Disciplined Methods of Software Specification: A Case Study*

Robert L. Baber, David L. Parnas, Sergiy A. Vilkomir
Software Quality Research Laboratory (SQRL)
Department of Computer Science and Information Systems, University of Limerick, Ireland
{Robert.L.Baber, David.Parnas, Sergiy.Vilkomir}@ul.ie

Paul Harrison, Tony O'Connor
Dell Products, Limerick, Ireland
{Paul_Harrison, Tony_O'Connor}@dell.com

Abstract

We describe our experience applying tabular mathematical approaches to software specifications. Our purpose is to show alternative approaches to writing tabular specifications and to help practitioners who want to apply such methods by allowing them to pick the best one for their problem.

The object for the case study is software used by Dell Products for testing the functionality of the keyboards on notebook computers. Starting from informal documents, we developed a variety of tabular representations of finite state machine specifications and tabular trace specifications. We found that the discipline required by these methods raised issues that had never been considered and resulted in documents that were both more complete and much clearer. The various tabular representations are compared from a user's point of view, i.e., clarity, consistency, unambiguity, completeness, suitability, etc

Keywords: *software, tabular specifications, finite state machine, traces, trace specifications.*

1. Introduction

Although both researchers and developers have long believed that it is valuable to have a precise statement of a program's requirements, there is surprisingly little evidence to support this belief. Producing a specification that is both precise and complete requires a lot of effort and is rarely done. Many doubt that it is worth doing. This paper reports on our experience in producing a precise requirements specification for a successful piece of software. We wanted to answer two types of questions:

1. Would the effort reveal issues or questions about the software that had been ignored or missed during its development and use?
2. Which of several available approaches would produce the most readable and useful document?

This paper reports on what was learned simply by preparing the specification. We are trying to give readers the benefit of our experience and discussions during the first phase of our project. The whole project is intended to improve our inspection and testing techniques and to see how effective they are in practice. The specification was written as a prelude to testing and inspecting previously developed software that is already in use. The project is continuing to see what can be learned by systematic, document based, testing and inspection.

It was important to us that the software was the product of a good conventional development approach. It is easy to show advantages of any method over a "strawman". We wanted to see if our mathematics based approaches could add value for a product where the users were already completely satisfied. It was also important that the software had been developed by people who were not interested in our methods.

It has been suggested [5, 10, 6, 4] that methods using tabular mathematical expressions are more suited for practical use than other mathematical approaches. In the present paper, we consider the practical application of two mathematical tabular approaches:

- Tabular representation of finite state machine (FSM) specifications (based on the original classical ideas of Moore [9] and Mealy [8]).
- Tabular trace specifications [2, 7, 13, 12].

* This work was supported by Science Foundation Ireland under SFI Grants 01/P1.2/C009 and 03/CE3/1405.

In this paper we compare various forms of known tabular approaches using a real piece of software as a common example. These tabular representations are considered from a user's point of view, i.e., the aspects of interest are their clarity, consistency, unambiguity, completeness, suitability, etc.

We developed tabular specifications for an existing program starting with informal statements of requirements and descriptions. As the object for the case study, we chose Streamlined Test On Portables (STOP) software by Dell Products; it is used for testing the functionality of different devices on portable (notebook) computers, in particular Keyboards. This program has been in use at Dell's Limerick Manufacturing Operation since early in 2003.

Both the developers and the users consider the program successful and all known issues had been resolved. STOP is important to the efficiency of the plant and the quality of the distributed product. STOP also had the advantage of being small enough and self-contained enough that it could be studied by a small group in a reasonable amount of time. Finally, STOP was accompanied by what appeared to be a well-written document that described what it was expected to do.

The remainder of this paper is structured as follows. An informal description of the STOP software is given in Section 2. Section 3 discusses the ambiguities that were discovered as we prepared the tabular specifications. Section 4 presents various FSM specifications. We consider tables for groups of states and groups of inputs as well as representations with multilevel headers, with conditions separated for states and inputs and with merging similar cells. Section 5 compares the FSM approaches with one that is "theoretically" equivalent but based on external descriptions known as "traces". Section 6 offers conclusions.

2. The case study: Notebook Keyboards Test software

Dell's facility in Limerick (Ireland) manufactures over 6000 notebook computers on an average day. Dell prides itself on delivering a superior experience to its customers. Its direct business model facilitates this by building a computer only when the customer's requirements are known. The computer is then to be built with no defects whilst minimizing the time to deliver to the customer. All aspects of testing (including notebook keyboard testing) that assure quality must also be performed quickly to ensure a good experience for the customer.

