CompEuro 1992 Proceedings

Computer Systems and Software Engineering

May 4–8, 1992 *The Netherlands*

Sponsored by

IEEE Computer Society

IEEE Region 8

IEEE Benelux Section

IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

VFSM Executable Specification

F. Wagner

BALZERS AG

Abstract

*The paper presents a software design method based on a **virtual finite state machine** (**VFSM**) concept. The concept defines a virtual environment that allows the finite state machine to be an entirely table driven software module. A hybrid finite state model is used to achieve a superior design clarity. The presented method separates the control part of the design problem from the data manipulation part. The specification of the control part is directly executable. The control part is not coded; it is expressed in a table that is executed by the **virtual finite state machine executor**.*

1 Virtual Environment

1.1 Software design

In the very beginning of any project there are requirements. They are general, expressed in an implementation independent way. A software project starts with a specification [1] [5] [11] that should describe the requirements in a formal but still implementation independent form. The implementation that consists of a design and coding should be an exact translation of the specification into programming language instructions. The code includes implementation specific aspects dictated by the language and operating system.

If the transition from a specification to a design and later code is done by hand, or more precisely speaking, by the mind of a programmer, some deviations from the specification are unavoidable. Changes in the requirements introduced when the project is well advanced tend to bypass the specification and design, and are done directly in the code.

During development the role of specification and design documents decreases. At the end of a software project the only reliable source of information "what the software does" is the code itself.

This typical software development scenario repeats project after project. Different administrative means try to improve the situation but they fail for several reasons:

- in the end only the code really counts as it is the end product,
- any document made by hand that is not processed by programs that verify its correctness (compile, assemble, test) is not reliable.

1.2 Introducing the idea

Let us specify a simple task of controlling an air conditioner in an apartment:

> The air conditioner should be switched on if the temperature is too high (above 80 degrees) and all windows are closed. It should be switched off if the temperature is low enough (drops below 75 degrees) or any window stays open too long (1 minute).

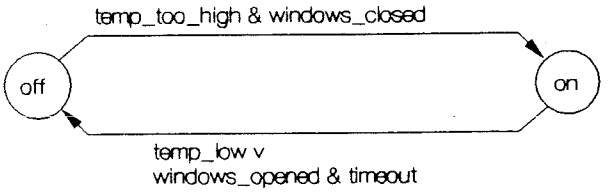

Figure 1 State transition diagram for an air conditioner control

The state specification table shown in Figure 1 presents the concept of the air conditioner control using a finite state machine with two states: *off* and *on*. The air conditioner control is fully specified in a specification table (Table I).

> Entering state *off* the finite state machine switches the air conditioner off: (*E: air_cond_off*). If the temperature is too high (*temp_too_high*) and all

0-8186-2760-3/92 $03.00 © 1992 IEEE

Table I Specification table for an air conditioner control

off (state)		E: air_cond_off (entry action)
on (next state)	temp_too_high & windows_closed (transition condition)	
on (state)		E: air_cond_on (entry action) X: stop_timer (exit action)
	windows_closed (input action condition)	stop_timer (input action)
	windows_open (input action condition)	start_timer (input action)
off (next state)	temp_low v windows_open & timeout (transition condition)	

windows are closed (*windows_closed*) the finite state machine changes state to *on*. Entering state *on* the finite state machine switches the air conditioner on: (*E: air_cond_on*). If the temperature drops low enough (*temp_low*) the finite state machine changes its state to *off*. Opening any window starts a timer (*start_timer*). If the timer elapses (*timeout*) and the window is still open (*windows_open*) the finite state machine changes its state to *off*. Leaving the state *on* the finite state machine stops the timer (*X: stop_timer*).

The specification is abstract - it uses names that describe the essence of control requirements. A change in these names or conditions are highly improbable. If they happen they would mean a major revision of the requirements. The details of the requirements that are more likely to be changed (the actual temperature values or number of windows) are not present in the specification. Similarly, this specification does not address the details of the air conditioner or timer control. On the other hand it includes all details of the control requirements - how the air conditioner should be switched on and off by changing temperature and opening windows.

