

**Practical Guidelines for Constructing and Simplifying Tables
for
Finite State Machines (FSMs)
and
Trace Functions (TFs)**

SQRL, Robert L. Baber, 2005 June 14

Table of Contents

1. Introduction	2
2. Finite State Machines (FSMs).....	2
2.1. Brief review of FSMs.....	2
2.2. Normal form of an FSM table.....	3
2.3. Reducing the number of states in an FSM	4
2.4. Grouping rows and columns of an FSM table.....	5
2.5. Merging identical adjacent cells.....	7
2.6. Transition variants and state invariants.....	8
2.7. An industrial example of constructing and manipulating FSM tables.....	8
2.7.1. Constructing an FSM table from an English language statement of the problem....	8
2.7.2. Reducing the FSM table.....	16
2.8. Summary of main guidelines.....	18
3. Trace Functions (TF).....	19
3.1. Trace functions vs. FSMs.....	19
3.2. Commonly used auxiliary functions on traces	20
3.3. An example of a TF.....	20
3.3.1. Constructing a table from the English language specification of the desired TF ..	20
3.3.2. Moving conditions between rows and columns	25
3.4. An industrial example of manipulating TF tables.....	26
3.4.1. Constructing a table from the English language specification of the desired TF ..	26
3.4.2. Systematic examination of the constructed table	32
3.4.3. Restructuring the conditions in the table.....	39
3.4.4. Reordering and grouping rows in the TF table	40
3.5. Review of the main guidelines for constructing and restructuring a TF table	46
Appendix A. Finite State Machines (FSMs): mathematical definitions	48
Appendix B. Equivalence of two FSMs.....	49
Appendix C. Deriving an FSM to Compute a Given Trace Function (TF)	53
Appendix D. References	70



1. Introduction

Requirements and specifications for computer programs must be formulated unambiguously, i.e. mathematically, if misunderstandings are to be avoided and a correct program is to be implemented. Often the specifications must state what sequence of outputs is to be delivered for every possible sequence of inputs. Finite state machines (FSMs) and trace functions (TFs) are convenient mathematical objects for specifying such input-output behaviour.

FSMs and TFs can both be defined with tables. Typically, any particular FSM or TF is characterized by any one of many different but mathematically equivalent tables. One table may best serve one purpose, e.g. communicating between people involved in specifying the program in question, while a different but equivalent table may best serve some other purpose, e.g. communicating between other people or planning and designing the program. It is, therefore, often desirable and useful in practice to construct a table representing the specification for the program and then to transform that table into other equivalent tables.

The goal of this document is, therefore, to help software developers

- to construct tables representing finite state machines (FSMs) and trace functions (TFs) and
- to manipulate these tables into simpler or more convenient but mathematically equivalent tables.

This document is written for persons involved directly or indirectly with software development in industrial or commercial practice. Some general mathematical knowledge is assumed, but no mathematical or computer expertise in any particular subarea is assumed.

2. Finite State Machines (FSMs)

2.1. Brief review of FSMs

Informally, the FSM is based on the idea of a mechanism or process. The mechanism or process is in some *state* and upon receiving an *input*, the mechanism or process emits an *output* and undergoes a transition to a new state. Both the output and the new state depend upon the previous state and the input.

More formally, an FSM consists of a combination of

- a set S of states,
- a set IN of inputs,
- a set OUT of outputs,
- a function *output* mapping a state and an input to an output ($output : S \times IN \rightarrow OUT$),
- a function *nextstate* mapping a state and an input to a state ($nextstate : S \times IN \rightarrow S$) and
- an initial state s_0 , an element of the set S ($s_0 \in S$).

An FSM defines a function that maps a sequence of inputs to a sequence of outputs, i.e. a function that maps a sequence of elements of IN to a sequence of elements of OUT . An FSM is an iterative definition of this function. The states are intermediate terms in this definition.

See Appendix A below for mathematical definitions of an FSM, its function mapping an input sequence to an output sequence, and an example illustrating the relationships between an FSM's inputs and outputs.

The sets IN and OUT are often called the *input alphabet* and the *output alphabet* respectively.

In practical applications, the notion of “output” is often interpreted loosely or generally. An output can be a single symbol or a sequence of symbols, including an empty sequence. An output can also be interpreted as a command to perform some function.

2.2. Normal form of an FSM table

A normal table for an FSM contains one row for each state and one column for each input and, in addition, a header for the rows and a header for the columns. The header for the rows is itself a column, each cell of which contains the state. Correspondingly, the header for the columns is itself a row, each cell of which contains the input.

Each cell in the body of the table (i.e. every cell not in either of the headers) contains the output and the next state.

Example: The input to a program consists of a sequence of digits (0, 1, ... 9). If the first five digits in the input sequence are the digits 1 through 5 inclusive and in ascending order, the program should output “C” (for correct). If the first five digits are anything else, the program should output “E” (for error). Until one of these results has been determined, the program should output nothing (an empty string). After either “C” or “E” has been output once, the program should output nothing (an empty string) in response to additional input.

One possible FSM to satisfy this requirement is represented by the following table, where the initial state is state 1.

		Input digit									
		0	1	2	3	4	5	6	7	8	9
State	1 (initial)	7 “E”	2	7 “E”	7 “E”	7 “E”	7 “E”	7 “E”	7 “E”	7 “E”	7 “E”
	2	7 “E”	7 “E”	3	7 “E”	7 “E”	7 “E”	7 “E”	7 “E”	7 “E”	7 “E”
	3	7 “E”	7 “E”	7 “E”	4	7 “E”	7 “E”	7 “E”	7 “E”	7 “E”	7 “E”
	4	7 “E”	7 “E”	7 “E”	7 “E”	5	7 “E”	7 “E”	7 “E”	7 “E”	7 “E”
	5	7 “E”	7 “E”	7 “E”	7 “E”	7 “E”	6 “C”	7 “E”	7 “E”	7 “E”	7 “E”
	6	8	8	8	8	8	8	8	8	8	8
	7 (error)	7	7	7	7	7	7	7	7	7	7
	8 (correct)	8	8	8	8	8	8	8	8	8	8

When the FSM is in the state shown in the row header to the left and the input is the digit shown in the column header at the top, the cell in the corresponding row and column gives the next state of the FSM and the output the FSM emits. In those cases in which no output appears, the FSM outputs nothing (the empty string).

Notice that the size of a normal FSM table varies linearly with the number of different states and the number of inputs. In practical applications, both numbers, but especially the number of states, can become very large, leading to large and cumbersome tables. Techniques for reducing the size of FSM tables are, therefore, important in practice. These techniques fall into two categories, (1) reducing the number of states and (2) reducing the number of rows or columns without reducing the number of states. The second category reduces the size of the

table by grouping states (rows) or inputs (columns) in the table. Some of these techniques are illustrated in sections 2.3 and 2.4 below respectively.

2.3. Reducing the number of states in an FSM

Note that if the FSM in the last table in section 2.2 above is in either state 6 or state 8 and receives an input digit, the FSM outputs nothing and goes to state 8. I.e., the future behaviour of the FSM is identical once it reaches either state 6 or state 8. The two states 6 and 8 are, therefore, equivalent. They can be combined to eliminate one state, giving the following equivalent FSM.

		Input digit									
		0	1	2	3	4	5	6	7	8	9
State	1 (initial)	7 "E"	2	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"
	2	7 "E"	7 "E"	3	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"
	3	7 "E"	7 "E"	7 "E"	4	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"
	4	7 "E"	7 "E"	7 "E"	7 "E"	5	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"
	5	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	6 "C"	7 "E"	7 "E"	7 "E"	7 "E"
	6 (correct)	6	6	6	6	6	6	6	6	6	6
	7 (error)	7	7	7	7	7	7	7	7	7	7

Once this FSM reaches either state 6 or state 7, it outputs nothing and remains in the same state. I.e., the future behaviour of this FSM is identical once it reaches either state 6 or state 7. The two states 6 and 7 are, therefore, equivalent. They can be combined to eliminate another state, giving the following equivalent FSM.

		Input digit									
		0	1	2	3	4	5	6	7	8	9
State	1 (initial)	6 "E"	2	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"
	2	6 "E"	6 "E"	3	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"
	3	6 "E"	6 "E"	6 "E"	4	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"
	4	6 "E"	6 "E"	6 "E"	6 "E"	5	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"
	5	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "C"	6 "E"	6 "E"	6 "E"	6 "E"
	6 (end)	6	6	6	6	6	6	6	6	6	6

In general, two or more states are equivalent if, for every possible input, the FSM's output is the same and the next states are equivalent. To reduce the number of states in an FSM, one examines the table for such a pattern and postulates that the states in each such group identified are equivalent. Then one verifies that the condition in the first sentence of this

paragraph is satisfied for all the states in each group of postulated equivalent states. If this is the case, then the postulated equivalent states are actually equivalent.

It is apparent that no grouping of states in the last FSM table above satisfies the criteria for reducing the number of states. The above FSM, with its 6 states, cannot be reduced any further.

Reducing the number of states in an FSM and the equivalence of two FSMs are dealt with mathematically and in more detail in Appendix B.

2.4. Grouping rows and columns of an FSM table

In the FSM tables above, the columns for the input digits 0, 6, 7, 8 and 9 are identical. These columns can, therefore, be combined into one column representing all of these digits. The conditions in the headers of the columns being combined must be or-ed together to obtain the condition for the combined column. The FSM table then becomes:

		Input digit					
		1	2	3	4	5	other (0, 6, 7, 8 or 9)
State	1 (initial)	2	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"
	2	6 "E"	3	6 "E"	6 "E"	6 "E"	6 "E"
	3	6 "E"	6 "E"	4	6 "E"	6 "E"	6 "E"
	4	6 "E"	6 "E"	6 "E"	5	6 "E"	6 "E"
	5	6 "E"	6 "E"	6 "E"	6 "E"	6 "C"	6 "E"
	6 (end)	6	6	6	6	6	6

Notice the similar structure of the rows for states 1, 2, 3 and 4. If the number of the previous state is the same as the input digit, then the FSM goes to the next state in sequence and outputs nothing. Otherwise, "E" is output and the FSM goes to state 6. For state 5, the same structure is present, but "C" is output instead of nothing. In the case of state 6, the response is the same no matter what the input is, so input digits may be grouped in any convenient way. These observations suggest parameterizing the state and the input digit and introducing a case distinction in the header for the input digit, in order to make the rows identical, so that they may be combined. The steps in this process are shown below.

		Input digit d	
		d=s	d≠s
State s	1 (initial)	s+1	6 "E"
	2	s+1	6 "E"
	3	s+1	6 "E"
	4	s+1	6 "E"
	5	s+1 "C"	6 "E"
	6 (end)	6	6

The rows for states 1, 2, 3 and 4 are now identical, so they can be combined in the same way that columns were combined in the previous example. Correspondingly, the conditions in the headers of the rows being combined must be or-ed together to obtain the condition for the combined row. The expression $(s=1 \vee s=2 \vee s=3 \vee s=4)$ can also be written $(s \in Z \wedge 1 \leq s \leq 4)$, leading to the following table for this FSM.

		Input digit d	
		d=s	d≠s
State s	$s \in Z \wedge$ $1 \leq s \leq 4$	s+1	6 "E"
	s=5	s+1 "C"	6 "E"
	s=6 (end)	6	6

The last FSM table above has only three rows in the body. This number would not increase if the number of states were increased, e.g. to verify that the first 6, 7, 8 or 9 input digits are consecutive and increasing. Reducing or eliminating the table size's dependence on the number of states can be important and can even make the difference between an FSM specification being feasible or infeasible in a particular application.

Note that in the last FSM table above, there are still 6 different states, even though there are only 3 rows. The first row groups 4 different states, and these states are not equivalent. This technique for reducing the number of rows in an FSM table does not reduce the number of states in the FSM. Similarly, there are still 10 different input digits, even though the table above has only 2 columns.

Note that in all of the tables above, the conditions in the row headers are mutually exclusive and exhaustive, that is, no two conditions overlap (can be true simultaneously), and together, they cover the entire range of the variables in question. The same applies to the conditions in the column headers. In general, one should ensure that the conditions in the headers of a table satisfy these properties. It is not absolutely necessary that they do so, but if they do not, then

one must verify consistency of the entries in the cells and one must add entries for uncovered cases.

Generalizing expressions involving states and inputs can be carried to an extreme. In fact, by rewriting the expressions in the cells in the body of an FSM table with conditional expressions appropriately, one can make all of the cells identical. The body of the table can then be reduced to a single cell. Such a single cell table can be difficult to read, so this extreme is not necessarily the best choice. In fact, it is seldom the best choice.

The best choice for compacting an FSM table often depends on the intended readership and the purpose of the table. A highly compressed table such as the last one above is often very useful for purposes of analyzing the requirements and designing the program. A less compressed table, such as one of the earlier ones presented above, might appeal more to some readers, whose main desire is to understand the specification and assess its suitability for the application in question.

2.5. Merging identical adjacent cells

When horizontally or vertically adjacent cells in a table have identical contents, such cells can be merged into one. This often highlights certain relationships and makes the table more readable.

For example, in the table

		Input digit d	
		d=s	d≠s
State s	$1 \leq s \leq 4$	s+1	6 "E"
	s=5	s+1 "C"	6 "E"
	s=6 (end)	6	6

the two cells at the lower right have the same contents, so they can be merged. The two cells at the upper right of the body of the table can also be merged. The above table can, therefore, be rewritten as:

		Input digit d	
		d=s	d≠s
State s	$1 \leq s \leq 4$	s+1	6 "E"
	s=5	s+1 "C"	
	s=6 (end)	6	

Often – but not always – merging cells in this way improves the readability of the table. As pointed out above, the preferred form can depend on the intended purpose. One form might be better for reading by people, while another form might be more convenient for analysis or program design purposes.

2.6. Transition variants and state invariants

Often it is possible and useful to define an expression whose value is either never decreased, never increased, always increased or always decreased by state transitions. State variables and possibly other variables appear in such expressions. The values of such expressions must be elements of an ordered set. An expression of this type is called a transition variant or simply a *variant*. (To be precise, the word “monotonic” should be used in connection with “variant”, but it is, by convention, understood and omitted.) In order to prove that a given expression is a variant of the FSM in question, one must examine every state transition, i.e. every cell in the body of the table, and show that under the conditions in the headers of the row and column in question, the value of the expression changes in a way according to the type of variant.

A particularly simple example is the expression s for the last table in section 2.5 above. Every state transition either increases the value of s or leaves the value of s unchanged (e.g. in state 6). Therefore, the expression s is a never decreasing variant.

A state *invariant* is a Boolean expression (an expression that evaluates to either “false” or “true”) whose value is always true, i.e. is true in every state. To prove that a given expression is an invariant, one must show that

- its value is true in the initial state and that
- if it is true before every state transition, it is true after that transition.

The last table in section 2.5 above gives a trivial example of a state invariant: $1 \leq s \leq 6$. The initial value of s is 1. Its value never decreases, so it is always true that $1 \leq s$. No transition increases its value to more than 6, so $s \leq 6$. Thus, $1 \leq s \leq 6$ is true before and after every state transition. If the table above contained a row for state $s=7$, this invariant would show that the state $s=7$ cannot be reached and could be removed. In this simple case this is obvious, but complex tables exist in which it is not so clear that a state is unreachable and, therefore, can be removed from the table.

Important characteristics of an FSM can often be proved using a transition variant or a state invariant, e.g., that a particular state cannot be reached from the initial state, that a particular state variable is redundant and can be eliminated, or that a table can be simplified in a certain way. Examples in section 2.7 and Appendix C below illustrate some uses of transition variants and state invariants.

2.7. An industrial example of constructing and manipulating FSM tables

2.7.1. Constructing an FSM table from an English language statement of the problem

This example is the construction of a slight variation of Table 4 in [1] based on the English language specification in sections 2 and 4.1 in [1]. This FSM is part of the specification of a program in industrial use. The application in question involves testing a keyboard and its connection to a computer by manually pressing its keys in order. See also section 3.4.1 below.

To transform an English language specification into a mathematically precise definition of an FSM, one must (1) identify suitable state variables of the FSM and (2) for each state and each input term, determine the next state and the output. A person with limited or no prior

experience extracting such information from English text will find it best to analyze the text in at least two passes, identifying the state variables in the first pass and completing the table showing the next states and the output in the second pass. One or more previous passes, just to read the text, is sometimes also advisable. With experience one will often be able to transform the English text into an FSM table in fewer passes through the text.

To identify the state variables of the FSM, the following tactics are generally useful.

Tactic 1: To identify candidates for state variables for an FSM corresponding to a statement of requirements written in natural language, look for *noun phrases or clauses representing significant conditions or situations that change during the test and upon which the next step in the test depends*. The clauses should be clauses of being or state, not action, i.e. whose verb represents a state (e.g. a form of “to be”), not an action. The clause can be abbreviated, e.g. with the verb absent but implicit. Especially if the requirements text refers to steps in the process of calculating the desired result is this tactic likely to be effective.

A verb of action is seldom if ever appropriate in a phrase or clause identifying candidates for state variables. If a verb of action seems to be appropriate, use its (passive) past participle in the role of an adjective. Do not use its (active) present participle. (This guideline might have to be further qualified for a language which, unlike English, has both present and past participles in both the passive and the active voice.)

Tactic 2: Check for redundancy among state variables as follows. After identifying candidate state variables, read the text carefully to identify invariant relationships between the state variables. For each state variable, check whether or not its value can always be determined from the values of the other state variables. If it can, that state variable is redundant and is not needed.

Because the analysis outlined above is based on natural language text, which might contain not yet detected ambiguities or inconsistencies, one might not want to eliminate apparently redundant state variables yet. In this case, note (1) the tentative state invariant expressing the redundancy and (2) the suspected redundancy. After a formal mathematical specification has been formulated, analyze it with the goal of identifying and eliminating redundant state variables, e.g. using a suitable invariant.

When constructing an FSM table, one first identifies the state variables as described above. Then, one looks for information on the values of the next state and the output and the corresponding *conditions* on the previous state and input term. To do this, one looks in the text for conditions upon which the subsequent behaviour of the FSM depends and the behaviour under those conditions. The conditions relate to the previous state and the input term, i.e. identify the relevant row and column. The behaviour described in the text relates to the next state and the output term, i.e. the contents of the cell in the relevant row and column.

Note that the process outlined above and illustrated with the example below is neither precisely defined nor completely mechanistic. However, these guidelines help considerably, and make an otherwise very implicit process much more systematic and explicit and, thereby, also more reproducible.

The original English version of the specification for this industrial example appears in **blue Helvetica font** below. The interspersed passages in black normal font are intermediate

comments and interpretation of the original text written in the process of formulating a mathematically precise specification.

The main requirements of the keyboard sequence test are the following:

- Keys of a keyboard should be pressed in a specified order during keyboard testing, i.e. there is only one key that is expected to be pressed at each moment (the expected key).
- The inputs to the program are described by the sequence number of the key pressed.

We assume that the “expected” key is the key that should be pressed next. This noun phrase *expected key* fulfills the criteria of Tactic 1 above, so is a candidate state variable.

We further assume that the last bulleted item above means that the key identification numbers are the sequence numbers, i.e. that the identification number for the first key that should be pressed is “1”; for the second key that should be pressed, “2”; for the third key that should be pressed, “3”, etc. We will, therefore, represent the state for the *expected key* by the variable n , which will take on integer values beginning with 1.