The keyboard testing is performed by employees who press all keys in a specified sequence to ensure that they are functioning correctly. STOP software supervises this testing to make sure that all functions are tested and that all quality requirements are met. It provides continuing guidance to the employee as the test proceeds.

The STOP software allows testing keyboards for the following defects:

1. Keyboard ribbon not connected
2. Key action signal not received (each key sends a 'down' signal to the operating system (OS) when pressed down and an 'up' signal to the OS when it is released)
3. Key 'Caps' (plastic covers) are not firmly attached.

Defect type 1 requires only a few key presses to ensure that the extremities of the ribbon are tested for connection to the computer (it is assumed that if the extremities are connected then middle is connected). However defect types 2 and 3 require that every key be depressed and released.

The STOP software provides two kinds of keyboard tests: a sequence test and a stuck key test. For the sequence test, only the 'down' signals are important, for the stuck key test, the 'up' signals are important as well. In the framework of our investigation, we formalized both tests as well as conditions of a transfer between tests, specifications for display, etc. Size limits for the present paper only allows us to consider here the keyboard sequence test. For that reason, we will ignore the 'up' signal.

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 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.
- 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. 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.

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

3. Anomalies found in the natural language requirements

As we began to work with the program, we found ambiguities in the English; we used alternate forms of the specifications to communicate the various possible interpretations. These were given to the developers with a "Which one did you want?" memo. The experience has shown the ability of the developers to read and write some of the specifications in the actual project.

Among the benefits we gained by translating the English statements of the requirements into mathematical language – and tabular form in particular – were the following:

- Lack of ambiguity: This process revealed ambiguities in the English text that we probably would have otherwise overlooked. For example, if the last key on the keyboard has been pressed in sequence and subsequently an "incorrect" key is pressed, must the operator press the last key again, or should this test be terminated as soon as the last key is pressed in sequence? This ambiguity raised the even more fundamental question, "How is the end of the test to be recognized?". The answer to this question was needed in order to specify precisely the conditions for starting the next test (not discussed in this paper).
- Completeness: The fact that needed information is missing is more apparent when one must fill in every cell in a table than when one only reads the English text, looking for missing information. Especially in complicated situations is it easy to overlook certain cases, but even with a very large table, it is easy to check that no cell is left empty. In this case study it was found that the initial informal requirements did not specify what action to take if the incorrect key was pressed when testing the first key in the test sequence. This specification had been assumed by the program designer and accepted by the program user without it explicitly being brought to their attention.
- Consistency: Contradictions in the English statement of requirements can be overlooked, especially if the logic is complicated or the document is long. When filling in cells in tables, contradictory information converges onto a single cell of the table, where the contradiction becomes readily apparent. In this example, some parts of the requirement documentation strongly suggested that pressing the escape key out of sequence should cause the expected key to revert to the preceding key, while other parts indicated otherwise.
- Suitability: Expressing the English requirements in mathematical notation forces one to think more deeply and extensively about the specification itself. Overlooked but desirable functionality is often identified. In this example, there is no guarantee that the test will ever end; the operator can continue to press "incorrect" keys indefinitely. (Certain hardware errors could lead to such a situation.) This raises the question whether or not the specification should be modified to prevent this possibility, e.g. by imposing a time limit on the test or time limits on pressing each key.
- Clarity: Long passages of text from scattered places in the original documents were expressed more succinctly in the tables.

We discovered the above anomalies in the process of formulating the documented requirements mathematically. The relevant documents were statements of requirements, descriptions of the earlier version of the keyboard test program and instructions for testing the program, all in English text. The descriptions of the previous program became part of the specification for the new one because of the requirement that the new program do at least as much as the old one did while improving at least its non-functional performance, e.g. speed.

