From the IEE “Computing and Control Engineering” journal, February,2003.

A Modern Real-Time Software Design Tool:

Applying Lessons from Leo

By Ferdinand Wagner & Peter Wolstenholme

Summary:

The special CCEJ issue on REAL TIME presented many new initiatives that are currently being developed. This short article introduces a software design tool that is currently being used on a variety of real time projects and which is based on principles similar to those employed 50 years ago on the LEO computer system. The Leo programming process was very effective in producing reliable software. A modern version of this, called StateWORKS, is based on abstract, finite state-machine models. In contrast to top-level modelling tools such as UML, which leave some coding to be done in the final stages, StateWORKS is an efficient method of implementing the final software and of avoiding much of the coding process.

1. Bread-boarding.

50 years ago, many designs were "bread-boarded" (built on a large, flat and very accessible panel on the bench) and the engineer would fiddle until his project seemed to function. This could take a long time, but the worst aspect was that the end product might not function under some marginal conditions. Testing is not a good substitute for meticulous design.

The modern programming process seems to be, to write the programs, test them, and fiddle until they work. The successive versions made during the software design process are not seen by third parties, and leave almost no trace behind. The "design" of software by dozens of iterations came into its own when batch programming was replaced by on-line terminals, and has continued to be common practice into the era of the superb, modern I.D.E.s .

But, like breadboarding, it is still wrong in many circumstances. It may be an acceptable way of developing the program for a self-contained, and purely numeric, algorithm, where the testing can be reasonably complete, but it falls down when the software will need to function in complex and ever-changing circumstances, subject to external errors. Software for most embedded control systems clearly comes into the latter category, and one might imagine that, in view of the common crashes and malfunctions, even software for the benevolent environment of the desk-top PC is working in a more complex environment than foreseen at the design stage.

Edsger Dijkstra has stated that one can never hope to make software reliable by testing the final implementation [1]. We concur. We are convinced that lessons from the hardware side of engineering apply to software and that the good software designer needs to know **why** his software will be correct. Code written in modern languages such as C++ can be quite impenetrable and unreadable, when it is trying to cope with very complex situations. There is still a software crisis, and initiatives such as Extreme Programming are, in our view, an expression of despair.

In the meantime, hardware bread-boarding has been largely replaced by high-level design methods and simulation software: for example, integrated-circuit designers now seem able to succeed with very, very few design iterations. IC complexity, following Moore's Law, has been able to rise at about 57 % per annum, hardware designer productivity at perhaps 20 % per annum, and software designer productivity at, perhaps, 5 % per annum, which is probably a generous estimate.

2. Programming for Leo Computers.

In the IEE Review for September 2001, a paper [2] described the programming process for the Leo computer. To quote three extracts:

"A firm discipline was established to ensure program correctness. There was clear and complete documentation of what the system was to do: clear enough to be understood by the laymen who were the customers and complete enough to provide a prescriptive definition for the programmers."

"The whole process was gone through whenever there was a change. The specification, the flow charts and the coding sheets were all brought into line."

" …….the fact is that the amount of computer time spent on getting programs completely right was very, very, low by comparison with present norms. "

So where did we go wrong? We submit that the fault lies in our obsession with better ways of coding, involving more and more complex programming languages. These are well suited to the purposes for which they were invented: scientific computations (FORTRAN onwards) and business data processing (COBOL onwards) which take place in rather closed and benevolent environments - although even this is changing. But they are not suited to dealing with very intricate and complex behaviour of software.

The quotations above give strong hints about two possible approaches to a solution.

Firstly, the program specification needs to be sufficiently simple to be discussed between the laymen and the experts. If it is embodied in "code" this can not happen, as even a programmer can never see all the intricacies, and check for behaviour in obscure circumstances. A model of the process, at an abstract level, is

required. To meet this requirement, UML goes some way but is far too complex, as it tries to be a high level programming language. The behavioural aspects of the software can nevertheless be isolated from the various numerical calculations, and expressed in very straightforward finite-state-machine (FSM) terms. A hierarchy of a number of FSMs will be needed as a rule. Although these are a trifle technical, a program designer can use them to take his customer, or his project manager, though the behaviour of the system, and respond to searching questions, by reference to state transition diagrams, and with help from a simulator.

The second hint is that a full specification, which includes a "prescriptive definition" of the final software and leaves no freedom to the programmer, would imply that the final implementation might be done quite automatically, avoiding conventional code generation, and not by a human being. This possibility has been generally neglected, despite the great advances in computing over the last half century.

