TOOLS, METHODS AND LANGUAGES FOR
SCIENTIFIC AND ENGINEERING COMPUTATION
B. Ford, J.C. Rault, F. Thomasset (editors)
Elsevier Science Publishers B.V. (North-Holland) / © INRIA, 1984

Software Engineering for Engineering Software

Génie Logiciel et Logiciel pour l'ingénierie

Bertrand Meyer

Electricité de France, Direction des Etudes et Recherches
1 Avenue du Général de Gaulle 92141 Clamart (France)

ABSTRACT

This discussion paper surveys the current state of software for scientific and engineering applications, and the foreseeable evolutions. It argues that a serious change in the attitude of scientists and engineers is necessary in order to master the growth of problem size. The discussion includes aspects of methodology, tools and languages ; we feel that much progress is needed in all three areas. The current actions of the ANSI Fortran committee are seen as particularly obnoxious with respect to the overall goal of providing scientific programmers with adequate tools.

1 - INTRODUCTION

1.1 - Background

This paper can be viewed as one computer scientist's reflections about software for scientific and engineering applications : what it now looks like, how it will evolve (for the better or worse), and what may be done about it. I am grateful to the organizers of this Conference for providing me with an opportunity to express these views.

The ideas expressed below result from an eight-year experience as in-house software engineering specialist in a large scientific computation center ; the author and his group were in programming education, design and implementation of some software tools, procurement of those which were available on the market (some publications which reflect this activity are [4, 7, 8, 10]). Such a situation, that of a computer scientist working in an environment

where computers are considered as mere tools, implies repeated misunderstandings : misunderstandings with users whose official professional interests lie with some application domain other than programming per se, but also with other computer scientists who may look down on what they consider to be uninteresting problems associated with the use of obsolete tools and techniques. As it is often the case with hybrid situations, this one also has its advantages, and we think it has provided us with some insights into the real problems of software engineering.

Needless to say, the views expressed in this paper are those of the author.

1.2 Methods, tools, languages

The view of software engineering which forms the basis for the ideas expressed in this paper implies in particular that three aspects should be given equal weight when discussing software : methods, tools, languages. We think this is more than just a cliché. In many cases there is a tendency to give undue consideration to one or two of these factors. Roughly speaking, one can say that as far as academic research in computer science is concerned, the 60's were the decade of languages (when all discussions about programming would focus on language features), the 70's were the decade of programming methodology, and the current interest in software tools is obvious. Whereas one can make some progress by switching to a better language, or by using adequate design methods with possibly imperfect languages and tools, or by gaining access to powerful tools, we think it is impossible to make any real breakthrough in software quality without advancing in all three areas. The discussion will thus focus on the three aspects.

2 - SCIENTIFIC AND ENGINEERING SOFTWARE, AS IT IS.

We outline below some of the characteristics of scientific and engineering software, as we perceive them and as they distinguish this type of software from others such as business software (accounting, transaction processing and the like), real-time software (command/control etc.), systems software (compilers, operating systems, teleprocessing etc.) or office information systems. These characteristics relate to the form and contents of the programs and to the way they are produced and used.

In the sequel, the term "scientific software" refers to programs developed for scientific and engineering applications. It does not cover basic software tools which may be developed in connection with such programs, e.g. memory management primitives, language preprocessors etc. ; although such tools are often written as part of scientific software projects, their characteristics are rather like those of system software.

2.1 - Language

The most obvious feature of scientific programs is the lan-

guage in which they are written : to an overwhelming majority, they use Fortran. Some competition has come from PL/I and APL, the latter being popular in some circles especially for the implementation of prototypes, "quick and dirty" versions, etc. ; both, however, remain marginal.

Many sites have done some experiments with Pascal in order to assess a fashionable language, but few have used it on actual projects, since most scientific programmers who have tried it deeply resent the lack of features they consider essential. More important here than functions such as exponentiation or direct-access I/O (which can be implemented through external procedures) are project management tools such as separate compilation, which is indeed impossible to implement in the strict Pascal framework (if static type checking and the whole system of types are to be retained). Also, the notion of Fortran-like conformant arrays, now included as an optional feature in the ISO standard, but still seldom available on existing compilers, is essential for scientific computation if useful procedures are to be written.

So Fortran is still king. It should be noted, however, that the world is not so simple as it used to be : Fortran means different things to different people. The Fortran 77 standard has not completely taken over ; in many cases, what is available is still either a compiler based on the 66 standard, usually complemented by machine-dependent extensions, or some hybrid between the 66 and 77 versions. At the same time, some manufacturers are taking (high-risk) bets on the next standard being concocted by ANSI. This results in a rather fuzzy situation ; as S. Feldman had foreseen in his 1976 criticism of the Fortran 77 draft, which is still good reading [6], Fortran undoubtedly gained in many respects by becoming Fortran 77, but it also lost in two of its essential qualities, simplicity and universality.

