Less is more. Why Oberon beats mainstream in complex applications

F V Tkachov

Institute for Nuclear Research, Russian Academy of Sciences, Moscow 117312 Russia

E-mail: ftkachov@inr.ac.ru

Abstract. The second part of the two-part account of the enabling technologies behind a successful completion of the Troitsk-nu-mass experiment, summarizes an almost 20 years' experience of using Oberon/Component Pascal in complex applications ranging from the cutting-edge computer algebra (b-quark decays etc.) to experimental data processing (neutrino mass etc.) to exploratory algorithm design work (the optimal jet finder etc.) to systematic computer science education (the international project Informatika-21).

1. Introduction

The currently best ν mass bound was published by the Troitsk-ν-mass experiment [1]. This happened after the experiment had been plagued by the so-called "Troitsk anomaly" for about 10 years. The anomaly melted away after a reanalysis that was made possible by two enabling technologies: the statistical method of quasioptimal weights for parameter estimation (see a companion talk [2]) and the Oberon technologies for software development reviewed in the present talk.

In regard of software, everything could in theory have been done with fortran or even C++ but at a much higher cost. A proper *technique* is about minimizing effort while attaining the best result; such a technique is rooted in how well one understands the basic principles of what one is doing. The success of the reanalysis proves soundness of the adopted approach towards software development.

In fact, a complete reanalysis of the Troitsk-ν-mass data [1] was not planned. The initial purpose was only to implement a new statistical method [2] and to play with real data with a view to clarify the treatment of the Poisson background as a test of the new method. The software development tool was chosen to be a derivative of the Oberon System [3] known as the BlackBox Component Builder [4], [5], [6]. It was known from a previous experience with the BEAR computer algebra engine [7] to be easy to use, efficient, safe, flexible and producing a clean compiled code (a combination of features shared by all Oberons). The resulting piece of software proved — as a result of adhering to the principles of the underlying system [3], [8] — to be easy to operate, safely extensible, adaptable and efficient enough that the testing of the new method grew into a complete reanalysis of the data. The successful completion of [1] within the severe manpower and other limitations that the Troitsk-ν-mass team operated under, is proof of the sound rationality behind the minimalistic Oberon as opposed to the prodigal complexity of the mainstream tools.

The software used in [1] was designed to be generic enough so that it could be used as a supplemental material for my university programming course. The course is part of the educational project «Informatika-21» [9]) that promulgates scientific rationality for the professional and general (physicists', linguists' etc.) IT education in the form of a unified backbone of algorithmics/program-

ming courses spanning students of ages 11-20 and based on a single perfectly designed programming platform — Oberon. In the capacity of coordinator of the «Informatika-21» project, I enjoyed the privilege of discussing the matters of software development in general and programming education in particular, with a considerable number of excellent experts (rocket scientists, compiler optimization gurus, UAV designers, school teachers, university professors, etc.; cf. [10]). As a result, my views of the subject have sharpened compared with the earlier talk [5].

2. The IT bubble of excessive complexity

I would like to discuss software development from the point of view of a "non-professional" programmer, i.e. one for whom computer programming is not a full-time job but part of application problem solving. That using a computer inevitably leads to a full-blown programming in complex applications is seen from the fact that all significant software packages develop script languages. Such languages are usually poorly designed, inefficient and unreliable. They also limit access to the low level programming and thus do not allow one to utilize the full power of the computer. The resulting two-tier programming model (an "efficient" system level with the notoriously unsafe C/C++ and a somewhat less unreliable but inefficient application level with a multitude of special purpose languages) represents an unnecessarily complicated environment that prevents non-professional programmers from exploiting their knowledge of the application problem domain to the full. There is, as a result, a vast grey area where programming problems are excessively hard for non-professionals to tackle with the complicated mainstream tools, yet a recruitment of a professional programmer cannot be justified.

