Closing the Gap Between Software Modelling and Code

F. Wagner
Free-lance consultant
ferdinand.wagner@stateworks.com

T. Wagner
Free-lance consultant
thomas.wagner@stateworks.com

P. Wolstenholme
CYDON Technology
p.wolstenholme@computer.org

Abstract

If a software implementation is to be generated fully automatically from a model, then the model must be detailed and totally complete. For the definition of software implementing system behaviour, through finite state machines, we propose a well-proven method for the creation of such models and an associated XML expression of them.

1. Modelling

Construction of software by means of modelling techniques is gaining much attention, particularly with the release of Version 2 of the Unified Modelling Language (UML 2) and various initiatives in Model-Driven Architecture (MDA). UML is designed as a top-down modelling method, which permits large complex processes and software products to be described in a standard way and explored by simulation. It has always been hoped that it could lead to the automatic production of software: in fact the automatic generation of program code.

The most difficult area of software for embedded systems, for telecommunications, and for "reactive systems" in general is to ensure that the behaviour of the software in all circumstances is well controlled, including the presence of unusual combinations of external stimuli and of errors. This area is best expressed in terms of finite state machines (FSM), which are much easier to design, understand, and discuss than would be the corresponding program code.

UML tools from some vendors make serious attempts to generate code from such FSM models expressed as state charts, but there are major difficulties. Unless all the intimate details are in some way defined in the model, the code can not be complete, and in the typical case only "header files" or some equivalent code skeleton can be produced. This is inherent in the UML top-down approach to design. Speaking about UML 1, Cris Kobryn of Telelogic states that "UML 1 was very inept at taking a large architecture and chunking it down into sub-systems and components" [1].

There is hope that UML evolution will improve the situation, but some serious difficulties remain.

2. Agile Programming.

Proponents of agile methods believe that only the final code can represent the real software, as a model can never contain sufficient detail to express exactly what the code would do. So they tend to discount the UML process as too heavy and not sufficiently effective.

The authors of this paper agree, to some extent, with this view. An initial design of a process and of a corresponding software system must be done, in terms of a high-level model. The final implementation needs to be constructed in a bottom-up sense, where basic, simple functions are used to make more complex structures, working up until the complete project is implemented. This is common practice in software, where a large library of functions is available to developers, and they write code procedures or services around these, testing at each stage, and gradually building up the entire structure. In established software development environments the software is normally not written from scratch but rather in a frame applicable for standard applications. Commonly-used development tools (I.D.E., compiler etc.) offer ready-made frameworks for software development.

Unfortunately, for many applications such as embedded systems, there is in general no equivalent library. Also the software development frameworks are less standard. Some help may be found from the concept of “software patterns” but these only assist re-use to a limited extent.

3. Totally Complete Models

If we postulate that a fully-working, well-designed and quite bug-free code could be a good expression of the process as defined by the requirements, such code must express all the information required to run the application. Could a model contain all the information implied by that code, but in some other form? Only if we could construct such a totally complete model, could we expect to generate the exactly-corresponding code from it.

It seems most unlikely that UML can be used to generate such models, without introducing some new ideas.

So what ideas can be put forward? Over the past 13 years, work has been progressing on a suitable format for expressing behaviour of complex software as systems of FSMs. This has culminated in the development of a tool which we call StateWORKS [2][3], described in section 5 below and which indeed facilitates the creation of totally complete models, from which software is automatically generated.

