Modeling and Building Reliable, Re-useable Software

Ferdinand Wagner
Free-lance consultant
f.h.wagner@t-online.de

Peter Wolstenholme
Consultant: CYDON Technology
p.wolstenholme@computer.org

Abstract

Agile Software practices place great emphasis on coding, yet coding is time-consuming, difficult, and the source of many errors. The paper describes a way in which the specification and implementation processes can be unified, and much coding avoided as regards the behavioural aspects of the software. It shares much in common with Agile Methods, yet permits a significant degree of modeling to take place. This VFSM technique and its commercial implementation StateWORKS has been used for several years in a variety of projects, large and small, in industrial control and in telecommunications. It gives significant benefits in time-to-market, in reduced maintenance, and in accuracy of the final project documentation. It facilitates software re-use and system up-grading. It has potential to link up the "Executable UML" and "Agile Modeling" initiatives, to their mutual benefit.

1. Background

1.1 Desktop Packages Compared with Embedded System Software

Much software now in use is developed for PC and work-station applications, and, considering its complexity, is often quite reliable. This is achieved through very arduous testing by the production team, followed by much more testing by users, of beta versions, and then, after the issue of the third major release, the product becomes stable and reliable, although rarely perfect. Another factor helping to ease the development of such software is the existence of a huge software library, or A.P.I. library, holding functions which are well defined, and rather easily tested by their developers, and providing a possibility for massive re-use of such software components in a variety of projects.

Taking a look at the situation for software of embedded systems, which must control complex prcesses which are strongly influenced by the external environment, we see a less favourable situation. Although many tested A.P.I's will be available, for operator interfaces, input-output processing, signal processing and the like, the essential process will require software to be written from scratch. There will be no standard A.P.I. for "Run a blast furnace".

Such software will need to express the behaviour required of the system, and this will normally become very complex when all the various errors and problems which could arise are taken into account. It is usual to write it in a language such as C++ or C, and this code becomes very hard to read, for modification or for maintenance purposes: however, the code becomes the only expression of the system behaviour, and contains details which were never part of the initial written specifications. It is very hard to see what will happen in unusual, but possible, circumstances, and tests have to be arranged, but there is rarely enough time to do all possible tests.

Most software for embedded systems is in fact a prototype, which could never be tested as intensively as mass-market software, but the users expect high reliability. Even safety-critical software is often developed by companies using fewer resources than are available to mass production software firms. A more secure development process is becoming essential, although overdue.

1.1. Dangers of Coding

It is becoming clear that special coding for a new project is a dangerous activity, and that it is very hard to manage the development process. The code will be intrinsically unreliable, and often causes project delays while it is being fixed.

It is the purpose of this paper to outline an approach which eliminates some of the problems, by avoiding certain parts of coding, and imposing a strict discipline on the software structure. The techniques described below have been applied to a variety of projects over more than a decade, and are not limited to the implementation of software for embedded systems. Essentially, rather than coding this area of software, we express it in a formal and abstract way, as a specification, and then execute a fixed program to process the specification. Because the fixed program has been in use, without change, for some years, it is totally reliable, and it acts on data which **is** the specification in a condensed and specialised format, for efficiency.

Although this idea is not new, the way in which it is applied in combination with other features provides a very powerful tool for implementing software for complex systems.

2. Major Features.

2.1. Separation of Control Flow from Data

We consider that much complexity arises from the intermingling of computations of various sorts with the control statements, as this makes it difficult to retain awareness of all the assumptions implied in one part of the code, but which will affect other sections. The control flow, to which we give the classical meaning, namely the majority of statements of the nature of " if..then..else, do…while, switch….case..." defines the behaviour of the software. We extract all of this, separating it from all the numerical calculations or algorithms, and express it in a finite state machine form. (For those unfamiliar with this concept a short explanation is given in Appendix A.)

2.2. Finite State Machines

Although finite state machines (FSM) have long been considered a useful tool for the programmer, it is still common today to see allegations that, when applied to real-life projects, they become too unwieldy to program, and too complex to understand [1, 2]. This view is not correct, but certain techniques need to be employed if they are to be used successfully. Essentially, a software project, as regards the control flow, needs to be split into a number of FSMs, suitably inter-linked.

Several writers have tried to make this clear. David Harel, with StateCharts, suggests using a hierarchical structure, with inheritance [3]. We advise the use of a number of FSM arranged in a hierarchy, but do not insist on this in all cases. We dislike the "nesting" idea, where a state can contain a complete, subsidiary FSM, but rather suggest a flat arrangement where each FSM has its own, rather clean definitions. The basic rule is that a lower-level FSM presents its state to the upper levels, and the upper levels can pass commands to the lower levels: rather like an army. (There are some ways of getting round the rules, but they should be employed with discretion.) All upper-level machines have access to states of lower-level machines, and we prefer not to speak of inheritance in this context. Let it suffice to point out that, at any given instant, the state of the entire system is expressed as the set of all the states of the individual FSMs composing it.