- The test software provides interactive test diagnostics. An image of the keyboard is displayed on the screen. The expected key should be highlighted with a yellow box on the screen. When the next key in the sequence is pressed, the background colour of the key changes to blue, to indicate that the key has been successfully tested.

This part of the English text does not contain any noun phrase or clause of the type mentioned in Tactic 1, so does not contribute to identifying state variables. It does raise some questions of likely relevance later, for example: How should the images of the other keys be displayed? Is the desired display a function of the expected key only?

- Depending on the most recently pressed keys, there are three possible variations from the old to the new value of the expected key after some key has been pressed. For this purpose, the keys are divided into three groups:
 - The next key in the sequence (the expected key).
 - The escape key, but only if not the expected key.
 - Any other key (an incorrect key).
- If the expected key has been pressed, the next key in the test sequence becomes the expected key, i.e. the number of the expected key is increased by 1.
- If the escape key has been pressed, the number of the expected key remains the same.

The last two bulleted items above contradict each other in the case that the expected key is the escape key. Further above, the phrase “escape key, but only if not the expected key” appears. We assume that this qualification is intended to apply also to the last bulleted item above, i.e. that it was intended to read “If the escape key has been pressed *unexpectedly*, the number of the expected key remains the same.”

The clause “the number of the expected key is increased by 1” raises the question “What if there is no key on the keyboard corresponding to the resulting number?” This question highlights a significant gap in this English specification. We must keep this question in mind when examining the rest of the English specification.

The three bulleted items above contain three noun phrases of possible interest:

- “next key in the sequence” – synonymous with the already identified noun phrase *expected key*,
- “escape key (pressed unexpectedly)” and
- “other key” or, synonymously, “incorrect key”

In the above passage, these noun phrases are used in a way more suggestive of conditions for determining the next state (the next expected key), rather than of additional state variables. Because these conditions do affect the next state and, hence, the future behaviour of the FSM, we will keep them in mind while using them now as conditions for determining the next state:

state =	expected key pressed	escape key pressed unexpectedly	incorrect key pressed
n	n+1	n	?

In the table above, the row header (on the left) contains a condition on the previous state, the column header (on the top) contains conditions on the key most recently pressed, and each cell in the body of the table contains the next state when the conditions in the corresponding row and column header apply.

Tactic 3: Look for *conditions* on the previous state and input term to identify conditions for row and column headers. Later parts of the same sentence or subsequent sentences will typically indicate the next state and output to enter into the corresponding cell in the body of the table.

The column header in the above table can be restructured to reduce the possible ambiguity, leading to the following table:

state =	key pressed		
	expected	not expected	
		escape	not escape
n	n+1	n	?

- If an incorrect (unexpected) key has been pressed first time after the expected key, the previous key becomes expected, i.e. the number of the expected key is decreased by 1.
- If an incorrect key is pressed later, the expected key remains unchanged until the expected key is pressed.

Again here a contradiction can arise. If the most recently pressed key is unexpected but the escape key, and the previously pressed key was the expected one, the first of the two bulleted items above states that the new value of *expected key* (the state variable n) should be n-1. An earlier bulleted item above states that the value of n should remain unchanged. Further above a definition of “incorrect” is given: unexpected and not the escape key. We assume that this definition is meant here also, i.e. that the last bulleted item above should begin “**if an incorrect (unexpected and not the escape) key has been pressed ...**”.

The two bulleted items above are not completely clear regarding how unexpected depressions of the escape key are to be regarded. Presumably they mean that if, after an expected key is pressed, pressing an incorrect key will cause the value of n (representing the expected key) to be decreased by one, but only once. Subsequent depressions of an incorrect key will leave the

value of n unchanged, until an expected key is pressed, starting this cycle over. Unexpected depressions of the escape key are disregarded in this, as they leave the value of n unchanged.

What precisely does the word “later” mean in the last bulleted item above? Is the phrase “until the expected key is pressed” significant? Considering the context set by the previous bulleted item above, “later” presumably means “after an incorrect” key was pressed. This is an assumption that should be verified with appropriate persons.

We can rewrite the two bulleted items above into a form describing the condition or situation prevailing before the depression of the incorrect key in question as follows. The color green below indicates a reformulated version of the original English text.

- If no incorrect key has been pressed since the most recent depression of an expected key, pressing an incorrect key results in decreasing the expected key by 1.
- Otherwise, pressing an incorrect key again results in no change in the expected key, until the expected key is subsequently pressed, in which case the above bulleted item applies again.

A clause of state representing a condition or situation that can change during the keyboard test and upon which subsequent steps depend (cf. Tactic 1 above) appearing in the above text is: “no incorrect key has been pressed since the most recent depression of an expected key”. Positively expressed conditions are often easier to read and understand than negatively expressed ones, especially when the analysis leads to an additional negation being applied, as is often the case. We choose, therefore, “*an incorrect key has been pressed since the most recent depression of an expected key*” as a clause of state defining a new state variable, which we will represent by the variable name w . The state variable w will take on Boolean values (“F” for false, “T” for true). Initially, no key has been pressed, so the initial value of w is F.

Alternative definition: At this point it should be noted that this new state variable w could be defined differently, in terms of its effect, i.e. whether pressing an incorrect key should cause n to be decreased by one or not. This view would lead to defining the new variable as the truth value of the statement “reduction of n required on incorrect key” or, formulated negatively, “no reduction of n required on incorrect key”. A state variable defined by the latter statement would be equivalent to the state variable defined in the previous paragraph above except that its initial value would be T. We will not pursue this possible alternative further here, but later, in section 2.7.2 below, we will see that the approach we do follow leads to an FSM that can be simplified to the one that would result from this alternative definition.

Incorporating the new state variable w and the additional rules for determining the next state into the above table, we obtain:

state (n, w) =	key pressed		
	expected	not expected	
		escape	not escape (“incorrect”)
(n, F)	($n+1, F$)	(n, F)	($n-1, T$)
(n, T)	($n+1, F$)	(n, T)	(n, T)

The initial state is, as mentioned above, $n=1$ and $w=F$, i.e. the state (1, F).

If the first key pressed is not the expected key and not the escape key, the above table indicates that the value of n (the expected key), which is initially 1, should be reduced to 0. However, there is no key numbered 0, so this does not make sense. We must, apparently, distinguish between states in which $n=1$ and states in which $n>1$. If no later statement in the English version of the specification clarifies this situation, we will have to assume some appropriate next state.

Introducing a distinction between $n=1$ and $n>1$, our table becomes:

state (n, w) =	key pressed		
	expected	not expected	
		escape	not escape ("incorrect")
(1, F)	(2, F)	(1, F)	(1?, T)
(1, T)	(2, F)	(1, T)	(1, T)
(n, F), where $1 < n$	(n+1, F)	(n, F)	(n-1, T)
(n, T), where $1 < n$	(n+1, F)	(n, T)	(n, T)

where the question mark indicates a situation not resolved by the English specification.

- The only way to pass the keyboard sequence test is to successfully test all keys.

Almost hidden here is the answer to the question above, "What if there is no key on the keyboard corresponding to the resulting number?", i.e. what if the value of n is greater than the number of keys on the keyboard? The bulleted item above seems to say that if the last key on the keyboard is pressed when it is the expected key, then the keyboard passes the test. This suggests the rule: As soon as the value of n reaches $L+1$, where L is the sequence number (the identification number) of the last key on the keyboard, the keyboard passes the test.

The output of the FSM being constructed has not been mentioned yet in the English specification. We assume here that when the keyboard has passed the test, the FSM should output the string "Pass", and only once. Similarly, we assume that when the test fails, the FSM should output the string "Fail", again, only once. These assumptions should be verified with the client.

Detecting passing the test requires introducing an additional distinction on n in the table. When and only when $n=L$ and the expected key (L) is pressed, then the output "Pass" should be generated. The table then becomes:

state (n, w) =	key pressed		
	expected	not expected	
		escape	not escape ("incorrect")
(1, F)	(2, F)	(1, F)	(1?, T)
(1, T)	(2, F)	(1, T)	(1, T)
(n, F), where $1 < n < L$	(n+1, F)	(n, F)	(n-1, T)
(n, T), where $1 < n < L$	(n+1, F)	(n, T)	(n, T)
(L, F)	(L+1, F) "Pass"	(L, F)	(L-1, T)
(L, T)	(L+1, F) "Pass"	(L, T)	(L, T)

In two cells of the body of the table above we have introduced a new state, (L+1, F). We must add a row for this state. The English specification says nothing about how the FSM should behave once this state is reached. Under our assumption above (that the output "Pass" or "Fail" should be generated only once), the FSM should never produce a subsequent output. We can define the entries for the new state (L+1, F) so that any input will leave the FSM in this state, without emitting any output. The table becomes:

state (n, w) =	key pressed		
	expected	not expected	
		escape	not escape ("incorrect")
(1, F)	(2, F)	(1, F)	(1?, T)
(1, T)	(2, F)	(1, T)	(1, T)
(n, F), where $1 < n < L$	(n+1, F)	(n, F)	(n-1, T)
(n, T), where $1 < n < L$	(n+1, F)	(n, T)	(n, T)
(L, F)	(L+1, F) "Pass"	(L, F)	(L-1, T)
(L, T)	(L+1, F) "Pass"	(L, T)	(L, T)
(L+1, F)	(L+1, F)	(L+1, F)	(L+1, F)

- The only way to fail the keyboard sequence test is to press the escape key twice in a row.

Again, we assume that by "press the escape key twice in a row" here the qualification on "escape" is meant: "press the escape key unexpectedly twice in a row". This assumption should be verified with the client.

We need to reformulate the above part of the English specification to distinguish between conditions or situations prevailing when the escape key is (unexpectedly) pressed and upon which the failure or non-failure depends. One possibility is:

- If the most recently pressed key was unexpectedly the escape key, and then the next key pressed is also, unexpectedly, the escape key, the keyboard fails the test. This is the only way the keyboard test can fail.

The clause of being or state that describes a condition or situation upon which the effect of pressing the escape key unexpectedly depends is *the most recently pressed key was unexpectedly the escape key*. We will, therefore, use this clause as a definition of a new state variable that we will introduce and name “e”. This variable will take on the values F or T, depending on whether the clause is false or true, respectively. Initially, the clause is false, so the initial value of the state variable e is F.

Adding this new state variable to the table above results in the following table for the FSM we are constructing. If, in a state in which e=T, the escape key is pressed unexpectedly, the FSM should output “Fail” and go to the state in which no more output can be generated, i.e. (L+1, F, F). Otherwise, the next state is as given in the table above with the value of the new state variable e being either T or F, depending on whether the key just pressed was unexpectedly the escape key or not, respectively – except in the state in which no more output can be generated, in which the key just pressed is disregarded.

state (n, w, e) =	key pressed		
	expected	not expected	
		escape	not escape (“incorrect”)
(1, F, F) (initial state)	(2, F, F)	(1, F, T)	(1?, T, F)
(1, T, F)	(2, F, F)	(1, T, T)	(1, T, F)
(1, F, T)	(2, F, F)	(L+1, F, F) “Fail”	(1?, T, F)
(1, T, T)	(2, F, F)	(L+1, F, F) “Fail”	(1, T, F)
(n, F, F), where 1<n<L	(n+1, F, F)	(n, F, T)	(n-1, T, F)
(n, T, F), where 1<n<L	(n+1, F, F)	(n, T, T)	(n, T, F)
(n, F, T), where 1<n<L	(n+1, F, F)	(L+1, F, F) “Fail”	(n-1, T, F)
(n, T, T), where 1<n<L	(n+1, F, F)	(L+1, F, F) “Fail”	(n, T, F)
(L, F, F)	(L+1, F, F) “Pass”	(L, F, T)	(L-1, T, F)
(L, T, F)	(L+1, F, F) “Pass”	(L, T, T)	(L, T, F)
(L, F, T)	(L+1, F, F) “Pass”	(L+1, F, F) “Fail”	(L-1, T, F)
(L, T, T)	(L+1, F, F) “Pass”	(L+1, F, F) “Fail”	(L, T, F)
(L+1, F, F)	(L+1, F, F)	(L+1, F, F)	(L+1, F, F)

Reorganizing the table above to separate the conditions on n from the state representation in the left column and representing the sequence number of the key pressed by the variable “k”, we rewrite the table as follows.

Conditions	Previous state (n, w, e)	k = n	k ≠ n	
			k = esc	k ≠ esc
n = 1	(1, F, F) (initial)	(2, F, F)	(1, F, T)	(1?, T, F)
	(1, T, F)	(2, F, F)	(1, T, T)	(1, T, F)
	(1, F, T)	(2, F, F)	(L+1, F, F) "Fail"	(1?, T, F)
	(1, T, T)	(2, F, F)	(L+1, F, F) "Fail"	(1, T, F)
1 < n < L	(n, F, F)	(n+1, F, F)	(n, F, T)	(n-1, T, F)
	(n, T, F)	(n+1, F, F)	(n, T, T)	(n, T, F)
	(n, F, T)	(n+1, F, F)	(L+1, F, F) "Fail"	(n-1, T, F)
	(n, T, T)	(n+1, F, F)	(L+1, F, F) "Fail"	(n, T, F)
n = L	(L, F, F)	(L+1, F, F) "Pass"	(L, F, T)	(L-1, T, F)
	(L, T, F)	(L+1, F, F) "Pass"	(L, T, T)	(L, T, F)
	(L, F, T)	(L+1, F, F) "Pass"	(L+1, F, F) "Fail"	(L-1, T, F)
	(L, T, T)	(L+1, F, F) "Pass"	(L+1, F, F) "Fail"	(L, T, F)
n = L+1	(L+1, F, F)	(L+1, F, F)	(L+1, F, F)	(L+1, F, F)

Except for the order of the rows and of the columns (which has no effect on the meaning of the table) and the cells for the transitions from the states (L, T, F) and (L, T, T), this table is the same as the first table in section 2.7.2 below, which is, in turn, a slight variation of Table 4 in [1]. The states (L, T, F) and (L, T, T) are unreachable from the initial state (1, F, F), so the contents of the cells for these states are irrelevant, see section 2.7.2 below.

In summary, this section 2.7.1 shows how the FSM table above can be deduced from the English language version of the specification for the keyboard test.

2.7.2. Reducing the FSM table

The following example illustrates the application of combinations of the techniques introduced in earlier sections above. This example is a slight variation of Table 4 in [1]. The FSM is part of the specification of a program in industrial use.

The state of the FSM is given by the values of three state variables (n, w, e). The state variable n is an integer between 1 and L+1 inclusive, where L > 1. The state variables w and e have the values F (false) or T (true). The initial state is (1, F, F). The sequence number of the key pressed is represented by the variable "k" in the table below.

The cells containing "N/A" represent transitions that cannot occur, because the states (L, T, F) and (L, T, T) are unreachable, i.e. cannot arise. This is proved below with the help of a state invariant.

Conditions	Previous state (n, w, e)	k = n	k ≠ n	
			k ≠ esc	k = esc
n = 1	(1, F, F) (initial)	(2, F, F)	(1, T, F)	(1, F, T)
	(1, F, T)	(2, F, F)	(1, T, F)	(L+1, F, F) “Fail”
	(1, T, F)	(2, F, F)	(1, T, F)	(1, T, T)
	(1, T, T)	(2, F, F)	(1, T, F)	(L+1, F, F) “Fail”
1 < n < L	(n, F, F)	(n+1, F, F)	(n-1, T, F)	(n, F, T)
	(n, F, T)	(n+1, F, F)	(n-1, T, F)	(L+1, F, F) “Fail”
	(n, T, F)	(n+1, F, F)	(n, T, F)	(n, T, T)
	(n, T, T)	(n+1, F, F)	(n, T, F)	(L+1, F, F) “Fail”
n=L	(L, F, F)	(L+1, F, F) “Pass”	(L-1, T, F)	(L, F, T)
	(L, F, T)	(L+1, F, F) “Pass”	(L-1, T, F)	(L+1, F, F) “Fail”
	(L, T, F)	N/A	N/A	N/A
	(L, T, T)	N/A	N/A	N/A
n=L+1	(L+1, F, F)	(L+1, F, F)	(L+1, F, F)	(L+1, F, F)

If $w=T$, then $n<L$. Reformulating this as a mathematical expression gives one of the several state invariants for this FSM:

$$w=T \Rightarrow n<L$$

This expression is clearly true in the initial state. The proposition that every cell’s transition maintains the truth of this expression can be verified straightforwardly. Examine every cell whose transition results in $w=T$. In each such case, the resulting value of n is less than L (because $1<L$). Therefore, the value of the expression ($w=T \Rightarrow n<L$) is true after every state transition. If $n=L$ and $w=T$, the value of this expression would be false, which cannot be the case, so the two states (L, T, F) and (L, T, T) can never be reached. Cells in the body of the table in the corresponding rows represent transitions that cannot occur. This is indicated by the entry “N/A” in these cells.

The transitions and the outputs from the states $(1, F, T)$ and $(1, T, T)$ are identical. Therefore, these two states are equivalent. Because these states are equivalent, states $(1, F, F)$ and $(1, T, F)$ are also equivalent to each other (but not to $(1, F, T)$ and $(1, T, T)$). If the initial state is changed from $(1, F, F)$ to its equivalent state $(1, T, F)$, the states $(1, F, F)$ and $(1, F, T)$ are unreachable * (see the next paragraph below), i.e. no sequence of inputs will ever cause the FSM to enter either of these states. The transitions and outputs in the cells of the rows for these two states are, therefore, arbitrary and can be changed to anything. If changed to the rows for the states (n, F, F) and (n, F, T) , the four rows for the states in which $n=1$ become the same as the four rows for the states $1<n<L$, and these two groups of four rows each can be combined. Thus, the first four rows of the table can be eliminated.

* One way of proving that the states $(1, F, F)$ and $(1, F, T)$ are unreachable from state $(1, T, F)$ is to identify an appropriate transition variant and to show that a transition or a sequence of transitions from $(1, T, F)$ to either $(1, F, F)$ or $(1, F, T)$ is inconsistent with the variant. Notice that whenever n is decreased, w is “increased” from F to T . This suggests considering the

expression whose value is n if $w=F$ or $n+1$ if $w=T$. If we define the function val such that $val(F)=0$ and $val(T)=1$, we can write the desired expression as $n+val(w)$. By examining the table above it can be seen that the value of this expression is never decreased by any transition. But the value of this variant in the state $(1, T, F)$ is $1+val(T)=2$, while the value of this variant in either state $(1, F, F)$ or $(1, F, T)$ is $1+val(F)=1$. A transition from state $(1, T, F)$ to either state $(1, F, F)$ or $(1, F, T)$ would require the value of the never decreasing variant to decrease, which is not possible. Thus, the states $(1, F, F)$ or $(1, F, T)$ are unreachable from state $(1, T, F)$.

In terms of the state reduction procedure and the theorem in Appendix B below,

S_1 is the set of all states in the table above

$S_2 = S_1 \setminus \{(1, F, F), (1, F, T)\}$,

$r((1, F, F)) = (1, T, F)$,

$r((1, F, T)) = (1, T, T)$,

$r(s) = s$, for every other $s \in S_1$, i.e. for every $s \in S_2$

$output_2(s, i) = output_1(s, i)$ for every $s \in S_2$ and every $i \in \mathbb{IN}$,

$nextstate_2(s, i) = r(nextstate_1(s, i))$ for every $s \in S_2$ and every $i \in \mathbb{IN}$, and

$s_{20} = (1, T, F)$, i.e. $s_{20} = r(s_{10})$.