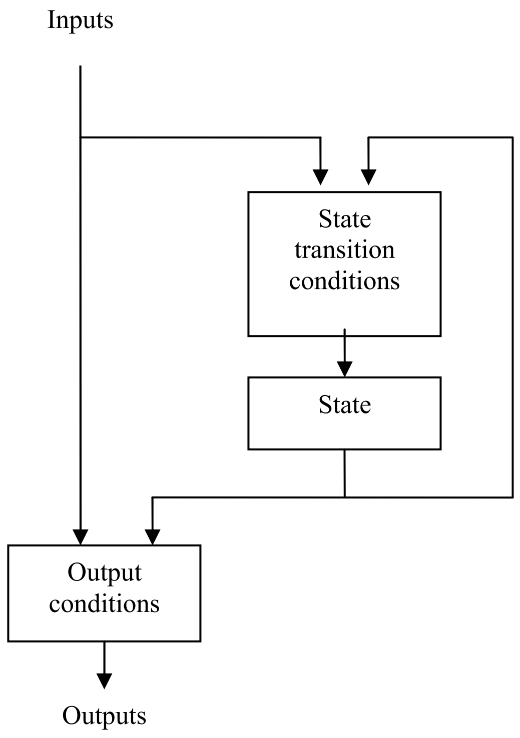

Figure 1: Generic Finite State Machine

4. Finite State Machine Misunderstandings

The topic of finite state machines in software has attracted a number of misconceptions over the half century since the FSM concept was intensively studied in relation to hardware and other general systems. Regrettably, in software design this has caused many practitioners to conclude that the FSM concept is difficult to apply to real projects. Some remarks aimed at correcting this erroneous impression follow. Figure 1 shows the generic FSM, whose state sequencing depends on the inputs and also on the history of those inputs as evidenced in the state.

Many students are introduced to the FSM in software only in the context of the parsing of regular expressions, or in general, text streams. The FSM used for this purpose is a specialised subset of the general case, and is often "deterministic" in that it operates to produce a result, and then ceases processing the input. Furthermore, the input is a character stream, which is a very limited form of FSM input. Based on this case, one is easily led to conclude that a transition table is a very straightforward affair, and that it would be easily to use it as a basic for automatic code generation. But in a typical control-system application, the transition table for a FSM needs to take account of a much larger number and variety of inputs, often requiring several variables to be combined in expressions, and to process this table for automatic code generation, in a general way, becomes a nightmare.

The typical software-textbook definition of the FSM omits to mention "actions" which, in control systems, need to be generated at various stages in the operation of the FSM; for instance on entering a specific state. Yet the classical theoretical studies of FSM models explain the "Moore Model" and Mealy Model" of FSM output, which were studied half a century ago: these concepts seem irrelevant to many software practitioners. In some cases they are unknown: one published proposal to the W3C group [4] takes an oversimplified FSM as a basis, and elaborates on this to produce the new concept of "transition networks" which are nothing more than the classical FSM as understood by many hardware designers

It is obvious that an FSM operates as a result of various changes to its input, and this leads to the assumption that an FSM is "event driven" – which is in a sense true. But the concept should not be taken too far, for control systems, although it is valid for the text parsing examples often encountered by programmers. Published proposals for generating code for FSMs [5] have included schemes in which each incoming event is "consumed" by the FSM as it operates. An event which does not cause a transition is discarded in these schemes, and this leads to a need to store events which might be needed in future, by ensuring that they always cause a transition to a new state. With this scheme, the number of states required for even a moderately complex process becomes prohibitively large, and the FSM impossible to grasp intuitively. We stress that an FSM must have access to all the inputs which will determine its transitions, at the instant when any transition expression is being evaluated.

To take an example of this point, consider an out-door security lamp. This should light when movement is detected, and remain lit for, say, 40 seconds after movement ceases. But it should not light in daylight, so it incorporates a sensor of the ambient light level. Using a classical FSM model, we need just two states, the initial OFF state and the ON state (Figure 2). The lamp controller passes to ON when movement is detected, but only when the ambient light level is low. An entry action to the ON state starts a timer, and an "input action" re-starts the timer whenever movement is detected. The transition to the OFF state occurs when the timer reaches the time-out delay, and perhaps also when the ambient light level has increased.

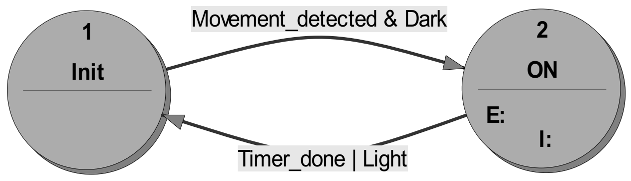

Figure 2: Lamp Control by Classical Finite State Machine

Now consider the equivalent state transition diagram for a purely event-driven version (Figure 3). It requires 4 states and 7 transitions! Of course, in practice there are ways around the problem, by means of "guarded transitions" but these are in fact a dilution of the event-driven principle.

