F. Wagner

A Virtual Environment For Table-Based
Control Software

Abstract

Software can perform more simply and effectively if it is driven by a set of uniform input signals. This paper presents the concept of a virtual environment which separates heterogeneous real-world signals from a uniform virtual world of names. The concept uses sets to express boolean functions and is presented with details for implementing logical functions. The virtual environment concept allows a finite state machine be totally table driven.

1. Introduction

Programs written in high-level procedural languages, like PASCAL, C, MODULA-2 or ADA, can only be formed from a few types of statements, as defined by the syntax rules of the language. The statements can be grouped into three categories:

— assignment, loops: for, while, repeat;

— if then else, case (switch);

— go to.

These statements control the execution of the program: select other statements to be executed, perform loops, and transfer program control. The disadvantages of the "go to" statement had been understood some 20 years ago, and led to the idea of a structured programming style, which replaced the previous, rather chaotic, style. The "go to" statement has remained in the syntax definition of programming languages as a kind of relic, but it is rarely used in practice.

Structured programming has not solved all software problems. The software crisis is a current reality. A long list of problems contribute to this situation. Programs are difficult to read and to change. The "logic" of programs is expressed by "if then else" and "case" statements. More complex logical bindings, when expressed in these statements, are difficult to understand (see example of a Pascal program in Appendix A). Any change of the logical expression requires a change to the code, which may not really be under control, and may have unexpected side effects.

A key point made in this paper is that we must try to avoid use of the "if then else" and "case" statements. Replacing them with tables dramatically improves the situation. A program which consists only of assignments and loop statements is easier to read and understand, because the logic of the problem is concentrated in tables, which are easier for a human being to understand. In addition, table-driven software is modifiable without recompilation, even on-line in some instances.

This paper discusses the usage of tables in writing control software. The idea of table-driven software has long been known but has not been fully exploited. In addition to some psychological barriers, there is also a purely technical problem. Namely, the use of tables requires uniform inputs. This paper introduces the concept of a virtual environment where all input signals are known by names, which correspond to the control essences of the signals.

The virtual environment allows us to introduce a new method of expressing boolean functions. Conventional software structures (arrays of boolean) for the truth table of a logical function must

be completely defined. They include, in many cases, don't care conditions, i.e., redundant information required by the syntax rules of a programming language. The "table of sets" form of a boolean function, presented in this paper, allows the size of tables be kept minimal - the size is determined by the essential information only.

Sections 2 and 3 summarize basic features of table-driven software. Section 4 defines a new method for expressing boolean functions by sets. Section 5 introduces the concept of a virtual environment, which allows us to exploit fully the "table of sets" form of boolean functions. Section 6 documents, by example a possible software implementation of the virtual environment and the "table of sets" concept. Section 7 describes the results of applying the virtual environment for designing a table-driven finite state machine. Conclusions in Section 8 close the paper.

2. Logical expressions as software tables

In high-level languages a table can be specified by a structured constant or an initialized variable. The second form will be used in the examples in this paper.

Let us examine some possible software expressions of a logical function:

y = (a AND b) OR c

for which the truth table is presented in Figure 1.

a b c	y		
0 0 0	0		
0 0 1	1		
0 1 0	0		
0 1 1	1		
1 0 0	0		
1 0 1	1		
1 1 0	1	where:	0 → FALSE
1 1 1	1		1 → TRUE

Figure 1. Truth table of the logical function y.

The obvious software implementation in PASCAL is shown in Figure 2.

```
FUNCTION Logical (a,b,c : BOOLEAN) : BOOLEAN;
BEGIN
  Logical := (a AND b) OR c;
END;
```

Figure 2. Coded implementation of the logical function y.

If the result of a problem analysis is presented in a control flow diagram, programmers will tend to translate it directly into a code. This approach leads to programs with long, unreadable "if-then-else" statements (see Appendix A).

The statement in Figure 2 is the simplest expression of the Logical function. If this is the only form of the function, it is the best solution. However, in many practical applications it is not only still more complex than necessary; but also it must be changed from time to time, which may introduce errors. Any change in the truth table requires reanalysis of the function.

