F. Wagner, P. Wolstenholme August 2005

StateWORKS - specifying control software instead of coding

Why are CASE Tools condemned to fail?

The goal of each software development is to build from an imprecisely defined task an error-free functioning application. The "usual procedure" has therefore always the same characteristic features: first the specification is analyzed for some time, in order to get the formal release of programming. Afterwards a phase of programming marked by innumerable iterations begins. The first presentations of the results provoke reactions, which lead to more exact definitions of the design aim. Constant minor changes to the requirements often require major changes to the software. This happens in particular, if it should turn out that in the context of the already-created architecture new or changed requirements cannot be realized. This procedure always leads to the incalculability of the completion date and unreliability of the application: the usual case in software development.

This unsatisfactory situation has caused software manufacturers to revise the way in which software specifications are handled, in an attempt to improve the development process. The necessary methods and tools have been known for approximately 30 years under the generic name CASE Tools and have culminated in UML. CASE Tools propose methods which allow a task be specified and results to be generated to facilitate coding. Usually graphic editors are offered, with which beautiful diagrams are drawn. Formal methods, which one obviously needs during the conversion of verbal descriptions to the diagrams, are touched upon but far too briefly. The CASE tools produce documents, which contains some drawings and accompanying text. It is better to begin the programming work with such introductory documents than with unofficial notes on some scraps of paper.

Even so, many programmers struggle against investing time into such activities. They believe that they might learn, during the elaboration of a slightly more formal specification, to understand the application better; however they are afraid that the investment is too high. One can have some sympathy with this view, although it is clear that in some cases it is only offered as a rationalisation of deeply-held personal prejudices. Sometimes the true reason for the refusal is that programmers like coding, which is fun for them. Obviously there is no fun in working on a specification. A further reason is that the verification and a search for logical errors as well as a consistency test of such a specification are difficult tasks. The validity of the specification document decreases rapidly as the coding proceeds. The many changes, which became necessary after starting the programming phase, flow rarely into the specification, which loses thereby its function as a precise specification of the software.

In order to improve the usefulness of CASE Tools, one strives to convert the results of the specification automatically into the final code. This seems to be however an insurmountable task, contradictory in itself. If we succeeded to develop a specification language, with which one could describe a task **completely**, the specification could be converted directly into the machine code. This means nothing else than that a programming language of a new generation would have been invented, which would not be comparable with today's object-oriented languages. We are still far away from this ideal. For the moment code mock-ups are being produced, with which programming is to begin. In simple cases some finished functions are also generated, which must be merged into the software somehow. The alleged reduction of programming effort through automatically generated files, which are to be used as a development basis, causes more problems than assistance. For example the additional expenditure which must be invested, in order to maintain the files for Reverse Engineering is annoying. This functions anyway only partly and in the course of the development the program diverges from the specification.

It should not be forgotten also that a specification method and the associated tools must be learned. If this investment resembles learning a programming language, the reluctance of the programmers is

to be understood, if they must master two completely different languages in order to do the one task twice over.

The sad experience of many enterprises showed that all efforts to create support with CASE Tools during the software development have supplied no convincing results. In special situations, where one made a large expenditure on CASE Tools, in particular on UML, certain results can be shown. If however one looks behind the scenes, the alleged results turn out to be some theoretical formalities, which are signs of (unfulfilled) hopes but do not reveal the reality. In everyday life it is "business as usual" and the expensive specification tools vegetate on a shelf, in order to arouse a good impression on an innocent visitor or on the managers who spent the money for them.

The essential reason for the modest results of all CASE Tools lies in the necessity to “specify” for the second time in a standard programming language the already, laboriously specified software.

Prerequisites for an genuinely executable specification

The solution to the problems described above lies in creating an executable specification which does not require an intermediate code, which must be changed or completed by hand. As stated above, it seems impossible nowadays to define such a strong specification language. Hence, another way must be formulated. The other way must be based on a ready-made run time system which can **implement** a specification. The result of the specification is then no longer program code in some intermediate stage of completion, but rather data, which describe the application. The data are interpreted by the run time system.