Many text books, and some published schemes for generating code, seem only to consider a single FSM. In real control systems, and in telecommunications systems, it is usually essential to split the system into a number of FSMs, all coordinated in some way, as otherwise a single FSM becomes far too complicated to understand. Error conditions need to be handled in real life, and these are another cause of growth of the number of states: a process which seems easily expressed as a single FSM, when first designed for the case when everything works, might require three or more FSMs to provide a totally safe and reliable implementation. A well thought-out strategy for dealing with large numbers of concurrent FSMs in a single system is required.

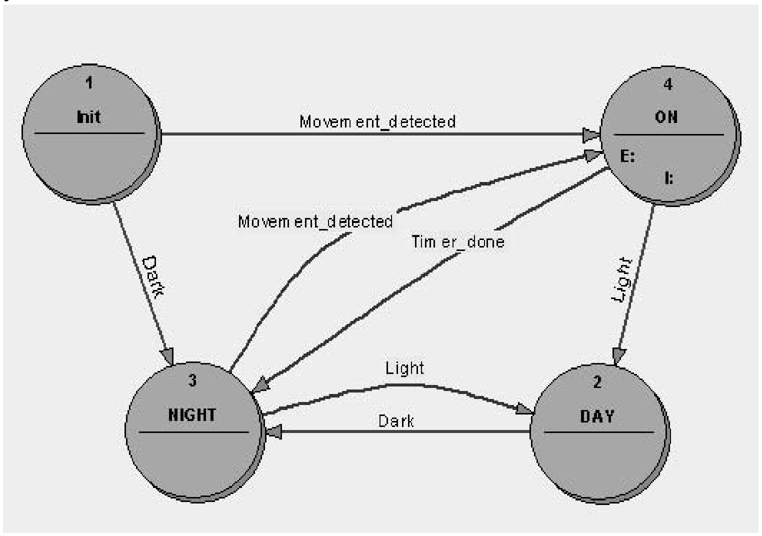

Figure 3: Lamp Control by Purely-Event-Driven Finite State Machine

A practical, but non-trivial problem is that it can be hard to display all the details pertaining to an FSM on the "transition diagram" which is a popular design tool.

All of the difficulties outlined above can be overcome, quite simply, but only with the aid of a few important concepts such as those developed for StateWORKS.

5. The StateWORKS Approach

This approach is based on a separation between data manipulation and system behaviour, and is devoted to the development of totally precise FSM models, which can then be used with no need to generate code by hand. These models control the behaviour of the software, which is expressed as a set of FSMs, preferably arranged in a hierarchy. Each FSM is sufficiently simple to be examined, understood and discussed by designers and their colleagues, and usually also by project managers and customers.

The essential problem in automating FSM code generation in software lies with the "Transition Tables" which will define, for each FSM, and for each of its states, the conditions which could provoke a transition to a different state. Although this can appear easy for some simple examples, in an embedded system there is usually a requirement for many of the transitions to be controlled by quite complex expressions, involving a variety of parameters such as limit switch states, fluid levels, temperatures, etc. in combination. Without some new ideas, these conditions will only be expressed in models as comments, and the programmer would have to deal with each on an ad-hoc basis.

The StateWORKS FSM models essentially handle no numerical data: they run in a "Virtual Environment" where they only see “Virtual Input” (a set of control flags) representing states of other FSMs, states of inputs, received messages or commands: transitions are triggered by occurrences of various events. Of course, real data needs to be processed to provide such "Virtual Input" and StateWORKS provides a variety of ways of doing that.

To give an example, one might be measuring a temperature T19 in some process. Perhaps T19 ought to lie within limits 130° to 155°. The VFSM does not know or care about these physical temperature values, but only receives state information, in the form of "Input Names": T19_LOW, T19_GOOD, T19_HIGH, T19_ERROR where the last item is caused by a detected fault in the measuring device. (These names are assigned by the designer, of course.)

