free of industrial pressures and prejudices, I could design what I considered right, and not necessarily what was requested, conventional, or simply fashionable.

I also emphasize that ETH's long tradition of keeping teaching and research closely tied together was of considerable importance. I immediately profited from the necessity to keep concepts and construct simple, clear, explicable for effective teaching. Thereby I realized the long-term value of simple, straight-forward design. Our research environment had been ideal.

References

1. P. Naur, B. Randell, *"Software Engineering"*. Nato Science Committee, Conference Report, Garmisch, Oct. 1968.
2. N. Wirth, "The Programming Language Pascal," *Acta Informatica, 1* (Jun 1971) pp. 35-63.
3. O.-J. Dahl, E. W. Dijkstra, and C.A.R. Hoare, "*Structured Programming*", Academic Press, 1972.
4. D. L. Parnas, "On the Criteria to be used in Decomposing Systems into Modules". *Comm. ACM, 15,* (Dec. 1972), pp. 1053 – 1058.
5. B. Liskov and S. Zilles, "Programming with Abstract Data types", *Proc. ACM Conf. on Very High Level Languages, SIGPLAN Notices 9,* 4 (April 1974), pp. 50-59.
6. C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", *Comm. ACM 17,* 10, pp. 549 – 557, (Oct. 1974).
7. N. Wirth, "Modula: A language for modular multiprogramming," *Software - Practice and Experience, 7,* pp. 3-35 (1977).
8. --, "The use of Modula," *Software - Practice and Experience, 7,* pp. 37-65 (1977).
9. --, "Design and Implementation of Modula," *Software - Practice and Experience, 7,* pp. 67 - 84 (1977).

10. C. Geschke, J. Morris, and E. Satterthwaite, “Early Experience with Mesa,” *Comm. ACM 20,* 8, August 1977.

11. J. Mitchell, W. Maybury, and R. Sweet, “*Mesa Language Manual* (version 5.0),” Xerox PARC Technical Report CSL-79-3, April 1979.

12. N. Wirth, “Lilith: A Personal Computer for the Software Engineer,” *Proc. 5th Int'l Conf. Software Engineering.* San Diego, 1981. IEEE 81CH1627-9.

13. --, “*Programming in Modula-2*.” Springer-Verlag, Heidelberg, New York, 1982. ISBN 0-387-50150-9

14. W. Teitelman, “A Tour through Cedar”. *IEEE Software, 1*, 2 (April 1984) pp. 44-73.

15. T. L. Andersen, “Seven Modula compilers reviewed“. *J. of Pascal, Ada, and Modula-2.* March-April 1984.

16. M. L. Powell, “A portable, optimizinmg compiler for Modula-2”. *SIGPLAN Notices 19* (6), (June 1984) pp. 310 – 318.

17. N. Wirth, “History and Goals of Modula-2.” *BYTE,* Aug. 84, pp. 145-152.

18. J. Gutknecht, “Tutorial on Modula-2”. *BYTE,* Aug. 1984, pp. 157-176.

19. R. Ohran, “Lilith and Modula-2” *BYTE,* Aug. 1984, pp. 181-192.

20. J. Gutknecht, W. Winiger, „Andra: The Document Preparation System of the Personal Workstation Lilith”. *Software - Practice & Experience, 14,* (1984), pp. 73-100.

21. G. Pomberger, “Lilith and Modula-2”. Hanser Verlag, 1985. ISBN 3-446-14328-9.

22. J. Gutknecht, “Concepts of the Text Editor Lara”. *Comm. ACM, 28,* 9 (Sept. 1985), pp. 942-960.

23. P. H. Hartel and D. Starreveld, “Modula-2 Implementation Overview”. *J. of Pascal, Ada, and Modula-2.* July/Aug. 1985, pp. 9-23.

24. J. Gutknecht, “Separate Compilation in Modula-2”. *IEEE Software,* Nov. 1986, pp. 29-38.

25. P. Rovner, “Extending Modula-2 to Build Large,” Integrated Systems. *IEEE Software,* Nov. 1986, pp. 46-57.

26. N. Wirth, “Type Extension”. *ACM TOPLAS, 10,* 2 (April 1988), pp. 204-214.

27. --, “From Modula to Oberon”. *Software - Practice and Experience, 18,* 7, (July 1988), pp. 661-670.

28. --, “The Programming Language Oberon.” *Software - Practice and Experience, 18,* 7 (July 1988), pp. 671-690.

29. --, “Oberon: A System for Workstations”. *Microprocessing and Microprogramming 24* (1988) pp. 3-8.

30. N. Wirth and J. Gutknecht, “The Oberon System.” *Software - Practice and Experience, 19,* 9, (Sept. 1989), pp. 857-893

31. N. Wirth, “Designing a System from Scratch.” *Structured Programming, 10,* 1 (Jan. 1989), pp. 10-18.

32. --, “Ceres-Net: A Low-Cost Computer Network”. *Software - Practice and Experience, 20,* 1, (Jan. 1990), pp. 13-24.

33. Greg Nelson, ed. “*Systems Programming with Modula-3”*, Prentice Hall 1991.

34. H. Mössenböck and N. Wirth, “The Programming Language Oberon-2.” *Structured Programming, 12* (1991), pp. 179-195.

35. M. Reiser, “*The Oberon System”*. Addison-Wesley 1991. ISBN 0-201-5442-9

36. M. Reiser and N. Wirth, “*Programming in Oberon: Steps beyond Pascal and Modula*.” Addison-Wesley, 1992. ISBN 0-201-56543-9

37. N. Wirth and J. Gutknecht, “*Project Oberon”*. Addison-Wesley, 1992. ISBN 0-201-54428-8

38. N. Wirth. “Recollections about the development of Pascal,” in History of Programming Languages-II, Thomas J. Bergin, Jr. and Richard G. Gibson, Jr. (eds), ACM Press and Addison-Wesley Publishing Company (1993), pp. 97-120.

39. H. Mössenböck, “Extensibility in the Oberon System”. *Nordic Journal of Computing 1* (1994), pp. 77 – 93.

40. M. Brandis, R. Crelier, M. Franz and J. Templ, “The Oberon System Family”. *Software - Practice and Experience, 25,* (Dec. 1995), pp. 1331-1366.

41. N. Wirth, “A Plea for Lean Software”. *IEEE Computer,* Feb. 1995, pp. 64-68.

42. A. Fischer, H. Marais, “The Oberon Companion”. 1998. ISBN 3 7281 2493 1.

43. M. Franz, “Oberon – The Overlooked Jewel”, in *The School of Niklaus Wirth.* L. Böszörmenyi, J. Gutknecht, G. Pomberger, eds. d-punkt.verlag. Heidelberg, 2000. ISBN 1-55860-723-4

44. N. Wirth, “Pascal and its Successors”. In M. Broy and E. Denert, *“Software Pioneers”*, Springer-Verlag, 2002, pp. 109 – 119, ISBN 3-540-43081-4.

http://www.inf.ethz.ch/personal/wirth

http://www.oberon.ethz.ch