When talking about Fortran with respect to scientific software, it is impossible not to mention an apparent paradox : in spite of its almost undisputed position as a vehicle for writing numerical software and its pretentions to portability, Fortran does not as yet offer any tool for controlling the numerical accuracy of programs in a portable fashion. The recommendations of IFIP Working Group 2.5 on numerical software did not make their way into the 78 standard ("Fortran 77"), although they are being included in the next standard (but see below). It is interesting to note that the only widely publicized language which does offer machine-independent facilities in this area (following some attempts made in PL/I) is Ada.

2.2 - Program size

Scientific programs vary considerably in size. A typical range is between 5,000 and 50,000 source lines (whether or not one counts comments usually has a marginal influence on the evaluation). There are bigger programs, but they are not so common ; some packages reach 300,000 lines or more, but one seldom hears about sizes comparable to

what is often quoted about e.g. telephone exchange software (500,000 to 1 million or more). Thus much scientific software can be characterized as "medium-size". There are many signs, however, that these figures may be growing steadily. This apparent tendency is likely to bring about much concern regarding the scaling up of the methods used for program writing and project management.

2.3 - Contents

There is still a widely held view that scientific programs are essentially computation-oriented. In our experience, this is inaccurate. Of course, most scientific programs include some non-trivial arithmetic computation. If, however, one looks at the actual code, one frequently finds out that the part which actually performs numerical computation is relatively small in size (if not in execution time), the bulk of the program text being concerned with manipulation of data structures, storage management, input and output, pre- and post-processing, etc. **For a large part, scientific programs are data manipulation programs.** In most cases, this part is growing much faster than the purely numerical one, which is often relatively stabilized ; many developments have to do with improvements in the user interfaces, inclusion of interactive facilities, graphical input and output, uses of data base managements systems, etc.

This aspect of scientific programs should be understood by those who design new machine architectures, programming languages, software tools or methods aimed at this area.

2.4 - Human aspects.

An important feature of scientific software, which distinguishes it quite dramatically from, say, business or systems software, is that a large share of the programs in this area is written by people whose official job title does not include words such as "programmer", "analyst", etc. Rather, they are considered to be specialists in other areas, e.g. engineers, physicists, etc. On the other hand, many of these people are programmers by any objective criterion ; it is hard to decide by what other name one should call a person who has been devoting four-fifths of his time in the past few years to writing, maintaining and modifying programs.

This frequent discrepancy between official classification and actual work has many negative consequences. One is the inevitable dissatisfaction of people who feel computers prevent them from spending their time on their "real" job. Another is the difficulty encountered in properly training people who do not want to engage in a deep study of what they consider to be just a tool.

When discussing the human aspect of scientific programs, it is necessary to mention the overwhelming influence of the Fortran

tradition. It does have some positive aspects, such as the systematic use of separate compilation, which may be considered a good information hiding device (although it also implies breaches in the already weak mechanisms for type checking). These are, however, largely balanced by some very obnoxious characteristics ; the worst two, in our opinion, are the purely static nature of the language (which prompts programmers to develop ad hoc storage management tools, intermixed with the application code) and the lack of data structuring mechanisms (everything must be described using arrays).

2.5 - Programming Methodology.

The appearance of many scientific programs is a sufficient witness of the methods, or lack thereof, used for writing such software and managing the corresponding projects. This is all the more interesting since, in the past fifteen years, researchers in programming methodology have developed techniques which, to some extent, make it possible to apply the criteria of scientific rigor to software development. It looks like the area of scientific software will be one of the last to be influenced by these developments. One is always impressed by the amount of erratic program text which Ph. D's and high-bred engineers are able to produce, and in some cases be proud of.

2.6 - Tools.

The use of programming tools, beyond such standard ones as editors and compilers, is fairly limited in many installations. It is remarkable to see, for example, how often the machine-format dump still plays the role of the basic debugging aid. Here again, the discrepancy in levels of abstraction between the sophistication of the applications and the people who conceive them, on the one hand, and the characteristics of the underlying software, on the other hand, are striking. Also, one can again notice the negative effect of the language : although Fortran is much more primitive by its concepts than, say, Pascal, Lisp or Simula, it is often less amenable to language-dependent tools such as syntax-directed editors, symbolic debuggers etc. because of its baroque features, strange format and irregular structure.

It should be mentioned, however, that one kind of tool oriented towards scientific software can claim a fair amount of success : subroutine libraries. Several numerical libraries now exist which in our opinion count among the best pieces of reusable software ever designed in any application area. That these libraries are still not used as much as they should is a fact to which many installations can testify ; several reasons can be found to this situation, some of which are probably just connected with human laziness, others having deeper technical roots.