A powerful feature of the StateWORKS system is that expressions governing state transitions, and also any expressions governing actions, use such "Input Names", combined as appropriate using OR operators, AND operators, and brackets. The NOT operator is not permitted, because to say that T19 is not in a particular state, of its four possible states, has no precise meaning. Expressions have no practical limit on complexity. If a very complex expression is needed, then one may use some additional “Input Names” which are themselves derived from sub-expressions, to improve readability.

Another important feature of an FSM is the set of "Actions" which are triggered by the FSM as it operates. These are typically generated when a new state is entered, but can also be triggered on exit from a specific state. Furthermore, an action might be desired, when in a certain state, on detection of a specific pattern of inputs. StateWORKS can handle all these possibilities. (In other terms, it can express both the Moore and Mealy FSM models.) The expressions to control such actions are of similar nature to those controlling state transitions.

Using the StateWORKS development system, a dual-mode editor permits convenient graphical construction of state-transition diagrams, while at the same time full details are saved in tables. For each state in the state transition diagram a corresponding table can be opened for examination or editing. An FSM can be designed in this way as a rather abstract entity, and in fact we call this a Virtual FSM (VFSM). A project editor window permits each VFSM to be configured to meet the requirements of the projects, and linked to the input/output, to other VFSMs in the system, and to issue commands. One VFSM can be used in many instances: for example a movement control VFSM might be designed, and used several times in a project. Of course, it can also be re-used in other projects, as a component (a new object type if you prefer).

We recommend constructing FSM systems in a hierarchical way, by which upper-level FSMs have knowledge of the states of lower-level FSMs, and can issue commands to them. The design tools support this scheme, and a diagram of “State Machine Systems” organisation is drawn for documentation purposes.

Does it work? Well, the concept has been in use for over 12 years, and always very successfully[2,3]. The StateWORKS approach involves specifying each FSM in total, intimate detail, with no ambiguity or loose ends to be patched later. In fact, we prefer to avoid generating code to run FSMs, but instead there is a "VFSM Executor" program which runs all the FSMs, by understanding a cleverly-coded version of the FSM specifications. This "executor" program has been improved slightly over a 12 year period, and no bugs have been encountered in the past few years. It can deal with large systems which employ dozens, hundreds, or even thousands, of FSMs and is totally reliable.

The fact that it has been possible to run software for many industrial control systems, various instrumentation devices and also telecommunications switches by means of a single, invariant "VFSM Executor" program, which interprets the formal specifications, demonstrates that those specifications are totally complete.

There are some subtle aspects of this technique, for example the use of a defined “execution model” which defines how the finite state machines should operate, and for which a set of strategies is incorporated in the “executor”. One important point, for a general FSM, which will not be noticed for an event-driven equivalent, is that after a transition from one state to a second state, the inputs may direct an immediate transition to a third state. StateWORKS will perform the two transitions without a pause, not allowing any “input actions” defined for the intermediate state to be produced although “entry actions” will occur. This point took many months of discussion and experiment to settle. All the strategies which have been adopted are aimed at ensuring completely deterministic and reliable behavior. Operation is always identical between the test environment and the final run-time environment. Note that, in contrast, programmers writing code based on models find themselves building such strategies into the software, sometimes without realising it, and the resultant code is liable to occasional failure without any obvious cause. Various published proposals for generating code from models seem to be at risk in this way.

There are many other aspects of StateWORKS, such as the testing and monitoring facilities, the ability to operate over TCP/IP and other networks, and the way in which the Real Time Data Base is set up, which are outside the scope of this paper. We merely wish to remark here that all the diverse practical aspects of linking the StateWORKS functions to other parts of the system have been dealt with, and that in all cases the StateWORKS section provides a useful structure for the entire project, so saving a great deal of effort for the programmers.

6. Automatic Code Generation from XML Formats.

If a system is implemented by means of the StateWORKS development tools, totally complete and tested specifications are generated as the result, and these, in principle, could then be coded automatically. The point of this paper is to present an engineering method for construction of sufficiently complete models of system behaviour. In fact, StateWORKS eliminates the need to generate code in the usual sense of the word.

The StateWORKS I.D.E. automatically generates XML output files, which contain all the information of the models and can be used by any other tools as desired.