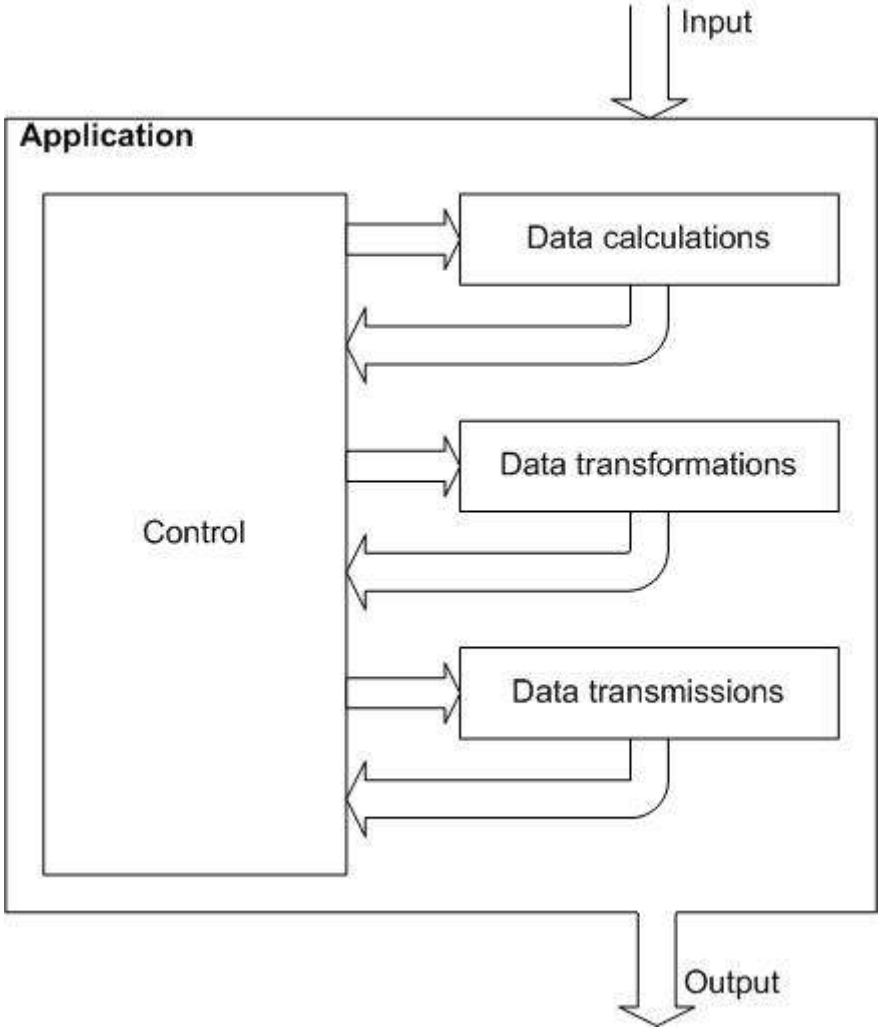

Figure 1 Control and Data flow in the software

A solution available today supplies: a method, specialised development tools and a run time system. The implementation of the idea is based on the Virtual finite State Machine (VFSM) concept.

Software is a sequence of activities which are triggered by externally and internally generated events. Therefore one can represent software as two separate data flows as in Figure 1, where the data flow (activities: computations, data transformation, data transmission, operator interface, etc.) is managed by the control flow. The control flow represents or directs the behaviour of the application. The VFSM method uses a finite state machine model of the sequential activities, in order to define the behaviour. A complete specification of the control flow can be only achieved, if all control-relevant information is covered. This is solved in VFSM by definition of the control values for each kind of data. In order to be able to use the control values, a positive logical algebra was defined. The StateWORKS system is the practical implementation of the VFSM method, i.e. StateWORKS is both: a development and an execution environment. In this note we do not describe the method; a reader may find that information in other publications, also on the web site www.stateworks.com. We show only by an example, how simple the development tools make it to completely specify an application. With a single state machine only small tasks can be solved. Practical applications are specified as systems of several, or perhaps very large numbers of, state machines. The VFSM method offers here a Master-Slave interface between state machines, which permits to build systems of any size that are however at all times controllable.