Figure 3 presents a table solution as a PASCAL function. In cases where the truth table may be changed, Figure 3 is the better solution in comparison with the coded implementation. Because, in fact, the function in Figure 3 represents ***any*** logical function of three variables - the adjustment to a specific function is done by appropriate initialization of the variable c_logical. As the variable c_logical can be changed at any time in the program, the function Logical can be modified during execution of the program.

```
FUNCTION Logical (a,b,c : BOOLEAN) : BOOLEAN;

TYPE
   a_logical = ARRAY [BOOLEAN, BOOLEAN, BOOLEAN] OF BOOLEAN;

VAR
   c_logical : a_logical :=  ( ( ( FALSE,
                                   TRUE ),
                                 ( FALSE,
                                   TRUE ) ) ,
                               ( ( FALSE,
                                   TRUE ),
                                 ( TRUE,
                                   TRUE ) ) );

BEGIN
   Logical := c_logical [a,b,c];
END;
```

Figure 3. Table implementation of the Logical function.

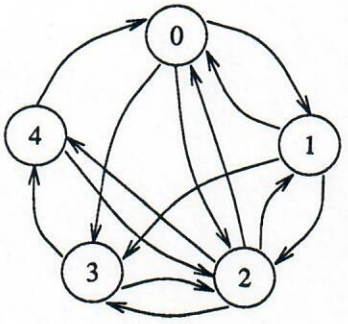
3. Table driven automata

Tables can be used to implement automata. The advantages of tables are evident in situations where the automaton is not just a counter but has a somewhat more complex transition diagram. As an example, let us discuss an automaton which has a variable number of states. In other words, the automaton generates sequences of numbers. It has two inputs: seq_sel = 0..3 which determines one of four possible sequences, and the current state (cur_state). A sequence can have 2, 4 or 5 elements (numbers) - the details are shown in Figure 4 as a C language function.

a)

seq_sel	sequence of states (numbers)
0	2-4
1	0-3-2-1
2	0-1-3-2
3	0-1-2-3-4

b)



c)

```
#define         MAX_SEQ       4          /* number of sequences */
#define         MAX_STATES    5          /* highest number of states */


        generate (seq_sel, cur_state)      /* generate next state of a sequence */

        int seq_sel, cur_state;            /* seq_sel selected sequence */
                                           /* cur_state current state of automata */

        {
          int next_state;                  /* next state of automata */

          state int c_count [MAX_SEQ] [MAX_STATES] =

                        {
                          { 2,2,4,2,2 },        /* sequence 2-4 */
                          { 3,0,1,2,0 },        /* sequence 0-3-2-1 */
                          { 1,3,0,2,0 },        /* sequence 0-1-3-2 */
                          { 1,2,3,4,0 }         /* sequence 0-1-2-3-4 */
                        };

          next_state = c_count [seq_sel] [cur_state];

          return next_state;

        }
```

Figure 4. Sequence generator: states sequences (a), state diagram (b), implementation as a function in C (c).

Table-driven software automata are considerably easier to design then their coded equivalents. Each will represent a whole class of automata, the specific one being determined by the content of the table, in a specific case. Designing automata with more complex transition functions is not so obvious. With this introduction, let us now turn to a method by which more complex logical functions can be implemented efficiently in tabular form.

4. The use of sets in boolean expressions

Let us name the values of a boolean variable x_i:

x_i^F stands for the FALSE value

x_i^T stands for the TRUE value

Let the AND operation on boolean variables $x_1 \cdots x_k$ be expressed as a set $\{\tilde{x}_1,\ldots,\tilde{x}_k\}$ (see Appendix B) where:

$\tilde{x}_i = x_i^F$ for $\overline{x}_i$

$\tilde{x}_i = x_i^T$ for x_i

Example 1:
a AND $\overline{b}$ AND c <==> $\{a^T, b^F, c^T\}$

Assume that a logical function $f(x_1, \cdots, x_k\}$ has the boolean disjunctive form. Then the function can be expressed as an array of sets:

$$f(x_1,\ldots,x_k) = \left| \begin{array}{l} \{\tilde{x}_1,..,\tilde{x}_k\}_1 \\ \ldots \\ \{\tilde{x}_1,\ldots,\tilde{x}_k\}_n \end{array} \right|$$

where n is equal to number of terms (AND expressions) in the f function.

Example 2:
f = a AND b OR a AND $\overline{b}$ AND c OR c <==> $\left| \begin{array}{l} \{a^T, b^T\} \\ \{a^T, b^F, c^T\} \\ \{c^T\} \end{array} \right|$

The value of a logical function in the "table of sets" form can be calculated as:

$$\begin{aligned} f(x_1,\ldots,x_k) = & (\{\tilde{x}_1,\ldots,\tilde{x}_k\}_1 \subseteq \text{ actual\_variable\_set}) \text{ OR} \\ & \ldots \\ & \ldots \\ & (\{\tilde{x}_1,\ldots,\tilde{x}_k\}_n \subseteq \text{ actual\_variable\_set})) \end{aligned}$$

where operator $\subseteq$ = "is a subset of", and
actual_variable_set = set of names which correspond to actual variables' boolean values

In other words the (boolean) value of the function is evaluated by testing whether any of the "AND-sets" in the table is a subset of the actual set of variable names, as determined by the current environment.