A good textbook explanation of FSM applications is given in Ref. [4] , and the author provides quite a wide range of examples of FSM application, both singly and with a hierarchical structure.

We use a very simple and classical text-book form of the FSM [5], limited to a set of states, transitions, and associated actions (Figure 1) and similar to that proposed for finite state machines in Executable UML [6]. In the transition diagram the states are the vertices of a graph, and the transitions may be arranged as needed. Several transitions from any specific state to one other are allowed. Situations where the same transition might occur for several rather different reasons are catered for by allowing a complex logic expression to govern each transition.

2.3. Virtual Inputs

For practical and theoretical reasons, the expressions which regulate the transition conditions of each FSM are expressed in a special form of binary algebra, which we call positive-logic algebra. Each term is an assertion of some aspect of the system outside the FSM, and is called a "Virtual Input" because it is derived from a real input signal which may be digital, analog, a timer, an event etc. Various terms may be combined in a binary expression, using AND and OR operators only. (AND takes precedence over OR.) The NOT operator is forbidden. This is because an absence of an assertion, such as "Valve_Open", does not necessarily have a meaning such as "Valve_Closed" as there could be some other input state such as "Valve moving" or "Valve_Position_Unknown". Many quantities are analog, and need to be transformed into virtual inputs such as "Pressure_Low, Pressure_Good, Pressure_High".

Although it is obvious that the NOT operator will be hard to use in situations where the value has three or more meaningful states, we prohibit its use even when the input is clearly binary, so that the transition conditions are always made up of positive assertions of various sorts. The programmer is, naturally, quite free in the way he wishes to generate and use them. By use of the positive-logic algebra, the software for enumeration of all the expressions becomes very efficient indeed.

Another way of regarding the Virtual Inputs is to consider them as a virtual lower layer, presenting an FSM-style interface to the real FSM's we are to design. The Virtual Input to the entire system is a set of input states, each under the control of some specialised input/output software described below. The states are visible to the upper layers, for use in the transition expressions. For example, a motorised valve might have five possible states such as OPEN, CLOSED, OPENING, CLOSING, FAULT. A NOT operator, applied to any of these states, is rather meaningless as it could imply any one of the other four states.

On account of the use of Virtual Inputs, the techniques were given the name "Virtual Finite State Machine" (VFSM).

2.4. Actions

At various stages in the operation of the software, the FSM must act on other parts of the system, by issuing commands, setting new parameters, sending signals, starting timers etc. We use the term "Action" for such stimuli, and there are three basic categories (Figure 1):

- Entry actions occur at entry to a given state, and correspond to the classical Mealy automata in FSM theory.
- Exit actions occur at the exit from a state: these are slightly less commonly used.
- Input Actions are actions which, although associated with some form of input condition or signal, do not necessarily cause a state change, but need to cause an output of some nature. These correspond to the Moore model of automaton.

It must be stressed that these actions are not associated with specific transitions but with states.

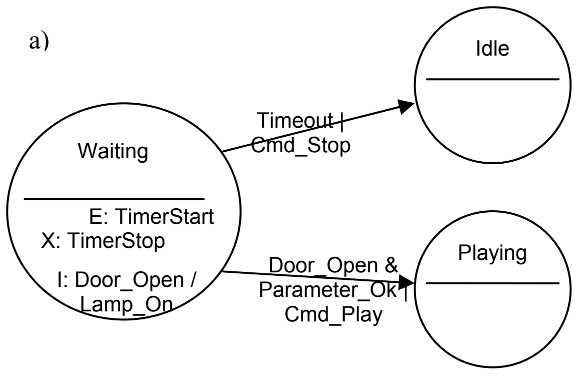
The designer will need to establish an "Output Name" for each action he requires in the system, and the FSM operations will cause "Virtual Outputs" to be generated. These are handled elsewhere.

2.5. Direct Execution without Coding

In design of the various FSMs in his system, the programmer will need to specify all the Virtual Inputs, using Input Names. He will have prepared all the transition expressions, and defined actions to be performed or invoked at various stages in the process. All this work is performed using a specialised editor, with a graphical presentation of a state-transition diagram coupled to a tabular presentation of all the fine detail. At the conclusion of this process, a file is generated, expressing the complete structure in a compressed format. This is called the "Control Specification".