Specifying a control problem with StateWORKS

A state machine controls equipment, process, measuring procedure, etc. within an application. The behaviour of the state machine is defined with the help of a state transition diagram; Figure 2 shows an example[1].

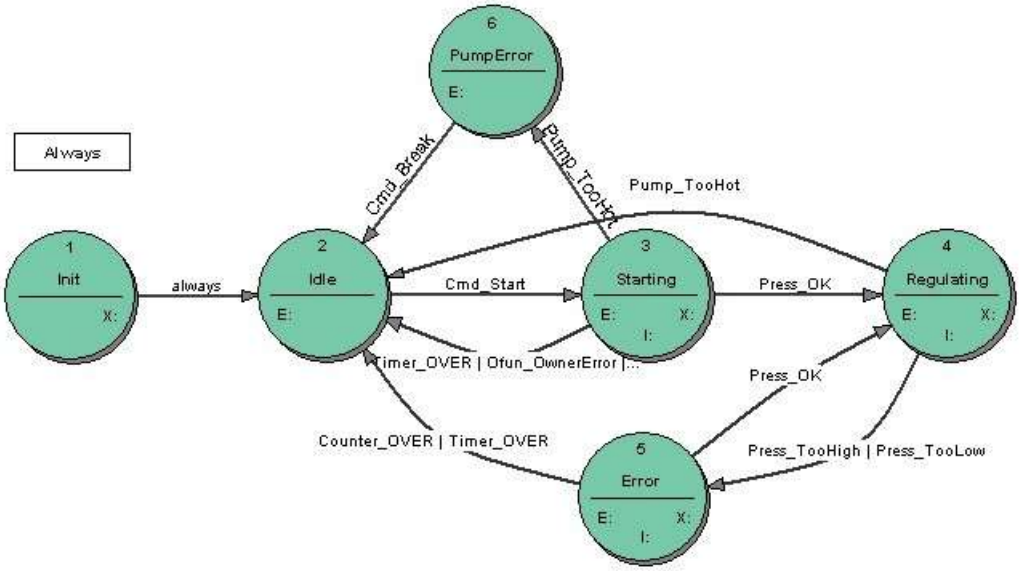

Figure 2 StateWORKS: Representation of a state transition diagram

This representation is understandable and therefore often used, but it conveys only a limited part of the specification: the states, the state transitions and the state transition conditions. For a complete state presentation state transition tables are used; an example is shown in Figure 3. Additionally to the information, which is shown already in the state transition diagram, the table contains the full description of the state machine's activities: setting output values, starting timers, generating alarms, etc.

An application needs several state machines. Those state machines function not as separate control

1 The diagrams and table are taken from the book “Modeling Software with Finite State Machines: A Practical Approach” to be soon published.

units but form a coupled control system. The preferred form of such a system is a hierarchy, as is shown in the example in Figure 4. The diagram shows a Main state machine, acting as a master and three Slaves: Pressure1, Pressure2 and Device1 (whereby Pressure1 and Pressure2 are instances of the same state machine, having common behaviour specifications). Additionally the Diagram contains units, which define the interface to peripheral devices (DI8:01, DO8:01, NI4:01, NO4:01) and the application-specific output functions (OfuLimit:01, OfuLimit:02). The output functions are software interface components that permit to merge application-specific data processing modules into the StateWORKS run time system. The shown diagrams and tables are produced by StateWORKS Studio. StateWORKS Studio has editors to specify individual state machines and a system of state machines too. Beyond that StateWORKS Studio contain a simulator to test the specified application.