On first reading, the available documents seemed to be clear and, except for a few of conflicts, unambiguous, but when we started to write mathematical expressions for the specification, ambiguities and gaps became apparent. One example was the criterion for ending the sequence test mentioned. The requirements stated that the next test (the stuck key test mentioned in section 2 above), should be performed when the sequence test is successfully completed. This statement seems clear, but what constitutes "completed"? The sequence test is, in a sense, complete when the last key on the keyboard is detected as having been pressed

Table 1. An FSM table in a basic form

	Input_1	Input_2	Input_3
Previous_state_1	New_state_1_1	New_state_1_2	New_state_1_3
Previous_state_2	New_state_2_1	New_state_2_2	New_state_2_3
Previous_state_3	New_state_3_1	New_state_3_2	New_state_3_3
Previous_state_4	New_state_4_1	New_state_4_2	New_state_4_3

when expected. But if the stuck key test is performed immediately thereafter, before the operator has released the key, that key will be detected as still down, i.e. stuck, causing a false failure of the stuck key test. Some delay between completing the sequence test and starting the stuck key test is obviously required, but such a delay was not defined in the requirements documents.

We looked at the program code and noticed that the stuck key test is started as soon as any key goes up after the last key on the keyboard was pressed in sequence. If the key going up is the last key on the keyboard, all is fine, but if it is some other key, which is possible, a false failure of the stuck key test would occur. These possibilities highlighted the need to specify the criteria for completing of the sequence test and for starting the next test precisely, unambiguously and appropriately. The fact that this possibility had not caused false failures of keyboards in actual productive use of the program could be attributed to the layout of the keyboards and the way one swipes a finger over the keyboard during the test. However, this explanation does not apply to the layout of a new model keyboard being introduced. Thus, because of our disciplined approach, a possible problem was discovered before it arose in the production use of the program.

In order to aid the decision making process, we wrote several specifications, each representing a different interpretation or assumption about the ambiguous or missing information.

4. Tabular representations of finite state machines specifications

4.1 Informal comments

The FSM is a widely used model for describing computer systems and software. An FSM may be represented either by a graph in which each node represents a state and each edge represents a transition from one state to the next state or in a variety of tabular formats. In the approach illustrated below, each row in the table corresponds to a particular previous state, each column corresponds to a particular input value and the cell at the intersection of the row and the column contains the new state of the system. An example of such a table with 3 inputs and 4 states is presented in Table 1. A similar table can describe the outputs of the machine, showing how they depend on values of states and inputs.

FSMs arising in real systems often have a large number of states and inputs, making this type of table impractical. A more practical approach is to partition the states and inputs into classes characterised by predicate expressions, significantly reducing the size of the table. In the resulting table, each row corresponds to a group of previous states and each column corresponds to a group of inputs. The entries are expressions which evaluate to the next state. We used this approach to specify the STOP software.

For the STOP software, the main component of the software state is the sequence number of the expected key. However, the behavior of the program also depends on whether or not an incorrect key or the escape key has been pressed previously. The state of the program is completely determined by the history of keystrokes. There are two possibilities (yes-no) for pressing an incorrect key and two possibilities for pressing the escape key so the total number of situations to consider is four times the number of keys in the sequence. In other words, we can partition all states into four groups where each group contains a state for every key.

The inputs of the STOP software are described by the sequence number of the key pressed and can be divided into three main groups: an expected key, the escape key (only if not expected) and an incorrect (any other) key. The contents of these groups depend on the most recently passed key. The creation of FSM specifications for the sequence test of the STOP software is discussed in Section 4.2.

4.2 FSM tabular representation

We created an FSM table for the STOP software using the following model:

- Each intermediate state S of the test program is represented by a triple (n, w, e), where
  - n is the number of the expected key,
  - $w \in \{T, F\}$ indicates whether an incorrect key has been pressed and

Table 2. STOP software FSM tabular representation

Previous State S	$k = n \wedge n \neq L$	$k = n \wedge n = L$	$k \neq n \wedge k \neq esc \wedge n = 1$	$k \neq n \wedge k \neq esc \wedge n \neq 1$	$k \neq n \wedge k = esc$
(n, F, F)	(n+1, F, F)	Pass	(n, T, F)	(n-1, T, F)	(n, F, T)
(n, F, T)	(n+1, F, F)	Pass	(n, T, F)	(n-1, T, F)	Fail
(n, T, F)	(n+1, F, F)	N/A	(n, T, F)	(n, T, F)	(n, T, T)
(n, T, T)	(n+1, F, F)	N/A	(n, T, F)	(n, T, F)	Fail