At any time during the process, the designer may run some simulations and other tests, in order to see how the system will behave in various conditions. Such simulations run on the same data - the Control Specification - as will be used in the target application. When the designer is happy, and has perhaps also shown the running process to his supervisor or to his customer, the Control Specification is loaded to the target system, together with the program which we call the VFSM Executor. Some form of input-output software package is also needed, and this is described below.

a)



b)

Waiting	E:	Timer_Start
	C:	
	X:	Timer_Stop
	Door_Open	Lamp_On
Idle	Timeout | Cmd_Stop	
Playing	Door_Open & Parameter_Ok | Cmd_Play	

Figure 1. State specification: graphical (a) and tabular (b) forms

It is important to realise that no run-time code is compiled in this process. In a sense, no new software is generated, and we see software re-use pushed to an extreme. (A similar approach is described in [7]). Of course, one might consider the Control Specification to be a high-level language, but it is not the usual procedural language, as it may have many concurrent activities rather than expressing a single thread, and indeed a major system might have some hundreds of FSMs in operation, all implemented by means of the single VFSM Executor, and existing at the same time.

The Control Specification is an expression of a very few, basic concepts such as states, virtual inputs, and transitions, and it is extremely efficiently implemented by the VFSM Executor. Many thousands of state transitions per second can be handled in a practical system, using a modern c.p.u., depending on what other functions need to be dealt with at that time, of course. The VFSM concept can be easily applied to multiple-processor systems, using a LAN.

This technique goes far beyond the conventional idea of setting out a state-transition table and implementing FSMs by means of an interpreter, as a state-transition table of classical structure can not easily cope with very complex transition expressions and there is a risk that these will have to be implemented and documented elsewhere. The Virtual Input concept is the essential key to both making the execution process efficient, and showing adequate detail about transition conditions at the abstract level of the state machine design.

Because there is no compilation of code, all revisions have to be made at the abstract level of the specifications, which means that these specifications really express all the details and can not be out of date.

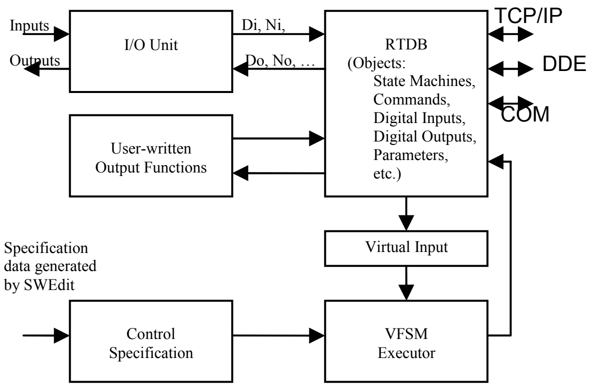


Figure 2: StateWORKS Execution Environment

2.6. Treatment of Inputs

The VFSM executor runs in what we term a "virtual environment" whereby the inputs which reach each FSM are not the real, varied and sometimes complicated items found in the system, but have been transformed into virtual inputs, as described in section 2.3 above. (Similar considerations apply to the outputs.)

At any time, the *Virtual Input* contains the complete information about the control conditions in the system, and it is in fact a set. For instance, the virtual input *{DI_LOW , TI_STOP}* means that the digital input has the control value LOW, the timer has been stopped and the control values of other inputs are unknown.

We introduce a control property for each object and its usage in the following way:

Each object has a control value which is a number. Each object has its definition of the control value that describes the calculation algorithm and range. To make it more understandable we use names for control values instead of numbers. The name of the control value is the object control property.

An example might be a measurement of the level of liquid in a tank. Rather than using the numerical value of the measurement, such as gallons, we establish “switch points” from which we derive control properties such as “LEVEL_LOW, LEVEL_OK, LEVEL_HIGH” which are available to supply the virtual input. By this means, the specific details of inputs and outputs, which might require much more complex treatment than implied in the above examples, such as parsing messages, are isolated from the VFSM design work. This approach has a number of advantages:

1. Simplification of the VFSM design work by removal of extraneous considerations.

2. Easy alteration of parameters if a design change is required at a later stage.

3. Easy alteration of parameters if required during operation of a process.

4. Isolation of input signal processing, which allows a great deal of software re-use when many similar functions are encountered in a system.

5. The VFSM editor (Figure 3) is made aware of all the control properties, input names, output functions etc. which have been set up, and can check for consistency of the entire design.

6. System behavior can be examined before much specialized programming work starts, by just assuming that the virtual environment can be specified at the start but will be implemented in detail at a later stage.

7. Apparent simplification of a complex project, allowing it to be handled when previous approaches have failed. (Obviously, it will still be complex and require a good deal of design work !)

There is therefore a need to generate all the control properties from real-world inputs such as analog-to-digital converters, timers, counters, digital sensors, events, commands, messages etc. and a considerable set of routines for this has been developed over the years. Originally, these were assembled into a package known as the I/O Unit, but in recent years a more versatile approach has been developed, using a Real Time Data Base (RTDB) and the entire product is commercialised under the name of StateWORKS (Figure 2). It is written in C++.