The surging complexity of the IT industry tools is visualized in the following figure (based on one borrowed from [12])

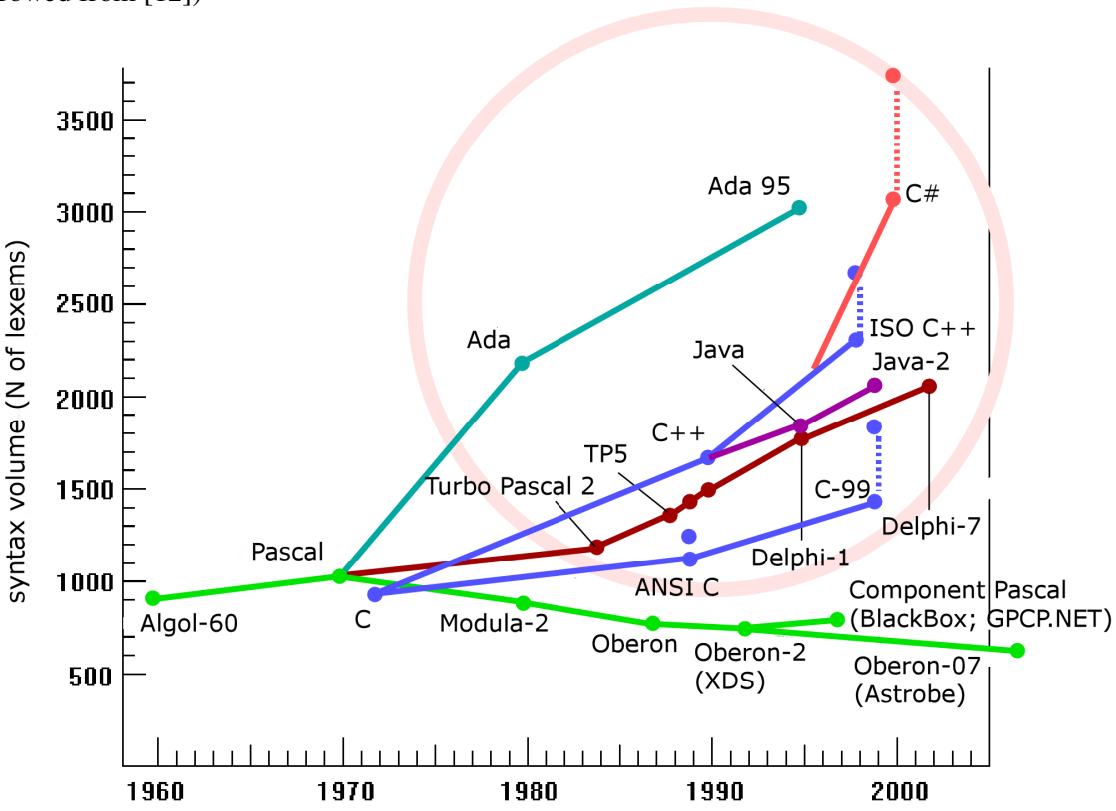

Figure 1. The syntax volume of some programming languages.

The figure is not intended to compare the complexity of various programming languages but only to convey an idea of the IT industry's growing bubble of complexity. The figure is complemented by the so-called Wirth's Law stating that the software gets slower faster than the hardware gets faster.

The only line that goes down in the figure is the one corresponding to the languages designed by Niklaus Wirth and his school at ETH, Zürich, who for many years have been combatting the complexity of programming and programming languages. But let us discuss the bubble first; it is important to understand its nature because, quite obviously, some fundamental forces are at play.

3. Understanding the IT complexity bubble

There are two causes of the excessive IT complexity bubble visualized in Figure 1.