Table 3. FSM tabular representation with a multilevel header

Previous State S	$k = n$		$k \neq n$		
			$k \neq esc$		$k = esc$
	$n \neq L$	$n = L$	$n = 1$	$n \neq 1$	
(n, F, F)	(n+1, F, F)	Pass	(n, T, F)	(n-1, T, F)	(n, F, T)
(n, F, T)	(n+1, F, F)	Pass	(n, T, F)	(n-1, T, F)	Fail
(n, T, F)	(n+1, F, F)	N/A	(n, T, F)	(n, T, F)	(n, T, T)
(n, T, T)	(n+1, F, F)	N/A	(n, T, F)	(n, T, F)	Fail

after this the expected key has not been pressed (in this situation $w = T$, otherwise $w = F$),

- o $e \in \{T, F\}$ indicates whether or not the most recently pressed key was the escape key (if it is the escape key, $e = T$, otherwise $e = F$).

- k is the sequence number of the current input key.
- 'Pass' and 'Fail' are final (absorbing) states, i.e. the result of the test program.
- $S_0 = (1, F, F)$ is the initial state of the model.
- 'esc' is the sequence number of the escape key in the sequence keys of a keyboard (can be different for different types of keyboards).
- L is the total number of keys on a keyboard including special buttons. The keys are numbered from 1 to L, so k and n take up any value in the set $\{1, ... L\}$.

Table 2 gives a formal FSM representation of the STOP software behavior, which was informally described in Section 2. Because the situations $n = 1$ and $n = L$ require special consideration, the number of different groups of inputs increases. "N/A" corresponds to situations which cannot arise in reality because if $w = T$ then $n \neq L$.

As an output of the STOP program we consider the function, which returns the number of the expected key for all intermediate states and results of the test ('Pass' and 'Fail') for the final (absorbing) states. The output table can be easily derived from Table 2.

4.3 Representation with a multilevel header

The conditions defining the columns of the table should partition the input space, i.e. they should be mutually exclusive and exhaustive. To make the partitions clearer and consequently reduce the number of mistakes, a modified version of the FSM table with a multilevel header was produced.

To create a multilevel header, the whole input domain is first divided into two subdomains, forming level 1 of the header. Then each subdomain is divided

Table 4. FSM tabular representation with separated conditions

Conditions	Previous State S	k = n	k ≠ n	
			k ≠ esc	k = esc
n = 1	(1, F, F)	(2, F, F)	(1, T, F)	(1, F, T)
	(1, F, T)	(2, F, F)	(1, T, F)	Fail
	(1, T, F)	(2, F, F)	(1, T, F)	(1, T, T)
	(1, T, T)	(2, F, F)	(1, T, F)	Fail
1<n ∧ 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)	Fail
	(n, T, F)	(n+1, F, F)	(n, T, F)	(n, T, T)
	(n, T, T)	(n+1, F, F)	(n, T, F)	Fail
n=L	(L, F, F)	Pass	(L – 1, T, F)	(L, F, T)
	(L, F, T)	Pass	(L – 1, T, F)	Fail
	(L, T, F)	N/A	N/A	N/A
	(L, T, T)	N/A	N/A	N/A

Table 5. FSM tabular representation with merged cells

Conditions	Previous State S	k = n	k ≠ n	
			k ≠ esc	k = esc
n = 1	(1, F, F)	(2, F, F)	(1, T, F)	(1, F, T)
	(1, F, T)			Fail
	(1, T, F)			(1, T, T)
	(1, T, T)			Fail
1<n ∧ n<L	(n, F, F)	(n+1, F, F)	(n-1, T, F)	(n, F, T)
	(n, F, T)			Fail
	(n, T, F)		(n, T, F)	(n, T, T)
	(n, T, T)			Fail
n=L	(L, F, F)	Pass	(L – 1, T, F)	(L, F, T)
	(L, F, T)			Fail
	(L, T, F)	N/A	N/A	N/A
	(L, T, T)			

into new subdomains, which forms level 2 of the header, etc. The fact that conditions at each level are mutually exclusive and exhaustive can be seen at a glance.

In that way, the logical expression for each column is represented as a tree. Connection of all corresponding with a column conditions from the every level using the logical conjunction forms a full condition for a column. An FSM table with a multilevel header for STOP software is given in Table 3.

To reduce the number of levels, auxiliary predicates can be introduced. This technique is considered below in Section 5.2 for trace tabular specifications but it can be used for FSM specifications too.