Scientific software has also been the prime target for other successful tools : Fortran static (and, to a lesser extent, dynamic)

analyzers. Again, these tools are underused ; it is clear, however (in particular from our own experience with General Research's RXVP) that they can provide a host of services which, although conceptually limited, are extremely useful in connection with the development, acquisition, debugging and documentation of scientific software. Although it is true that some of the checks performed by Fortran static analyzers (e.g. type checking) are only needed because of the language's deficiencies, this is is only part of the picture ; some of the ideas could be profitably adapted to more elaborate languages, which are still lagging behind Fortran with respect to availability of such tools.

3 - HOPES AND FEARS FOR SCIENTIFIC SOFTWARE

We now turn to the future and try to ascertain what will happen of scientific software and of the methods, tools and languages used to develop it.

3.1 - Characteristics of future software

It is likely that the average size and complexity of important scientific programs will grow considerably in the coming years. We see two main reasons for this evolution :

- increased demand for sophisticated user interfaces, graphics processing etc. ;

- new ambitions generated by faster machines (in particular vector and parallel processors) with increased storage, solid-state auxiliary memory, which will trigger requests for developments of programs for problems which were previously considered intractable.

If these predictions come true, the current techniques of software development will not permit to control the added complexity of scientific software. It is well-known that the difficulty of writing a program and managing its development is not a linear function of its size ; much of today's scientific software seems to be just below one of the "complexity barriers" which have been met in other application domains.

A change of attitude on the part of scientists and engineers is necessary if the hopes which they put in computers are to be fulfilleld. It is interesting to note that some of the more lucid members of that community are now realizing, for example [11], that Fortran may not be the last word. Such a change of attitude will be necessary if some of the hopes which scientists and engineers place in computers are to be fulfilled.

3.2 - Methodology

One area in which progress is badly needed is, clearly, programming methodology. We now mention the advances in this area which we consider most relevant to the area under study. We shall list three : abstract data types ; formal specification techniques ; assertion-guided program construction.

Abstract data types are one of those powerful ideas that look so simple once they have been invented. Abstract data types make it possible to describe data structures through their external properties, defined by the operations which outside users (usually programs) may perform on them and the properties of these operations, without any reference to the physical representation of the corresponding objects.

Abstract data types provide very helpful tools for specifying precisely yet abstractly the objects which are manipulated by a program. They are particularly useful as a basis for the modular decomposition of software systems ; they yield modular structures which are often more solid, error- and change-resistant than those obtained with the classical procedural decomposition. This idea forms the basis for the very promising technique of **object-oriented programming**.

Formal specification is the application of mathematical formalism to the description of programs and program objects. This technique, combined with abstract data types, is nothing else than the adaptation of the classical axiomatic method to programming. However, the two can be applied separately. Practicing programmers, even with a higher education in mathematics, are often reluctant to use formal techniques ; they fear the benefit will not be worth the amount of work needed. Only through regular application of these techniques does one begin to appreciate the help they bring with respect to problem understanding, inter-programmer communication, existence of a fixed and unambiguous basis against which the eventual programs can be validated.

Assertion-guided program construction, a field of study based on Dijkstra's work [5], is concerned with methods of program construction which work in a systematic way, starting with the text of the formal specification. Although there have been no reports of systematic application of such methods to the design of large programs, they undoubtedly provide a framework in which it becomes much easier to reason explicitly and precisely about the program design process.

One of the major problems which remain to be solved is, to my opinion, the design of a practical axiomatic system which would make it possible to apply the Hoare-Dijkstra methods for formal reasoning about programs to programs involving floating-point computations. There have been several attempts at this. The kind of questions

involved includes determination of rules for computing assertions such as

$$wp(x > y,\quad x := y + z)$$

where wp (P, A) is the weakest precondition which will ensure validity of P after execution of A.

3.3 - Tools

It is soon realized that in a sufficiently rich software environment, the merits of individual tools become almost less important than the consistency of the various available tools. Consistency here means not only ability to communicate, but also sharing of a common set of calling conventions, user interfaces and general philosophy. Only if these conditions are met does the buzzword "integrated" apply. For this to be the case, a strong unifying concept must be found. One of the most promising areas of research focuses on the idea that a programming language may play this role, at least if viewed with a sufficient level of abstraction, hence the importance of the notion of **abstract syntax**. One key development in systems such as the INRIA's Mentor or Carnegie-Mellon's Gandalf has been to show that, for software environments, **language-based does not mean language-bound** : the language can be a parameter, in such a way that adaptation of a language-based environment to a new language is a relatively simple task. In such a framework, the definition of a language may be considered to include such aspects as coding standards, comment conventions etc.