Example 3: Calculate the value of the function

f = (a AND b) OR (a AND $\overline{b}$ AND c) OR c

for a = TRUE, b = FALSE, c = TRUE
which corresponds to
actual_variable_set = $\{a^T, b^F, c^T\}$

f = ($\{a^T, b^T\} \subseteq \{a^T, b^F, c^T\}$) OR

($\{a^T, b^F, c^T\} \subseteq \{a^T, b^F, c^T\}$) OR

($\{c^T\} \subseteq \{a^T, b^F, c^T\}$) =

FALSE OR TRUE OR TRUE = TRUE

The empty set is a special case - by convention, it means "always FALSE". In addition, the set of all input names must be completed with an "always" name which is used to express "always TRUE". Hence, in a set representation of an AND term:

$\{x^T,...\}$	means:	the term is TRUE if x=TRUE and . . .
$\{x^F,...\}$	means:	the term is TRUE if x=FALSE and . . .
{}	means:	the term is always FALSE
$\{always\}$	means:	the term is always TRUE

Example 4: Express in a "table of sets" form functions described in the specification in the following way:

f_1 should be TRUE if
(x_1 is TRUE and x_5 is FALSE) or
(x_2 is FALSE) or
(x_3 is FALSE and x_4 is TRUE and x_6 is TRUE)

f_2 should be TRUE if
(x_1 is TRUE and x_4 is TRUE)

f_3 should be TRUE if
(x_3 is TRUE) or
(x_1 is FALSE) or
(x_2 is TRUE and x_6 is TRUE)

f_4 should be always FALSE

f_5 should be always TRUE

Solution:

$$
\mathrm{f} = \left|\begin{array}{lll|l}
\{x_{1,}^{T}\ x_5^{F}\} & \{x_2^{F}\} & \{x_{3,}^{F}\ x_{4,}^{T}\ x_6^{T}\} & (f_1)\\
\{x_{1,}^{T}\ x_4^{T}\} & \{\} & \{\} & (f_2)\\
\{x_3^{T}\} & \{x_1^{F}\} & \{x_{2,}^{T}\ x_6^{T}\} & (f_3)\\
\{\} & \{\} & \{\} & (f_4)\\
\{always\} & \{\} & \{\} & (f_5)
\end{array}\right.
$$

The "table of sets" form of a logical function can be used to express any logical function of many variables in a table form. The size of the table is determined by the largest expected number of terms of the function in disjunctive form. The minimal form of the function is preferable but not necessary - any form which fits in the assumed maximal size will do. The "table of sets" form corresponds to the usual way used by designers to describe logical conditions in a disjunctive form: writing factors which are essential for the function and neglecting the complexity of the function.

Notice that the function f_5 in Example 4 must have at least one {always} set - the content of the other sets is then irrelevant (disjunctive form).

The table in Example 4 presents five logical functions of six variables. Even for this small problem, the human mind has difficulty in comprehending the direct boolean form as used in Figure 3. This difficulty is not due to the size of the expression, but rather is due to limits on the human ability to understand and operate software structures.

The "table of sets" is a direct translation of the specification from the form conceived by human minds into software structure. It is readable and easy to change without introducing errors. A "table of sets" is a valuable form in cases where its limitation - the number of factors - is acceptable, which is the case in practical problems.

The use of sets for expressing a boolean function is not limited to a disjunctive form. The same idea can be applied to a conjunctive form, the table consisting of OR instead of AND terms. The conjunctive form is evaluated by testing whether all of the "OR-sets" in the table are subsets of the actual set of variable names. In this case, an empty set has also, by convention, a special meaning - always TRUE. But there is one exception - all empty sets in the table are used to express an always FALSE value of the function.

The application of "table of sets" for expressing more complex forms of boolean functions is imaginable, but seldom necessary.

