Oberon – The Overlooked Jewel

Michael Franz

University of California, Irvine

Abstract

Niklaus Wirth has received much deserved fame for the creation of Pascal, but in many ways, he subsequently became a victim of Pascal's success. In an age of rising specialization, in which most researchers are trying to define themselves as experts in increasingly narrow domains, Wirth stands out as a rare generalist, almost an "Universalgenie" of our discipline. Sadly, the larger computer science community has been unable or unwilling to recognize Wirth's broader horizon as a builder of systems, and throughout his career has pigeonholed him as a "language and compiler guy".

But ever since Pascal, the language aspect has been almost secondary to Wirth's work. Modula(-2) and Oberon both came out of larger system-level design projects that simultaneously also developed workstation computers, modular operating systems, and suites of innovative application programs. Unfortunately, these other important contributions were overshadowed by the programming languages they were associated with, and hence never received the recognition they deserved.

This article presents selected facets of Project Oberon, the latter of Wirth's two large system-level design efforts. The leitmotiv of this project was a quote from Einstein, "make it as simple as possible, but not simpler". And if any further evidence was still needed, Oberon provided the conclusive proof for Wirth's mastery of The Art of Simplicity.

1 Introduction

The world at large will remember Niklaus Wirth primarily as "that language guy", and associate his name with *Pascal*, the programming language he created in the late 1960s. But the fame that the widespread success of Pascal brought to Wirth was at best a mixed blessing, as it overshadowed all of his subsequent accomplishments, some of them even greater than Pascal itself.

This was, of course, partially a problem of Wirth's own making. In his modest Swiss Engineer's manner, and quite untypical for a celebrated academic, Wirth consistently avoided exploiting the initially high-speeding Pascal bandwagon for the

purpose of "marketing" his subsequent ideas. Indeed, he even refrained from giving his later programming-language creations names such as "Pascal-2", "Pascal Plus", or "Pascal-2000" but instead opted first for "Modula", and then "Oberon". Both of these languages would probably have been considerably more successful if the connection to the Pascal legacy had been made in their name[1], but this would have run counter to Wirth's view that a marketer he is not.

Unfortunately, in an academic culture of increasingly flatulent self-promotion, such modesty eventually almost inevitably leads to a certain degree of self-imposed obscurity. In Wirth's case, the world continued to clamor for "more Pascal" for quite a while, but Wirth had long moved on to new horizons and more or less ignored these requests. Eventually, the world lost interest, and it, too, moved on to new horizons, which unfortunately were not always compatible with Pascal and its successor languages.

The principal reason why Wirth did not immediately respond to clamors for "more Pascal" was of course simply that he did *not* view himself primarily as a "language guy". Indeed, Wirth's publication record quite conclusively demonstrates that his talent has always been much broader and focused chiefly at the *system* level. Twice in his career, Wirth was instrumental in building a complete computer system "from scratch", each consisting of a hardware platform, a modular operating system and related systems-level software such as compilers and assemblers, and also a limited amount of highly innovative application software.

Both of these systems, **Lilith**/Medos/Modula[2] in 1981, and Ceres/**Oberon** in 1988 were years ahead of their time and could at the point of their inception be rivalled by only a small number of far more complex research systems being developed at places such as Xerox' Palo Alto Research Laboratory[3]. For example, the Oberon System of almost ten years ago anticipated such innovations as a hypertext-based user interface, living "applets" embedded within documents, target-machine independent mobile code, and document-centric computing. And Oberon provided not just the vision, but also executed that vision in a robust implementation

---

[1] Wirth himself makes this observation in [Wir93].

[2] The Lilith/Medos/Modula-2 system is usually summarily referred to as the "Lilith system" while the "Ceres/Oberon" system is commonly known as "The Oberon System". This probably reflects the greater relative importance of the hardware aspects at the time of Lilith, as well as the fact that Oberon soon emancipated itself of the Ceres hardware platform by being ported to several alternative architectures as well.

[3] The Lilith and Oberon projects were in fact both inspired by systems that Wirth had encountered at Xerox PARC, namely the Mesa and Cedar systems. Wirth succeeded in replicating many of the key characteristics of these advanced systems with considerably less effort, achieving this remarkably reduction in complexity to some extent by leaving out some inherently complex features, such as preemptive multitasking. As a consequence, comparisons with the much larger research systems existing at the time aren't entirely fair.

that could actually be and consequently *was* used on a day-to-day basis—quite unlike some of today's "research" systems.