2.7. The RTDB

The RTDB centralises all information regarding states, signals, inputs, outputs and messages. It holds the Control Specification and all Object Properties. In Microsoft Windows NT/2000 systems it acts as a DDE or a COM server, so that other software in the system may interact with the StateWORKS components. Versions for operating systems based on UNIX, such as Linux, Sun Solaris or VxWorks are provided with similar facilities, using POSIX or operating system specific calls as a rule. TCP/IP communication is supported. The RTDB holds the linkages between the physical input/output signals and the virtual environment: these are set up using the StateWORKS editor, and in many instances standard library items can be taken, and merely configured.

User interface software, and local or remote monitoring and debugging features, are easily connected to the RTDB. A wide range of standard API functions is provided, so that in some extreme cases a highly complex control system may be developed with almost no classical coding. A comprehensive API manual is supplied. New functions may be easily added as required. The RTDB contains a variety of standard object types with the control values shown in Table 1.

There are two types of objects in the RTDB: input and output objects. Input objects can influence system behaviour. and they have control values. Output objects have no influence on control and therefore they do not have control values.

Table 1: RTDB Objects

Object	Description	Value	Control value
VFSM	State machine	State (Number > 0, User defined Name)	State (Number > 0, User defined Name)
CMD	Command	Command (Number > 0, User defined Name)	Command (Number > 0, User defined Name)
TI	Timer	Counter state	OVER, OVERSTOP, RESET, RUN, STOP
CNT, ECNT	Counter Event counter	Counter state	OVER, OVERSTOP, RESET, RUN, STOP
UDC	Up/Down counter	Counter state	CHANGED, DEF, INIT
SWIP	Switchpoint	Object to be supervised	HIGH, IN, LOW, OFF
XDA	Memory	Number > 0	Number > 0
OFUN	Output function	Number	Number
DI	Digital input	Boolean value	LOW, HIGH
NI, DAT, PAR	Numerical input, Data, Parameter	Value	CHANGED, DEF, INIT
DO	Digital output	-	-
NO	Numerical output	-	-
AL	Alarm	-	-
TAB	Table	-	-
UNIT	I/O Unit	-	-

3. The Development Process

3.1 General Strategy

The development process for StateWORKS is, to some extent, a matter of style and of experience, and there can be several reasonably good ways of designing any given system. There are some aspects of the process which are different from common practice in high-level modelling. Although any design needs first to be studied at the overall level, we suggest following the rule "top-down design: bottom-up implementation" which held for coding. Thus, the initial overview identifies some functional elements which should be implemented separately before combining into the whole.

In a typical embedded-system project, one then starts with the lowest level of FSMs, directly controlling or monitoring the hardware. Then the next level up can be designed, to issue commands to the bottom level and to monitor its activities. The complete design proceeds in this way, and at frequent intervals the FSMs are tested, by a simulator, and their behaviour in all possible situations examined, and indeed discussed with other members of the project team whenever a relevant issue arises. As this process continues, it will very often be found that the initially-planned structure has to be revised, because the complete system specifications which were the starting-point of the project were not complete in all respects. Readers familiar with the concepts of Agile Software Development will recognize some of them in the above description.

Of course, other FSM structures may be needed, and we do not wish to impose a strict hierarchy for all situations. Several groups of FSMs might be required in a project, with no strong inter-communication between them. Another problem might arise in knowing where and how to start, if there is no clear base level: this was encountered recently in the implementation of a very complex communications protocol, where the designer just had to start somewhere, and work through the system until he arrived at a good solution, using five FSMs in fact.

The design process outlined above can be completed before many of the algorithms which will eventually be needed have been developed, as it avoids handling data. Testing and design reviews can then be performed in the virtual environment, until the designer is confident that he has an effective solution. Following this, the various RTDB objects need to be finalised, and remaining functions programmed in the usual way. In practice, there will be some degree of iteration as detail design issues can impact on the FSM structure, to a limited extent.

3.2. Comparison with Other Methods

We are in agreement with proponents of Statecharts and UML to the extent that we believe that software development methods must move towards automatic implementation of abstract specifications, and away from coding. But in practice, such methods have a major problem, in that when one starts with a high-level project specification, and works gradually downwards, the final result can never be complete. As a rule, top-down design does not work for control flow: we have to start with a vague idea about the main (upper) state machine and do the detailed specification of the lower level state machines. Then we get some feeling for the intermediate levels of state machines and work through these until we reach the main state machine.

It is quite impossible to foresee, from the start, all the small details which will have a critical influence on the way in which the final code will work. The end result is that, after going as far as possible with, say, UML, one is obliged to start coding, and this coding diverges from the beautiful UML model one created. Finally, only the code represents the true specification, and it needs to be understood whenever changes are required, or software has to be re-used in a new generation of hardware. Theoretically, reverse engineering should assure that the code and specification are synchronized but: firstly, reverse engineering never works reliably and secondly, which is much more important, we are presented with the same problem as with coding. Once a programmer starts coding, there is no way to force him to take care about documentation, specification, and similar non-vital activities.