5. Virtual environment

The use of "table of sets" requires the conversion of real signals into "named" (virtual) signals, which create the virtual environment. The formal specification, as well as its software implementation operate exclusively on these names. The method of expressing boolean functions in the "table of sets" form has been discussed using boolean signals, but virtual signals are not restricted to boolean ones. The virtual signals are just names and their origin is irrelevant for the boolean function. For instance a verbal specification:

> the "motor" should be "switched off" if the "oil temperature" has been "too high" for at least 120 seconds, or the "emergency switch" has been pressed, or the "command off" has been received, or the "command idle" has been received while the machine has reached the "end position"

can be expressed as:

$$\text{motor\_off} = \left| \begin{array}{l} \{\text{oil\_temp\_too\_high, timeout}\} \\ \{\text{emergency\_on}\} \\ \{\text{cmd\_off}\} \\ \{\text{cmd\_idle, position\_end}\} \end{array} \right|$$

This is the formal specification which can be directly used in an appropriate software structure and which corresponds to the "table of sets" form. Some of the invented names (emergency_on, position_end) will correspond to real boolean signals, others - do not. The specification covers the control flow only; it uses symbolic virtual names and does not cover other information irrelevant for control, such as the temperature timeout value. Any multivalued signals, such as: commands, switchpoints, etc., may be translated (see Appendix C). The conversion of real signals into virtual signals can be achieved by means of tables. Hence, the entire system consists of translation tables and logical function tables, as shown in Figure 5.

The inputs in the form of events and input data (for instance, parameters) are translated into names which create the virtual input. The translation process may have several forms, basically:

— one input produces one virtual input entry;

— one input produces two or more virtual input entries;

— many inputs produce one virtual input entry.

The virtual input is used by the function table to produce the output; in this case the logical function.

To explain better this point let us examine the example of a string of digits coming from a telephone. Considering the data flow the string represents a number. For control purposes the string consists of separate digits, some of them carrying essential control information, for instance: "first digit", "last digit", "zero digit", "any other digit". In fact the actual value of the digit is irrelevant for the control flow.

As an another example let us take the typical signal for control processes - the boolean one. If the control process needs, for instance, only the true value of the boolean signal, the signal would be represented in the virtual input by one name: signal_name_true. If the control process uses both true and false values, then both names will appear in the virtual inputs: signal_name_false and signal_name_true.

The conversion of real inputs to virtual inputs allows the software to operate in a uniform, virtual world containing only the essential facts, as compared to the multi-format, often redundant world of signals from the external environment. Once again, abstraction has produced software simplicity.