In retrospect, both the Lilith and Oberon systems were probably *too far* ahead of their time, which is why they did not receive the enthusiastic responses they should have deserved. In both cases, the situation was exacerbated by the fact that the research community was expecting a *language* from Wirth, while he delivered a *system*. As a consequence, the predictable result in both instances was that the world at large focused only on the language component of Wirth's offering, and more or less ignored the remainder that was at least as important.

The story of Lilith is for someone else to tell. This article presents my hindsight view of The Oberon System, from the highly subjective perspective of having been one of the early members of Wirth's Oberon group (since early 1989)[4]. I will try to illuminate certain aspects of Oberon that were relevant then and still are relevant today, concentrating on the lesser-known facets rather than the previously published ones. The most important influence throughout was of course Wirth's insistence on simplicity of design, which finally seems to be finding a resonance in the world of computing. I believe that Wirth's ultimate impact will be felt even stronger ten years from now than it is today, and that the legacy of his accomplished career devoted to The Art of Simplicity will be enduring.

2 The Oberon Compiler

Niklaus Wirth is widely known as the creator of several programming languages. What is less well known is that he also personally wrote several compilers, including the first single-pass compiler for Modula-2 that later evolved into the initial compiler for Oberon[5]. These compilers distinguished themselves by their particularly simple design – they didn't aspire to tickle the last possible bit of achievable performance out of a piece of code, but aimed to provide adequate code quality at a price-point of reasonable compilation speed and compiler size. At the time, this was in stark contrast to almost all other research in compilers, which generally had been characterized by an enormous and ever-increasing complexity of optimizations to the detriment of compilation speed and overall compiler size.

In order to find the optimal cost/benefit ratio, Wirth used a highly intuitive metric, the origin of which is unknown to me but that may very well be Wirth's own

---

[4] To the best of my knowledge, when I started my Diploma Thesis in the Fall of 1988, I actually became only the third day-to-day user of Oberon (after J. Gutknecht and N. Wirth himself). At that time, all other members of the Institute for Computer Systems were still using Modula-2, and the Oberon System and language themselves were still undergoing minor revisions.

[5] The initial compilers for Pascal were written by E. Marmier, U. Ammann, and R. Schild [Wir93], and the initial Modula-2 compiler by L. Geissmann.

invention. He used the compiler's *self-compilation speed* as a measure of the compiler's quality. Considering that Wirth's compilers were written in the languages they compiled, and that compilers are substantial and non-trivial pieces of software in their own right, this introduced a highly practical benchmark that directly contested a compiler's complexity against its performance. Under the self-compilation speed benchmark, only those optimizations were allowed to be incorporated into a compiler that accelerated it by so much that the intrinsic cost of the new code addition was fully compensated[6].

And true to his quest for simplicity, Wirth continuously kept improving his compilers according to this metric, even if this meant throwing away a perfectly workable, albeit more complex solution. I still vividly remember the day that Wirth decided to replace the elegant data structure used in the compiler's symbol table handler by a mundane linear list. In the original compiler, the objects in the symbol table had been sorted in a tree data structure (in identifier lexical order) for fast access, with a separate linear list representing their declaration order. One day Wirth decided that there really weren't enough objects in a typical scope to make the sorted tree cost-effective. All of us Ph.D. students were horrified: it had taken *time* to implement the sorted tree, the solution was *elegant*, and it worked *well* – so why would one want to throw it away and replace it by something *simpler*, and even worse, something as prosaic as a linear list? But of course, Wirth was right, and the simplified compiler was both smaller and faster than its predecessor.

And fast these compilers were – not just Wirth's original compiler, but also the Oberon compilers that were subsequently developed by his students for a variety of target architectures [BCF95]. Every single one of these compilers outperformed the respective manufacturer's recommended compiler for the particular architecture by several orders of magnitude in compilation speed, and on several of the platforms, also in code quality. Only on the newer RISC platforms could the Wirth-style compilers not quite compete, mainly because they used a linear-scan register allocation mechanism rather than the more advanced graph-coloring variants employed by their competitors [Cha82]. It is quite ironic that in the current trend towards "just-in-time" compilation for Java, in which compilation speed is more important than the last bit of execution speed, Wirth-style compilation is making a grandiose comeback – unfortunately sometimes without proper attribution[7].