Advocates of Agile Methods [8], such as Extreme Programming have understood this problem, so they advise just getting started with coding, because only it will express the final specifications. There are two main problems with XP. The first is that modern programming languages are gaining complication faster than they are gaining power, so that code behavior is almost impossible to understand, in all but the more straightforward circumstances. The second, perhaps explained most clearly by the late Edsger Dijkstra, is that we can never hope to produce reliable software by means of a testing and patching process, without help from some other processes. A good feature of XP is the way in which user stories are employed, to generate tests, and are built up gradually into a complete test suite, but we must emphasize that these tests will need to cover all possible fault paths, and not just the desired scenarios.

StateWORKS avoids the difficulties described above by steering a middle way. A StateWORKS user simultaneously defines abstract models and implements them, generating data rather than code. Furthermore, the

StateWORKS FSM model is very basic and easy to learn, so that programmers need not "lose" a lot of time learning complex UML tools which might not in the end save much effort.

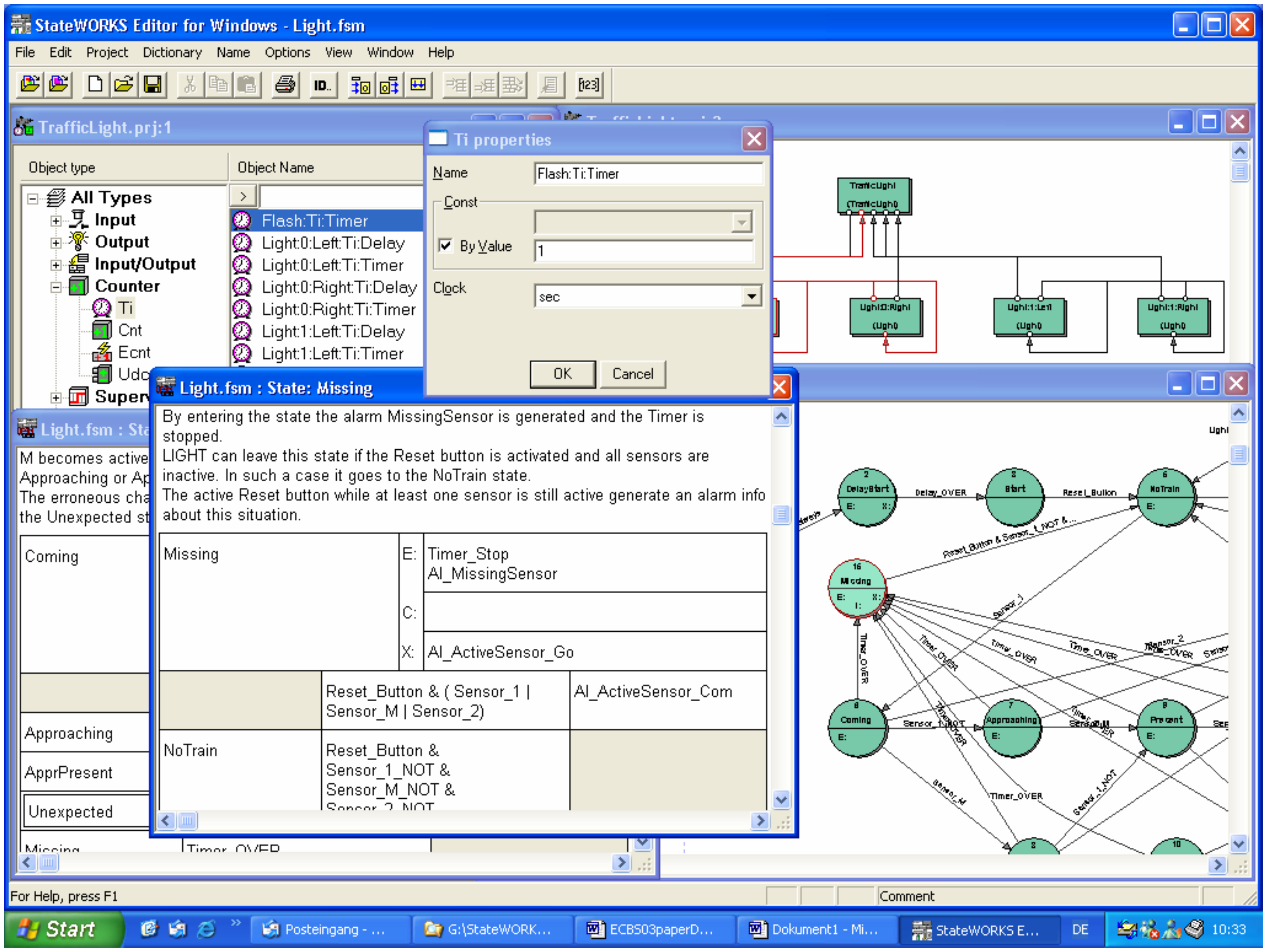
The StateWORKS designer, using FSM transition diagrams and the associated textual data, is in fact working on the implementation, which is not coded of course, but corresponds at the same time to the real behavioural specification - what the software will really do. He is able to design at a more abstract level than code. Devotees of Agile Methods can employ StateWORKS as one powerful tool among several in their arsenal, in the same way that they might use a State-Machine Compiler (SMC) [8 - Pages 429 to 431].