Note that the above satisfies all the common application conditions except d – the initial states of the two FSMs are not the same. (See Appendix B)

Thus, by the theorem in Appendix B below, the FSM table above with initial state $(1, F, F)$ is equivalent to the smaller table

Conditions	Previous state (n, w, e)	k = n	k ≠ n	
			k ≠ esc	k = esc
$1 \leq n < L$	(n, F, F)	(n+1, F, F)	(n-1, T, F)	(n, F, T)
	(n, F, T)	(n+1, F, F)	(n-1, T, F)	(L+1, F, F) “Fail”
	(n, T, F)	(n+1, F, F)	(n, T, F)	(n, T, T)
	(n, T, T)	(n+1, F, F)	(n, T, F)	(L+1, F, F) “Fail”
n=L	(L, F, F)	(L+1, F, F) “Pass”	(L-1, T, F)	(L, F, T)
	(L, F, T)	(L+1, F, F) “Pass”	(L-1, T, F)	(L+1, F, F) “Fail”
n=L+1	(L+1, F, F)	(L+1, F, F)	(L+1, F, F)	(L+1, F, F)

where the initial state is $(1, T, F)$.

2.8. Summary of main guidelines

The following summarizes the main guidelines identified earlier in this section 2.

Construct an FSM table from an English language statement of the problem:

- Read carefully and pedantically, looking for ambiguities, inconsistencies, missing information, etc.

- Identify suitable state variables by looking for *noun phrases or clauses representing significant conditions or situations that change during the test and upon which the next step in the test depends* (see Tactic 1 above).
- Check for redundancy among state variables (see Tactic 2 above).
- Identify alternative state variables.
- Look for *conditions* on the previous state and input term
 - to identify conditions for row and column headers and
 - to identify the next state and output to enter into the corresponding cell in the body of the table (see Tactic 3 above).

Reduce the number of states in an FSM by identifying equivalent states:

- Look for rows with identical outputs in each column.
- Check that transitions are to equivalent states.
- See the criteria in section 2.3 above. Apply the theorem in Appendix B below.

Reduce the size of an FSM table without reducing the number of states:

- Unify (make identical) and group rows and columns of an FSM table by
 - parameterizing states or inputs
 - generalizing expressions
 - or-ing the conditions

Improve the readability of an FSM table:

- Organize conditions in the headers hierarchically to show clearly that they are mutually exclusive and exhaustive.
- Merge identical adjacent cells.
- Remove unreachable states, impossible conditions (conditions that are always false, never satisfied).
- Simplify according to (1) the intended readership and (2) purpose of table (e.g. communication between humans, logical analysis, program design).

Transition variants and state invariants:

- Identify and use to facilitate the above ways of simplifying an FSM table.

3. Trace Functions (TF)

3.1. Trace functions vs. FSMs

This chapter deals with functions mapping an input sequence (a *trace*) to an output value (not a sequence of output values). In the terminology of section 2.1 above, a trace function maps a sequence of elements of IN (an element of IN*) to an element of OUT. See also section 2.1 above and Appendix A below.

Also an FSM defines a function mapping a sequence of inputs to an output, see section 2.1 above and Appendix A below. An FSM is, mathematically, an iterative definition of the output function. The states of the FSM are intermediate terms in this iterative definition. When defining a function of input sequences with an FSM, one must, therefore, define the states of the FSM. If, in this process, one selects inappropriate states, the resulting FSM will not define the function one really wants to define.

Alternatively, one can take a more direct approach and define the function by referring to the terms in the input sequence, but not to any intermediate states or other mathematical objects. Sometimes, the trace function can be defined by referring to the last few terms in the input

sequence only. By referring in addition to a few of the previous outputs, the number of previous terms in the input sequence that must be referred to in the definition of the function can sometimes be reduced. In the first example below, the definition refers to the last term in the input sequence and to the previous output value only.

3.2. Commonly used auxiliary functions on traces

Definitions of the auxiliary functions r , p and len : When defining trace functions, it will often be helpful to use the auxiliary functions r , p and len . The value of the function $r(T)$ is defined to be the most recent (latest, newest) term in the trace T . The value of $p(T)$ (precursor) is the remainder of T after removing $r(T)$. $r([])$ is undefined and $p([])$ is defined as the empty trace $[]$. Note that $T=p(T).[r(T)]$ for all $T\neq[]$.

The value of $len(T)$ is the length of the trace T (the number of terms in T).

Thus, the function r maps a trace to a term of the trace. The function p maps a trace to a trace. The function len maps a trace to a non-negative integer.

3.3. An example of a TF

3.3.1. Constructing a table from the English language specification of the desired TF

Example: Consider a slight variation of the example in section 2.2 above. The input to a program consists of a sequence of digits (0, 1, ... 9). If the first five digits in the input sequence are the digits 1 through 5 inclusive and in ascending order, the program should output “C” (for correct). If the first 5 digits are anything else, the program should output “E” (for error). If the input trace is shorter than five digits and an error is not present, the program should output the digit expected next. The function calculated by this program is called *out*. Note that *out* maps a sequence of digits to an element of the set {C, E, 1, 2, 3, 4, 5}.

On first glance at such a specification in English text, translating it into a TF table often appears easy – easier than it really is. Natural language text is notorious for ambiguities and missing information. One should read such a specification very carefully and pedantically – be picayune. If at first you do not see ambiguities or gaps, look again; they are there. When ambiguities or gaps are identified, ask the author, “user” or client for clarification. List explicitly any assumptions made.

To transform the English language specification to a mathematically precise specification, we begin by translating the above text, part by part, into more precise statements about the value of $out(T)$ for various T , i.e. T meeting various conditions.

The original English version of the specification appears in **blue Helvetica font** below. The interspersed passages in black normal font are intermediate comments and interpretation of the original text written in the process of formulating a mathematically precise specification.

If the first five digits in the input sequence are the digits 1 through 5 inclusive and in ascending order, the program should output “C” (for correct).

If $len(T)\geq 5$ and $first5(T)=[1, 2, 3, 4, 5]$, then $out(T)='C'$. Here, the function $first5$ is defined to map a trace to the trace consisting of the first five terms in the original trace.

If the first five digits are anything else, the program should output “E” (for error).

If $\text{len}(T) \geq 5$ and $\text{first5}(T) \neq [1, 2, 3, 4, 5]$, then $\text{out}(T) = \text{"E"}$.

If the input trace is shorter than five digits and an error is not present, the program should output the digit expected next.

The word “error” here is not defined and is ambiguous. We assume that it means a trace that cannot, by appending further terms, lead to the “correct” trace $[1, 2, 3, 4, 5]$. So a trace in which no error is present would be a trace that can become $[1, 2, 3, 4, 5]$ by appending additional terms. I.e., a trace in which no error is present would be either the empty trace $[], [1], [1, 2], [1, 2, 3]$, or $[1, 2, 3, 4]$.

The term “digit expected” is also undefined. What does it mean? We assume that it means the digit that would be the next one in sequence, i.e. 1 greater than the last digit in the input trace.

The last part of the English specification of the function out then becomes

If $T = []$, then $\text{out}(T) = 1$.

If $T = [1]$, then $\text{out}(T) = 2$.

If $T = [1, 2]$, then $\text{out}(T) = 3$.

If $T = [1, 2, 3]$, then $\text{out}(T) = 4$.

If $T = [1, 2, 3, 4]$, then $\text{out}(T) = 5$.

The above “specification” leaves the question still open what the value of out should be if an error is present in an input trace shorter than five digits. The earlier statement “If the first five digits are anything else, the program should output “E” (for error)” seems to suggest that this applies even if fewer than five digits are present in the trace, but strictly speaking, it does not say so. We assume that the value of out should be “E” in such cases and add the catch all condition “else” with a value of “E”:

Else, $\text{out}(T) = \text{"E"}$.

Collecting the above conditions and values of out into a list gives the following table.

line number	condition on T	out(T)
1	$\text{len}(T) \geq 5 \wedge \text{first5}(T) = [1, 2, 3, 4, 5]$	“C”
2	$\text{len}(T) \geq 5 \wedge \text{first5}(T) \neq [1, 2, 3, 4, 5]$	“E”
3	$T = []$	1
4	$T = [1]$	2
5	$T = [1, 2]$	3
6	$T = [1, 2, 3]$	4
7	$T = [1, 2, 3, 4]$	5
8	else	“E”

Line 2 in the table above can be subsumed into the catch all line 8:

line number	condition on T	out(T)
1	$\text{len}(T) \geq 5 \wedge \text{first5}(T) = [1, 2, 3, 4, 5]$	“C”
2	$T = []$	1
3	$T = [1]$	2
4	$T = [1, 2]$	3
5	$T = [1, 2, 3]$	4
6	$T = [1, 2, 3, 4]$	5
7	else	“E”

At each step in the development of such a table of conditions on T and values of out, one should aim for conditions that are mutually exclusive, i.e., do not overlap. Conditions that violate this requirement should be examined with a view to eliminating the overlap. If the overlapping conditions lead to different values for out, the table is inconsistent and the inconsistency must be resolved. The conditions in the final table should be exhaustive, i.e. cover all possibilities. In the table above, the condition “else” ensures that this is the case. If the conditions are not exhaustive, then the table is incomplete. This will often be the case when only part of the English text has been examined, i.e. in the middle of the process of reformulating the English text into a mathematically precise table.

The goal is to express out(T) as a function of r(T), p(T) and, if necessary, earlier terms in the trace, e.g. r(p(T)), p(p(T)), r(p(p(T))), p(p(p(T))), etc. Ultimately, we would like the terms p(T), p(p(T)), etc. to appear only as arguments of out or r, i.e. in the form out(p(T)), out(p(p(T))), ... r(p(T)), r(p(p(T))), etc.

Therefore, we begin by introducing columns for p(T), out(p(T)) and r(T) into the table above, giving:

line number	condition on T	p(T)	out(p(T))	r(T)	out(T)
1	$\text{len}(T) \geq 5 \wedge \text{first5}(T) = [1, 2, 3, 4, 5]$				“C”
2	$T = []$	[]	1	not defined	1
3	$T = [1]$	[]	1	1	2
4	$T = [1, 2]$	[1]	2	2	3
5	$T = [1, 2, 3]$	[1, 2]	3	3	4
6	$T = [1, 2, 3, 4]$	[1, 2, 3]	4	4	5
7	else				“E”

The entries for p(T), out(p(T)) and r(T) in line 1 are not unique, so have been left blank above. Therefore, we split this line in order to distinguish between the shortest trace for which the value of out is “C” and longer traces. I.e., we begin with the cases $\text{len}(T) = 5$ and $\text{len}(T) > 5$.

When $\text{len}(T) = 5$, the condition in line 1 simplifies to $T = [1, 2, 3, 4, 5]$. We put the row for this case lower in the table to fit the pattern in rows 2 through 6.

line number	condition on T	p(T)	out(p(T))	r(T)	out(T)
1	$\text{len}(T) > 5 \wedge$ $\text{first5}(T) = [1, 2, 3, 4, 5]$		“C”	anything	“C”
2	$T = []$	$[]$	1	not defined	1
3	$T = [1]$	$[]$	1	1	2
4	$T = [1, 2]$	$[1]$	2	2	3
5	$T = [1, 2, 3]$	$[1, 2]$	3	3	4
6	$T = [1, 2, 3, 4]$	$[1, 2, 3]$	4	4	5
7	$T = [1, 2, 3, 4, 5]$	$[1, 2, 3, 4]$	5	5	“C”
8	else				“E”

Considering line 1, the values of p(T) and r(T) are still not uniquely determined. However, the value of out(p(T)) is determined uniquely; it is “C”. When $T = [1, 2, 3, 4, 5]$, $\text{out}(T) = \text{“C”}$. For every longer T, regardless of the last term r(T), out(T) is also equal to “C”.

The values of out(p(T)) and r(T) alone determine the value of out(T) in all lines except line 8. To be sure that no counterexample lurks in the catch all line 8, we must split line 8 into the various ways the condition “else” can arise. In particular, we consider those cases in which the next input digit (r(T)) is not the expected one (out(p(T))):

line number	condition on T	p(T)	out(p(T))	r(T)	out(T)
1	$\text{len}(T) > 5 \wedge$ $\text{first5}(T) = [1, 2, 3, 4, 5]$		“C”	anything	“C”
2	$T = []$	$[]$	1	not defined	1
3	$T = [1]$	$[]$	1	=1	2
3a	$T = [d], d \neq 1$			$\neq 1$	“E”
4	$T = [1, 2]$	$[1]$	2	=2	3
4a	$T = [1, d], d \neq 2$			$\neq 2$	“E”
5	$T = [1, 2, 3]$	$[1, 2]$	3	=3	4
5a	$T = [1, 2, d], d \neq 3$			$\neq 3$	“E”
6	$T = [1, 2, 3, 4]$	$[1, 2, 3]$	4	=4	5
6a	$T = [1, 2, 3, d], d \neq 4$			$\neq 4$	“E”
7	$T = [1, 2, 3, 4, 5]$	$[1, 2, 3, 4]$	5	=5	“C”
7a	$T = [1, 2, 3, 4, d], d \neq 5$			$\neq 5$	“E”
8	else				“E”

The notation “ $T = [\dots, d], d \neq \dots$ ” means that T is of the stated form with the digit d not equal to the following value. The comma “,” should be read as “where” as it is often used in mathematical writing.

Lines 1 through 7a now cover all possible T except those T for which $\text{out}(p(T)) = \text{“E”}$. I.e. $\text{out}(p(T)) = \text{“E”}$ (and r(T) is any value) are the only “else” cases left. We can, therefore, insert these entries into the corresponding cells in line 8 of the above table. Then the column for “p(T)” becomes superfluous and can be deleted. The resulting table specifying the function out then becomes:

line number	condition on T	out(p(T))	r(T)	out(T)
1	$\text{len}(T) > 5 \wedge \text{first5}(T) = [1, 2, 3, 4, 5]$	"C"	anything	"C"
2	$T = []$	1	not defined	1
3	$T = [1]$	1	=1	2
3a	$T = [d], d \neq 1$		$\neq 1$	"E"
4	$T = [1, 2]$	2	=2	3
4a	$T = [1, d], d \neq 2$		$\neq 2$	"E"
5	$T = [1, 2, 3]$	3	=3	4
5a	$T = [1, 2, d], d \neq 3$		$\neq 3$	"E"
6	$T = [1, 2, 3, 4]$	4	=4	5
6a	$T = [1, 2, 3, d], d \neq 4$		$\neq 4$	"E"
7	$T = [1, 2, 3, 4, 5]$	5	=5	"C"
7a	$T = [1, 2, 3, 4, d], d \neq 5$		$\neq 5$	"E"
8	else	"E"	anything	"E"

The condition $T = []$ is useful for distinguishing line 2, in which $r(T)$ is not defined. Otherwise, the entries in the column "condition on T" are now superfluous and can be deleted. Restructuring the above table accordingly yields the following table.

If T meets the conditions below,		then out(T)=	
$T = []$		1	
$T \neq []$	out(p(T))="C"	"C"	
	out(p(T))="E"	"E"	
	out(p(T))=1	$r(T) = \text{out}(p(T))$	2
		$r(T) \neq \text{out}(p(T))$	"E"
	out(p(T))=2	$r(T) = \text{out}(p(T))$	3
		$r(T) \neq \text{out}(p(T))$	"E"
	out(p(T))=3	$r(T) = \text{out}(p(T))$	4
		$r(T) \neq \text{out}(p(T))$	"E"
	out(p(T))=4	$r(T) = \text{out}(p(T))$	5
		$r(T) \neq \text{out}(p(T))$	"E"
	out(p(T))=5	$r(T) = \text{out}(p(T))$	"C"
		$r(T) \neq \text{out}(p(T))$	"E"

By writing equivalent expressions for the values 2, 3, 4 and 5 of out(T) in the table above, the values of out in the corresponding pairs of rows become identical:

If T meets the conditions below,		then out(T)=	
T=[]		1	
T≠[]	out(p(T))="C"	"C"	
	out(p(T))="E"	"E"	
	out(p(T))=1	r(T)=out(p(T))	r(T)+1
		r(T)≠out(p(T))	"E"
	out(p(T))=2	r(T)=out(p(T))	r(T)+1
		r(T)≠out(p(T))	"E"
	out(p(T))=3	r(T)=out(p(T))	r(T)+1
		r(T)≠out(p(T))	"E"
	out(p(T))=4	r(T)=out(p(T))	r(T)+1
		r(T)≠out(p(T))	"E"
out(p(T))=5	r(T)=out(p(T))	"C"	
	r(T)≠out(p(T))	"E"	

and the table can be compressed to the following form.

If T meets the conditions below,		then out(T)=	
T=[]		1	
T≠[]	out(p(T))="C"	"C"	
	out(p(T))="E"	"E"	
	out(p(T))∈ {1, 2, 3, 4}	r(T)=out(p(T))	r(T)+1
		r(T)≠out(p(T))	"E"
	out(p(T))=5	r(T)=out(p(T))	"C"
		r(T)≠out(p(T))	"E"

3.3.2. Moving conditions between rows and columns

In the last table above, all conditions are shown on the left, with one row per condition. That table is essentially a one dimensional table, or a list. The table can be restructured to make better use of the two dimensional nature of a table on a plane (e.g. a piece of paper or a display screen). Such restructuring often leads to a table in which the dependencies between the various variables are easier to understand.

The above definition of the function out is essentially an iterative definition; it defines out(T) in terms of out(p(T)) and r(T) – the value of out for a shorter trace and the value of the last term in the trace. The base case, out([]), is defined separately in the first row in the body of the table. Placing the conditions on out(p(T)) in the rows and the conditions on r(T) in columns (or vice versa) results in a convenient alternative tabular definition of the function out.

The following table defines the function out for arguments $T \neq []$,

out(T)=	r(T)=out(p(T))	r(T)≠out(p(T))
out(p(T))="C"	"C"	
out(p(T))="E"	"E"	
out(p(T))∈ {1, 2, 3, 4}	r(T)+1	"E"
out(p(T))=5	"C"	"E"

where out([])=1.

The following minor variation reduces repetition of text in the table headers, but is structurally the same as the above table. The reduced clutter improves readability, at least for some readers.

out(T)=		r(T)	
		=out(p(T))	≠out(p(T))
out(p(T))	= "C"	"C"	
	= "E"	"E"	
	∈ {1, 2, 3, 4}	r(T)+1	"E"
	= 5	"C"	"E"

3.4. An industrial example of manipulating TF tables

The following example illustrates the application of the techniques introduced in the sections above in a more typical situation. This example is a slight variation of the example in Section 5 and Table 6 in [1]. This TF is part of the specification of a program in industrial use.

The English language specification of the TF is taken from sections 2 and 4.1 in [1]. The application in question involves testing a keyboard by manually pressing its keys in order.

The input trace is a sequence of key identification numbers. The desired value of the output (of the TF) is the identification number of the key that should be pressed next if the keyboard test is not complete. If the test is complete, the value of the TF should be "Pass" or "Fail", depending on whether the keyboard passed or failed the test. The desired TF, named *out*, must map an input trace to a key identification number or to "Pass" or "Fail".