---

[6] A more detailed discussion of this and related metrics, and an example application of the metric, can be found in [Fra94b].

[7] For example, Adl-Tabatabai et al. [ACL98] describe a compilation strategy for Java that they might just as well have copied from one of Wirth's books [WG92, Wir96], but they present it as a new invention and give it the new name "lazy code selection". Wirth has been teaching the identical method at least since the mid-1980's, calling it "delayed code emission".

3. The Oberon Libraries and Application Program Interface

The fact that they had no prior experience in operating system design turned out to be a blessing when Niklaus Wirth and Jürg Gutknecht were designing the interfaces of the core modules of the Oberon system. Unconstrained by any preconceived notions of how "things should work", they designed a thoroughly novel system architecture that exhibits quite a few little strokes of genius throughout.

For example, anyone who has ever had the opportunity to use Oberon's file system is likely to find most other file system interfaces lacking. Unlike conventional file systems, Oberon's differentiates between the abstractions of a *data file* and an access mechanism to it. The latter is called a "Rider" and maintains the current position[8]. An arbitrary number of these Riders may operate concurrently on the same file, sharing just one set of buffers with full synchronization. In other file systems, these two abstractions are intertwined, making the "current position" a property of the file itself. Worse still, these other file systems often permit the creation of several unsynchronized access paths to the same file by allowing "multiple opening of the same file"– this may of course lead to corruption of the associated data file if the ranges of several concurrent access paths inadvertently overlap.

Oberon also separates the directory operations from the file operations. When a new file is created in Oberon, no entry is made into the directory, making the file anonymous. A name can be entered into the directory at any time, which makes the file permanent on the disk and possibly shadows an existing file with the same name. At any given moment, a directory lookup will simply return the instance of the file that was last registered. In my experience, this separation of directory and file operations makes programming much more intuitive and simple.

As a third example from among the many contributions that simplify the task of programming, the Oberon System allows an arbitrary number of files to be open simultaneously, and they neither need to be explicitly closed, nor deleted. Instead, file closing (with concomitant buffer flush to disk) is delegated to the garbage collector: files whose descriptors become unreachable from the executing code are automatically closed, and if such a file is also no longer accessible via the directory (either because it has never been registered or because another file with the same name was subsequently registered) it is automatically deleted and its storage space reclaimed.

The file system is only one example of how Oberon solved some age-old problems in a novel and practical manner. Future system designers would do well to study Oberon's application program interfaces more closely.

---

[8] C. Szyperski [Szy92b, Szy98] later generalized this separation of concerns into the "Carrier/Rider" design pattern and applied it successfully to domains other than the file system.

4 Oberon's Hypertext-Based User Interface

At the time of its creation, Oberon's user interface was highly unusual and outright intimidating to newcomers. In the initial absence of introductory tutorials for beginning users, it also had a very steep learning curve—in short, this was a system for "power users" (and yet was used very successfully for many years by ETH's undergraduate population).

However, revisiting the Oberon user interface ten years later, one finds that it no longer seems so intimidating. Oberon-style interaction has become ubiquitous, since today every web browser and even Microsoft's Windows desktop has many similarities with the Oberon system of yore. Furthermore, people have become much more accustomed to initiating complex commands by mouse actions.

But even today, Oberon's design is unrivalled in its simplicity and complete avoidance of modal dialogs. For example, a file being displayed in a Viewer (window) could be saved under a different name in the original Oberon system simply by editing the name displayed in the Viewer's title bar and then clicking the save command in the menu. To this date, I find this more intuitive than any "save as" command offered in any modern system that then will pop up a modal dialog box to enter a new name.

Oberon's steep learning curve was due to the fact that it used a three-button mouse, in which almost all combinations of simultaneous mouse-clicks (and "inter-clicks") had separate meanings. Oberon also differentiated between a current focus (a user-set coordinate on a screen, most often used to mark a whole window for a subsequent command), a text entry position, and multiple text selections. In almost all other systems, these points of attention are combined, but separating them is what makes powerful commands (such as "replace all occurrences of the second most recent text selection in the document currently marked by the focus by the text selected most recently") possible.