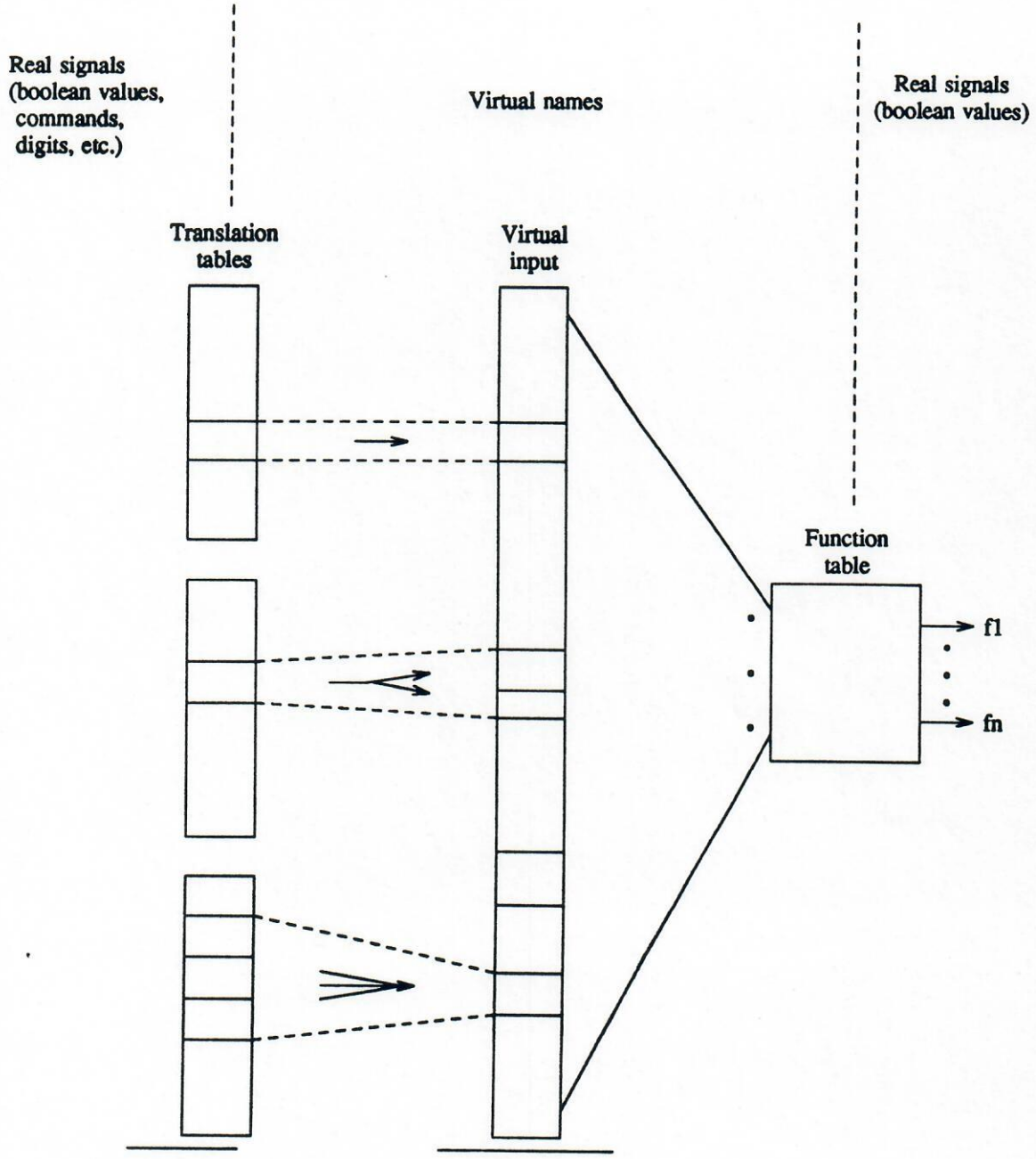


Figure 5. System for calculating logical functions based on "table of sets" representation

6. Implementation example

Implementation of the "table of sets" technique depends on the language used. If the language has the set type and the set operations in its syntax definition, then there exists a direct correspondence between the specification and implementation. The following example presents one implementation of the discussed technique.

Example 5.
A system has several inputs which belong to four categories: cmd (stands for command), digit, count, and timer. It calculates three logical functions:

f_1 = "cmd read came"

f_2 = "cmd idle came" or
"first digit is zero" or
"time_2 elapsed"

f_3 = "first digit came" or
"last digit came and is not zero" and "cmd idle came"

Figure 6 shows all input/output signals and the virtual environment. The implementation in a PASCAL-like language follows. Such details as implementation of the digit counter, the timers, resetting of the counter and timers, etc., are omitted for clarity. The example presents detailed implementation for the "table of sets" specification of logical functions. It consists of:

— constant declarations;

— type declarations (begin with a letter describing the type: e - stands for enumeration, a - for array, s - for set, i - for integer, r - for record);

— variable declarations, which are translation and function tables;

— virtual input and logical function declaration (the first letter v - stands for virtual, c - for constant, as the c_variables are used, in fact, as constants);

— procedure to actualize the virtual input;

— procedure to calculate the logical function.

We note in passing that the program structure is very regular, and hence, well suited for creation from specification by a program generator.