This specification describes the control of an air conditioner using only names and two logical operators: AND (&) and OR (v). The specification is abstract enough to remain stable for a long time. It could be used for many types of input temperature sensors, window position sensors, any number of windows, air conditioner types and timers. It has been achieved by avoiding implementation details.

1.3 Names as virtual signals

A virtual environment is created by inventing names of signal values. The names must cover all data relevant for the problem to be solved. The names are the only information that can be used by specifying the problem.

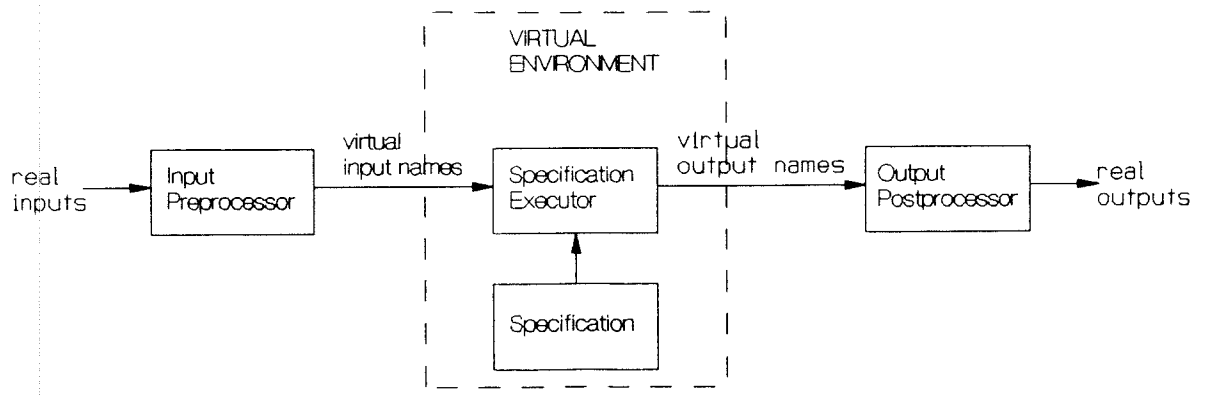

Figure 2 A control system with a specification executor

The true nature of the real signals does not matter - they can be numbers, boolean values, analog values, attributes, parameters, descriptive names of situations, complex conditions, etc. Irrespective of their origin, names are equal in the virtual environment. They express some value or feature of signals. They describe some more complex input/output dependencies. They define some combined features of many signals.

2 Virtual Finite State Machine

2.1 VFSM Control System

Let us now imagine an implementation system presented in Figure 2. It comprises a standard engine **Specification Executor** that is able to execute abstract specifications similar to that shown in Table I. Real inputs are preprocessed to a form required by the **Specification Executor**. Outputs produced by the **Specification Executor** must be transformed by an **Output Postprocessor** to a form suitable for controlling outside actuators. The **Specification Executor** as well as the **Specification** are in a **Virtual Environment.**

Thus, the concept of control software presented in this paper leads to partitioning of the problem into well defined pieces:

- input preprocessing procedures that map the real inputs or input conditions into uniform *virtual input names*,
- output postprocessing procedures that trigger actions defined by *virtual output names* produced by the **Specification Executor**,
- the specification of the control problem.

The Specification covers the control flow of the designed system. It includes the entire knowledge about the behavior of the system, but it is kept on an abstract level to assure that it is fairly readable as it does not include any implementation specific details. Moreover, the specification is not susceptible to input/output implementation changes as it is implementation independent and therefore portable.

The **Specification Executor** is a constant engine written only once and used for all control specification in the system (finite state machines).