We believe that StateWORKS can fulfil many of the requirements of Executable UML [6], in a way which can also utilise the essentials of the Agile Programming working practices to good effect, while avoiding the need to generate complex program code. (We would, however, prefer transitions to be governed by more generalised inputs than the evanescent signals proposed by Mellor & Balcer.)

3.3. FSM Creation

The FSM design process, carried out using the StateWORKS editor (Figure 3), involves defining and naming each FSM and creating states, which are named and then fully specified. A graphical editor is usually preferred, but a tabular text presentation is also developed in parallel and this holds the complete FSM definitions. States are added to the transition diagram, and transition vectors drawn between them. The StateWORKS I.D.E. provides a context-aware editor, which can display the pure FSM transition diagram in graphical form and also bring up detail in text form, as for instance all the detailed actions associated with a state, as well as the transition conditions. The editor is aware of all the defined objects which might be invoked, and simultaneously links FSMs to the RTDB. It is able to deal with multiple FSMs in a project, and with the various communication mechanisms provided.

Figure 3. StateWORKS editor



Flash:Ti:Timer
Light:0:Left:Ti:Delay
Light:0:Left:Ti:Timer
Light:0:Right:Ti:Delay
Light:0:Right:Ti:Timer
Light:1:Left:Ti:Delay
Light:1:Left:Ti:Timer

Name: Flash:Ti:Timer
Const: [✓ By Value] 1
Clock: sec

By entering the state the alarm MissingSensor is generated and the Timer is stopped.
LIGHT can leave this state if the Reset button is activated and all sensors are inactive. In such a case it goes to the NoTrain state.
The active Reset button while at least one sensor is still active generate an alarm info about this situation.

Missing	E:	Timer_Stop AI_MissingSensor
	C:	
	X:	AI_ActiveSensor_Go
	Reset_Button & (Sensor_1 | Sensor_M | Sensor_2)	AI_ActiveSensor_Com
NoTrain	Reset_Button & Sensor_1_NOT & Sensor_M_NOT & Sensor_2_NOT	

M becomes active
Approaching or Ap
The erroneous cha
the Unexpected st

Coming
Approaching
ApprPresent
Unexpected
Missing Timer_OVER

Concurrently with the FSM design, as all the transition conditions are developed, the designer will need to give a formal name to each term in a condition: these are chosen to be meaningful, are held in an Input Name list by the editor, and can be used in other expressions. I/O objects are held in the RTDB, in fact, and they have a defined structure, with an I/O Object ID and a set of Values. A timer, for example, as an input, has values corresponding to RESET, STOP, RUN, OVER and OVERSTOP, while it can receive outputs for Reset, Stop, Start etc. which can be actions from an FSM state.

For each state entry actions, exit actions and input actions may be defined, in a similar way, by adding Output Names to a list. These details are not normally shown on the transition diagram, as they would make it too complex to read, but are instantly available when any state is selected.

For most of the inputs and outputs which one might use, there exist already-defined objects for the RTDB. This can be used at two skill levels, in that a complete ready-compiled RTDB with all the common objects is available, and there is also a StateWORKS RTDB Class Library for use by skilled programmers with special requirements.

We should point out that the availability of many standard RTDB objects implies a high degree of software re-use for many input-output functions, so saving much time for the programming team.

As the design proceeds, the corresponding Control Specification is created, and the RTDB populated. These may be tested at various stages, either in the development environment or in a target system. A basic simulation capability is offered by a tool called SWLab, and this will often be adequate, but more complex simulations might require some ancillary software to be written, perhaps in Visual Basic if the I.D.E. is hosted in a Microsoft Windows NT/2000/XP system. It is common to test each FSM design against the perceived requirements, using "User Stories" or 'Use Cases" to generate the tests, after a first visual check – a sort of mental simulation. Frequently the visual run-through of various cases is sufficient to show up errors in the design, including the effects of various faults. Running tests, simulations and reviews will frequently cause the entire structure to be revised, several times, during development.