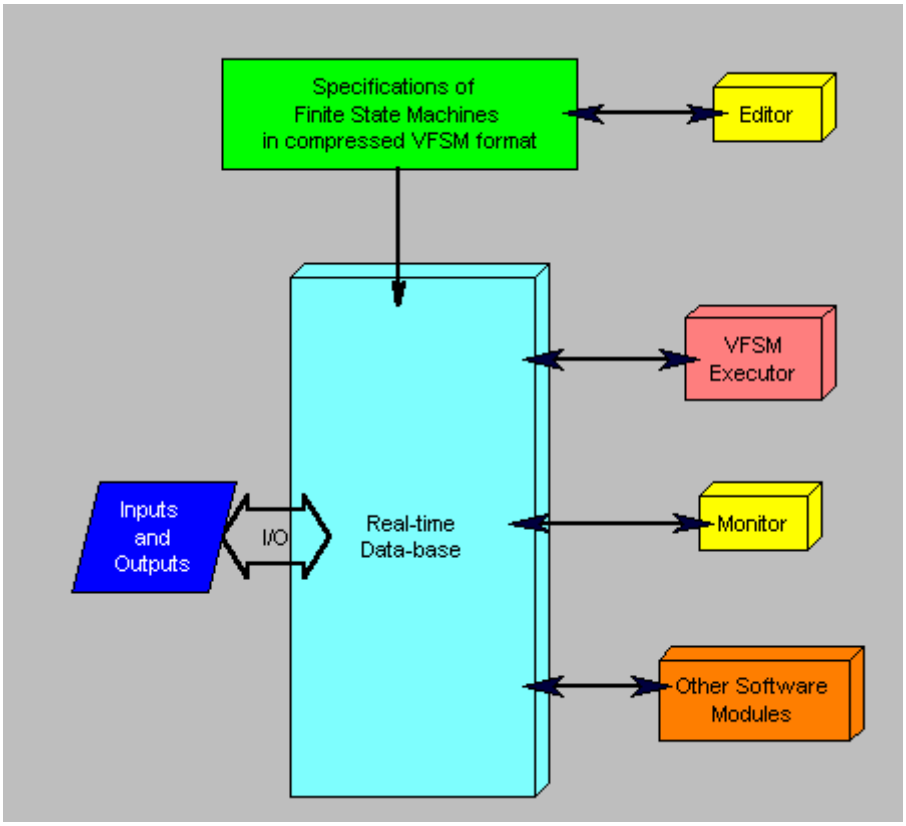
3. StateWORKS

The tool set known as "StateWORKS" employs the approaches outlined above, namely to express software behavioural aspects as FSMs and then to implement the software by executing the FSM specifications directly.

Associated with the basic concept is an input-output processor, which can deal with the various digital and analogue signals used in real life, and present the binary "assertions" which control each FSM. We use the term "assertion" because a real-life input might not be binary: a valve might be open or closed, but its state might also be in transition or even unknown. So the FSM transition expressions use a "positive-logic algebra" where the NOT operator is forbidden, being unclear in its significance.

Over the decade or more since this programming method was conceived [3], it has been used in a number of fields including telecommunications switching [4], process control, and specialised instrumentation. The input-output processor software has evolved into a real-time data-base which provides all the communications between the FSMs and the outside world, as well as those between FSMs, and with other software packages in the system. The "VFSM Executor" program, which directly executes the FSM specifications, is able to handle large numbers, in the hundreds, of FSMs for such real-life projects.

Some past experiences with modelling software processes as finite state machines have been discouraging, and have given the FSM technique a bad reputation in some quarters. This is because a complex process must be represented by a correspondingly complex FSM, which becomes as hard to understand as program code would be. By breaking the system up, into a number of FSMs, arranged in a well-structured hierarchy, as well as by isolating the behavioural aspects from the data processing routines, it becomes possible to manage this complexity, and to avoid "state explosion" difficulties.



At a given instant, each FSM is in one specific state, and in fact the overall system is itself in one state, which is the set of all the states of the individual FSMs, at that instant. Of course, this strategy of considering a complex system in terms of simpler parts, so as to be able to understand it, is well known in all fields of engineering, including software design ("Structured Programming" for example, or "Object Orientation"). Regrettably, in the software field it is still usual to produce code, which is hard to read and to understand, but which is finally the only true expression of how the system will behave.

The StateWORKS method replicates the good features of the Leo software design method, including exact correspondence between the abstract FSM model and the final implementation, as there is, at least in the critical regions, no conventional code to "tweak". The models are very straightforward and can be simulated and verified. Productivity increases of between 40% and 300% have been reported, but there is also the less tangible benefit of higher reliability in service. Using conventional methods, involving extensive testing, there is a high risk of not testing for the more obscure situations: life is too short. Using an abstract specification, it is much easier to take these into consideration from the start of the design. The software development process starts to resemble the design process in other fields of engineering, at last, and it becomes conceivable to manage it.

Further details can be found at the Web site: www.stateworks.com . Feedback would be most welcome, as this topic ought to be discussed widely, in both theoretical and practical aspects.

References:

1. Edsger W. Dijkstra: "On the Cruelty of Really Teaching Computer Science" Comm. ACM, Vol. 22 No. 32 , December 1989, pp. 1398-1404.

2. D. Caminer: "Putting Computers to Work", IEE Review, September, 2001, pp. 27-29.

3. F. Wagner: "VFSM Executable Specification", Proc. Int'l. Conf. on Computer System and Software Engineering, The Hague, Netherlands, 1992, pp. 226-231.

4. A. Flora-Holmquist et al, "The Virtual Finite State Machine Design and Implementation Paradigm": Bell Labs Technical Journal, Winter, 1997, pp. 96-113. http://www.lucent.com/minds/techjournal/pdf/winter_97/paper08.pdf

---

Note: - *not in the published paper* –
More papers about Leo can be found in the July-September, 2000 issue of the IEEE "Annals of the History of Computing" which is available on-line.