1. The first cause is the **combinatorial nature of normal intellect**. (The adjective normal is chosen to correlate with the normal science of Thomas Kuhn [13].) This was observed in a multitude of experiments conducted by primatologists and zoopsychologists starting from Wolfgang Koeler in the early XX c. This means that the basic mechanism of intellect is to find a (finite) combination of known actions with the objects within view in order to achieve a desired goal. If a combination is not found whereas the goal remains in view, one gets restless and roams around, which brings new objects into the scope of one's attention thus increasing chances of finding the solving combination. Thus, the normal intellect tends to pile up things and objects, to add patches and props to bring the prospective solution closer to the target — a phenomenon well seen e.g. by observing how kids do their programs. They are reluctant to go back to their program in order to simplify it once it seems to do what is required. A habit of cleaning up the mess — which may require a critical ability in order to separate what is important and to be left from what is not and to be discarded — has to be learned either by painful experiences (the Kalashnikov Principle: Excessive complexity is vulnerability) or via a proper training. The parallels with Figure 1 are quite obvious, with an additional complication that the cleaning up of the mess in that case may mean redesigning the language. Such a redesign the industry — together with their customers — are reluctant to do for stated reasons of compatibility, even if a redesign would in the long run be beneficial to all.

A related point worth making here to explain the steep rise of the curves in Figure 1, is that with the IT, the normal/combinatorial intellect has for the first time in the history of Humanity been largely freed from the restrictions of the resistance of materials.

2. The other, social reason is that the excessive complexity creates what economists call **asymmetry of information**. This serves as a source of an extra economic and social rent derived by IT experts (cf. the Nobel Prize for economics awarded to Akerlof, Spence and Stiglitz in 2001 for their analysis of the market of used cars with "the bad driving out the good" in the market). It is not, for example, uncommon to observe a youngster who proudly pontificates about software development and programming educaton in virtue of having simply acquired an amount of knowledge about obscure features of an overly complex mainstream tool. The youngster's less computer savvy companions may indeed start to regard and treat him as a true expert. This example is actually a paradigmatic one for a good part of the IT industry.

Alchemy preceded chemistry and astrology preceded astronomy. In the absence of an established IT education system based on a scientific foundation, there is no reason why the fast expanding IT sphere should be any different. Indeed, the bubble visualized in Figure 1 testifies to the fact that the IT industry passes through a parascientific stage characterized by a paradoxical if implicit assumption that the growing complexity of programming languages and tools might help one solve the key problem of software development — the one of controlling the complexity of software.

4. What is Oberon/Component Pascal

Oberon remains the only systematic research project to have successfully identified a minimal rational core of technologies for efficient software development. It perhaps deserves to be called anti-language due to its contrast with the typically bloated mainstream languages.

Oberon descends from the old Pascal of 1970 via Modula-2 (in fact, Modula-2 might have been described as Pascal-80, and Oberon, as Pascal-86). However, one should beware of myths and wrong associations because Oberon is a fundamentally more powerful language than either Pascal or Modula-2 because of its automatic garbage collection. That feature, however, does not prevent it from being a fast compiled language: the garbage collection is only invoked when dynamic memory allocation is used. Oberon is also a fundamentally safer language (no segviols, ever) because it extends the strict static type safety mechanism to dynamic records.

There is a number of Oberon dialects (cf. the fact that the exact position of an optimum may be hard to pinpoint precisely due to its instability with respect to minor changes of the optimum criteria): the original ETH Oberon [3], Active Oberon [14], Oberon-2 (available as XDS Oberon [15]), Component Pascal (a 1997 refinement of Oberon-2 from Oberon microsystems, Inc. available with the BlackBox Component Builder [4] as well as the Gardens Point Component Pascal compiler for Microsoft.NET [16]), Oberon-07 (with a 2011 revision available from [17]). Most of the above are available in open source. All these languages are very small and for the purposes of the present talk will be referred to as Oberon/Component Pascal.

— Oberon/Component Pascal can be described as a “vanishing” imperative programming language with everything that can be put away into libraries excluded from the language; the remaining language core is designed with utmost care, with unlimited, safe and efficient extensibility in mind.

— The language is very small, its syntax rules fit on one page, and the language report is only about twenty pages long plus or minus a few pages depending on the dialect.

— A very fast one-pass compiler produces a clean compiled code with no compiler-generated overhead which normally requires an optimization to get rid of as is the case e.g. with C++, (Floating point optimization is available via an external tool [15].)

— The language is highly readable in the Pascal tradition, and carefully designed to be robust against typing errors.

— It is statically type safe, including dynamic records.