As problems arise in the implementation, the designer might wish to show the FSM transition diagram, and indeed the full data, to colleagues, and discuss what should occur in special cases. The presentation is relatively easy to comprehend, as there are very few symbols to be learned and the designer can show his colleagues what is happening even if they do not fully comprehend the FSM techniques.

3.4 The Run-Time System

When the designer has created the full system, written any special software that is needed, and tested and simulated all he can, the results can be loaded to the target system and run there. There is no compilation or equivalent process required for the StateWORKS part of the project, and the run-time system executes the specifications directly.

4. StateWORKS in Practice

StateWORKS concepts have been applied to a wide variety of projects, over a 12-year period [9, 10]. These have ranged from industrial controls and specialised measurement systems to telecommunications switching and protocol handling. The projects have not been trivial: the first implementation of the VFSM principle was for a semiconductor production line, using mini-computers, in 1988-1990. It is in wide use at Lucent for their international telecom switching products [11, 12] and has been shown to reduce development cost and also improve quality.

The full StateWORKS system with the RTDB has been in use for about five years, and has been very successful wherever applied. A typical project undertaken was where a team of 10 programming staff had great difficulty in coping with maintenance of many versions of a basic product, and new projects were becoming hard to deal with. A grass-roots project was commenced, with the aim of replacing all the existing software by new versions based on StateWORKS. This took 18 months, as there were special safety requirements, a need for a sophisticated user interface, and specialised computations for the data acquired, but at the end a team of 5 persons was able to deal with all needs, and the software was considered the best package in that particular market segment.

Because the StateWORKS tools impose a certain structure on the software, it has sometimes been possible to successfully and rapidly complete a project which had been though to be impossible, on account of its apparent complexity. If a complex project is hard to comprehend, then coding in the classical way, with any available language, will not make it easier to understand, and failure can result.

At the time of writing, StateWORKS is being introduced into two major multi-national companies. A list of past projects may be seen at the StateWORKS web site [13].

Because the StateWORKS concepts are very simple and basic, the programmer need not invest too much time in learning them, and can start applying them very quickly. This contrasts with some more complex tools, which require a major investment to master, and then produce results which are not always felt to justify that effort.

5. Further Work.

The already very extensive StateWORKS Class Library, which is the foundation for the RTDB, has new functions added from time to time, and this process will continue.

A certain number of improvements are being made to the Editor and other aspects of the I.D.E.

Work is already quite advanced on defining a standard implementation-independent XML data type description, which will simplify the passing of FSM specifications between various tools, including UML, and we are looking for a way to start discussions of this topic.

6. Conclusions

The concepts presented in this paper have resulted in a new modelling and implementation paradigm for control systems. Although the designs use a conventional system behavior description in the form of state machines, the implementations are very different in comparison with typical control software. The software control flow is not programmed any more, but it is implemented by a universal execution unit that executes the control specification in the form of binary or text files. The specification model is prepared and tested by means of development tools including editors, validators, simulators and monitors [13].

StateWORKS offers a very high degree of software re-use, and of avoidance of coding. This can yield great increases in productivity, which we shall need if great advances in the complexity of the hardware we can now build at low cost (cf. Moores Law for IC density) are to be fully utilised. Companies offering a range of similar products may re-use many of their FSM designs with confidence, and can adapt existing designs to new target hardware without the need to analyse complex code written for the previous generation.

Why is it so effective? The main reason is surely that the software is constructed in simple units, and the FSM model we employ forces each FSM in particular to conform strictly to the definition of its interfaces. Similar concepts have been used for many years in other branches of engineering.

A major problem encountered in most initiatives to reduce or automate coding is the diversity of input/output signals in embedded systems: these normally require laborious programming. The StateWORKS concepts of the "virtual environment" and of "positive-logic algebra" permit the software behaviour to be fully and rigorously specified while isolating the input/output particularities from that specification, thus reducing the perceived complexity of the software.

Though the technical concept has been tested and proven in many applications the introduction of these new development and execution tools has not been an easy matter. The main problem for programmers seems to be the acceptance of tools that are not freely programmed and which limit their task to the preparation of a formal specification. A true software engineer should be glad to avoid the coding of the control flow [14].

7. Acknowledgements

The idea was born many years ago. To prove its usefulness and to reach a stage of working development and execution tools required the involvement of many others. R. Schmuki has had the most influence on the proper definitions of object properties. Due to his support and work we have now working tools. A. Flora-Holmquist, J. Dobrowolski, D.Smith and N. Chaffee were the driving persons at AT&T and made the introduction of the concepts into the telecommunication sphere a success. To them, as well as many other persons who helped us during innumerable discussions, we should like to express our acknowledgments. We are also indebted to the anonymous reviewers for their helpful suggestions and comments.

8. References

[1] Edward A. Lee: "Embedded Software - An Agenda for Research", *UCB ERL Memorandum M99/63* (December 1999). (Available on-line at eecs.berkeley.edu/publications.)