2.2 Boolean expression as table of sets

A **virtual finite state machine** is an entirely table based program. There are two barriers that limit the use of tables in software design. First, the use of tables requires uniform inputs. Second, the tables grow exponentially with increasing number of inputs. The concept of a virtual environment [9] [10] solves the problem allowing tables to be used on a much larger scale. The virtual environment allows logical conditions to be expressed as a *table of sets*. It seems that for any practical purpose, size of a *table of sets* does not exceed acceptable values, and they can be used to implement finite state machines.

Let us assume that:

- input variables of a logical function are set elements;
- a set of some input variables represents an **AND** operation on these variables;
- a table of sets (of input variables) represents an **OR** operation on these variables.

A more detailed description of the *table of sets* method can be found in [9]. For the purpose of this paper this definition will be supported by an example. A logical function in a *table of sets* form:

$$aircond\_on = \left| \begin{matrix} \{temp\_low\} \\ \{windows\_open,\ timeout\} \end{matrix} \right|$$

expresses the logical function (taken from the Table I):

$$aircond\_on = temp\_low \vee windows\_open \wedge timeout$$

The value of a logical function in the *table of sets* form is calculated by checking whether the sets in the table are a subset of the actual input variable set. If at least one of the sets in the table is a subset of the actual input variable set the logical function is **TRUE**, otherwise it is **FALSE**.

In the above example *aircond_on* is **TRUE** if either the name *temp_low* or both names: *windows_open* and *timeout* are in the input variable set.

2.3 VFSM execution model

Combining the features of Mealy and Moore automata [2][6][7][8] leads to combined models. There are several combined models imaginable. The model used for the **virtual finite state machine** has evolved from practical experiments and experiences rather than from any deeper theoretical analysis. This model is not dictated by the concept of a virtual environment; any other model will do. Anyway, this model will be further named as a *virtual finite state machine* (*VFSM*) model.

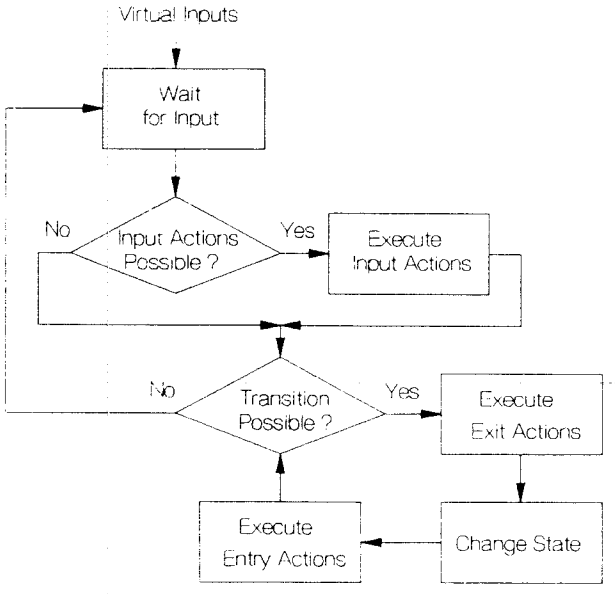

Figure 3 VFSM execution model

A **VFSM** model uses three types of actions: entry, exit and input actions. The entry and exit actions represent the Moore nature of the model - they are state dependent. The input action represent the Mealy feature of the model - it is a function of a state and inputs.

The **VFSM** model is described by a flowchart shown in Figure 3. The important issue is that a transition function is rather a boolean condition and not just a single event. Keeping this in mind, three possible scenarios may happen with none (A), single (B), and multiple (C) state changes.

In case A, the input change causes only an input action and the **VFSM** returns to the waiting point - there are no conditions to change the state.

In case B, the input change triggers a state change. First, an input action is performed. Since the transition conditions are fulfilled an exit action is carried out, the state is changed, an entry action in the new state is performed, and the **VFSM** returns to the waiting point.

In case (C), the input change triggers a serious of input changes. The beginning is exactly as in case (B): an entry action, an exit action, state change and entry action. Since in the new state the transition function is also fulfilled the **VFSM** continues the loop: exit action, state change, entry action. The loop terminates when the **VFSM** enters a state in which there no more condition to change the state, and the **VFSM** returns to the waiting point.