4.4 Representation with separated conditions

An advantage of the FSM tables above is their compactness. The shortcomings that we noted include that the logical expressions in the headers of columns are a mixture of conditions for states and conditions for inputs. The modification of the FSM table below eliminates this shortcoming, but at the cost of increasing the size of the table. Table 4 separates logical conditions for inputs and for states. Conditions for inputs are placed into headers of columns and conditions for states are placed into headers of rows.

This separation creates a more natural partition of the input domain and makes tables more easily understood and used.

4.5 Representation with merged cells

Some adjacent cells in Table 4 have the same content. To avoid duplicating information, such cells can be merged as in Table 5.

Table 6. Trace specifications: values of N(T) for traces T of length 2 or more

Condition							Value N(T))
N(p(T)) = Pass							Pass
N(p(T)) = Fail							Fail
N(p(T))≠ Pass ∧ N(p(T))≠ Fail	r(T)= N(p(T))	N(p(T)) = L					Pass
		N(p(T)) ≠ L					N(p(T))+1
	r(T)≠ N(p(T))	r(T)≠ esc	r(p(T))≠ N(p(p(T)))	r(p(T))= esc	p(p(T))= _		N(p(T))
					p(p(T)) ≠_	r(p(p(T)))= N(p(p(p(T))))	N(p(T))–1
						r(p(p(T))) ≠ N(p(p(p(T))))	N(p(T))
				r(p(T)) ≠ esc			N(p(T))
			r(p(T)) = N(p(p(T)))				N(p(T))-1
		r(T)= esc	r(p(T))=esc	N(p(p(T))) = esc			N(p(T))
				N(p(p(T))) ≠ esc			Fail
			r(p(T)) ≠ esc				N(p(T))

Merging adjacent cells containing the same information improves clarity and readability by illustrating more clearly which cases are handled in the same way, by reducing the density of the text in the table, and by reducing general visual clutter.

5. Tabular trace specifications

5.1 Trace specifications without canonical traces

In our discussion of the finite state machine representations we mentioned that the state was entirely determined by the sequence representing the history of the keystrokes. If that is the case, it should be possible to avoid the step of picking a state representation and write the specification entirely in terms of the keystroke history and the outputs. This is the approach that was called the "Trace Assertion Method", which has been studied in a sequence of papers beginning with Bartussek & Parnas' paper [2]. This section explores the use of a trace-based model.

Most papers on trace specifications exploit the concept of equivalence of traces. We are describing finite state machines, but the set of traces is infinite. Consequently, the traces can always be partitioned by an equivalence relation and a finite set of traces can be represented as canonical representatives of the equivalence classes. In the present paper we use a different approach, namely trace specifications without defining a class of canonical traces. In this approach, a function, that gives the output of a new trace for every trace and every input, should be determined. The practical possibility to determine such function depends on the "depth" of dependency of the output on input traces.

The following terms and functions are defined in addition to those in Section 4.2:

T is a trace denoting a sequence of key depressions. I.e., T is a finite sequence of key numbers.

' _ ' is the empty trace.

N(T) is an output function, returning the number of the expected key after trace T or the result of the test. The range of N(T) is {1, ... L} ∪ 'Pass' ∪ 'Fail'.

To address trace specifications without canonical traces for STOP software, we only need to determine values of N(T) for all possible traces T. We define the value of N(T) inductively, firstly for an empty trace and for a trace of length 1:

N(_) = 1

N(k) =

Condition	Value N(k)
k = 1	2
k ≠ 1	1

and then in Table 6 for a trace of length 2 or more. In Table 6 two standard functions on traces are used. The value of the function r(T) is the most recent (latest, newest) term (sequence number of a key) 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≠_.

5.2 Using auxiliary predicates

An advantage of the table above is that all conditions are given in the clear explicit multilevel form and for each level it is easy to check that the conditions partition the whole domain (are mutually exclusive). The shortcoming of such tables is that for a large

Table 7. Trace specifications using auxiliary predicates

Condition			Value N(T))
$N(p(T)) = Pass$			Pass
$N(p(T)) = Fail$			Fail
$N(p(T)) \neq Pass \wedge N(p(T)) \neq Fail$	$r(T) = N(p(T))$	$N(p(T)) = L$	Pass
		$N(p(T)) \neq L$	$N(p(T)) + 1$
	$r(T) \neq N(p(T)) \wedge r(T) \neq esc$	FirstAbnormal	$N(p(T)) – 1$
		¬ FirstAbnormal	$N(p(T))$
	$r(T) \neq N(p(T)) \wedge r(T) = esc$	ReadyToFail	Fail
		¬ ReadyToFail	$N(p(T))$

number of levels of condition the table becomes too complicated and it is hard manually to track all chains of conditions (but tools could help in such situations).