3.4.1. Constructing a table from the English language specification of the desired TF

The original English version of the specification appears in [blue Helvetica font](#) below. The interspersed passages in black normal font are intermediate comments and interpretation of the original text written in the process of formulating a mathematically precise specification. Cf. section 3.3.1 above, in which the same was done for a simpler example, and section 2.7.1 above, in which an FSM table was constructed from the same English specification of this industrial example.

To transform the English language specification to a mathematically precise specification, we begin by translating the text, part by part, into more precise statements about the value of out(T) for various T, i.e. for T meeting various conditions. In the process, we build up a table of conditions on T with the corresponding values of out(T), whereby the conditions on T ultimately involve r(T), out(p(T)), r(p(T)), out(p(p(T))), etc. only. In each step of building the

table, we ensure that the table is consistent, e.g. by ensuring that the conditions are mutually exclusive, i.e. do not overlap.

The main requirements of the keyboard sequence test are the following:

- Keys of a keyboard should be pressed in a specified order during keyboard testing, i.e. there is only one key that is expected to be pressed at each moment (the expected key).
- The inputs to the program are described by the sequence number of the key pressed.

We assume that the “expected” key is the key that should be pressed next. We further assume that the last bulleted item above means that the key identification numbers are the sequence numbers, i.e. that the identification number for the first key that should be pressed is “1”; for the second key that should be pressed, “2”; for the third key that should be pressed, “3”, etc.

- The test software provides interactive test diagnostics. An image of the keyboard is displayed on the screen. The expected key should be highlighted with a yellow box on the screen. When the next key in the sequence is pressed, the background colour of the key changes to blue, to indicate that the key has been successfully tested.

How should the images of the other keys be displayed? Is the desired display a function of the expected key only? If not, will another TF be needed to control the display?

- Depending on the most recently pressed keys, there are three possible variations from the old to the new value of the expected key after some key has been pressed. For this purpose, the keys are divided into three groups:
 - The next key in the sequence (the expected key).
 - The escape key, but only if not the expected key.
 - Any other key (an incorrect key).
- If the expected key has been pressed, the next key in the test sequence becomes the expected key, i.e. the number of the expected key is increased by 1.

If the most recently pressed key was the one expected when it was pressed, the value of out should be the previous value of out plus one. I.e. if $r(T)=out(p(T))$, then $out(T)=out(p(T))+1$. *Question:* What if out(T) is not a sequence number of any key, i.e. what if out(T) is greater than the number of keys on the keyboard?

The first entry in the table of conditions on T and values of out(T) is:

Condition on T	out(T)
$r(T)=out(p(T))$	$out(p(T))+1$

- If the escape key has been pressed, the number of the expected key remains the same.

The last two bulleted items above contradict each other in the case that the expected key is the escape key. Further above, the phrase “escape key, but only if not the expected key” appears. We assume that this qualification is intended to apply also to the last bulleted item above, i.e. that it was intended to read “If the escape key has been pressed *unexpectedly*, the

number of the expected key remains the same.” Under this assumption, if the most recently pressed key was not the one expected but was the escape key, then the value of out should be the previous value of out. I.e. if $r(T) \neq out(p(T))$ and $r(T) = esc$, then $out(T) = out(p(T))$, whereby we define *esc* to be the sequence number of the escape key. Our table of conditions on T and values of out(T) becomes:

Condition on T	out(T)
$r(T) = out(p(T))$	$out(p(T)) + 1$
$r(T) \neq out(p(T)) \wedge r(T) = esc$	$out(p(T))$

- If an incorrect (unexpected) key has been pressed first time after the expected key, the previous key becomes expected, i.e. the number of the expected key is decreased by 1.

Again here a contradiction can arise. If the most recently pressed key is unexpected but the escape key, and the previously pressed key was the expected one, the last bulleted item states that the new value of out should be $out(p(T)) - 1$. The previous bulleted item above states that the value of out should be $out(p(T))$. Further above a definition of “incorrect” is given: unexpected and not the escape key. Assuming that this latter definition is meant, i.e. that the last bulleted item above should begin “If an incorrect (unexpected and not the escape) key has been pressed ...”, the first sentence in the last bulleted item above leads to the rule: If $r(T) \neq out(p(T))$ and $r(T) \neq esc$ and [r(T) has been pressed first time after the expected key], then $out(T) = out(p(T)) - 1$. We must now translate the phrase in square brackets. This phrase appears to mean “the key pressed before r(T) was the one expected then. Expressed in terms of the trace T and previous values of out, the condition in square brackets is: $r(p(T)) = out(p(p(T)))$.”

Then the complete rule is: If $r(T) \neq out(p(T))$ and $r(T) \neq esc$ and $r(p(T)) = out(p(p(T)))$, then $out(T) = out(p(T)) - 1$.

The following sequence of terms and values of out illustrates this condition and the relationships between the various terms in T and the values of out for corresponding substraces (at intermediate points in the trace). Note that $out(p(T))$ is the key expected for r(T), for any non-empty trace T. Below, “incorrect” means “not expected and not the escape key”.

input terms	$r(p(p(T)))$		$r(p(T))$ expected		$r(T)$ incorrect	
values of out		$out(p(p(T)))$ expected $r(p(T))$		$out(p(T))$ expected $r(T)$		$out(T)$ expected next key

Our table of conditions on T and values of out now becomes:

Condition on T	out(T)
$r(T) = out(p(T))$	$out(p(T)) + 1$
$r(T) \neq out(p(T)) \wedge r(T) = esc$	$out(p(T))$
$r(T) \neq out(p(T)) \wedge r(T) \neq esc \wedge r(p(T)) = out(p(p(T)))$	$out(p(T)) - 1$

- If an incorrect key is pressed later, the expected key remains unchanged until the expected key is pressed.

What precisely does “later” mean in the above sentence? Is the phrase “until the expected key is pressed” significant? Considering the context set by the previous bulleted item above, “later” presumably means “after an incorrect” key was pressed. This is an assumption that should be verified with appropriate persons. The following sequence table illustrates this situation.

input terms	$r(p(p(T)))$		$r(p(T))$ incorrect		$r(T)$ incorrect	
values of out		$out(p(p(T)))$ expected $r(p(T))$		$out(p(T))$ expected $r(T)$		$out(T)$ expected next key

I.e., if $r(T) \neq out(p(T))$ and $r(T) \neq esc$ and $r(p(T)) \neq out(p(p(T)))$ and $r(p(T)) \neq esc$, then $out(T) = out(p(T))$. Our table of conditions on T and values of out(T) becomes:

Condition on T	out(T)
$r(T) = out(p(T))$	$out(p(T)) + 1$
$r(T) \neq out(p(T)) \wedge r(T) = esc$	$out(p(T))$
$r(T) \neq out(p(T)) \wedge r(T) \neq esc \wedge r(p(T)) = out(p(p(T)))$	$out(p(T)) - 1$
$r(T) \neq out(p(T)) \wedge r(T) \neq esc \wedge r(p(T)) \neq out(p(p(T))) \wedge r(p(T)) \neq esc$	$out(p(T))$

- The only way to pass the keyboard sequence test is to successfully test all keys.

Almost hidden here is the answer to the question above, “What if out(T) is not a sequence number of any key, i.e. what if out(T) is greater than the number of keys on the keyboard?”. The bulleted item above seems to say that if the last key on the keyboard is pressed when it is the expected key, then the keyboard passes the test. This suggests the rule: If $r(T) = out(p(T))$ and $r(T) = L$, then $out(T) = \text{“Pass”}$, where L is the sequence number (the identification number) of the last key on the keyboard. This leads to the following table of conditions on T and values of out.

Condition on T	out(T)
$r(T) = out(p(T))$	$out(p(T)) + 1$
$r(T) \neq out(p(T)) \wedge r(T) = esc$	$out(p(T))$
$r(T) \neq out(p(T)) \wedge r(T) \neq esc \wedge r(p(T)) = out(p(p(T)))$	$out(p(T)) - 1$
$r(T) \neq out(p(T)) \wedge r(T) \neq esc \wedge r(p(T)) \neq out(p(p(T))) \wedge r(p(T)) \neq esc$	$out(p(T))$
$r(T) = out(p(T)) \wedge r(T) = L$	“Pass”

If $r(T) = L$, both the first and the last conditions in the table above are satisfied, and the table doubly defines out(T) to be L+1 and “Pass”, an inconsistency. We assume that new rule about passing the test and the corresponding last line in the table above take precedence, i.e. that the first condition in the table above should be further restricted by the additional condition $r(T) \neq L$. This will also resolve the question identified earlier by preventing the value of out to exceed the number of keys on the keyboard. After applying the additional condition to the first condition in the table above and reordering the lines in the table, we obtain the following table:

Condition on T	out(T)
$r(T)=out(p(T)) \wedge r(T)=L$	“Pass”
$r(T)=out(p(T)) \wedge r(T)\neq L$	$out(p(T))+1$
$r(T)\neq out(p(T)) \wedge r(T)=esc$	$out(p(T))$
$r(T)\neq out(p(T)) \wedge r(T)\neq esc \wedge r(p(T))=out(p(p(T)))$	$out(p(T))-1$
$r(T)\neq out(p(T)) \wedge r(T)\neq esc \wedge r(p(T))\neq out(p(p(T))) \wedge r(p(T))\neq esc$	$out(p(T))$

- The only way to fail the keyboard sequence test is to press the escape key twice in a row.

This text suggests that the following sequence should result in “Fail” as the value of out:

input terms	$r(p(p(T)))$		$r(p(T))$ esc		$r(T)$ esc	
values of out		$out(p(p(T)))$ expected $r(p(T))$		$out(p(T))$ expected $r(T)$		$out(T)$ expected next key

which would lead to the rule: If $r(T)=esc$ and $r(p(T))=esc$, then $out(T)=\text{“Fail”}$. This condition overlaps with the first, second and third conditions in the above table of conditions and values of out. An earlier statement in the English text indicated that pressing the escape key was to be interpreted as an escape event only if the escape key was not the expected key. This suggests – and we assume here – that the above text should read “The only way to fail the keyboard sequence test is to press the escape key *unexpectedly* twice in a row”, more specifically, that in neither case was the escape key the expected key. This leads to the revised rule: If $r(T)=esc$ and $r(T)\neq out(p(T))$ and $r(p(T))=esc$ and $r(p(T))\neq out(p(p(T)))$, then $out(T)=\text{“Fail”}$. Our table of conditions on T and values of out is, after rewriting the new condition to better match the form of an existing condition in the table and inserting the new condition next to a similar existing condition:

Condition on T	out(T)
$r(T)=out(p(T)) \wedge r(T)=L$	“Pass”
$r(T)=out(p(T)) \wedge r(T)\neq L$	$out(p(T))+1$
$r(T)\neq out(p(T)) \wedge r(T)=esc$	$out(p(T))$
$r(T)\neq out(p(T)) \wedge r(T)=esc \wedge r(p(T))\neq out(p(p(T))) \wedge r(p(T))=esc$	“Fail”
$r(T)\neq out(p(T)) \wedge r(T)\neq esc \wedge r(p(T))=out(p(p(T)))$	$out(p(T))-1$
$r(T)\neq out(p(T)) \wedge r(T)\neq esc \wedge r(p(T))\neq out(p(p(T))) \wedge r(p(T))\neq esc$	$out(p(T))$

Now only the third and fourth conditions in the table above overlap. The third condition came from the earlier English language statement “If the escape key has been pressed, the number of the expected key remains the same.”, which we modified to “If the escape key has been pressed *unexpectedly*, the number of the expected key remains the same.” in order to resolve a contradiction. Presumably, the intention is that pressing the escape key unexpectedly once leaves the value of out (the next expected key) unchanged, but pressing it again unexpectedly fails the keyboard test. Assuming that this is the case, we must further qualify the condition for the single unexpected depression of the escape key to exclude the possibility that the previously depressed key was the escape key (pressed unexpectedly). This additional condition will be the negation of the expression $r(p(T))\neq out(p(p(T))) \wedge r(p(T))=esc$. The table of conditions on T and values of out becomes

Condition on T	out(T)
$r(T)=out(p(T)) \wedge r(T)=L$	“Pass”
$r(T)=out(p(T)) \wedge r(T)\neq L$	$out(p(T))+1$
$r(T)\neq out(p(T)) \wedge r(T)=esc \wedge \neg[r(p(T))\neq out(p(p(T))) \wedge r(p(T))=esc]$	$out(p(T))$
$r(T)\neq out(p(T)) \wedge r(T)=esc \wedge r(p(T))\neq out(p(p(T))) \wedge r(p(T))=esc$	“Fail”
$r(T)\neq out(p(T)) \wedge r(T)\neq esc \wedge r(p(T))=out(p(p(T)))$	$out(p(T))-1$
$r(T)\neq out(p(T)) \wedge r(T)\neq esc \wedge r(p(T))\neq out(p(p(T))) \wedge r(p(T))\neq esc$	$out(p(T))$

This completes the construction of the table of conditions on T and values of out from the English text specifying the TF out. At this point, one should review this table carefully against the individual English language statements. Check that all ambiguities, contradictions, inconsistencies and gaps have been resolved in a reasonable way, list all assumptions made and ask for confirmation of all assumptions.

The last two parts of the English specification above describe the criteria for passing or failing the keyboard test, see pages 29 and 30 above respectively. These sentences do not clearly state what the value of out should be for traces containing additional terms beyond the points at which a “Pass” or “Fail” was determined. It would seem likely, however, that once a trace has led to a “Pass” or a “Fail” result, subsequent terms should not change the result, leading to the rules:

If $out(p(T))=“Pass”$, then $out(T)=“Pass”$.

If $out(p(T))=“Fail”$, then $out(T)=“Fail”$.

and that these rules should take precedence over all other rules. The author, “user” or client should be asked for clarification on this point. We assume here that these rules should be added to the English specification of the TF out.

We must, therefore, add the restriction

$$out(p(T))\neq“Pass” \wedge out(p(T))\neq“Fail”$$

to every condition already in our table of conditions on T and values of out. We must also add the two rules above to the table. Our table then becomes:

Condition on T	out(T)
$\text{out}(p(T)) = \text{"Pass"}$	"Pass"
$\text{out}(p(T)) = \text{"Fail"}$	"Fail"
$\text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) = \text{out}(p(T)) \wedge r(T) = L$	"Pass"
$\text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) = \text{out}(p(T)) \wedge r(T) \neq L$	$\text{out}(p(T)) + 1$
$\text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) = \text{esc} \wedge \neg[r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc}]$	$\text{out}(p(T))$
$\text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) = \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc}$	"Fail"
$\text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) = \text{out}(p(p(T)))$	$\text{out}(p(T)) - 1$
$\text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) \neq \text{esc}$	$\text{out}(p(T))$

3.4.2. Systematic examination of the constructed table

The constructed table must still be examined to ensure that its entries are

- always defined
- mutually exclusive
- exhaustive

The conditions in the table will be mutually exclusive if each step in the development of the table maintained this property. Even if this is the case, however, this aspect of the table should be verified again on completion of the table.

When conducting this examination, it is useful to recall again the domain and range of the TF in question, here out. The function out maps a trace (a sequence of key sequence numbers) to either a key sequence number, "Pass" or "Fail". I.e., the domain of out is the set of all sequences of key sequence numbers (identification numbers). Each key sequence number is an integer between 1 and L inclusive. The range of out is the set of key sequence numbers augmented by the elements "Pass" and "Fail". Thus,

$$\text{domain of out} = \{1, \dots, L\}^*, \text{ where } L > 1$$

$$\text{range of out} = \{1, \dots, L, \text{"Pass"}, \text{"Fail"}\}$$

Notice that $r(T)$ appears in all conditions but two in the above table of conditions on T and values of out. The term $r(T)$ is defined if and only if the length of T is at least 1 (if and only if $\text{len}(T) \geq 1$). Several conditions contain $r(p(T))$, which is defined if and only if $\text{len}(T) \geq 2$. We must, therefore, extend our table for the case $\text{len}(T) = 0$ and subdivide our table into sections for $\text{len}(T) = 1$ and $\text{len}(T) \geq 2$.

When $\text{len}(T) = 0$, $T = []$. The English text above does not deal explicitly with this case. One of the preliminary comments indicates that the value of out should be the sequence number of the next expected key when the test is not complete. When $T = []$, the test has not yet started, and the next expected key is presumably 1. Thus, we assume that $\text{out}([]) = 1$.

When $\text{len}(T) = 1$, then $p(T) = []$ and $\text{out}(p(T)) = 1$ (hence, $\text{out}(p(T)) \neq \text{"Pass"}$ and $\text{out}(p(T)) \neq \text{"Fail"}$). Because $L > 1$, $L \neq 1$ and $\text{out}(p(T)) \neq L$, so the first three conditions in the last table above of conditions on T and values of out will always be false. The fourth condition in

that table can be true, giving rise to the rule: If $\text{len}(T)=1$ and $r(T)=\text{out}(p(T))$, then $\text{out}(T)=\text{out}(p(T))+1$. Alternatively, this can be written: If $\text{len}(T)=1$ and $r(T)=\text{out}(p(T))$, then $\text{out}(T)=2$.

What, however, if $\text{len}(T)=1$ and $r(T)\neq\text{out}(p(T))$? This situation is not clearly covered by the English text. Furthermore, the English text further subdivides this situation into two cases depending on whether or not $r(T)$ is the (unexpected) escape key. Therefore, it appears that we need to distinguish between the cases

- a) $\text{len}(T)=1 \wedge r(T)\neq\text{out}(p(T)) \wedge r(T)=\text{esc}$
- b) $\text{len}(T)=1 \wedge r(T)\neq\text{out}(p(T)) \wedge r(T)\neq\text{esc}$

Case a) was covered by the original English text “[If the escape key has been pressed, the number of the expected key remains the same](#)”, corresponding to the fifth line in the last table above before we extended that condition to exclude two successive unexpected depressions of the escape key. When $\text{len}(T)=1$, there was no previous depression of any key, so the value of out should be unchanged, i.e. 1.

In case b), an incorrect key has been pressed initially, i.e. the key is neither the expected one nor the escape key. The English text regarding pressing an incorrect key refers to the previous key depressions, but in case b), there is none. The English text fails to define the value of out when an incorrect key is pressed initially. The English text indicates that the value of out either remains unchanged (if an incorrect key was previously pressed) or is decreased by 1 (if the previously pressed key was the expected one). Decreasing the value of out would leave $\text{out}(T)=0$, but this value of out makes no sense: It is not in the defined range of out and there is no key with the sequence number 0. We assume, therefore, that the value of out in case b) should be 1. This assumption, as all others, should be clarified with the the author of the English text, the “user” or the client.

Both cases a) and b) lead to a value of 1 for out. The distinction between cases a) and b) is therefore unnecessary. Thus, when $\text{len}(T)=1$, the rules become

If $\text{len}(T)=1$ and $r(T)=\text{out}(p(T))$, then $\text{out}(T)=2$.

If $\text{len}(T)=1$ and $r(T)\neq\text{out}(p(T))$, then $\text{out}(T)=1$.