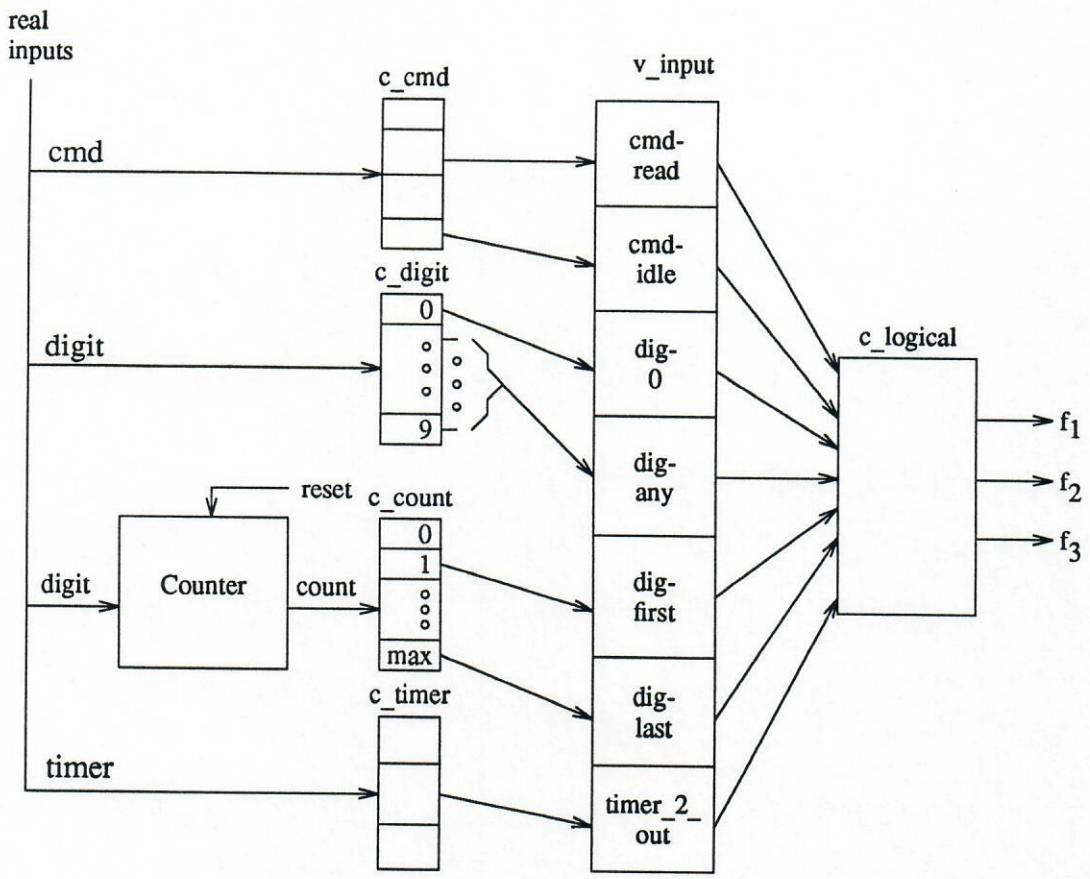


Figure 6. System for calculating functions in the program "example".

```
PROGRAM example;

CONST
    count_max = 15;
    func_max  = 3;
    prod_max  = 3;

TYPE
        { real inputs declaration }
    e_cmd   = ( off, read, write, idle );
    i_digit  = 0..9;
    i_count = 0..count_max;
    e_timer = ( time_1, time_2, delay );

    { events }
    e_event = (cmd, digit, count, timer);
    r_event = RECORD
    CASE event : e_event OF
      cmd     :   (cmd_event : e_cmd);
      digit   :   (digit_event : i_digit);
      count   :   (count_event : i_count);
      timer   :   (timer_event : e_timer);
    END;

        { virtual input declaration }
    e_input = ( cmd_read,
                cmd_idle,
                dig_0,
                dig_any,
                dig_first,
                dig_last,
                tim_2_out );
    s_input = SET OF e_input;

        { translation tables declaration }
    a_cmd   = ARRAY [e_cmd] OF s_input;
    a_digit  = ARRAY [i_digit] OF s_input;
    a_count = ARRAY [i_count] OF s_input;
    a_timer = ARRAY [e_timer] OF s_input;

        { function table declaration }
    a_logical = ARRAY [1..func_max, 1..prod_max] OF s_input;

VAR
        { translation tables }
    c_cmd   : a_cmd   := ([ ], [cmd_read], [ ], [cmd_idle]);
    c_digit  : a_digit  := ([dig_0], 9 OF [dig_any]);
    c_count : a_count := ([ ], [dig_first], (count_max-2) OF [ ], [dig_last]);
    c_timer : a_timer := ([ ], [tim_2_out], [ ]),

        { auxiliary variables }
    c_cmd_all   : s_input := [cmd_read, cmd_idle];
    c_digit_all  : s_input := [dig_0, dig_any];
```

```
    c_count_all : s_input := [dig_first, dig_last];
    c_timer_all : s_input := [tim_2_out];

       { function table }
    c_logical : a_logical := (
       { f1 }          ([cmd_read], 2 OF [ ]),
       { f2 }          ([cmd_idle], [dig_0, dig_first], [tim_2_out]),
       { f3 }          ([dig_first], [dig_last, dig_any, cmd_idle], [ ])
                          );

       { real input }
    inp : r_event;

       { virtual input }
    v_input : s_input;

       { logical functions }
    logical : ARRAY [1..func_max] OF BOOLEAN;

PROCEDURE actualize_virtual_input;

BEGIN
 WITH inp DO
    CASE event OF
      cmd   : v_input := v_input - c_cmd_all + c_cmd [cmd_event];
      digit : v_input := v_input - c_digit_all + c_digit [digit_event];
      count : v_input := v_input - c_count_all + c_count [count_event];
      timer : v_input := v_input - c_timer_all + c_timer [timer_event];
    END;
END;


PROCEDURE calculate_function;

VAR func, prod : INTEGER;

BEGIN
    logical := ZERO;
    FOR func := 1 TO func_max DO
      FOR prod := 1 TO prod_max DO
        IF c_logical [func, prod] <> [ ]        { is not empty }
        THEN logical [func] := logical [func] OR
                               (c_logical [func, prod] <= v_input);
END;

BEGIN { main - program example }
    REPEAT
      wait_on_event_or_read_input; { not defined here }
      actualize_virtual_input;
      calculate_function;
      . . .
    UNTIL FALSE;
END.
```

7. Table-driven finite state machine

The concept of a virtual environment permits complex software to be constructed without specific programs, and has been successfully applied to development of standard modules for control purposes, implementing table-driven state machines.