Another kind of tools which should be of much help for the development of scientific software includes tools which are not as common in this area as they are e.g. in business software, namely what may be called "customizable packages", or "application generators". These are programs which will generate programs adapted to particular problems ; their position lies in-between fixed subroutine libraries (whose rigidity is one of the reasons why they are not universally used, as mentioned above), and general-purpose programming languages (which often provide too much freedom and too many possibilities when applied to a class of similar problems). Program generators are of course an old idea ; however, with the advent of new powerful devices for man-machine interaction, they may take on a new life. Many of the underlying concepts can be studied in the context of language-based environments as seen above.

3.4 - Languages

It is impossible to end this discussion without mentioning the appalling effort which has been going on in ANSI in order to find a successor to Fortran 77 ; the current working name is Fortran 8X. This incredible construction [1] results from adding to the Fortran framework all kinds of incoherent ideas which have occurred in the past five years to various people. We thus find, along with the classical Fortran constructs, such things as new control structures

(but there still isn't a **while** construct), Pascal-like record structures (but no pointers), array manipulation primitives (as part of the language, with a specific syntax), dynamic storage and recursive procedures, etc. Although the language "architecture" was supposed to have been fixed some time ago, new bright ideas seem to come in at every meeting ; for example, the last report [2] includes a proposal (adopted by "straw vote"), which adds to the language a concept called "bundle", which is supposed to implement abstract data types. The report contains minutes of a discussion over the possible conflicts of this notion with ... the doubling of quotes in FORMAT statements ! It is particularly sad to see the amount of time, money and energy which has been spent in many countries over the world to produce a document which looks like a survey of fashionable programming language concepts by a group of high-school students.

An idea of the elegance of the language will be readily obtained by considering the syntax for the primitives which give access to the characteristics of the host machine floating-point number system (probably one of the legitimate additions, given the nature of Fortran usage). A call such as PRECISION (X) returns the number of digits for objects of the type of the actual argument X : i.e. the number of digits for integers if X is a variable of type integer, etc. This quaint convention (which, to our knowledge, is a first in programming languages), is inconsistent with every known notion of parameter passing : what is passed here is neither X's value, nor its address, nor its name, but its type ! Incidentally, this convention makes it impossible for a local computation center to implement these facilities by anticipation without modifying the compiler.

Many lessons have been learned in the past fifteen years on the design of programming languages. In every university course on the subject, it is taught that a language design (as a program design) should be seen not as an accumulation of features, but as a homogeneous and regular construction. This is exactly the contrary of what the 8X proposal curently looks like.

One of the reasons why the current proposal is so bad is probably that few respectable computer scientists today will agree to participate in anything which has the name Fortran. I should like to point out that, whereas such an attitude is understandable, it is still necessary to have a few reasonable people participate in the necessary cleanup of the language. Fortran is one of the very few available languages which come close to achieving true portability. It is almost the realization of the UNCOL myth of the 1950's [9], i.e. a universal assembly language. It has, or had, in D. Barron's words [3], the rugged simplicity of a Ford Model-T. One should not let people add a nose, a tail and wings to it and pretend it has been transformed into a supersonic jet.

4 - CONCLUSION

The ideas expressed in this paper are clearly designed to stir up a fruitful discussion. I hope to have been able to convey ideas to members of two communities :

- to practicing scientific programmers, I hope to have shown that it may be useful to devote some time to the study of some formal aspects of programming in order to eventually spend less time programming, and to have given them some hints about Fortran not being the last word ;

- as for computer scientists, I have tried to emphasize that scientific computation, the oldest application field of computers, still has some interesting challenges to offer.

BIBLIOGRAPHY

[1] ANSI : Proposals approved for Fortran 8X (X3J3 / S6.81) ; March 2, 1981.

[2] ANSI : Minutes of 85th Meeting ; February 7-11, 1983.

[3] D. Barron (Ed.) : Pascal : The Language and its Implementation (Wiley, New York, 1981.

[4] A. Bossavit, B. Meyer : Methods for Vector Programming, in de Bakker and van Vliet (Eds.) : Algorithmic Languages (North-Holland, Amsterdam, 1981).

[5] E.W. Dijkstra : A Discipline of Programming (Prentice-Hall, 1976).

[6] S.I. Feldman : A Fortranner's Lament, SIGPLAN Notices, Dec. 1976, 25-34.

[7] B. Meyer, C. Baudoin : Méthodes de Programmation (Eyrolles, Paris, 1978).

[8] B. Meyer : Principles of Package Design ; Communications ACM, vol. 25, no. 7, July 1982, 419-428

[9] T.B. Steel : Uncol, the Myth and the Fact ; in R. Goodman (Ed.) : Annual Review in Automatic Programming, Vol. 2 (Pergamon Press, New York, 1961), 325-344.

[10] F. Vapné et al : 101 Conseils pour la Programmation en Fortran, EDF Report Atelier Logiciel n. 35, to appear in book form.

[11] K. Wilson : A Program of Computing Support for Physical Research, to appear.