It is normal practice for StateWORKS users to run the final system from the models automatically, and in that case the XML files, together with the graphics files of transition diagrams, are only used for documentation purposes. The I.D.E. automatically generates files which express the full specification in a special format – a sort of intermediate code - for direct execution in a target system. This process is more fully described in [2] and [3]. It does not generate any conventional "code" in a readable format: only the models, in their rather abstract format, can be used as documentation or for introduction of changes.

In a few projects, where it was impossible to use the StateWORKS Executor because of lack of time to implement a version for the specific operating system in use, system development using the StateWORKS I.D.E. has proved to be a valuable tool for saving time, and for assuring a reliable end product, even though the FSM structures had to be "coded by hand". We do not in fact propose any tools for generating code automatically from our XML files: we merely assert that this must be possible, on account of the completeness of the models, and leave the process to others to investigate. Our contention is that such a code-generation step, although very common practice in the software industry, is quite unnecessary and even dangerous.

StateWORKS models are "Platform Independent Models" (PIM) and it is not necessary to transform them to "Platform-Specific Models" (PSM) in the terminology of Model Development Architecture. The stage of translation of models into the working system, with some sort of tool, is also avoided. In fact, the Executor must be re-compiled or otherwise adapted in minor ways to any new target environment, but the models never change.

7. VFSMML

7.1 Overview

This section introduces the basic ideas and describes the overall design of VFSMML, which is the name given to the StateWORKS XML format for VFSM. Fuller details and examples may be downloaded from [3].

The VFSMML mark-up consists of about 27 elements. To completely describe a VFSM two sections are required: a section with definition of virtual input and output, and a section which specifies the state machine behavior.

7.2 Virtual input and output

The virtual input is a set of values (names) which are used in the state machine specification to describe behavior conditions, i.e. input actions or transitions. The virtual output is a set of values (names) which are set by the state machine based on a certain situation, i.e. when entering a state, exiting a state or as input actions. For instance to represent a simple on/off switch the following VFSM can be defined:

```
<VFSM>
  <Type>switch</Type>
  <Object>
    <Name>switch1</Name>
  </Object>
  <IOid>
    <Input>
      <Name>high</Name>
      <Value>1</Name>
    </Input>
    <Input>
```

```
            <Name>low</Name>
            <Value>0</Name>
         </Input>
      </IOid>
      <State>
         <Name>HIGH</Name>
      </State>
      <State>
         <Name>LOW</Name>
      </State>
   </VFSM>
```

The names "high" and "low" represent the virtual input of the switch VFSM. The names "HIGH" and "LOW" can be used as its virtual output (the current state). One can define state machines, which are commonly used, e.g. timer, digital input, digital output etc. Those state machines don't need to be defined in a VFSMML specification, as their virtual inputs and outputs are well known on the target system. VFSMML defines a set of known (predefined) VFSM. To use a predefined VFSM, only its object name definition is required.

7.3 State machine behaviour

The behavior of a state machine is given by the description of its states. Each state can set output values (names) based on certain conditions. The conditions are logical expressions created out of the input values (names). Entering or exiting a state can also be used as a condition to set an output value. For each state any condition-based transitions can also be specified. To support logical expressions to build conditions, MathML syntax is used. For instance to specify that the following state machine shall change to state "starting engine" when the air conditioning is running and the start switch is on, the description below can be used:

```
<VFSM>
   <Type>Engine</Type>
   <State>
      <Transition>
         <Condition>
            <apply>
            </and>
            <ci>airconditioning_runnin
              g</ci>
            <ci>switch_on</ci>
            </apply>
         <Condition>
         <StateName>
            StartingEngine
         </StateName>
      </Transition>
   </State>
</VFSM>
```

The input names used for conditions and output names used for actions are based on objects defined for the given VFSM. For instance the name "switch_on" could be created using the definition given in section 7.2:

```
<VFSM>
   <Type>Engine</Type>
   <IOid>
      <Type>
        switch1
      </Type>
      <Input>
         <Name>switch_on</Name>
         <Value>high</Value>
      </Input>
   </IOid>
   …
</VFSM>
```