Figure 7 shows a block diagram of such a state machine. The state machine works in a virtual environment processing input names and producing output names. Tables perform the interfacing to the real world: translating real inputs to input names and output names to real outputs. A transition table describes the functioning of the state machine using virtual names exclusively.

The logic of the State Machine is contained completely in the Virtual Transition Table. The content of the table depends neither on the language used nor on the Data Base or the Operating System. Thus the virtual names, and the tables which use them, create a highly portable environment in which the specification and implementation are merged.

Such state machines are not only rapidly and easily constructed; they have the immense merit of remaining comprehensible to the engineers who have originally specified the system behavior. This stands in contrast to conventional forms of programming, even in the most high-level and specialized of programming languages. Thus it is possible to "maintain" such software modules, in the sense of correcting errors of conception, or of adapting them to meet changed requirements, with far less effort and worry than is the case with conventional software.

This concept has been tested during the design of software for complex control systems. The most complicated system has 2,000 inputs and outputs and consists of 400 interconnected virtual finite state machines. The size of each machine varies from 20 to 60 states. Each system is highly parameterized because of customization requirements.

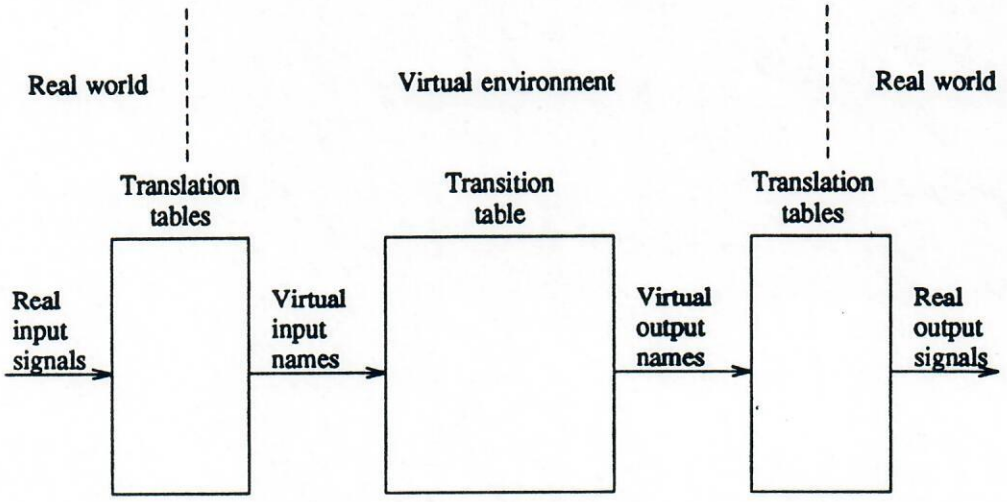


Figure 7. Totally table-driven finite state machine using a concept of virtual environment

8. Conclusions

The technique of representing boolean functions as tables of sets, when combined with the concept of a virtual environment, provides a powerful tool for expressing logical functions of many inputs, the inputs being of a variety of types.

The virtual environment is created by the system designer during the analysis and specification phase. The names which are invented during this phase represent the complete and only control information about the inputs. There is a one-to-one correspondence between the specification of a control problem and its implementation. Hence, automatic code generation is straightforward. A further advantage is that during debugging a designer stays in his specification environment. To summarize - in all stages of software development - from specification to code generation to run time the environment does not change.

Programs written using tables avoid the coding of "if then else" and "case" statements. A program which consists only of assignment and loop statements is easier to read and understand, because the logic of the problem is concentrated in tables for which human comprehensive is higher than for complex "if then else" conditions. Performance of table-based software increases, because tables represent conditions which are prepared off-line and are accessed during the program execution.

It is suggested that the use of software tools such as those described above will be the only way to generate reliable software for safety-critical applications. Rather than searching for formal tools for validation of software, let us rather eliminate the need for such tools, at least in project-specific form, so that no new, and thus incompletely proven, software has ever to enter service.

Acknowledgement

I have tested the virtual environment concept by designing table-driven software for BALZERS AG, a leading manufacturer of semiconductor production equipment. With my colleagues there, we applied this idea to build a hierarchical system of state machines for the purpose of process control. Especially the feedback from M. Matt and R. Schmucki helped me to implement the table-driven state machine. Acknowledgement also to Professor G. Franzkowiak for his contribution. J. Dobrowolski, H. Itzkowitz, L. Mekly, L. Schutte, D. Hong and P. Wolstenholme made many helpful suggestions in the process of writing this paper.