But the real power behind Oberon's original user interface[9] came from the pervasive use of *text as a unifying abstraction*. All text in Oberon behaved the same, whether it was a file name displayed in a Viewer's title bar, a command in a menu bar, the output of a "list directory" command, or the contents of an editor window. And almost all operations offered by the system operated on such abstract texts rather than on "passive" character arrays or data files. For example, if one had wanted to, one could have run the Oberon compiler directly on the text representing a Viewer's title bar, or one could have compiled the associated menu bar (although

[9] J. Gutknecht and H. Marais subsequently developed an enhanced graphical user interface for Oberon System 3 [Gut94] that makes the end users' experience far more pleasant. Not surprisingly, however, this increased end-user convenience comes at the price of a somewhat higher complexity on the programmers' side. It also brings with it a certain loss of the rigid regularity of the original text-based system, because the user interface now contains a multitude of data types other than text.

neither of this would have made much sense). Oberon's approach of incorporating text as a fundamental abstraction was a major improvement over the traditional "pipe" solution as found in Unix-like systems, simultaneously more powerful and intuitively simpler.

5 Extensibility and Document-Centric Approach

The Oberon System was designed with *extensibility* in mind—the ability to augment the system's functionality, possibly even at run-time, without having to modify or recompile any of the existing parts [PS94]. This naturally leads to an operational model founded on the principle of *dynamic loading* of code modules. Oberon refines this model considerably by allowing multiple entry points (so-called "commands") into each dynamically loaded module that can be activated by the end-user individually, and by preserving a module's global state across multiple invocations of such individual commands.

Remarkably, Oberon's architecture from the very beginning offered a still greater degree of extensibility, beyond the mere provision of additional commands in supplementary modules. By adopting a document-centric approach based on a generalization of the classic Model-View-Controller (MVC) design pattern [KP89], Wirth and Gutknecht arrived at a highly flexible and elegant system structure, permitting end-users to create wholly separate Model-View-Controller triplets that could nevertheless share the display space and other system resources with the built-in MVC-triplet supporting the text abstraction.

The key to this flexibility was a departure from the traditional object-oriented model of class-based dynamic dispatch with implementation inheritance[10]. Oberon replaced this model with an architecture in which the messages themselves were represented by objects[11]. This had two important consequences: First, the message hierarchy used in the system became independent of the object hierarchy and could be evolved independently of it. This in turn made a *generic broadcast mechanism* possible by which messages could be routed through the inner layers of the system even though they originated and were ultimately processed in an outer layer[12].

---

[10] H. Mössenböck, J. Templ, and R. Griesemer later incorporated the class construct of traditional object-oriented programming languages (along with implementation inheritance) into a variant strain of Oberon [MJG89, MW92].

[11] In the classification by Gamma et al. [GHJ95], this is the *Command (233)* design pattern.

[12] In a traditional object-oriented design, the generic recipient of a message is modelled using an abstract class. This limits the range of messages that can be sent—although the concrete recipient in an extension of the original system may "understand" additional messages, these cannot be routed through the core system that predates the extension. For a more detailed discussion of this issue, see [Fra98].

It is precisely this architecture that makes Oberon so powerful at such comparably small size and complexity. Interestingly enough, the structure of the Oberon System cannot be replicated using class-based dynamic dispatch all by itself. Unfortunately, critics of the Oberon language who deplored its lack of automated type-based dispatch and method inheritance apparently never realized that these features would not have been of any use in supporting the architecture of the Oberon System[13].

Szyperski later demonstrated that Oberon's particular message dispatch mechanism was a sufficiently powerful foundation to support true "compound documents" [Szy92a], in which different and mutually unaware software components are jointly responsible for the display of a document containing a multitude of complex data items. Apparently, the Oberon system has always had all the mechanisms in place to be called a software component framework [Szy98], although Wirth and Gutknecht, in their dislike of marketing, neglected to advertise this fact properly.

6 Cross-Platform Computing

Niklaus Wirth and Jürg Gutknecht initially conceived the Oberon System as a didactic tool that could actually be understood in its entirety by a single person. As a consequence, although Oberon was written specifically as the native operating system for the Ceres workstation, its simplicity of design and orthogonality of concepts made it surprisingly simple to port onto other computer platforms.