— It features independently compiled modules that serve as units of information hiding, are dynamically linked and (un)loaded; as a result, the programming system gives one an interactive feel despite being based on a true compiler.

— It is object-oriented via a very efficient basic mechanism of single record extension.

— It is automatically garbage collected, which together with record extensibility makes it a synthetic language with flexibility comparable to that of functional languages (as proved e.g. by the BEAR computer algebra engine [5], [7]).

A popular argument in the discussions of programming languages is that functional languages are higher level than imperative ones. However, farther from hardware does not necessaritly mean closer to human, as is easily seen when one tries to teach kids. In fact, high level application problems are synthetic in nature, combining elements of imperative, functional and markovian (substitutions-based) paradigms. The hardware, however, is imperative, so a synthetic language based on the imperative paradigm, such as Oberon, offers a clear advantage in regard of efficient use of the hardware.

A superior readability and semantic transparency of Oberon are also relevant for the concept of open source. Indeed, access to program sources is essential for its adapation to changing requirements. The most adaptable is the code that is the most accessible. The most accessible code is one written in the simplest language. Ergo, the Oberon code is more open than the C++ code in proportion to how Oberon is simpler and more readable than C++.

5. Why Oberon technologies rather than just a programming language

Every tool — even as simple as a hammer — requires a proper technique to use it efficiently, i.e. to achieve the best results with minimal effort and maximal safety. Tools and techniques come together, and a proper balance is key. (If one is not aware of one's technique, it is probably best described as chiropractic.)

This observation is highly relevant for a complicated tool such as programming language. To make the most of Oberon the language one should use a proper development environment and possess a set of basic algorithmic skills (see below).

An Oberon development environment supports the *text-as-interface* paradigm: any text document can serve as a command prompt, and any text fragment can be designated as program input (similar to, but much more flexible than the command line parameters). Then text documents can serve as flexible, customizable menus, and simultaneously as storage for various sets of parameter values — a feature of great usefulness in applications involving parameter fitting, optimizations, etc.

Lastly, instead of learning defects and peculiarities of a big language, one invests effort into basic algorithmics [18], into learning Dijkstra's method to construct complex loops [19], and basic architectural patterns (Carrier-Rider, separation of interface from implementation, the message bus, etc.).

6. Bridging efficiency and flexibility

A popular conference topic in the 80s was the so-called numerical-analytical interface. The fundamental problem here is that numerical applications that are usually handled with fortran etc. tend to require flexibility of dynamic data structures as more adaptive intellect is built into the algorithms. Such a flexibility, however, used to be only available via the functional languages. On the other hand, large-scale computer algebra tasks, usually handled via dynamic languages, may benefit greatly from compiled routines. This is illustrated by the following figure:

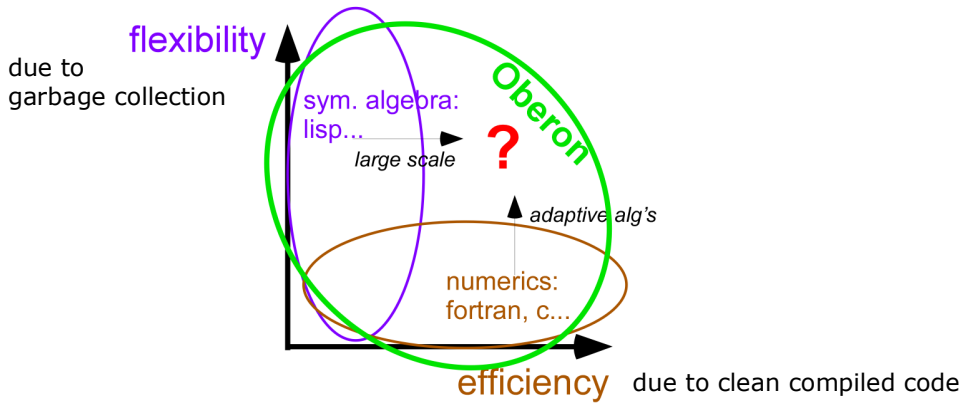

Figure 2. The flexibility *vs* efficiency dilemma.