The basic design philosophy of the **VFSM** is described by a definition of its actions:

a). Entering a state, the **VFSM** does an entry action and waits for a reaction of the controlled system. Normally, the reaction should lead to a state change.

b). Inputs which do not cause state change should trigger input action.

c). An exit action performed by leaving a state should have auxiliary character; it must not influence other states.

2.4 Table driven VFSM implementation

The algorithm presented by the flowchart in Figure 3 defines the behavior of the **VFSM** model. The flowchart defines the **VFSM Specification Executor**. The specification is represented as tables of sets and the **Executor** operators: *transition*, *do_input_action*, *do_entry_action*, *do_exit_action* use set operations to evaluate the specification control logic and output functions.

Figure 4 presents graphically a specification of one state. Three data types are used to describe the states:
- a number *state* that specifies the next state;
- a set of input names *VINPUT*;

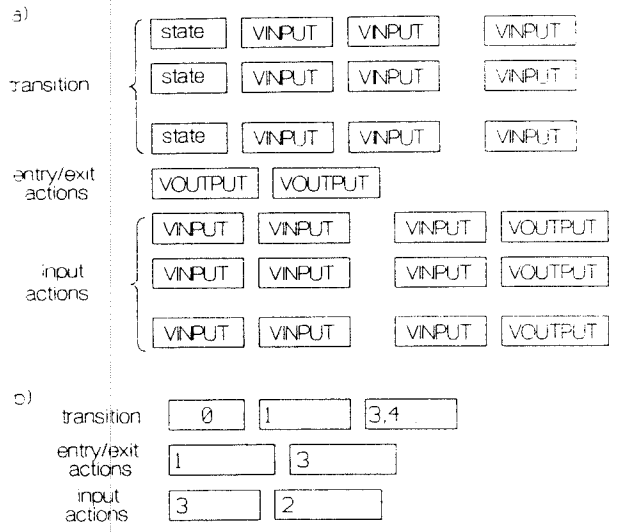

Figure 4 *Table of sets* representation of VFSM transition table (a) general, (b) state *on* in Table I

- a set of output names *VOUTPUT*.

For instance, to find out whether a transition is due the **Executor** checks whether any of the transition *VINPUT* sets are subsets of virtual input names variable. Similarly, this set operation is used to find out which of the input actions should be carried out.

3 Software Specification and Development System

3.1 Specification forms

Several FSM specification forms are known. At present, a **VFSM** is specified using a specification table in a form presented in example in Table I. The table comprises full **VFSM** specification, i.e. transitions and all actions: entry, exit and input.

A specification of a **VFSM** begins with definition of input, output, and state names. The names are defined as enumerations. These are the only names which can be used in the specification table.

3.2 Translation

The translation process is presented in Figure 5. The process has two phases. First, the specification table is tested for correctness (names and operators) and if it is correct a **VFSM string representation** *xxxx.va* is produced. In addition, a header file is generated which comprises all table sizes as a C constants. The translation program generates also several auxiliary files like: list of names and a source program for drawing a directed graph using DAG tool. The first phase translation programs are written in AWK.