Ironically, Gutknecht, and to some extent also Wirth had some serious reservations about this in the beginning, as they feared that due to the necessity of having to build on top of existing operating system software, the performance on other platforms would be so bad as to give Oberon a bad name. But since the group of graduate students around them was so enthusiastic about the prospect of bringing Oberon "to the rest of the world" (Ceres machines existed only at ETH in Zurich), they soon gave the go-ahead for alternative implementations of Oberon.

So some of us graduate students started porting Oberon to other platforms, first writing a compiler for the target architecture, and then re-building the Oberon system on top of the native operating system. And surprisingly, Oberon turned out to be not only highly portable, but (after some fine-tuning of the connections to the respective native operating systems[14]) also quite efficient on other platforms. By late

---

[13] On the contrary, Oberon's *architecture* is sufficiently interesting to perhaps warrant more specific language support of *its* particular mechanisms. The ongoing design of the experimental programming language Lag*oo*na [Fra97] is an attempt to lift some of these mechanisms to the source-language level.

[14] A representative selection of the engineering challenges (and their solutions) encountered when porting the Oberon System to run "on top of" another operating system is presented in [Fra93].

1990, several versions of Oberon were available, for Apple Macintosh computers based on MC68000 series microprocessors, for Sun Microsystems workstations using the SPARC architecture, and for MIPS-based workstations from Digital Equipment Corporation. Eventually, there were also implementations for Intel i86-based personal computers, for IBM's RISC System/6000, and for Hewlett-Packard's HP-RISC architecture[15].

Unfortunately, the cross-platform aspect of the Oberon project has never been fully appreciated by the wider research community. This is unfortunate because Oberon offered a level of cross-platform portability that was unprecedented and with the possible exception of Java has never been achieved again. Excluding a handful of low-level modules, every Oberon source program could be compiled and executed on any of the platforms, and produce exactly the same results. Wirth and Gutknecht realized the importance of portability early on, and after the first Oberon ports were finished and the implementors came back to report incompatibilities and inconsistencies, they went back to refine the Oberon system interface to remove these obstacles to true portability. Quite remarkably, for example, about half of the machines that Oberon ran on used little-endian byte-ordering, while the other half used big-endian addressing, yet this was abstracted away completely at Oberon's application program interface.

Oberon's extraordinary cross-platform capabilities extended not only to the source code, but also to the data formats used within the system. For example, complex documents could be shared freely among platforms, preserving not only all of the formatting across all target platforms, but even supporting the same editor extensions (today we would call them "applets") on every platform. In contrast, there are still commercial software packages today that use different file formats for each of the platforms on which they are available.

7 Machine-Independent Mobile Code

Another domain in which Oberon provided pioneering work was in the area of machine-independent mobile code along with just-in-time code generation. As early as 1991, I had published a paper [FL91] at the annual Modula-2 conference (with Stefan Ludwig, who was doing his Diploma Thesis under my supervision at the time), advocating *dynamic compilation at load-time* from a target-architecture independent, compiler-oriented intermediate representation.

This work eventually led to my dissertation [Fra94a], the central thesis of which was that one could obtain *machine independence effectively for free* if only one could manage to find a sufficiently dense intermediate representation so that the additional computational effort required for just-in-time code generation could be

---

[15] For a more detailed history and implementation background, see Brandis et al. [BCF95].

compensated entirely by reduced I/O overhead due to much smaller "object files". My dissertation proposed such a representation, subsequently christened "Slim Binaries" [FK97].

True to Wirth's spirit of creating *usable* research, this work in the ensuing period matured beyond being simply a research prototype, as would perhaps have been its fate almost everywhere else but ETH. With the help of Thomas Kistler, who later became my own student at UC Irvine, the Slim Binary format was turned into the *main* software distribution format for the two different processor architectures that Apple's Macintosh operating system ran on. This meant that almost all of the Oberon system was dynamically compiled at load-time on these platforms, with the only major exception of the dynamic compiler itself, and the Oberon storage allocator and garbage collector.