[2] Aki Nimura: "SCC-II Microsequencer", Xilinx "*XCELL Journal*" issue 42, Spring 2002, page 72. (Available on-line at www.xilinx.com/literature.)

[3] David Harel: "Executable Object Modeling with Statecharts", IEEE "*Computer*" July 1997, pp. 31-42.

[4] Miro Samek, "*Practical Statecharts in C/C++*" CMP Books, 2002.

[5] J. Caroll, D. Long, *Theory of Finite Automata with an Introduction to Formal Languages*. Prentice Hall, Englewoods Cliffs, N.J., 1989.

[6] Stephen J. Mellor and Marc J. Balcer, *Executable UML*. Addison-Wesley, 2002.

[7] Shige Wang and Kang G. Shin: "An Architecture for Embedded Software Integration Using Reusable Components", *Proc. International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, ACM, November 2000.

[8] Robert C. Martin, *Agile Software Development*. Prentice-Hall, 2002.

[9] F. Wagner, "VFSM Executable Specification" *Proceedings of the International Conference on Computer System and Software Engineering*. The Hague, Netherlands, 1992, pp.226-231.

[10] F. Wagner, *The Virtual Finite State Machine: Executable Control Flow Specification*. Rosa Fischer-Löw Verlag, Gießen, 1994.

[11] A. R. Flora-Holmquist, J.D. O'Grady, M.G. Staskauskas, "Telecommunications Software Design Using Virtual Finite-State Machines", *Proceedings of the International Switching Symposium (ISS ,95)*. Berlin, Germany, 1995, pp. 103-107.

[12] A. R. Flora-Holmquist, E. Morton, J.D. O'Grady, M.G. Staskauskas, "The Virtual Finite-State Machine Design and Implementation Paradigm". *Bell Labs Technical Journal*. Winter 1997, pp. 96-113. (Available on-line at www.lucent.com/minds/ - search for VFSM).

[13] For further information and references, see http://www.stateworks.com.

[14] Terri Maginnis, "Engineers Don't Build," *IEEE Software*, Jan.-Feb.2000, pp.34-39.

Appendix A: Finite State Machines (FSM).

A finite state machine, sometimes called a finite automaton, is a system whose condition depends not just on external stimuli, but on the history of those stimuli.

A very simple example is a keyboard, which might be in the normal, initial state, or the caps-lock state, depending on the number of times the Caps Lock key has been pressed. Although programmers are often introduced to FSMs in the context of parsing input text for compilers, the concept is very much more general, and applies to most "reactive systems" in which internal processes are governed or influenced by external events. In such systems the FSM does not merely run through a sequence, producing an end result, but it normally operates throughout the period when the system is able to function.

The academic definition of an FSM is a "quintuple" A = <Σ, S, S0, δ, F> where Σ is an alphabet, S is a finite, non-empty set of states, S0 is a set of initial states,

$\delta: S \times \Sigma \rightarrow \rho(s)$ is a transition function, and F is the set of accepting states (perhaps empty). This is perhaps not too helpful to the practitioner, but quite an amount of theory can be found in the various text-books if he is mathematically inclined. A key point is, however, the input alphabet Σ, which defines the stimuli to which the FSM will react, and this is discussed in some detail in the body of the paper.

A weak point of the above definition is the absence of actions. One might think that the task of the state machine is to change states until it reaches an end state Fn, and there is a class of state machines called "deterministic" which need to do this. (For example, parsing text and reporting when specific sequences are detected.) The true task of the generalised state machine is to trigger actions according to situations defined by the present state and stimuli.

FSMs are normally described in a diagrammatic form, using a circle to represent each state, and lines with arrow heads to represent transitions between the various states. The addition of detail explaining what will provoke any transition is often difficult to achieve, and a text description of the FSM is then needed.

As the FSM functions, changing state from time to time, it will provoke actions in other parts of the system, as required for the specific project.

An FSM will often seem to be very easy to design, and will require a modest number of states - say about a dozen - to perform its task. Then, when the designer considers what might go wrong in various ways with the external system, he is forced to add many states to handle these errors, and the whole FSM becomes very hard to understand or deal with. This is referred to as "state explosion"[1] and there is a solution: split the FSM into several different FSMs which are linked together, and where each one deals with a part of the problem. In very large systems, one will find that many of the error-handling processes are almost identical, and this can save time in the development phase by permitting re-use of some FSM designs.

Although the essential concepts of the FSM are quite easy to learn, their use to design complex systems will necessarily require some experience, and even talent, if effective and elegant solutions are to be created.

[1] The term "state explosion" is also used in validation techniques of protocols, effectively state machines (see for instance *Design and Validation of Computer Protocols* by Gerald J. Holzmann) but it has there a different meaning.