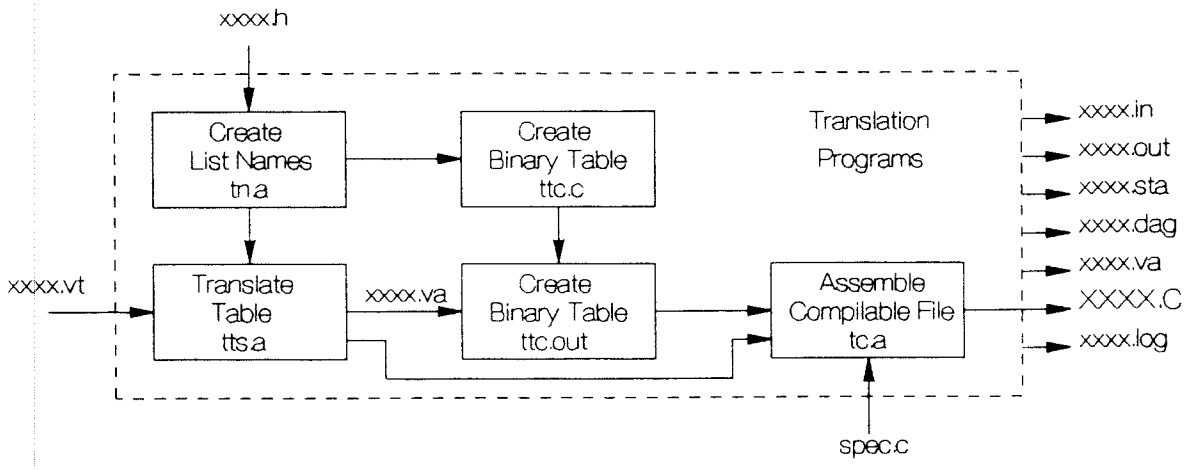

Figure 5 VFSM translation tool

The second translation phase is accomplished by AWK and C programs which generate a compilable C specification file *XXXX.C*. This C specification file consists of a *spec* variable and access functions. The *spec* variable is the binary specification table, and it is initialized to values defined by the **VFSM** string representation. The access functions are used by the **VFSM executor** to read data from the spec variable.

The *spec* variable has a header which comprises pointers to access function. The **VFSM executor** is called with a pointer to the *spec* variable header which enables it to access the specification data. In such a way, the **VFSM executor** is completely separated from the specification. Hence, one **VFSM executor** serves any number of **VFSMs**.

4 Implementations

The concept of a virtual environment has been utilized for the design of complex software control systems. The most complicated system has 2000 inputs and outputs and consists of 400 interconnected virtual finite state machines. The size of each machine varies from 20 to 60 states.

Experiences with process control and telecommunication software have shown that the specified finite state machines have the immense merit of remaining comprehensible to the designers who originally specified the system behavior. It is possible to maintain such software with far less effort and worry than is the case with conventional software.

Acknowledgments

The virtual environment concept and the VFSM Technology have evolved over many years. Without innumerable discussions with my friends in Europe and USA I would not have been able to reach the point of the working paradigm and design system. I would like to express my gratitude to engineers in BALZERS AG where the idea was conceived and at&T where I was able to find it workable in a totally different environment. Especially, I would like to mention the encouraging and fruitful discussions with R. Schmuki and A. Flora-Holmquist.

References

[1] T. De Marco, Structured Analysis and System Specification. Prentice-Hall, 1978.

[2] A. Gill, Introduction to the Theory of Finite_State Machines. McGraw-Hill, 1962.

[3] W. Grass, Steuerwerke. Entwurf von Schaltwerken mit Festwertspeichern, Springer Verlag, 1978.

[4] D. Harel, "Statecharts. A Visual Formalism for Complex Systems", Science of Computer Programming, No.8, 1987, pp. 231-274, Elsevier Science Publishers B.V. (North-Holland).

[5] D.J. Hatley, I.A. Pirbhai, Strategies for Real-Time System Specification. Dorset House Publishing, 1988.

[6] F.C. Hennie, Finite-State Models for Logical Machines. John Wiley & Sons, Inc., 1968.

[7] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, 1979.

[8] Z. Kohavi, Switching and Finite Automata Theory. McGraw-Hill, 1978.

[9] F. Wagner, A Virtual Environment for Table Based Control Software. Report prepared for AT&T, 1990.

[10] F. Wagner, Method of and Apparatus for Constructing a Control System and Control System Created Thereby (patent pending), 1991.

[11] E. Yourdon, L.C. Larry, Structured Design: Fundamentals of a Discipline of Computer Program and System Design. Prentice-Hall, 1975.