As a result, by the end of 1994, the users of MacOberon were routinely employing dynamic just-in-time compilation at load time from a machine-independent representation, although most of them probably simply didn't notice (nor should they have noticed!). At the time, dynamic compilation was largely considered a "cute curiosity" without many practical consequences. Until, of course, in 1995, Java "happened", which had not only the effect of making just-in-time compilation a "hot topic", but whose publicity assault also more or less completely drowned all prior or simultaneous research of a similar nature.

8 Daily Life With Oberon

Oberon never became a household name, but even among those who did learn of Oberon, only a select few realized that it was not just simply a research prototype, but actually the one and only system that almost all members of the Institute for Computer Systems (including the secretary!) used on a daily basis for all their computing tasks. And as a result of the daily use by its designers, Oberon became extraordinarily reliable[16].

I believe that at the time, the members of the Institute for Computer Systems never fully appreciated the reliability of the Oberon System, because they had nothing else to compare it with. But those of us who have since been forced to work with commercial software will look back to our Oberon years as a golden era

[16] True to Wirth's maxim that software designers should be forced to use the products of their labor themselves, each of the Ph.D. students who had ported Oberon onto a new platform subsequently used Oberon on that particular platform as his main (and often sole) work environment. Only recently is this sentiment making a comeback in the software developer community, under the somewhat flippant moniker of "eating one's own dog-food". Hence, after having created the code-generating loader for my machine-independent code-transportation format, I actually wrote my doctoral dissertation using a version of Oberon's document processing software that was generated afresh on-the-fly by dynamic compilation each time that I started it.

without the frequent glitches, unexpected crashes, and data losses that users of other systems are accustomed to.

And the phrase that Oberon was used "for all computing tasks" should be taken literally. For example, almost every scientific article that came out of the Institute for Computer Systems in the 1990's was written using one of Oberon's several document-processing systems. Almost incredibly, even the *fonts* used to type-set these articles on the Oberon-driven laser printer were designed in-house by the resident typographer, H. E. Meier[17]. Similarly, the Institute ran its own network protocols, file servers, print servers, and mail servers, all of which (along with the respective client software) had been developed entirely in-house, and to the greatest extent by N. Wirth himself. I estimate that the Institute spent less than one thousand dollars on commercial software during the whole decade.

Moreover, few people know that Ceres and Oberon not only provided the backbone of all *research* at the Institute for Computer System, but also for most of the *education*. ETH was probably the only university in the Western world that in the 1990's conducted the majority of its undergraduate education in Computer Science using workstation computers (of the Ceres family) built in-house, running an operating system (Oberon) developed in-house, and teaching a programming language (Oberon) created by one of the resident professors. And the resulting education was arguably better than anywhere else, because instead of merely explaining to students *how to use* the available educational computer systems, at ETH the educational system's architects were at hand to explain *the motivation* behind individual design decisions.

On the research side, having a system developed in-house meant independence. Rather than calling a telephone hot-line when a problem occurred, one could simply walk over to the author of the code that caused the problem. And in the resulting dialogue, the problem would usually be identified and corrected on the spot. Alternatively, since the complete source code of the Oberon System was readily available to everyone within the Institute (which is currently a highly fashionable idea), one would correct the error oneself and then inform the original author of the correction so that the master copy of the code was updated. Either way, the code base soon achieved its legendary robustness.

9 Conclusion

The Oberon System was well ahead of its time, anticipating many developments in software that most of the rest of the world is beginning to address only now. It is to be hoped that future researchers study Oberon closely, because in many instances, they will find inspirations that will prevent them from having to re-invent a wheel

[17] The fonts and font design system go back to the earlier Lilith project, but H. E. Meier continued working at the Institute for Computer Systems throughout most of the Oberon project.

here or there. And just like a wheel is superior to more complex alternatives, Oberon's design contains many simple and elegant solutions that are unlikely to ever be surpassed, at any level of complexity.

Dedication

This article is dedicated to N. Wirth, whose radiant example has shaped my professional thinking profoundly. It was his books that I read during high-school that prompted me to choose ETH to study for my undergraduate degree. It was Wirth's classes there that encouraged me to continue onwards for a Ph.D. And finally, it was his example as a Ph.D. advisor and professor that persuaded me to pursue an academic career of my own. Thank you, Niklaus, for leading the way.

