Programming for Behaviour *

Wladyslaw M. Turski
Institute of Informatics
Warsaw University
Banacha 2, 02-097 Warsaw

Abstract. Traditional programming is for computations. Many contemporary applications suffer when squeezed into programming paradigm. This presentation investigates desirable properties of another paradigm: programming for behaviours and explores some technical consequences of a possible formalism to describe such programs.

1 A view of history

For centuries a computer was a person performing strictly prescribed operations on numbers recorded on a sheet of paper. The object of the computation — as far as the computer was concerned — was to evaluate an algebraic formula, perhaps repeatedly, for a number of data sets. In a computation, the major portion of human effort was consumed by acts of executing arithmetical operations. The evolution of computing machinery concentrated on ever faster means of executing the four arithmetical operations.

With the invention of electronic digital circuits the long established proportions between the time needed to execute arithmetical operations and that consumed by human operator in following the prescribed sequence of operations changed dramatically. Further increases in the speed of executing arithmetical operations would be aimless if the control actions were not performed at approximately the same rate.

The concept of programmed computer was a natural and necessary consequence of the spectacular speed-up of arithmetical operations. The observation that programs can be stored just as easily as data and in the same medium (present already in Babbage's designs, but usually attributed to von Neumann) led to the stored-program computers.

With these machines becoming ever more readily available and faster, the appetite for computations soon met another barrier that could not be breached by improving the hardware technology. The programs controlling the order in which arithmetical operations were performed, the machine language programs, were exceedingly time-consuming to prepare. However, this limitation was not quite universal. Most acute in the environments characterised by the need to perform a variety of ever fresh computations, it was hardly at all felt by those who were satisfied with repeated executions of the same programs.

* The work reported here was supported in part by the Polish Research Council under grant KBN-211999101.
At this point in development of programming two different attitudes, indeed two subcultures emerged. One was concerned primarily with program construction, another with program use.

When the use of a program is the main concern, one tends to consider it as a part of machinery which, once installed, is expected to provide a continued service. In other words, a program is treated as (a part of) a product, purchased for its continued utility in a work environment. Its quality is measured with respect to the services it provides, and therefore is subject to empirical evaluation (testing).

When the main concern is with the program-making, one's chief objective is to reduce the effort of program production. This implies two things: (i) inventing and using program-production tools, and (ii) applying such production methods which provide for quality control through design and its implementation rather than by the a posteriori testing of finished products.

At first glance, no serious controversy follows from such separation of concerns. But appearances — at least in this case — are highly misleading.

The best, most productive program-making tools and the most reliable methods of program design and implementation are based on the assumption of inviolability of program specification. There is no proof that there is no other way, but the weight of circumstantial evidence makes this conjecture highly probable. The program makers tend to swear by the principle: the program is no better than its specification.

On the other hand, program users tend to regard a program good if its execution on a computer provides a satisfactory service. Since the notion of satisfaction is related to the current needs which, just as computer configurations, tend to vary in a less than predictable way, the program users put a premium on program flexibility.

The problem is that with the traditional view of a program "flexibility" is hard to specify. The controversy about the value of calculational (formal) methods of programming is a symptom of an essential divergence between the actual use of computers and the traditional paradigm of computing. Formal methods of programming are geared to making programs for computations, where fixed specifications are possible and natural, whereas computers are increasingly used for service purposes, for which the very notion of fixed specifications is ill-fitting.

The traditional paradigm of computing did not consider any interaction between the process of computing and its environment. This is not to say that computations were undertaken for no practical purpose, merely that the process of computation and the application of its outcome were physically separate. The necessary link was often established by various tables produced by computers and used in an application domain, such as science and commerce, but also in some control applications, such as navigation and artillery.

To facilitate the field use of tables many ingenious techniques were applied, both analytic and computational. Some of them were so closely associated with a particular problem that they became inseparable from it. Even today, when the availability of the outline computing technology vitiates any need for such techniques, we are following them because we were taught that they constitute a part of the solution or, in extreme cases, of the problem itself. A prime example of such habits, formerly necessary and — despite their present technical irrelevance — faithfully maintained, is the presentation (and measurement) of physical process variables at regular time intervals, rather than at occurrences of significant events (as, e.g., when we cook an