One small, simple and reliable Oberon covers both bases, and does what is usually achieved via complex (and inherently less reliable) combinations such as C++ and python, or fortran and Mathematica, etc.

The success of Java at the expense of C++ in the late 90s proved the latter's vulnerability due to its complexity — in a complete agreement with the above mentioned Kalashnikov Principle. Yet the high-energy physics community chose C++ as a successor to fortran. Note that by 1992, the Oberon System that never had a segviol had been ported to a dozen platforms, an optimizing XDS compiler was already available, and in 1993 the full Oberon functionality was made available in the popular operating systems. In view of these facts and after 20 years' experience of software development, it is hard not to regard the adoption of C++ as a gross failure of the HEP community as a ***scientific*** community.

7. Conclusions

A central problem of software development is controlling the complexity of software. The problem is partly rooted in the way the human brain operates and is fostered by the asymmetry of information in the absence of a proper programming education system. **Containing the gratuitous growth of complexity must be a permanent concern whenever the information technologies are involved.** Such a containment starts with a proper choice of the programming language and related tools. There is abundant and various experience to support the notion that a minimalistic language designed with utmost care is key to efficient development of reliable software. Such a minimalistic language is bound to resemble Oberon modulo insignificant syntactic variations that are more likely to be defects of design than anything else (cf. the recent Google's Go [20]). Such defects would be least tolerated by non-professional programmers and in the educational context.

References

[1] Aseev V N et al. 2011 *Phys. Rev. D* **84** 112003 (arXiv:1108.5034).
[2] Tkachov F V 2013 *Quasi-optimal weights: a versatile tool of data analysis.* Talk at ACAT 2013, Beijing.
[3] Wirth N and Gutknecht J 1992 *Project Oberon*. Addison-Wesley.
[4] Oberon microsystems, Inc. http://www.oberon.ch/blackbox.html
[5] Tkachov F V 2001 *From novel mathematics to efficient algorithms. Do we have proper SD foundation to build future?* Talk at CPP2001, Japan (arXiv:hep-ph/0202033).
[6] *Workshop Oberon Day at CERN* 2004 http://www.inr.ac.ru/~blackbox/Oberon.Day/
[7] Blokland I et al. 2004 *Phys. Rev. Lett.* **93** 062001 (arXiv:hep-ph/0403221);
Blokland I et al. 2005 *Phys. Rev. D* **71** 054004 (arXiv:hep-ph/0503039);
Blokland I et al. 2005 *Phys. Rev. D* **72** 033014 (arXiv:hep-ph/0506055);
Czarnecki A, Slusarczyk M, Tkachov F V 2006 *Phys. Rev. Lett.* **96** 171803 (arXiv:hep-ph/0511004).
[8] Szyperski C (with Gruntz D and Murer S) 2002 *Component Software – Beyond Object-Oriented Programming* Addison-Wesley / ACM Press.
[9] International educational project «Informatika-21» since 2002 http://www.inr.ac.ru/~info21
[10] *Workshop Oberon Day* 2011 http://www.oberonday2011.ethz.ch/
[11] *Educational project «Informatika-21»* (since 2002) http://www.inr.ac.ru/~info21/ (in Russian)
[12] Sverdlov S Z 2007 *Programming Languages and Compilation Methods* Piter Press (in Russian).
[13] Kuhn Th S 1962 *The Structure of Scientific Revolutions* University of Chicago Press.
[14] Prof. Jurg Gutknecht's group at ETH Zurich: http://nativesystems.inf.ethz.ch/
[15] XDS Oberon (optimizing) http://www.excelsior-usa.com/xdsx86.html
[16] Gardens Point Component Pascal: http://gpcp.codeplex.com/
[17] Oberon-07 for embedded applications: http://www.astrobe.com/
[18] Wirth N 2012 *Algorithms and data structures. Version for Oberon* http://www.inf.ethz.ch/personal/wirth/books/AlgorithmE1/AD2012.pdf
[19] Dijkstra E W 1976 *A Discipline of Programming* Prentice-Hall.
[20] http://golang.org