References

[ACL98] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth; "Fast, Effective Code Generation in a Just-In-Time Java Compiler", in *Proceedings of the 1998 ACM Sigplan Conference on Programming Language Design and Implementation (PLDI)*, published as *ACM Sigplan Notices*, 33:5, pp.280-290; 1998.

[BCF95] M. Brandis, R. Crelier, M. Franz, and J. Templ; "The Oberon System Family"; *Software–Practice and Experience*, 25:12, pp.1331-1366; 1995.

[Cha82] G. J. Chaitin; "Register Allocation & Spilling via Graph Coloring"; *Proceedings of the ACM Sigplan '82 Symposium on Compiler Construction*, published as *ACM Sigplan Notices*, 17:6, pp.98-105; 1982.

[FK97] M. Franz and T. Kistler; "Slim Binaries"; *Communications of the ACM*, 40:12, pp.87-94; 1997.

[FL91] M. Franz and S. Ludwig; "Portability Redefined"; in *Proceedings of the Second International Modula-2 Conference*, Loughborough, England, pp.216–224; 1991.

[Fra93] M. Franz; "Emulating an Operating System on Top of Another"; *Software–Practice and Experience*, 23:6, pp.677-692; 1993.

[Fra94a] M. Franz; *Code-Generation On-the-Fly: A Key to Portable Software* (Doctoral Dissertation No. 10497, ETH Zürich); Verlag der Fachvereine, Zürich; 1994.

[Fra94b] M. Franz; "Compiler Optimizations Should Pay for Themselves"; in P. Schulthess (Ed.), *Advances in Modular Languages: Proceedings of the Joint Modular Languages Conference*, Universitätsverlag Ulm, Germany, pp.111-121; 1994.

[Fra97] M. Franz; "The Programming Language Lagoona: A Fresh Look at Object-Orientation"; *Software–Concepts and Tools*, 18:1, pp.14–26; 1997.

[Fra98] M. Franz; "On the Architecture of Software Component Systems"; in R. Nigel Horspool (Ed.), *Systems Implementation 2000, (Proceedings of the IFIP TC2 WG2.4 Working Conference on Systems Implementation 2000: Languages, Methods and Tools, Berlin, Germany)*, Chapman & Hall, ISBN 0-412-83530-4, pp. 207–220; 1998.

[GHJ95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides; *Design Patterns–Elements of Reusable Object-Oriented Software*; Addison Wesley;1995.

[Gut94] J. Gutknecht; “Oberon System 3: Vision of a Future Software Technology”; *Software–Concepts and Tools*, 15:1, pp.26-33; 1994.

[KP89] G. E. Krasner and S. T. Pope; “A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-89”; *Journal of Object-Oriented Programming*, 1:3, 26-49; 1989.

[MTG89] H. Mössenböck, J. Templ, and R. Griesemer; *Object Oberon: An Object-Oriented Extension of Oberon*; Report No. 109, Department Informatik, ETH Zurich; 1989.

[MW92] H. Mössenböck and N. Wirth; “The Programming Language Oberon-2”; *Structured Programming*, 12:4; pp.179-195; 1991

[PS94] D. Pountain and C. Szyperski; “Extensible Software Systems”; *BYTE*, 19:5; 1994.

[Szy92a] C. Szyperski; “Write-ing Applications: Designing an Extensible Text Editor as an Application Framework”. In *Proceedings of the 7th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'92)*, Dortmund, Germany; Prentice Hall; 1992.

[Szy92b] C. Szyperski; *Insight Ethos: On Object Orientation in Operating Systems. Clemens Szyperski*. (Doctoral Dissertation No. 9884, ETH Zurich). Verlag der Fachvereine, Zürich; 1992.

[Szy98] C. Szyperski; *Component Software – Beyond Object-Oriented Programming*; Addison-Wesley; 1998.

[WG92] N. Wirth and J. Gutknecht; *Project Oberon: The Design of an Operating System and Compiler*; Addison-Wesley; 1992.

[Wir93] N. Wirth; “Recollections about the Development of Pascal”, in T. J. Bergin and R. G. Gibson, *History of Programming Languages*. (Proceedings of the Second Conference on the History of Programming Languages, Cambridge, Massachusetts, April 1993). Addison-Wesley, pp.97-120; 1996.

[Wir96] N. Wirth; *Theory and Techniques of Compiler Construction*. Addison-Wesley; 1996.