7.4 Example of VFSMML

The XML below corresponds to the state machine of Figure 2.

```
 <?xml version="1.0" ?>
 <?xml-stylesheet   href="vfsmml.xsl"
type="text/xsl"?>
 <!DOCTYPE vfsmml SYSTEM
   "vfsmml.dtd" >
 <vfsmml>
 <VFSM type="vfsm">
   <Type>VFSM_C</Type>
   <Prefix>VFS</Prefix>
   <IOid>
     <Name>MyCmd</Name>
     <Type>CMD-IN</Type>
   </IOid>
   <IOid>
     <Name>Timer</Name>
     <Type>TI</Type>
     <Input>
       <Name>
         Timer_done
       </Name>
       <Value>OVER</Value>
     </Input>
     <Output>
       <Name>
         Timer_Re_Start
       </Name>
       <Value>
         ResetStart
       </Value>
     </Output>
     <Output>
```

```
      <Name>
        Timer_Start
      </Name>
      <Value>Start</Value>
    </Output>
  </IOid>
  <IOid>
    <Name>Lighting</Name>
    <Type>DI</Type>
    <Input>
      <Name>Dark</Name>
      <Value>LOW</Value>
    </Input>
    <Input>
      <Name>Light</Name>
      <Value>HIGH</Value>
    </Input>
  </IOid>
  <IOid>
    <Name>Movement</Name>
    <Type>DI</Type>
    <Input>
      <Name>
        Movement_detected
      </Name>
      <Value>HIGH</Value>
    </Input>
  </IOid>
  <IOid>
    <Name>Light</Name>
    <Type>DO</Type>
  </IOid>
  <State>
    <Name>OFF</Name>
    <Transition>
      <Condition>
        <apply>
        <and/>
        <ci>Movement_detected</ci>
        <ci>Dark</ci>
        </apply>
      </Condition>
      <StateName>
        ON
      </StateName>
    </Transition>
  </State>
  <State>
    <Name>ON</Name>
    <EntryAction>
      Timer_Start
    </EntryAction>
    <InputAction>
      <Condition>
        <ci>Movement_detected</ci>
      </Condition>
      <Action>
        Timer_Re_Start
      </Action>
    </InputAction>
    <Transition>
      <Condition>
        <apply>
        <or/>
        <ci>Timer_done</ci>
        <ci>Light</ci>
        </apply>
      </Condition>
      <StateName>
        Init
      </StateName>
    </Transition>
  </State>
 </VFSM>
</vfsmml>
```

8. Conclusions

A concept has been presented which permits the construction of totally complete platform-independent models of software system behaviour. We suggest that only such a concept can really bridge the present gap between the models and the final, reliable implementation of working software.

We propose an XML representation which is able to express the full detail of such models.

We deprecate the current obsession with coding as the only way to implement software. Although the StateWORKS executor is in fact a form of interpreter, of an intermediate code expressing the complete platform-independent models or specifications, the processing which it undertakes is very efficient and compares well with the speed of any alternative coded solution. If coding of great complexity is avoided, the specifications remain as the only form of documentation, and remain valid through the life of the projects in which they are used.

We suggest that the StateWORKS concepts could help to fill a gap in the present ideas about UML, and provide a path towards "Executable UML". In other words, if there is ever to be any hope of generating software automatically from UML or any other specifications, then ways of generating totally complete models from them, in the StateWORKS style, need to be developed.

Furthermore, StateWORKS could no doubt be applied with advantage to the generation of code for "Programmable Logic Controllers" for which current techniques are becoming inadequate to implement the more complex systems.

9. References.

[1] Chris Edwards: "Modelling standard gets ready for second round", IEE *"Electronic Systems and Software*" Oct/Nov 2003.

[2] F. Wagner, P. Wolstenholme: "Modeling and Building Reliable, Re-useable Software" *Proc. 10th. IEEE Symposium on Engineering of Computer-Based Systems* (ECBS'03) Huntsville, April 2003.

[3] See http://www.stateworks.com/papers

[4] "*XTND – XML Transition Network Definition*" W3C Note, Nov. 2000.

[5] S. J. Mellor, M. J. Balcher: "Executable UML: a Foundation for Model-Driven Architecture" Addison-Wesley, 2002.