Combining the above for $\text{len}(T)=0$, $\text{len}(T)=1$ and the last table above for $\text{len}(T)\geq 2$, our table of conditions on T and values of out becomes

Condition on T	out(T)
$\text{len}(T)=0$	1
$\text{len}(T)=1 \wedge r(T)=\text{out}(p(T))$	2
$\text{len}(T)=1 \wedge r(T) \neq \text{out}(p(T))$	1
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) = \text{"Pass"}$	"Pass"
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) = \text{"Fail"}$	"Fail"
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \wedge r(T) = \text{out}(p(T)) \wedge r(T) = L$	"Pass"
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \wedge r(T) = \text{out}(p(T)) \wedge r(T) \neq L$	$\text{out}(p(T))+1$
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) = \text{esc} \wedge \neg[r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc}]$	$\text{out}(p(T))$
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) = \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc}$	"Fail"
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) = \text{out}(p(p(T)))$	$\text{out}(p(T))-1$
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) \neq \text{esc}$	$\text{out}(p(T))$

Examining the conditions above for completeness, it is evident that one case is missing:

$$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \\ \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc}$$

In this case, the last key pressed was incorrect (not expected, not escape) and the next to last key depressed was unexpectedly the escape key:

input terms		$r(p(p(T)))$...		$r(p(T))$ escape		$r(T)$ incorrect	
values of out	$\text{out}(p(p(p(T))))$ expected $r(p(p(T)))$		$\text{out}(p(p(T)))$ expected $r(p(T))$		$\text{out}(p(T))$ expected $r(T)$		$\text{out}(T)$ expected next key

For the previously pressed key $r(p(p(T)))$ there are four possibilities or subcases:

- 1) No key was pressed, i.e. $\text{len}(T)=2$.
- 2) The expected key was pressed.
- 3) The escape key was unexpectedly pressed.
- 4) An incorrect key was pressed (i.e. not the expected key and not the escape key).

Each of these four possibilities is considered in detail below.

Possibility 1: No key was previously pressed, i.e. $\text{len}(T)=2$, and $p(p(T))=[]$, the empty trace. This situation is similar to case b) above for a trace consisting of only one term representing an incorrect key. As there, we assume that the desired value of out is 1.

input terms		$r(p(T))$ escape		$r(T)$ incorrect	
values of out	$\text{out}(p(p(T)))=1$ expected $r(p(T))$		$\text{out}(p(T))=1$ expected $r(T)$		$\text{out}(T)=1$ expected next key

This gives rise to the rule:

If
 $\text{len}(T)=2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$
 $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc}$
 then $\text{out}(T) = \text{out}(p(T))$.

or, equivalently,

If
 $\text{len}(T)=2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$
 $\wedge r(T) \neq 1 \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq 1 \wedge r(p(T)) = \text{esc}$
 then $\text{out}(T) = 1$.

Possibility 2: The expected key was pressed [$r(p(p(T)))$ was the expected key].

input terms		$r(p(p(T)))$ expected		$r(p(T))$ escape		$r(T)$ incorrect	
values of out	$\text{out}(p(p(p(T))))$ expected $r(p(p(T)))$		$\text{out}(p(p(T)))$ expected $r(p(T))$		$\text{out}(p(T))$ expected $r(T)$		$\text{out}(T)$ expected next key

In this case, previously identified rules lead to:

$\text{out}(p(p(T))) = \text{out}(p(p(p(T)))) + 1$
 $\text{out}(p(T)) = \text{out}(p(p(T)))$

leaving open the question of the value of $\text{out}(T)$. The (modified) English text attempting to cover the situation in which an incorrect key has been just been pressed is

- If an incorrect (unexpected and not the escape) key has been pressed first time after the expected key, the previous key becomes expected, i.e. the number of the expected key is decreased by 1.
- If an incorrect key is pressed later, the expected key remains unchanged until the expected key is pressed.

An ambiguity could be seen in the use of the words “first time”. Does this phrase refer to (1) the first depression of an incorrect key after the expected key was last pressed in the (possibly distant) past or (2) the depression of an incorrect key immediately after (as the first key pressed after) the expected key? We assume (1) here, so that the value of out should be decreased by 1. Then

$\text{out}(T) = \text{out}(p(T)) - 1$

leading to the rule

If
 $\text{len}(T) > 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$
 $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc}$
 $\wedge r(p(p(T))) = \text{out}(p(p(p(T))))$
 then $\text{out}(T) = \text{out}(p(T)) - 1$.

Possibility 3: The escape key was unexpectedly pressed [r(p(p(T))) was the escape key but not the expected key].

input terms		r(p(p(T))) escape		r(p(T)) escape		r(T) incorrect	
values of out	out(p(p(p(T)))) expected r(p(p(T)))		out(p(p(T))) expected r(p(T))		out(p(T)) expected r(T)		out(T) expected next key

This situation cannot occur in the context in question because a trace ending in these three terms would have given rise by other already identified rules to the value “Fail” for out(p(T)) and, in turn, for out(T). However, one of the conditions defining the context under examination is out(p(T))≠“Fail”. I.e., formally, the full condition for the rule pertaining to this situation simplifies to the logical constant false, so the rule would never apply. The rule can, therefore, be dropped.

Possibility 4: An incorrect key was pressed [r(p(p(T))) was neither the expected key nor the escape key].

input terms		r(p(p(T))) incorrect		r(p(T)) escape		r(T) incorrect	
values of out	out(p(p(p(T)))) expected r(p(p(T)))		out(p(p(T))) expected r(p(T))		out(p(T)) expected r(T)		out(T) expected next key

Here the English text “[If an incorrect key is pressed later, the expected key remains unchanged until the expected key is pressed](#)” applies. Therefore, out(T)=out(p(T)). See also Possibility 2 above and previous sections referring to pressing an incorrect key.

The rule for this situation is

If
 $len(T) > 2 \wedge out(p(T)) \neq \text{“Pass”} \wedge out(p(T)) \neq \text{“Fail”}$
 $\wedge r(T) \neq out(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq out(p(p(T))) \wedge r(p(T)) = \text{esc}$
 $\wedge r(p(p(T))) \neq out(p(p(p(T)))) \wedge r(p(p(T))) \neq \text{esc}$
 then out(T)=out(p(T)).

Note, however, that the term r(p(p(T)))≠esc above is redundant, because r(p(p(T)))=esc is not possible, see Possibility 3 above. Therefore, that term can be dropped, leaving as the rule for this situation

If
 $len(T) > 2 \wedge out(p(T)) \neq \text{“Pass”} \wedge out(p(T)) \neq \text{“Fail”}$
 $\wedge r(T) \neq out(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq out(p(p(T))) \wedge r(p(T)) = \text{esc}$
 $\wedge r(p(p(T))) \neq out(p(p(p(T))))$
 then out(T)=out(p(T)).

More formally, by a previous rule,

$len(T) \geq 2 \wedge out(p(T)) \neq \text{“Pass”} \wedge out(p(T)) \neq \text{“Fail”}$
 $\wedge r(T) \neq out(p(T)) \wedge r(T) = \text{esc} \wedge r(p(T)) \neq out(p(p(T))) \wedge r(p(T)) = \text{esc}$
 $\Rightarrow out(T) = \text{“Fail”}$

for all traces T. In particular, the above formula applies for the trace p(T):

$$\begin{aligned} & \text{len}(p(T)) \geq 2 \wedge \text{out}(p(p(T))) \neq \text{"Pass"} \wedge \text{out}(p(p(T))) \neq \text{"Fail"} \\ & \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc} \wedge r(p(p(T))) \neq \text{out}(p(p(p(T)))) \wedge r(p(p(T))) = \text{esc} \\ & \Rightarrow \text{out}(p(T)) = \text{"Fail"} \end{aligned}$$

Reversing the implication, the above expression is equivalent to

$$\begin{aligned} & \text{out}(p(T)) \neq \text{"Fail"} \\ & \Rightarrow \text{len}(p(T)) < 2 \vee \text{out}(p(p(T))) = \text{"Pass"} \vee \text{out}(p(p(T))) = \text{"Fail"} \\ & \vee r(p(T)) = \text{out}(p(p(T))) \vee r(p(T)) \neq \text{esc} \vee r(p(p(T))) = \text{out}(p(p(p(T)))) \vee r(p(p(T))) \neq \text{esc} \end{aligned}$$

which implies the following expression formed by anding the same expression to both sides of the above implication

$$\begin{aligned} & \text{out}(p(T)) \neq \text{"Fail"} \\ & \wedge \text{len}(T) > 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \\ & \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc} \\ & \wedge r(p(p(T))) \neq \text{out}(p(p(p(T)))) \\ & \Rightarrow \\ & [\text{len}(p(T)) < 2 \vee \text{out}(p(p(T))) = \text{"Pass"} \vee \text{out}(p(p(T))) = \text{"Fail"} \\ & \vee r(p(T)) = \text{out}(p(p(T))) \vee r(p(T)) \neq \text{esc} \vee r(p(p(T))) = \text{out}(p(p(p(T)))) \vee r(p(p(T))) \neq \text{esc}] \\ & \wedge \text{len}(T) > 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \\ & \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc} \\ & \wedge r(p(p(T))) \neq \text{out}(p(p(p(T)))) \end{aligned}$$

which simplifies to (see earlier rules above), i.e. which is equivalent to

$$\begin{aligned} & \text{len}(T) > 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \\ & \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc} \\ & \wedge r(p(p(T))) \neq \text{out}(p(p(p(T)))) \\ & \Rightarrow \\ & [\text{out}(p(p(T))) = \text{"Fail"} \vee r(p(p(T))) \neq \text{esc}] \\ & \wedge \text{len}(T) > 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \\ & \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc} \\ & \wedge r(p(p(T))) \neq \text{out}(p(p(p(T)))) \end{aligned}$$

But by the previous rule above propagating failure, $\text{out}(p(p(T))) = \text{"Fail"} \Rightarrow \text{out}(p(T)) = \text{"Fail"}$, which is equivalent to $\text{out}(p(T)) \neq \text{"Fail"} \Rightarrow \text{out}(p(p(T))) \neq \text{"Fail"}$. The indented expression above simplifies then to the equivalent expression

$$\begin{aligned} & \text{len}(T) > 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \\ & \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc} \\ & \wedge r(p(p(T))) \neq \text{out}(p(p(p(T)))) \\ & \Rightarrow \\ & \text{len}(T) > 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \\ & \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc} \\ & \wedge r(p(p(T))) \neq \text{out}(p(p(p(T)))) \wedge r(p(p(T))) \neq \text{esc} \end{aligned}$$

The reverse implication is clearly true, so we have

$$\begin{aligned}
 & \text{len}(T) > 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \\
 & \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc} \\
 & \wedge r(p(p(T))) \neq \text{out}(p(p(p(T)))) \\
 & = \\
 & \text{len}(T) > 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"} \\
 & \wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc} \\
 & \wedge r(p(p(T))) \neq \text{out}(p(p(p(T)))) \wedge r(p(p(T))) \neq \text{esc}
 \end{aligned}$$

showing that the last term above $[r(p(p(T))) \neq \text{esc}]$ is redundant and can be dropped wherever it appears in the expression above the equals sign above.

(end of the four possibilities)

Adding the rules for possibilities 1, 2 and 4 above to the previous table of conditions on T and values of out, we obtain the table below.

Condition on T	out(T)
$\text{len}(T) = 0$	1
$\text{len}(T) = 1 \wedge r(T) = \text{out}(p(T))$	2
$\text{len}(T) = 1 \wedge r(T) \neq \text{out}(p(T))$	1
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) = \text{"Pass"}$	"Pass"
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) = \text{"Fail"}$	"Fail"
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) = \text{out}(p(T)) \wedge r(T) = L$	"Pass"
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) = \text{out}(p(T)) \wedge r(T) \neq L$	$\text{out}(p(T)) + 1$
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) = \text{esc} \wedge \neg[r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc}]$	$\text{out}(p(T))$
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) = \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc}$	"Fail"
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) = \text{out}(p(p(T)))$	$\text{out}(p(T)) - 1$
$\text{len}(T) \geq 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) \neq \text{esc}$	$\text{out}(p(T))$
$\text{len}(T) = 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc}$	$\text{out}(p(T))$
$\text{len}(T) > 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc}$ $\wedge r(p(p(T))) = \text{out}(p(p(p(T))))$	$\text{out}(p(T)) - 1$
$\text{len}(T) > 2 \wedge \text{out}(p(T)) \neq \text{"Pass"} \wedge \text{out}(p(T)) \neq \text{"Fail"}$ $\wedge r(T) \neq \text{out}(p(T)) \wedge r(T) \neq \text{esc} \wedge r(p(T)) \neq \text{out}(p(p(T))) \wedge r(p(T)) = \text{esc}$ $\wedge r(p(p(T))) \neq \text{out}(p(p(p(T))))$	$\text{out}(p(T))$

At this point, one should examine the table above again to ensure that all expressions are defined and that they are mutually exclusive and exhaustive.

The detailed analysis of the English text “specifying” the desired TF illustrates once again that more discrepancies lurk under the surface than one would expect from just reading the English text alone. One first recognizes how good or how poor the content of a piece of text is when one translates it into another (e.g. natural) language. Translating it into the language of mathematics appears to be an even tougher test of the substance of a piece of natural language text and particularly of its meaning. This reasonably well known fact is not as thoroughly recognized as it needs to be among software specifiers and developers. Every natural language specification badly needs to be vetted by translating it into mathematical language before turning it over to software designers and developers. Designing software directly from a natural language specification is not only asking – but is begging – for trouble, errors and grave consequential damage.

3.4.3. Restructuring the conditions in the table

The different terms $\text{len}(T)=2$, $\text{len}(T)\geq 2$ and $\text{len}(T)>2$ arise in the above table. If only one of these three terms were present, the structure of the table would be clearer and simpler. The conditions could then be factored into a hierarchical organization.

The following identities enable us to transform the conditions involving $\text{len}(T)=2$ and $\text{len}(T)>2$ into conditions in which $\text{len}(T)$ appears only in the form $\text{len}(T)\geq 2$.

$$(\text{len}(T)=2) = (\text{len}(T)\geq 2 \wedge p(p(T))=[])$$

$$(\text{len}(T)>2) = (\text{len}(T)\geq 2 \wedge p(p(T))\neq [])$$

Modifying the table above by substituting the above equalities and then factoring the conditions leads to the table below in which the conditions on T are hierarchically structured.

Condition on T				out(T)			
len(T)=0				1			
len(T)=1	r(T)=out(p(T))			2			
	r(T)≠out(p(T))			1			
len(T)≥2	out(p(T))=“Pass”			“Pass”			
	out(p(T))=“Fail”			“Fail”			
		r(T)= out(p(T))	r(T)=L		“Pass”		
			r(T)≠L		out(p(T))+1		
	out(p(T))≠“Pass” ^ out(p(T))≠“Fail”	r(T)≠ out(p(T))	r(T)=esc	¬[r(p(T))≠out(p(p(T))) ^ r(p(T))=esc]	out(p(T))		
				r(p(T))≠out(p(p(T))) ^ r(p(T))=esc		“Fail”	
			r(T)≠esc	r(p(T))=out(p(p(T)))		out(p(T))-1	
				r(p(T))≠esc		out(p(T))	
				r(p(T))≠out(p(p(T)))	p(p(T))=[]		out(p(T))
					p(p(T))≠[]		out(p(T))-1
r(p(T))=esc		r(p(p(T)))=out(p(p(p(T))))		out(p(T))			
		r(p(p(T)))≠out(p(p(p(T))))		out(p(T))			

3.4.4. Reordering and grouping rows in the TF table

Notice in the above table that the same values of out repeat often in the rightmost column. If the rows are suitably reordered to place rows with the same values of out together, adjacent rows can be combined. This might improve readability or facilitate the design of a program to calculate the value of out for a given trace T. The table below is one possible way to reorder the rows of the table above.

Condition on T				out(T)		
len(T)=0				1		
len(T)=1	r(T)=out(p(T))			2		
	r(T)≠out(p(T))			1		
len(T)≥2	out(p(T))="Pass"			"Pass"		
	out(p(T))="Fail"			"Fail"		
	r(T)=out(p(T))	r(T)=L		"Pass"		
		r(T)≠L		out(p(T))+1		
	r(T)≠out(p(T))	r(T)=esc	r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc		"Fail"	
			¬[r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc]		out(p(T))	
	r(T)≠out(p(T))	r(T)≠esc	r(p(T))≠out(p(p(T)))	r(p(T))≠esc		out(p(T))
				p(p(T))=[]		out(p(T))
			r(p(T))=esc		p(p(T))≠[]	r(p(p(T)))≠out(p(p(p(T))))
r(p(T))=out(p(p(T)))			r(p(p(T)))=out(p(p(p(T))))		out(p(T))-1	
r(p(T))=out(p(p(T)))				out(p(T))-1		

The block of cells enclosed in the double border in the table above is a prime candidate for reduction. Three of the four rows lead to the same value of out, suggesting that one combine the three corresponding conditions and simplify them. The condition for the TF value out(p(T))-1 then becomes

$$r(p(T))=esc \wedge p(p(T))\neq[] \wedge r(p(p(T)))=out(p(p(p(T))))$$

The condition for the TF value out(p(T)) is, loosely speaking, everything else, i.e. is the negation of the above condition, which can be written

$$r(p(T))\neq esc \vee p(p(T))=[] \vee r(p(p(T)))\neq out(p(p(p(T))))$$

Note that the values of the two expressions above are not strictly defined when $p(p(T))=[]$, because then $r(p(p(T)))$ is not defined. One common convention is to consider the value of a Boolean subexpression to be false when the expression contains a reference to an undefined value. This convention is appropriate in the situations arising here.

The reduced table is, then,

Condition on T			out(T)	
len(T)=0			1	
len(T)=1	r(T)=out(p(T))		2	
	r(T)≠out(p(T))		1	
len(T)≥2	out(p(T))="Pass"		"Pass"	
	out(p(T))="Fail"		"Fail"	
	r(T)= out(p(T))	r(T)=L	"Pass"	
		r(T)≠L	out(p(T))+1	
	out(p(T))≠"Pass" ∧ out(p(T))≠"Fail"	r(T)≠ out(p(T))	r(T)=esc	r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc ¬[r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc]
			r(T)≠esc	r(p(T))≠out(p(p(T)))
		r(p(T))≠esc ∨ p(p(T))=[] ∨ r(p(p(T)))≠out(p(p(p(T))))	out(p(T))	
			r(p(T))=esc ∧ p(p(T))≠[] ∧ r(p(p(T)))=out(p(p(p(T))))	
r(p(T))=out(p(p(T)))		out(p(T))-1		

The last two rows both lead to the TF value out(p(T))-1. They can be combined by or-ing their conditions. In order to do this, it is convenient to first split the cell enclosed by the double line in the above table and combine each new cell formed with the cell to its right. The result is the following table.