8. Conclusions

The technique of representing boolean functions as tables of sets, when combined with the concept of a virtual environment, provides a powerful tool for expressing logical functions of many inputs, the inputs being of a variety of types.

The virtual environment is created by the system designer during the analysis and specification phase. The names which are invented during this phase represent the complete and only control information about the inputs. There is a one-to-one correspondence between the specification of a control problem and its implementation. Hence, automatic code generation is straightforward. A further advantage is that during debugging a designer stays in his specification environment. To summarize - in all stages of software development - from specification to code generation to run time the environment does not change.

Programs written using tables avoid the coding of "if then else" and "case" statements. A program which consists only of assignment and loop statements is easier to read and understand, because the logic of the problem is concentrated in tables for which human comprehensive is higher than for complex "if then else" conditions. Performance of table-based software increases, because tables represent conditions which are prepared off-line and are accessed during the program execution.

It is suggested that the use of software tools such as those described above will be the only way to generate reliable software for safety-critical applications. Rather than searching for formal tools for validation of software, let us rather eliminate the need for such tools, at least in project-specific form, so that no new, and thus incompletely proven, software has ever to enter service.

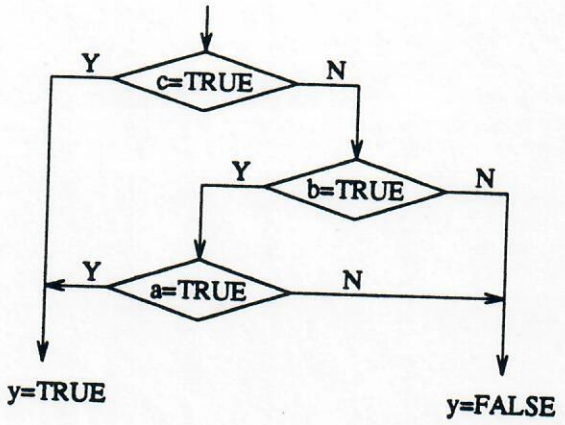
Acknowledgement

I have tested the virtual environment concept by designing table-driven software for BALZERS AG, a leading manufacturer of semiconductor production equipment. With my colleagues there, we applied this idea to build a hierarchical system of state machines for the purpose of process control. Especially the feedback from M. Matt and R. Schmucki helped me to implement the table-driven state machine. Acknowledgement also to Professor G. Franzkowiak for his contribution. J. Dobrowolski, H. Itzkowitz, L. Mekly, L. Schutte, D. Hong and P. Wolstenholme made many helpful suggestions in the process of writing this paper.

APPENDIX A
DIRECT TRANSLATION OF A CONTROL FLOW DIAGRAM INTO CODE

```
FUNCTION Logical (a,b,c : BOOLEAN) : BOOLEAN;

BEGIN
  IF c
  THEN
    Logical := TRUE
  ELSE
    IF b
    THEN BEGIN
      IF a
      THEN
        Logical := TRUE
      ELSE
        Logical := FALSE
    END
    ELSE
      Logical := FALSE
END;
```



APPENDIX B
NOTATION

$\{x_1,\ldots,x_n\}$ is the set comprising elements $x_1,\ldots,x_n$.

If X and Y are sets then:

$X \subseteq Y$ X is a subset of Y, i.e., every element of set X is a member of the set Y

X - Y is the set of elements which are in X but not in Y

X + Y is the set of elements which are in X or in Y

X * Y is the set of elements which are in X and in Y

APPENDIX C

VIRTUAL INPUT SIGNALS

Real inputs are multivalued and represent some physical signals. In the virtual environment they are replaced by a description - names. The following table shows examples of some typical input signals.

SIGNAL	VALUE	NAME
Timer	elapsed	tim_out
Binary	FALSE	di_f
	TRUE	di_t
Digit	0	dig_0, dig_first, dig_last
	1..9	dig_any, dig_first, dig_last
Command	off	cmd_off
	read	cmd_read
	write	cmd_write
	idle	cmd_idle
Parameter	mode = none	mode_none
	mode = current	mode_current
	mode = voltage	mode_voltage
Switchpoint*	high	swip_high, swip_out
	in_range	swip_in
	low	swip_low, swip_out
Other	(sub_sta_1 = off OR sub_sta_1 = idle) AND (sub_sta_2 = off OR sub_sta_2 = idle)	sub_sta_off_or_idle

* A switchpoint represents analog signals for control purposes; it signals whether the analog value is within a predefined band, or is outside that band (high or low).