Several activities are initiated. Waiting for Pressure acknowledgements. Due to a Timer missing acknowledgement leads to return to the Idle state. Too hot pump leads to the PumpError state. Both erroneous situation generate corresponding alarms. Error by accessing the output function returns the state machine also to the state Idle: it does not make sense to supervise the pressure without having proper pressure limits (corresponding alarms are generated in Always table).

Starting	Entry action	MyCmd_Clear SetPressure_Set Counter_ResetStart Timer_ResetStart Ofun_CalcLimit
	eXit action	Timer_Stop
	RequiredPress_CHANGED	Timer_ResetStart
	Timer_OVER	Al_PressureError
PumpError	Pump_TooHot	
Idle	Timer_OVER | Ofun_OwnerError | Ofun_ParameterError	
Regulating	Press_OK	

Figure 3 StateWORKS: Representation of a state transition table

In order to arrange testing in an effective and comfortable fashion, various monitors are made available. The monitors offer all feasible functions:

- Debugging of state machines and all objects involved
- Logging of alarms
- Automation of the test runs by command files

The application is inevitably documented at each instant: all diagrams, tables, comments can be printed or exported into other documents; a complete XML representation of the specification is likewise available. These documents are always up-to-date.

And now the most important feature: with the creation of the specification the development is terminated, if the appropriate I/O interfaces are present. The results in the form of files, which contain the structure of the system and the behaviour of all state machines in the system, can be

directly used by the StateWORKS run time system.

The StateWORKS run time system is application-independent standard software, which was tested for years in many different applications and with a variety of operating systems. The still possibly missing part - the user interface - is always application dependent and must be manufactured for each application. In order to support and facilitate this task, the StateWORKS run time system has a TCP/IP interface. A static and/or dynamic library is used, in order to realise communication with the run time system. This is relatively simply to manage, since the run time system is based on a real time data base (RTDB), which serves as a TCP/IP server. Clients communicate with the objects of the data base, whereby communication from view of a client contains reading and writing as well as receipt of events, which are sent automatically to registered clients.

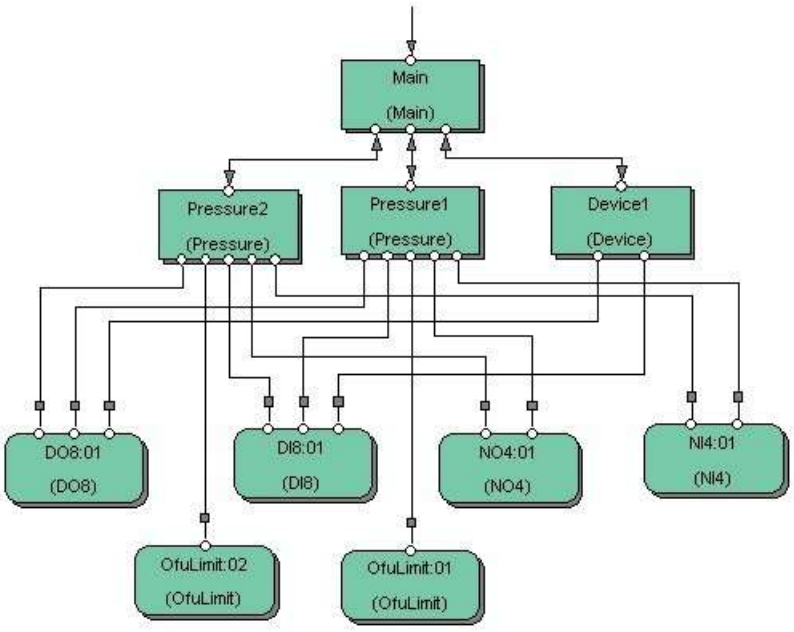

Figure 4 StateWORKS: Representation of a system of state machines

Conclusions

StateWORKS is thus a system, which realizes the idea of an executable specification. Since it does not use an intermediate code, the specification must be complete and any changes and removal of errors (which can only be logical, rather than coding errors) can be done only in the specification.