Condition on T				out(T)	
len(T)=0				1	
len(T)=1	r(T)=out(p(T))			2	
	r(T)≠out(p(T))			1	
len(T)≥2	out(p(T))="Pass"			"Pass"	
	out(p(T))="Fail"			"Fail"	
	out(p(T))≠"Pass" ∧ out(p(T))≠"Fail"	r(T)= out(p(T))	r(T)=L		"Pass"
		r(T)≠ out(p(T))	r(T)≠L		out(p(T))+1
	r(T)=esc		r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc		"Fail"
			¬[r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc]		out(p(T))
r(T)≠esc	r(p(T))≠out(p(p(T))) ∧ [r(p(T))≠esc ∨ p(p(T))=[] ∨ r(p(p(T)))≠out(p(p(p(T))))]			out(p(T))	
	r(p(T))=out(p(p(T))) ∨ [r(p(T))=esc ∧ p(p(T))≠[] ∧ r(p(p(T)))=out(p(p(p(T))))]			out(p(T))-1	

The two rows leading to the value out(p(T)) are now candidates for combining. Note that the each of the conditions differentiating these rows from one another are split in two different cells. Before combining these rows, each of the conditions must be in a single cell. This necessitates splitting the cells containing r(T)=esc and r(T)≠esc and then merging each pair of cells containing each condition:

Condition on T			out(T)	
len(T)=0			1	
len(T)=1	r(T)=out(p(T))		2	
	r(T)≠out(p(T))		1	
len(T)≥2	out(p(T))="Pass"		"Pass"	
	out(p(T))="Fail"		"Fail"	
	out(p(T))≠"Pass" ∧ out(p(T))≠"Fail"	r(T)= out(p(T))	r(T)=L r(T)≠L	"Pass" out(p(T))+1
		r(T)≠ out(p(T))	r(T)=esc ∧ r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc	"Fail"
	r(T)=esc ∧ ¬[r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc]		out(p(T))	
	r(T)≠esc ∧ r(p(T))≠out(p(p(T))) ∧ [r(p(T))≠esc ∨ p(p(T))=[] ∨ r(p(p(T)))≠out(p(p(p(T))))]		out(p(T))	
r(T)≠esc ∧ r(p(T))=out(p(p(T))) ∨ [r(p(T))=esc ∧ p(p(T))≠[] ∧ r(p(p(T)))=out(p(p(p(T))))]		out(p(T))-1		

Now the two rows leading to the value out(p(T)) can be combined by or-ing their conditions, after which the condition can be simplified:

$$\begin{aligned}
& r(T)=esc \wedge \neg[r(p(T))\neq out(p(p(T))) \wedge r(p(T))=esc] \\
& \vee r(T)\neq esc \wedge r(p(T))\neq out(p(p(T))) \wedge [r(p(T))\neq esc \vee p(p(T))=[] \vee r(p(p(T)))\neq out(p(p(p(T))))]
\end{aligned}$$

=

$$\begin{aligned}
& r(T)=esc \wedge [r(p(T))=out(p(p(T))) \vee r(p(T))\neq esc] \\
& \vee r(T)\neq esc \wedge r(p(T))\neq out(p(p(T))) \wedge [r(p(T))\neq esc \vee p(p(T))=[] \vee r(p(p(T)))\neq out(p(p(p(T))))]
\end{aligned}$$

=

$$\begin{aligned}
& r(T)=esc \wedge r(p(T))=out(p(p(T))) \vee \boxed{r(T)=esc \wedge r(p(T))\neq esc} \\
& \boxed{\vee r(T)\neq esc \wedge r(p(T))\neq out(p(p(T))) \wedge r(p(T))\neq esc} \vee r(T)\neq esc \wedge r(p(T))\neq out(p(p(T))) \wedge [p(p(T))=[] \vee r(p(p(T)))\neq out(p(p(p(T))))]
\end{aligned}$$

= {(A∧C ∨ ¬A∧B∧C) = (C∧(A ∨ ¬A∧B)) = (C∧(A ∨ B)) applied to the subexpression enclosed in boxes above}

$$\begin{aligned}
& r(T)=\text{esc} \wedge r(p(T))=\text{out}(p(p(T))) \\
& \vee r(p(T))\neq\text{esc} \wedge (r(T)=\text{esc} \vee r(p(T))\neq\text{out}(p(p(T)))) \\
& \vee r(T)\neq\text{esc} \wedge r(p(T))\neq\text{out}(p(p(T))) \wedge [p(p(T))=[] \vee r(p(p(T)))\neq\text{out}(p(p(p(T))))]
\end{aligned}$$

=

$$\begin{aligned}
& r(T)=\text{esc} \wedge r(p(T))=\text{out}(p(p(T))) \\
& \vee r(T)=\text{esc} \wedge r(p(T))\neq\text{esc} \\
& \vee r(p(T))\neq\text{esc} \wedge r(p(T))\neq\text{out}(p(p(T))) \\
& \vee r(T)\neq\text{esc} \wedge r(p(T))\neq\text{out}(p(p(T))) \wedge [p(p(T))=[] \vee r(p(p(T)))\neq\text{out}(p(p(p(T))))]
\end{aligned}$$

The resulting table is, then:

Condition on T			out(T)	
len(T)=0			1	
len(T)=1	r(T)=out(p(T))		2	
	r(T)≠out(p(T))		1	
len(T)≥2	out(p(T))="Pass"		"Pass"	
	out(p(T))="Fail"		"Fail"	
		r(T)= out(p(T))	r(T)=L	"Pass"
			r(T)≠L	out(p(T))+1
	out(p(T))≠"Pass" ∧ out(p(T))≠"Fail"	r(T)≠ out(p(T))	r(T)=esc ∧ r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc	"Fail"
			r(T)=esc ∧ r(p(T))=out(p(p(T))) ∨ r(T)=esc ∧ r(p(T))≠esc ∨ r(p(T))≠esc ∧ r(p(T))≠out(p(p(T))) ∨ r(T)≠esc ∧ r(p(T))≠out(p(p(T))) ∧ [p(p(T))=[] ∨ r(p(p(T)))≠out(p(p(p(T))))]	out(p(T))
		r(T)≠esc ∧ r(p(T))=out(p(p(T))) ∨ [r(p(T))=esc ∧ p(p(T))≠[] ∧ r(p(p(T)))=out(p(p(p(T))))]	out(p(T))-1	

Rows can be combined still further, e.g. the rows leading to the value 1 for the TF out, to the value “Pass” and to the value “Fail”. The procedures for doing so are the same as those used already in this section.

Which of the above equivalent tables – or still other possibilities – is the “simplest” or “best” depends on the purpose of the table and, in the case of reading by a person, on that person’s personal preference and way of examining and understanding the table and the problem. For some, none of the above tables in this section is better than the last table in section 3.4.3 above, which shows the hierarchical structure of the conditions particularly clearly. For a convincing examination and inspection, several different tables may be desired, each used for checking a different aspect of the logic against the requirements. One should not attempt to find the one and only “optimum” form for such a table, but instead, be able to convert any table into different forms, each supporting a different aspect of validating the correctness of the table or of using it to design the desired artifact, e.g. a computer program.

3.5. Review of the main guidelines for constructing and restructuring a TF table

Construct a TF table from an English “specification” of the function by performing the following steps:

- Read the English text in small sections, one by one. Read carefully, precisely, pedantically. Question the intended meaning of all words and phrases, especially unclear or undefined ones. Identify all potential ambiguities.
- Resolve all discrepancies found by asking the authors of the English text, the “users” and the clients for clarification.
- Reformulate each small piece of text into a condition on the trace (the argument of the TF) and the corresponding value of the TF, i.e. the value of the TF for arguments (traces) satisfying the condition.
- Reformulate the condition to refer to $r(T)$, $p(T)$, $r(p(T))$, $p(p(T))$, etc. only. If possible, limit references to $p(T)$, $p(p(T))$, etc. to be arguments of the function out, i.e. to $out(p(T))$, $out(p(p(T)))$, etc.
- Add the condition and the value of the TF (output) to a list of such conditions and values.
- Ensure that the conditions are mutually exclusive, i.e. that no two can be simultaneously satisfied (that no two “overlap”).
- Resolve any overlap between conditions in an appropriate way, e.g. by splitting one or both of the overlapping conditions into subconditions, none of which overlap, or by asking the authors of the English text, the “users” and the clients for clarification.
- After completing the analysis of the English text as described above, check that the conditions are defined and exhaustive. Check again that they are mutually exclusive.
- Restrict conditions containing possibly undefined terms to regions in which they are defined.
- If the conditions are not exhaustive, identify the set of traces remaining uncovered (i.e., identify the set of traces for which the TF is still undefined). Interpret the English text as it applies to all uncovered traces. Insert conditions and values of the TF for the previously uncovered cases into the table.
- Check again that the conditions are defined, exhaustive and mutually exclusive. Resolve discrepancies by repeating the relevant steps above. When no more discrepancies are found, the table is finished.
- Review the final table again against the English text. Resolve any discrepancies found as outlined above.

The table resulting from the above construction procedure can then be modified to improve its readability or its usefulness for other particular purposes (e.g. further analysis, program design) by transforming it in one or more of the following ways.

- Rows, columns: insert new, combine, delete, reorder
- Headers (row, column): restructure, e.g. hierarchically
- Cells: split, merge, insert comments (free form text)
- Conditions in headers: delete, restructure, regroup, move (e.g. from rows to columns), subdivide/refine
- Expressions in cells in body: reformulate, evaluate (partially or completely), *parameterize, unify*

See also section 2.8 above.

Appendix A. Finite State Machines (FSMs): mathematical definitions

As stated in section 2.1 above, an FSM is formally defined as the combination of

- a set S of states,
- a set IN of inputs,
- a set OUT of outputs,
- a function *output* mapping a state and an input to an output ($output : S \times IN \rightarrow OUT$),
- a function *nextstate* mapping a state and an input to a state ($nextstate : S \times IN \rightarrow S$) and
- an initial state s_0 , an element of the set S ($s_0 \in S$).

Example: The following table illustrates the relationships between each input, each output, each state and the sequences of inputs and outputs.

State	s_0		s_1		s_2		s_3		s_4		s_5		s_6	...
Input		in1		in2		in3		in4		in5		in6		...
Output		out1		out2		out3		out4		out5		out6		...

The output in each column is a function of the previous state and the current input (e.g. $out1 = output(s_0, in1)$).

The initial state is s_0 and each subsequent state is a function of the previous state and the most recent input (e.g. $s_1 = nextstate(s_0, in1)$).

The output sequence resulting from an input sequence is the sequence of individual outputs up to and including the output resulting from the most recent input. The function *outseq* is defined to be this mapping. E.g., $outseq([in1, in2, in3]) = [out1, out2, out3]$.

The state resulting from an input sequence is the state immediately after the last input in the sequence. We define *fstate* to be this mapping. E.g., $fstate([in1, in2, in3, in4]) = s_4$.

(end of example)

Mathematical definitions of outseq and fstate: The function that maps a sequence of elements of IN to a sequence of elements of OUT is called the *input/output sequence function* below. Formally, the input/output sequence function *outseq* for a given FSM ($S, IN, OUT, output, nextstate, s_0$) is defined as follows.

$$outseq : IN^* \rightarrow OUT^*$$

$$outseq([]) = [] \quad (\text{outseq maps the empty sequence to the empty sequence})$$

$$outseq(seq \& [i]) = outseq(seq) \& output(fstate(seq), i) \quad (\text{iterative definition of outseq})$$

where square brackets $[]$ denote a sequence, $\&$ denotes concatenation of sequences, seq is any sequence of elements of IN ($seq \in IN^*$), i is any element of IN ($i \in IN$), and the function *fstate* is defined as follows:

$$fstate : IN^* \rightarrow S \quad (\text{fstate maps an input sequence to a state})$$

$$fstate([]) = s_0 \quad (s_0 \text{ is the initial state})$$

$$fstate(seq \& [i]) = nextstate(fstate(seq), i) \quad (\text{iterative definition of fstate})$$

Appendix B. Equivalence of two FSMs

In section 2.3 above an example was given in which the number of states in an FSM was reduced. This possibility arises often when analyzing requirements and specifications for computer programs. Therefore, the mathematical basis for reducing the number of states in an FSM is given formally and in more detail below.

Two FSMs are equivalent if they define the same function mapping an input sequence to an output sequence (see Appendix A above). This notion of equivalence is practically meaningful only if the two FSMs have the same input alphabets and the same output alphabets.

Theorem on the equivalence of two FSMs: The input/output sequence functions $outseq_1$ and $outseq_2$ for two FSMs

$(S_1, IN, OUT, output_1, nextstate_1, s_{10})$ and

$(S_2, IN, OUT, output_2, nextstate_2, s_{20})$

respectively are equal if there exists a function r that maps each state in S_1 to a state in S_2 ($r : S_1 \rightarrow S_2$) such that

1. $r(s_{10}) = s_{20}$, i.e. r maps the initial state of the first FSM to the initial state of the second FSM,
2. $output_1(s, i) = output_2(r(s), i)$ for every $s \in S_1$ and every $i \in IN$ (outputs from equivalent states are equal) and
3. $r(nextstate_1(s, i)) = nextstate_2(r(s), i)$ for every $s \in S_1$ and every $i \in IN$ (all transitions from equivalent states are to equivalent states).

Sketch of proof: In the proof it must be shown that $outseq_1(seq) = outseq_2(seq)$ for every $seq \in IN^*$, using the definition of $outseq$ in Appendix A above applied to each of the two FSMs in this theorem. Tables of states, inputs and outputs as in the example in Appendix A above provide a sketch of the proof. The two tables for the two FSMs in the theorem differ only in that the states in the table for the second FSM are the equivalent states of the states in the first table. I.e., where s_0 appears in one table, $r(s_0)$ appears in the other; where s_1 appears on one table, $r(s_1)$ appears in the other, etc. The inputs and the outputs are the same in the two tables.

(end of theorem)

Usually, the state space S_2 is smaller than S_1 ($|S_2| < |S_1|$). The function r is called the *reduction function*.

Common application conditions: Often, when one applies this theorem in practice, one defines the second FSM (the reduced FSM) and the reduction function r more restrictively than required by the above theorem. Commonly, the second FSM and the reduction function r in the theorem are defined so that

- a. S_2 is a proper subset of S_1 (some states are deleted, but no new ones are introduced),
- b. $output_2$ is $output_1$ restricted to S_2 , i.e. $output_2(s, i) = output_1(s, i)$ for every $s \in S_2$ and every $i \in IN$,
- c. $nextstate_2(s, i) = r(nextstate_1(s, i))$ for every $s \in S_2$ and every $i \in IN$ (transitions to states being deleted are replaced by transitions to their equivalent states),
- d. $s_{20} = s_{10}$ (i.e. $s_{10} \in S_2$ and the initial state of the first FSM is also the initial state of the second FSM) and
- e. the function r maps states in S_2 to themselves (i.e., r restricted to S_2 is the identity function – within the set of remaining states no structural changes are made).

The function r maps states outside S_2 to equivalent states in S_2 .

Note that these common application conditions are additional conditions often met in practical applications. When satisfied, they facilitate applying the theorem above, but they are not necessary conditions for the equivalence of two FSMs. Examples exist in which an FSM can be reduced in ways not satisfying the common application conditions above.

How to reduce an FSM in practice: To reduce an FSM represented by a table such as any of those in chapter 2 above, one first identifies a suitable reduction function r . To do this, look for rows with the same outputs (cf. condition 2 of the theorem above). The state(s) represented by one of these rows must be assigned to S_2 , while one or more of the others are candidates for rows (states) to be dropped, i.e. are candidates for the set $S_1 \setminus S_2$. The reduction function r is then defined so that it maps such candidate states for $S_1 \setminus S_2$ to the state(s) assigned to S_2 .

Having identified a candidate reduction function r , one must check that the three numbered conditions of the theorem above are satisfied.

Condition 1 of the three conditions in the theorem will automatically be satisfied if the common application conditions d and e above are satisfied.

Condition 2 of the three conditions in the theorem will be satisfied because of the way the rows in the table were selected (the outputs are all the same), the way the reduction function r was defined (the state s_1 corresponding to any row being dropped is mapped by r to a state not being dropped for which the outputs are all the same) and common application condition b.

Condition 3 of the three conditions in the theorem will not always be satisfied when the application conditions above are satisfied and the reduction function r is chosen in the way described above. Condition 3 in the theorem must, therefore, be checked carefully. If the common application conditions c and e above are satisfied, condition 3 of the theorem will be satisfied for states in S_2 , but not necessarily for states in $S_1 \setminus S_2$, so only the latter must be checked. I.e., condition 3 in the theorem needs to be checked only for the rows to be deleted. Condition 3 in the theorem refers to both next state functions $nextstate_1$ and $nextstate_2$, but it would be more convenient to verify this condition using only the function $nextstate_1$. When application condition c above is satisfied, $nextstate_2(r(s), i) = r(nextstate_1(r(s), i))$ for every $s \in S_1$ (because the range of r is S_2 , so $r(s) \in S_2$). Then the condition 3 above to be verified becomes

$$r(nextstate_1(s, i)) = r(nextstate_1(r(s), i)) \text{ for every } s \in S_1 \setminus S_2 \text{ and every } i \in IN.$$

Candidate rows for deletion that do not satisfy this condition must be rejected; their states must remain in S_2 and hence in the reduced FSM. After revising the reduction function accordingly, one must check condition 3 in the theorem again. Candidate rows for deletion that do satisfy condition 3 in the theorem above can be deleted. Each next state s in the body of the resulting table is replaced by $r(s)$.

In summary, to reduce the number of states in an FSM, proceed as follows.

- i. Begin by looking for states (rows) with identical outputs. A group of states with the same outputs are candidates for reducing to one state (cf. condition 2 of the theorem above). Assign one of these states to the set S_2 and the others, to the set $S_1 \setminus S_2$, whereby the initial state must be assigned to S_2 . Repeat this for every group of states with identical outputs. If the initial state is one of any of these groups, assign it to S_2 .

- ii. Assign the remaining states, each of whose outputs differ from the outputs of other states, to S2.
- iii. Then define r so that it maps every state in S2 to itself and every state in S1\S2 to the state in S2 with the same outputs.
- iv. Check that

$$r(\text{nextstate1}(s, i)) = r(\text{nextstate1}(r(s), i))$$

(the modified version of condition 3 of the theorem above) is true for every state in S1\S2 (the states to be dropped) and every input in IN. Any state not satisfying this condition must be reassigned to S2 and the corresponding steps in the above process repeated.

- v. When a reduction is found satisfying the above conditions, delete the row(s) for the states in S1\S2 and replace all next states s in the table by r(s), thereby satisfying the application condition c above.

Example: In section 2.3 above, an FSM was informally reduced. Here we will reduce the same FSM using the more mechanistic procedure defined above. We begin with the last FSM table in section 2.2 above:

		Input digit									
		0	1	2	3	4	5	6	7	8	9
State	1 (initial)	7 "E"	2	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"
	2	7 "E"	7 "E"	3	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"
	3	7 "E"	7 "E"	7 "E"	4	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"
	4	7 "E"	7 "E"	7 "E"	7 "E"	5	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"
	5	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	6 "C"	7 "E"	7 "E"	7 "E"	7 "E"
	6	8	8	8	8	8	8	8	8	8	8
	7 (error)	7	7	7	7	7	7	7	7	7	7
	8 (correct)	8	8	8	8	8	8	8	8	8	8

Step i: The rows for states 6, 7 and 8 contain identical outputs (the empty string). We assign one, state 6, to set S2 and the others, states 7 and 8, to S1\S2. No other pair of states has the same outputs, so this step is finished.

Step ii: Every other state has a unique pattern of outputs, so they are assigned to S2. As a result of this and the preceding step, S2 = {1, 2, 3, 4, 5, 6} and S1\S2 = {7, 8}.

Step iii: The function r is now defined to be: r(1)=1, r(2)=2, r(3)=3, r(4)=4, r(5)=5, r(6)=6, r(7)=6 and r(8)=6.

The following table is a copy of the table above with arrows added illustrating the reduction function r. Green arrows show the mapping of states in S2 to themselves. Blue arrows show the mapping of states in S1\S2 to state 6 in S2.

		Input digit									
		0	1	2	3	4	5	6	7	8	9
State	1 (initial)	7 "E"	2	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"
	2	7 "E"	7 "E"	3	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"
	3	7 "E"	7 "E"	7 "E"	4	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"
	4	7 "E"	7 "E"	7 "E"	7 "E"	5	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"
	5	7 "E"	7 "E"	7 "E"	7 "E"	7 "E"	6 "C"	7 "E"	7 "E"	7 "E"	7 "E"
	6	8	8	8	8	8	8	8	8	8	8
	7 (error)	7	7	7	7	7	7	7	7	7	7
	8 (correct)	8	8	8	8	8	8	8	8	8	8

Step iv: We must check that the (modified) condition 3 of the theorem is true, i.e. that

$$r(\text{nextstate1}(s, i)) = r(\text{nextstate1}(r(s), i))$$

for every $s \in S1 \setminus S2$ and every $i \in IN$. $S1 \setminus S2 = \{7, 8\}$, so we must check that

$$r(\text{nextstate1}(7, i)) = r(\text{nextstate1}(r(7), i))$$

and

$$r(\text{nextstate1}(8, i)) = r(\text{nextstate1}(r(8), i))$$

for every $i \in IN$. Because $r(7)=6$, $\text{nextstate1}(7, i) = 7$, $\text{nextstate1}(6, i) = 6$, $r(6)=6$ and correspondingly for state 8, this condition is satisfied. The required conditions are satisfied, so the function r reduces the FSM of the table above.

Step v: Deleting the rows for states 7 and 8 in $S1 \setminus S2$ and replacing all next states s in the table by $r(s)$, we obtain the following table for the reduced FSM.

		Input digit									
		0	1	2	3	4	5	6	7	8	9
State	1 (initial)	6 "E"	2	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"
	2	6 "E"	6 "E"	3	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"
	3	6 "E"	6 "E"	6 "E"	4	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"
	4	6 "E"	6 "E"	6 "E"	6 "E"	5	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"
	5	6 "E"	6 "E"	6 "E"	6 "E"	6 "E"	6 "C"	6 "E"	6 "E"	6 "E"	6 "E"
	6 (end)	6	6	6	6	6	6	6	6	6	6

Note that the table above is the same as the last table in section 2.3 above, which was the result of a less formal procedure for reducing the number of states in the FSM.

Appendix C. Deriving an FSM to Compute a Given Trace Function (TF)

Consider the following table defining the TF out. This table is the last one in section 3.4.3 above (q.v.). Desired is an FSM that calculates the same function out.

Condition on T				out(T)			
len(T)=0				1			
len(T)=1	r(T)=out(p(T))			2			
	r(T)≠out(p(T))			1			
len(T)≥2	out(p(T))=“Pass”			“Pass”			
	out(p(T))=“Fail”			“Fail”			
		r(T)= out(p(T))	r(T)=L		“Pass”		
			r(T)≠L		out(p(T))+1		
	out(p(T))≠“Pass” ∧ out(p(T))≠“Fail”	r(T)≠ out(p(T))	r(T)=esc	¬[r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc]		out(p(T))	
				r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc		“Fail”	
			r(T)≠esc	r(p(T))=out(p(p(T)))		out(p(T))-1	
				r(p(T))≠out(p(p(T)))	r(p(T))≠esc		out(p(T))
r(p(T))=esc					p(p(T))=[]		out(p(T))
					p(p(T))≠[]		out(p(T))-1
		r(p(p(T)))=out(p(p(p(T))))		out(p(T))			
		r(p(p(T)))≠out(p(p(p(T))))		out(p(T))			

In order to derive an equivalent FSM, i.e. an FSM that calculates the same TF out, we need to identify suitable states for the FSM. The states must satisfy the criterion that upon inputting each term in the input trace T, the value of out must be calculable from the most recent term r(T) and the state only. Using the above table defining out, the next state must “save” all the information in the conditions above except the information about the term r(p(p(T))), because that term will not be needed when processing the next term in the input trace.

Viewed differently but equivalently, the value of the TF out is a function of the entire trace T: out(T). For non-empty T, T consists of p(T) and r(T), so out(T) can be written in a form with two arguments, e.g. outTF(p(T), r(T)), whereby often only some of the information in p(T) is required as the first argument. Desired is an (iterative) output function for an FSM with two arguments, a state s and the last term in T, i.e. outFSM(s, r(T)). We require that the states s be defined so that outTF(p(T), r(T)) = outFSM(s, r(T)) for all non-empty T. These considerations

lead to the following strategy for transforming a TF table into a corresponding FSM table: express $out(T)$ as a function of $r(T)$ and remaining terms. The remaining terms give the state argument to the outFSM function.

One possibility for the next state in the example shown in the table above is simply to “save” the values of $r(T)$, $r(p(T))$, $out(p(T))$, $out(p(p(T)))$ and, of course, the current output $out(T)$ as the state. However, this is more information than will be needed in the next step; only certain information about the relationships between these values is really needed.

In the above table for the TF out, each term in the input trace is compared with the previous value of out, which is the “expected” next term. This is illustrated in the figure below, in which each color highlights the term and value of out being compared in each condition.

input terms		$r(p(p(T)))$		$r(p(T))$		$r(T)$	
values of out	$out(p(p(p(T))))$ expected $r(p(p(T)))$		$out(p(p(T)))$ expected $r(p(T))$		$out(p(T))$ expected $r(T)$		$out(T)$ expected next key

Because only the result of each comparison is relevant for selecting the value of out, only that result must be embedded in the next state of the FSM. This observation will be used below in defining an appropriate function giving that result.

Note that all conditions comparing $r(\cdot)$ are of one of the forms

$$r(p^n(T)) \text{ [eqop] } out(p^{n+1}(T))$$

or

$$r(p^n(T)) \text{ [eqop] } esc$$

or

$$r(T) \text{ [eqop] } L$$

where $n=0, 1$ or 2 and [eqop] is either $=$ or \neq . The notation $p^n(T)$ means the function p applied to the argument T n times, i.e.

$$p^0(T) \text{ means } T, p^1(T) \text{ means } p(T), p^2(T) \text{ means } p(p(T)), \text{ etc.}$$

Because the form $r(T)$ [eqop] L involves only the most recent term in T , the state resulting from the preceding trace $p(T)$ cannot contain any information about it and we can disregard this expression in determining the next state.

The expressions $\{r(p^n(T)) [eqop] out(p^{n+1}(T))\}$ and $\{r(p^n(T)) [eqop] esc\}$ occur only in the following combinations:

$r(p^n(T))=out(p^{n+1}(T))$, corresponding to the expected key having been pressed,

and

$r(p^n(T))\neq out(p^{n+1}(T)) \wedge r(p^n(T))=esc$, corresponding to the escape key having been pressed unexpectedly,

and

$r(p^n(T))\neq out(p^{n+1}(T)) \wedge r(p^n(T))\neq esc$, corresponding to an “incorrect” (neither the expected nor the escape) key having been pressed,

In addition, the case in which there is no term $r(p^n(T))$ in the trace T (because T is too short) must be considered. This arises when $p^n(T)=[]$.

Therefore, we define a function kt (for “key type”) that maps values of n and T to one of the values X , E , I , or N , representing “expected key”, “escape key”, “incorrect key” or “no key” respectively. The function kt is defined by the following table:

If the condition below is true,		then the value of $kt(n, T)$ is ...	representing
$p^n(T)\neq []$	$r(p^n(T))=out(p^{n+1}(T))$	X	expected key pressed
	$r(p^n(T))\neq out(p^{n+1}(T)) \wedge r(p^n(T))=esc$	E	escape key pressed unexpectedly
	$r(p^n(T))\neq out(p^{n+1}(T)) \wedge r(p^n(T))\neq esc$	I	incorrect key pressed (i.e. neither of the above)
$p^n(T)=[]$		N	no key pressed (T too short)

where $n \in \{0, 1, 2\}$ and $T \in \{1, 2, \dots, L\}^*$ and where the functions out , r and p and the constants L and esc are as previously defined. Note that $kt(n+1, T)=kt(n, p(T))$ for all n and all T .

In preparation for constructing a table for the FSM to calculate the function out , we first regroup certain conditions in the table at the beginning of this appendix defining the TF out to form each combination in the above table for the function kt in a single cell.

Condition on T			out(T)		
len(T)=0			1		
len(T)=1	r(T)=out(p(T))		2		
	r(T)≠out(p(T))		1		
len(T)≥2	out(p(T))=“Pass”		“Pass”		
	out(p(T))=“Fail”		“Fail”		
		r(T)=out(p(T))	r(T)=L	“Pass”	
			r(T)≠L	out(p(T))+1	
	out(p(T))≠“Pass” ∧ out(p(T))≠“Fail”	r(T)≠out(p(T)) ∧ r(T)=esc	¬[r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc]	out(p(T))	
			r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc	“Fail”	
		r(T)≠out(p(T)) ∧ r(T)≠esc	r(p(T))=out(p(p(T)))		out(p(T))-1
			r(p(T))≠out(p(p(T))) ∧ r(p(T))≠esc		out(p(T))
			r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc	p(p(T))=[]	out(p(T))
				p(p(T))≠[]	r(p(p(T)))=out(p(p(p(T))))
	r(p(p(T)))≠out(p(p(p(T))))	out(p(T))			

In the above table we insert in brackets {} the equivalent conditions referring to the function kt and obtain

Condition on T			out(T)		
len(T)=0			1		
len(T)=1	r(T)=out(p(T)) {kt(0, T)=X}		2		
	r(T)≠out(p(T)) {kt(0, T)≠X}		1		
len(T)≥2	out(p(T))="Pass"		"Pass"		
	out(p(T))="Fail"		"Fail"		
		r(T)=out(p(T)) {kt(0, T)=X}	r(T)=L r(T)≠L	"Pass" out(p(T))+1	
		r(T)≠out(p(T)) ∧ r(T)=esc {kt(0, T)=E}	¬[r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc] {kt(1, T)≠E} r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc {kt(1, T)=E}	out(p(T)) "Fail"	
	out(p(T))≠"Pass" ∧ out(p(T))≠"Fail"	r(T)≠out(p(T)) ∧ r(T)≠esc {kt(0, T)=I}	r(p(T))=out(p(p(T))) {kt(1, T)=X}	out(p(T))-1	
			r(p(T))≠out(p(p(T))) ∧ r(p(T))≠esc {kt(1, T)=I}	out(p(T))	
			r(p(T))≠out(p(p(T))) ∧ r(p(T))=esc {kt(1, T)=E}	out(p(T))	
			p(p(T))=[] {kt(2, T)=N}	out(p(T))-1	
			p(p(T))≠[] {kt(2, T)≠N}	r(p(p(T)))=out(p(p(p(T)))) {kt(2, T)=X} r(p(p(T)))≠out(p(p(p(T)))) {kt(2, T)≠X}	out(p(T))

Because the each expression involving kt is equivalent to the expression involving r and out in the same cell, we may delete the latter expressions and retain only the former. The above table then becomes:

Condition on T				out(T)		
len(T)=0				1		
len(T)=1	kt(0, T)=X			2		
	kt(0, T)≠X			1		
len(T)≥2	out(p(T))=“Pass”			“Pass”		
	out(p(T))=“Fail”			“Fail”		
		kt(0, T)=X	r(T)=L	“Pass”		
			r(T)≠L	out(p(T))+1		
		kt(0, T)=E	kt(1, T)≠E	out(p(T))		
			kt(1, T)=E	“Fail”		
		out(p(T))≠“Pass” ∧ out(p(T))≠“Fail”	kt(0, T)=I	kt(1, T)=X	out(p(T))-1	
				kt(1, T)=I	out(p(T))	
				kt(1, T)=E	kt(2, T)=N	out(p(T))
					kt(2, T)≠N	kt(2, T)=X
		kt(2, T)≠X	out(p(T))			

From the above table it is evident that the only variable information needed to determine out(T) is the values of len(T), r(T), out(p(T)), kt(0, T), kt(1, T) and kt(2, T). Note that kt(0, T) can be calculated from r(T) and out(p(T)). This leaves the values of len(T), r(T), out(p(T)), kt(1, T) and kt(2, T) as the only values now needed to determine out(T).

The table for an FSM gives the next state and the output value for each succeeding input term in the trace. The FSM begins in a state corresponding to no input, i.e. corresponding to an input trace of length 0. The table is needed only for traces of length 1 or more. I.e., in the FSM table, a row for len(T)=0 is not needed. If that line is removed from the table above, the only information about len(T) still needed is whether it is 1 or greater than 1, and that information can be deduced from the value of kt(1, T): if kt(1, T)=N, then len(T)≤1, and if kt(1, T)≠N, then len(T)≥2. Therefore, the value of len(T) is not needed when the value of kt(1, T) is available.

Therefore, the only information needed to calculate out(T) for non-empty T is the values of r(T), out(p(T)), kt(1, T) and kt(2, T). The value of r(T) is the most recent term in the input trace, leaving the values of out(p(T)), kt(1, T) and kt(2, T) as the only information needed in the FSM state variables when determining the value of out and the next state. The following figure illustrates the state information before and after undergoing the transition resulting from inputting the most recent term in T.

kt	kt(2, T)		kt(1, T)		kt(0, T)		
input terms		r(p(p(T)))		r(p(T))		r(T)	
values of out	out(p(p(p(T)))) expected r(p(p(T)))		out(p(p(T))) expected r(p(T))		out(p(T)) expected r(T)		out(T) expected next key
state					previous state: (out(p(T)), kt(1, T), kt(2, T))		next state: (out(T), kt(0, T), kt(1, T))

Adding the next state to the previous table defining the TF out and evaluating the state expressions where possible gives the following table of values of out(T) and the next state:

Condition on T				out(T)	next state (out(T), kt(0, T), kt(1, T))			
len(T)=0				1	(1, N, N) {initial state}			
len(T)=1	kt(0, T)=X			2	(2, X, N)			
	kt(0, T)≠X			1	(1, kt(0, T), N)			
len(T)≥2	out(p(T))="Pass"			"Pass"	("Pass", kt(0, T), kt(1, T))			
	out(p(T))="Fail"			"Fail"	("Fail", kt(0, T), kt(1, T))			
		kt(0, T)=X	r(T)=L		"Pass"	("Pass", X, kt(1, T))		
			r(T)≠L		out(p(T))+1	(out(T), X, kt(1, T))		
		kt(0, T)=E	kt(1, T)≠E		out(p(T))	(out(T), E, kt(1, T))		
			kt(1, T)=E		"Fail"	("Fail", E, E)		
		out(p(T))≠"Pass" ∧ out(p(T))≠"Fail"	kt(0, T)=I	kt(1, T)=X		out(p(T))-1	(out(T), I, X)	
				kt(1, T)=I		out(p(T))	(out(T), I, I)	
				kt(1, T)=E	kt(2, T)=N		out(p(T))	(out(T), I, E)
					kt(2, T)≠N	kt(2, T)=X		out(p(T))-1
kt(2, T)≠X		out(p(T))	(out(T), I, E)					

The previous state is (out(p(T)), kt(1, T), kt(2, T)). Naming these three state variables s0, s1 and s2 respectively, we substitute s0 for out(p(T)), s1 for kt(1, T), and s2 for kt(2, T) in the table above to obtain the following table:

Condition on T and state (s0, s1, s2)				out(T)	next state (out(T), kt(0, T), kt(1, T))		
len(T)=0				1	(1, N, N) {initial state}		
len(T)=1	kt(0, T)=X			2	(2, X, N)		
	kt(0, T)≠X			1	(1, kt(0, T), N)		
len(T)≥2	s0="Pass"			"Pass"	("Pass", kt(0, T), s1)		
	s0="Fail"			"Fail"	("Fail", kt(0, T), s1)		
		kt(0, T)=X	r(T)=L		"Pass"	("Pass", X, s1)	
			r(T)≠L		s0+1	(s0+1, X, s1)	
		kt(0, T)=E	s1≠E		s0	(s0, E, s1)	
			s1=E		"Fail"	("Fail", E, E)	
	s0≠"Pass" ∧ s0≠"Fail"	kt(0, T)=I	s1=X		s0-1	(s0-1, I, X)	
			s1=I		s0	(s0, I, I)	
			s1=E	s2=N		s0	(s0, I, E)
				s2≠N	s2=X		s0-1
		s2≠X			s0	(s0, I, E)	

The above table defines an FSM that calculates the same function out as the TF tables above. This FSM table can be transformed to a more suitable and more typical form in several ways. As an FSM table, the line for len(T)=0 is irrelevant and can be dropped, as an FSM table applies to the transition resulting from a new input term in T, i.e. for traces T of length at least 1. The condition len(T)=1 is then equivalent to the condition that the input term is the first term in the trace T, i.e. that the previous state is the initial state, in which s1=N. The condition len(T)≥2 is equivalent to the condition that p(T) is not empty, i.e. p(T)≠[], in which case kt(1, T)≠N or, equivalently, s1≠N.

Finally, each of the several expressions involving kt(0, T) can be reexpressed in terms of r(T), see the table defining the function kt above. E.g.,

$$\begin{aligned}
 & \text{kt}(0, T)=X \\
 = & \quad \quad \quad \text{[definition of kt]} \\
 & r(T)=\text{out}(p(T)) \\
 = & \quad \quad \quad \text{[s0, the first component of the previous state, is the value of out}(p(T))\text{]} \\
 & r(T)=s0
 \end{aligned}$$

The other expressions involving kt(0, T) can be similarly rewritten.

Applying the changes mentioned above, the FSM table becomes

Condition on r(T) and state (s0, s1, s2)				out(T)	next state (out(T), kt(0, T), kt(1, T))		
s1=N	r(T)=s0			2	(2, X, N)		
	r(T)≠s0			1	(1, kt(0, T), N)		
s1≠N	s0="Pass"			"Pass"	("Pass", kt(0, T), s1)		
	s0="Fail"			"Fail"	("Fail", kt(0, T), s1)		
	r(T)=s0	r(T)=L			"Pass"	("Pass", X, s1)	
			r(T)≠L			s0+1	(s0+1, X, s1)
	r(T)≠s0 ∧ r(T)=esc	s1≠E				s0	(s0, E, s1)
		s1=E			"Fail"	("Fail", E, E)	
	r(T)≠s0 ∧ r(T)≠esc	s1=X			s0-1	(s0-1, I, X)	
		s1=I			s0	(s0, I, I)	
		s1=E	s2=N			s0	(s0, I, E)
			s2≠N	s2=X			s0-1
s2≠X				s0	(s0, I, E)		

where the initial state is (1, N, N).

A typical and more convenient structure for the table for this FSM would have the conditions on the states on the left, ideally beginning with s0 to the left and then proceeding to s1 and s2. All terms involving r(T), the most recent term in the input trace, would be in column headers. The next state and the output would be in corresponding cells. We proceed to transform the above table accordingly.

The first two rows (in which s1=N) apply only for the first term in the input trace. If s1=N, r(T) is the only term in T, i.e. p(T)=[], and out(p(T))=1. Because out(p(T))=s0, s0=1. The values 2 and 1 for out(T) can, therefore, be rewritten as s0+1 and s0 respectively.

Because s0=1 when s1=N (see the paragraph above), s0≠"Pass" ∧ s0≠"Fail" is also true when s1=N. Introducing this condition in the first two rows will perhaps facilitate combining later one or both of the first two rows with other row(s) below.

Note that the last component of the next state is always s1. In those rows in which a constant (e.g. E) appears as the last component of the next state, the constant can be replaced by the equivalent s1.

The expressions $r(T)=L$ and $r(T)\neq L$ appear only in rows in which $r(T)=s_0$. Therefore, in these expressions, s_0 can be substituted for $r(T)$.

In order to have conditions on the state involving the state variable s_0 to the left of such conditions involving s_1 , we can move the expressions $s_1=N$ and $s_1\neq N$ to the right in all rows.

In the cases of the two rows with $s_0=\text{“Pass”}$ or $s_0=\text{“Fail”}$, s_0 can be substituted for “Pass” and “Fail” in the columns for $\text{out}(T)$ and next state.

Note that the value of $\text{out}(T)$ is always the same as the value of the first component of the next state. Therefore, the value of $\text{out}(T)$ does not need to be repeated in every cell of the body of the FSM table, provided that a corresponding note accompanies the table.

Transforming the table above accordingly yields the following table for this FSM, where * indicates that the cell cannot apply because the conditions in its row and column are mutually exclusive, i.e. inconsistent with one another (their conjunction is false).

Condition on $r(T)$ and state (s_0, s_1, s_2)				$r(T)=s_0$	$r(T)\neq s_0$		next state ($\text{out}(T), \text{kt}(0, T), \text{kt}(1, T)$)	
					$r(T)\neq \text{esc}$	$r(T)=\text{esc}$		
$s_0\neq\text{“Pass”}$ $\wedge s_0\neq\text{“Fail”}$	$r(T)=s_0$	$s_1=N$		(s_0+1, X, s_1)	*	*	(s_0+1, X, s_1)	
	$r(T)\neq s_0$	$s_1=N$		*	$(s_0, \text{kt}(0, T), s_1)$	$(s_0, \text{kt}(0, T), s_1)$	$(s_0, \text{kt}(0, T), s_1)$	
$s_0=\text{“Pass”}$		$s_1\neq N$		$(s_0, \text{kt}(0, T), s_1)$	$(s_0, \text{kt}(0, T), s_1)$	$(s_0, \text{kt}(0, T), s_1)$	$(s_0, \text{kt}(0, T), s_1)$	
$s_0=\text{“Fail”}$		$s_1\neq N$		$(s_0, \text{kt}(0, T), s_1)$	$(s_0, \text{kt}(0, T), s_1)$	$(s_0, \text{kt}(0, T), s_1)$	$(s_0, \text{kt}(0, T), s_1)$	
$s_0\neq\text{“Pass”}$ $\wedge s_0\neq\text{“Fail”}$	$r(T)=s_0$	$s_1\neq N$	$s_0=L$	$(\text{“Pass”}, X, s_1)$	*	*	$(\text{“Pass”}, X, s_1)$	
			$s_0\neq L$	(s_0+1, X, s_1)	*	*	(s_0+1, X, s_1)	
	$r(T)\neq s_0 \wedge r(T)=\text{esc}$	$s_1\neq N$	$s_1\neq E$	*	*	(s_0, E, s_1)	(s_0, E, s_1)	
			$s_1=E$	*	*	$(\text{“Fail”}, E, s_1)$	$(\text{“Fail”}, E, s_1)$	
	$r(T)\neq s_0 \wedge r(T)\neq \text{esc}$	$s_1\neq N$	$s_1=E$	$s_1=X$	*	(s_0-1, I, s_1)	*	(s_0-1, I, s_1)
				$s_1=I$	*	(s_0, I, s_1)	*	(s_0, I, s_1)
			$s_2=N$	$s_1=E$	*	(s_0, I, s_1)	*	(s_0, I, s_1)
				$s_2\neq N$	$s_2=X$	*	(s_0-1, I, s_1)	*
	$s_2\neq X$	*	(s_0, I, s_1)	*	(s_0, I, s_1)			

The conditions on $r(T)$ in the row headers are now superfluous and can be eliminated. The column for the next state can be deleted, because the next states are shown in the other cells in the body of the table. To facilitate referring to particular rows, row numbers have been temporarily added. The result is the following table.

Condition on the state (s0, s1, s2)		r(T)=s0	r(T)≠s0		row number		
			r(T)≠esc	r(T)=esc			
s0≠“Pass” ∧ s0≠“Fail”	s1=N	(s0+1, X, s1)	*	*	1		
	s1=N	*	(s0, kt(0, T), s1)	(s0, kt(0, T), s1)	2		
s0=“Pass”	s1≠N	(s0, kt(0, T), s1)	(s0, kt(0, T), s1)	(s0, kt(0, T), s1)	3		
s0=“Fail”	s1≠N	(s0, kt(0, T), s1)	(s0, kt(0, T), s1)	(s0, kt(0, T), s1)	4		
s0≠“Pass” ∧ s0≠“Fail”	s1≠N	s0=L	(“Pass”, X, s1)	*	*	5	
		s0≠L	(s0+1, X, s1)	*	*	6	
	s1≠N	s1≠E	*	*	(s0, E, s1)	7	
		s1=E	*	*	(“Fail”, E, s1)	8	
	s1≠N	s1=X	*	(s0-1, I, s1)	*	9	
		s1=I	*	(s0, I, s1)	*	10	
		s1=E	s2=N	*	(s0, I, s1)	*	11
			s2≠N	s2=X	*	(s0-1, I, s1)	*
	s2≠X	*		(s0, I, s1)	*	13	

Rows 1 and 2 can be combined. The only distinction between them is made by conditions in the column header.

Except for the first condition, rows 3 and 4 are identical, so they can be combined by or-ing their conditions.

The conditions can be grouped better by moving the combination of rows 3 and 4 to the bottom of the table.

The value of $kt(0, T)$ depends only on $r(T)$ and $out(p(T))$, and $s0$ is $out(p(T))$. Thus, $kt(0, T)$ depends only on $r(T)$ and $s0$. The relationship between $r(T)$ and $s0$ is the subject of the column headers, so $kt(0, T)$ can be evaluated for each of the three columns whose headers reference $r(T)$. The function kt is defined in an earlier table above, which becomes, when the first argument is 0 and the second argument is a non-empty trace:

If the condition below is true,	then the value of $kt(0, T)$ is ...	representing
$r(T)=out(p(T))$	X	expected key pressed
$r(T)≠out(p(T)) \wedge r(T)=esc$	E	escape key pressed unexpectedly
$r(T)≠out(p(T)) \wedge r(T)≠esc$	I	incorrect key pressed (i.e. neither of the above)

Thus, $kt(0, T)$ is X in the first of the three columns whose headers reference $r(T)$, is I in the second, and is E in the third. These values can be substituted for $kt(0, T)$ in these three columns.

Applying the above described transformations, we obtain the following table.

Condition on the state (s_0, s_1, s_2)				$r(T)=s_0$	$r(T)\neq s_0$		row number		
					$r(T)\neq \text{esc}$	$r(T)=\text{esc}$			
$s_0\neq\text{"Pass"}$ $\wedge s_0\neq\text{"Fail"}$	$s_1=N$			(s_0+1, X, s_1)	(s_0, I, s_1)	(s_0, E, s_1)	1, 2		
	$s_1\neq N$	$s_0=L$		$(\text{"Pass"}, X, s_1)$	*	*	5		
		$s_0\neq L$		(s_0+1, X, s_1)	*	*	6		
	$s_1\neq N$	$s_1\neq E$		*	*	(s_0, E, s_1)	7		
		$s_1=E$		*	*	$(\text{"Fail"}, E, s_1)$	8		
	$s_1\neq N$	$s_1=X$		*	(s_0-1, I, s_1)	*	9		
		$s_1=I$		*	(s_0, I, s_1)	*	10		
		$s_1=E$	$s_2=N$		*	(s_0, I, s_1)	*	11	
			$s_2\neq N$	$s_2=X$		*	(s_0-1, I, s_1)	*	12
				$s_2\neq X$		*	(s_0, I, s_1)	*	13
	$s_0=\text{"Pass"} \vee s_0=\text{"Fail"}$				(s_0, X, s_1)	(s_0, I, s_1)	(s_0, E, s_1)	3, 4	

A rearrangement of the conditions on the left is appropriate. The first subdivision should separate the relevant subsets of s_0 : (1) numerical values between 1 and L-1 inclusive, (2) {L} and (3) either "Pass" or "Fail". The second subdivision should separate the four values of s_1 : X, I, E and N. The third subdivision should separate, where necessary, the three relevant subsets of s_2 : {X}, {N} and {E, I}. After constructing a table with the appropriate structure, each of its cells is filled in by copying the contents of the corresponding cell in the table above. The result is as follows.

next state (s0, s1, s2) =			r(T)=s0			
			r(T)≠s0			
			r(T)≠esc	r(T)=esc		
s0 ∈ {1, ... L-1}	s1=X		(s0+1, X, s1)	(s0-1, I, s1)	(s0, E, s1)	
	s1=I		(s0+1, X, s1)	(s0, I, s1)	(s0, E, s1)	
	s1=E	s2=N		(s0+1, X, s1)	(s0, I, s1)	("Fail", E, s1)
		s2=X		(s0+1, X, s1)	(s0-1, I, s1)	("Fail", E, s1)
		s2 ∈ {E, I}		(s0+1, X, s1)	(s0, I, s1)	("Fail", E, s1)
s1=N		(s0+1, X, s1)	(s0, I, s1)	(s0, E, s1)		
s0=L	s1=X		("Pass", X, s1)	(s0-1, I, s1)	(s0, E, s1)	
	s1=I		("Pass", X, s1)	(s0, I, s1)	(s0, E, s1)	
	s1=E	s2=N		("Pass", X, s1)	(s0, I, s1)	("Fail", E, s1)
		s2=X		("Pass", X, s1)	(s0-1, I, s1)	("Fail", E, s1)
		s2 ∈ {E, I}		("Pass", X, s1)	(s0, I, s1)	("Fail", E, s1)
s1=N		(s0+1, X, s1)	(s0, I, s1)	(s0, E, s1)		
s0 ∈ {"Pass", "Fail"}			(s0, X, s1)	(s0, I, s1)	(s0, E, s1)	

The initial state is (1, N, N) and the value of out(T) is the same as the first component of the next state.

Note that the line for s0=L and s1=N can never apply. Only in the initial state is s1=N, and then s0=1. When s0=L, s0≠1, because L>1.

Note also that the lines for s2=N and for s2 ∈ {E, I} contain the same entries in the body of the table, both when s0 ∈ {1, ... L-1} and when s0=L. The lines for s2=N and s2 ∈ {E, I} can, therefore, be combined.

In the block of lines for s0 ∈ {1, ... L-1}, the lines for s1=I and s1=N contain the same entries in the body of the table. These lines can, therefore, be combined.

A state invariant is $s1=E \wedge s2=E \Rightarrow s0="Fail"$. This expression is true in the initial state and its truth is maintained by every state transition. Note that every next state in which s1=E and s2=E, i.e. every next state of the form (-, E, E), arises from a prior state in which s1=E. All such prior states lead to a next state in which s0="Fail". Therefore, the combinations of s1=E and s2=E in the lines in which s0 is not "Fail" can be dropped.

When the table above is revised accordingly, the result is the table

next state (s0, s1, s2) =			r(T)=s0	r(T)≠s0	
				r(T)≠esc	r(T)=esc
s0 ∈ {1, ..., L-1}	s1=X		(s0+1, X, s1)	(s0-1, I, s1)	(s0, E, s1)
	s1 ∈ {I, N}		(s0+1, X, s1)	(s0, I, s1)	(s0, E, s1)
	s1=E	s2 ∈ {I, N}	(s0+1, X, s1)	(s0, I, s1)	("Fail", E, s1)
		s2=X	(s0+1, X, s1)	(s0-1, I, s1)	("Fail", E, s1)
s0=L	s1=X		("Pass", X, s1)	(s0-1, I, s1)	(s0, E, s1)
	s1=I		("Pass", X, s1)	(s0, I, s1)	(s0, E, s1)
	s1=E	s2 ∈ {I, N}	("Pass", X, s1)	(s0, I, s1)	("Fail", E, s1)
		s2=X	("Pass", X, s1)	(s0-1, I, s1)	("Fail", E, s1)
s0 ∈ {"Pass", "Fail"}			(s0, X, s1)	(s0, I, s1)	(s0, E, s1)

where the initial state is (1, N, N) and the value of out(T) is the same as the first component of the next state.

Once the FSM reaches either of the states "Pass" or "Fail", the distinction between X, I, and E for s1 and s2 is superfluous and can be suppressed. I.e., any (valid) values can be assigned to s1 and s2, e.g. s1=X and s2=X.

The expression $s0=L \wedge s2=N \Rightarrow s1=X$ is a state invariant. It is true for the initial state and every state transition maintains its truth. Therefore, the N in the line for s0=L, s1=E and s2 ∈ {I, N} cannot apply and can be dropped. (Alternative explanation: The state (L, E, N) would correspond to the state after only one key, the "escape" key, had been pressed. In this case, the value of s0 would be 1 and the state would be (1, E, N). But L cannot be 1. Therefore, the state (L, E, N) is not a possible state.)

These considerations lead to the table below.

next state (s0, s1, s2) =			r(T)=s0	r(T)≠s0	
				r(T)≠esc	r(T)=esc
s0 ∈ {1, ..., L-1}	s1=X		(s0+1, X, s1)	(s0-1, I, s1)	(s0, E, s1)
	s1 ∈ {I, N}		(s0+1, X, s1)	(s0, I, s1)	(s0, E, s1)
	s1=E	s2 ∈ {I, N}	(s0+1, X, s1)	(s0, I, s1)	("Fail", X, X)
		s2=X	(s0+1, X, s1)	(s0-1, I, s1)	("Fail", X, X)
s0=L	s1=X		("Pass", X, X)	(s0-1, I, s1)	(s0, E, s1)
	s1=I		("Pass", X, X)	(s0, I, s1)	(s0, E, s1)
	s1=E	s2=I	("Pass", X, X)	(s0, I, s1)	("Fail", X, X)
		s2=X	("Pass", X, X)	(s0-1, I, s1)	("Fail", X, X)
s0 ∈ {"Pass", "Fail"}			(s0, X, X)		

where the initial state is (1, N, N) and the value of out(T) is the same as the first component of the next state.

Consider the states (L, I, -). They can be reached in a single transition from states (L, I, -) and (L, E, I) only. The state (L, E, I) can be reached in a single transition from states (L, I, -) only. Thus, the states (L, I, -) and (L, E, I) can be reached from themselves only. They cannot be reached from the initial state (because $L \neq 1$ and also because the values of s1 are incompatible). Therefore, the states (L, I, -) and (L, E, I) can be removed from the table above. The group of states (L, E, -) reduces to the single state (L, E, X), so the condition $s2=X$ becomes superfluous. Applying these simplifications results in the following table.

next state (s0, s1, s2) =			r(T)=s0	r(T)≠s0	
				r(T)≠esc	r(T)=esc
s0 ∈ {1, ..., L-1}	s1=X		(s0+1, X, s1)	(s0-1, I, s1)	(s0, E, s1)
	s1 ∈ {I, N}		(s0+1, X, s1)	(s0, I, s1)	(s0, E, s1)
	s1=E	s2 ∈ {I, N}	(s0+1, X, s1)	(s0, I, s1)	("Fail", X, X)
		s2=X	(s0+1, X, s1)	(s0-1, I, s1)	("Fail", X, X)
s0=L	s1=X		("Pass", X, X)	(s0-1, I, s1)	(s0, E, s1)
	s1=E		("Pass", X, X)	(s0-1, I, s1)	("Fail", X, X)
s0 ∈ {"Pass", "Fail"}			(s0, X, X)		

where the initial state is (1, N, N) and the value of out(T) is the same as the first component of the next state.

The fact that only four rows appear in the table for $s_0 < L$ and only two rows for $s_0 = L$ indicates that a further reduction of the state variables is possible. In reducing the table to the above form, it was pointed out that some combinations of values of the second and third state variables (s_1 and s_2) cannot occur. Of the remaining combinations, some are equivalent in terms of the subsequent behaviour of the FSM, as can be seen by examining the above table. In fact, there are only four groups of equivalent combinations of the state variables s_1 and s_2 , so the state variables s_1 and s_2 could be combined into a single state variable with only four values (see below).

Compare the above table with the last table in section 2.7.2 above. The two FSMs implement the same key sequencing logic, so their tables have a similar structure – four rows for s_0 (or n) less than L , two rows for s_0 (or n) equal to L , and one row for the final state. The tables differ in that the one in section 2.7.2 above outputs “Pass” or “Fail” only once and does not output the number of the next expected key, while the FSM in the table immediately above in this section outputs a value after every new term in the input trace.

The four groups of equivalent combinations of values of s_1 and s_2 that can occur and the corresponding values of the state variables w and e in the last FSM table in section 2.7.2 above are:

Equivalent combinations of values of s_1 and s_2 (equivalence classes of the state variables s_1 and s_2) (s_1, s_2)	corresponding values of w and e (w, e)
(X, X), (X, I), (X, N), (X, E)	(F, F)
(I, X), (I, I), (I, N), (I, E), (N, N)	(T, F)
(E, X)	(F, T)
(E, I), (E, N)	(T, T)

Summary: Deriving an FSM from a given TF table

A general procedure for deriving an FSM table from a given TF table is: Examine the cells in the TF table and identify the various variables whose values are needed to determine the value of the output function. Then reduce the list by deleting those variables whose values can be calculated from others in the list. Then examine the expressions in the conditions with a view to identifying that information actually needed to calculate the value of the output function. For example, fairly long expressions involving terms containing $r(p(\dots p(\dots)))$, $out(p(p(\dots p(\dots))))$ appear repeatedly in the example above. However, only certain forms of these expressions appear, all of which are comparing a term in the input trace with the value of a previous output, where one of only four results is relevant. Reformulating these expressions appropriately as a simple, general function of (in this case) only two variables in effect extracts the information needed to be carried by the FSM’s state from one step to the next in scanning the input trace, in this case, fewer values than the number of variables from which they are calculated.

What if the output depends on a term arbitrarily far back?

In the above example, the trace function “out” could be expressed as a function of the most recent three terms in the input sequence and of the previous three values of out. In other cases, however, there may be no upper limit (such as “three” in the above example) to the number of earlier terms in the input sequence or the number of previous values of the output function upon which the next value of the output function depends. Such a situation could, for example, arise if the value of the output function could depend on the position of a key (either up or down) and, therefore, on the direction of the last movement of the key in question. The last movement of that key could be arbitrarily far back in the input sequence.

When the value of the output function can be a function of a term or previous value of the output function arbitrarily far back in the sequence, the following approach is indicated. Extend the output function to include the additional information needed and given by the possibly arbitrarily distant previous input term or output value. In the situation alluded to in the previous paragraph, this might be an output value indicating the position (up or down) of every key on the keyboard. By including such information in the value of the output function, it becomes available in the immediately previous output value, and no longer arbitrarily far back in the sequence. In turn, that information finds its way into the state of the FSM derived from the TF table.

Appendix D. References

- [1] Baber, Robert L.; Parnas, David L.; Vilkomir, Sergiy A.; Harrison, Paul; O'Connor, Tony; "Disciplined Methods of Software Specification: A Case Study", International Conference on Information Technology Coding and Computing ITCC 2005, Las Vegas, NV, U.S.A., April 4-6, 2005.