One way to reduce complexity of tables is using predefined auxiliary predicates. It is possible to combine traces for which the output function returns the same value in one group and to use one specific predicate for every group of traces. This approach could give very simple tables like

Condition	Value
Predicate 1	Value 1
Predicate 2	Value 2
Predicate 3	Value 3

but quite complicated predicates. In other words, we just move the complexity from the table part to the predicate part.

Often a better way is to share the complexity between a table and predicates. It this way, predicates describe some internal combinations of conditions. It gives possibility to have a clear structure of the table (small amount of condition levels) and not too complicated predicates. An example of such an approach appears below for STOP software.

The predicate *FirstAbnormal* describes a situation before first abnormal pressing (for a specific key). In this situation and after pressing an incorrect key it is necessary to repeat pressing last proper key, i.e. the number of the expected key is changed from n to n-1. If *FirstAbnormal* is wrong, the number of the expected key is not changed after wrong pressing. Predicate *ReadyToFail* describes a situation when the program fails after pressing the escape key. If *ReadyToFail* is wrong, the number of the expected key is not changed after pressing the escape key. Defining formally,

FirstAbnormal = $[r(p(T)) = N(p(p(T))) \vee (r(p(T)) \neq N(p(p(T))) \wedge r(p(T)) = \text{'esc'} \wedge p(p(T)) \neq \_ \wedge r(p(p(T))) = N(p(p(p(T)))))]$

ReadyToFail = $[r(p(T)) = \text{'esc'} \wedge N(p(p(T))) \neq \text{'esc'}]$

Then the Table 6 from the previous section can be modified to Table 7.

The table can become easier to read if we name even simple predicates to explain their meaning:

FinalKey = $[N(p(T)) = L]$,

NextKeyPressed = $[r(T) = N(p(T))]$, etc.

5.3 Trace function approach

The rules for the behaviour of the software being specified here are relatively complex requiring looking back several events in the history of the inputs. To make the specification more readable, we define several auxiliary predicates. We also separate the conditions involving these auxiliary predicates from the other conditions. In addition, the definitions of N(T) where T is empty or of length 1 are included in the same table. The result is Table 8.

The auxiliary predicates appearing in Table 8 are defined below, where equality is defined to be false if either or both sides are undefined where evaluated [11].

Name	Meaning	Definition
keyOK	most recent key is the expected one	$r(T)=N(p(T))$
keyesc	most recent key is the escape key	$r(T)=esc$
prevkeyOK	key before the most recent key was the expected one	$r(p(T))= N(p(p(T)))$
prevkeyesc	key before the most recent key was the escape key	$r(p(T))=esc$
preprevkeyOK	key two keys before the most recent key was the expected key	$r(p(p(T)))= N(p(p(p(T))))$
prevexpkeyesc	key expected before the most recent key was the escape key	$N(p(p(T)))= esc$

Table 8. Trace function

N(T)=			T = _	¬(T = _) ∧		
				N(p(T))=1	**1<N(p(T))<L**	**N(p(T))=L**
keyOK				2	N(p(T))+1	Pass
¬keyOK	¬keyesc	(¬prevkeyOK ∧ prevkeyesc ∧ preprevkeyOK) ∨ prevkeyOK			N(p(T))-1	N(p(T))-1
		¬prevkeyOK ∧ prevkeyesc ∧ ¬preprevkeyOK			N(p(T))	N(p(T))
		¬prevkeyOK ∧ ¬prevkeyesc	1	1	N(p(T))	N(p(T))
	keyesc	¬prevkeyesc		1	N(p(T))	N(p(T))
		prevkeyesc ∧ ¬prevexpkeyesc		Fail	Fail	Fail
		prevkeyesc ∧ prevexpkeyesc			N(p(T))	N(p(T))

Table 8 defines a function whose domain is a set of traces, including the empty trace. The specification covers traces during the test only, not before it is started or after it is finished. This is different from the previous trace tables and from the FSM tables in which Pass and Fail are treated as states and the behaviour in those states is fully defined.

The empty cells in Table 8 represent cases that cannot arise because the conditions in the column header and the row header cannot occur at the same time (i.e. the conjunction of these conditions is always false).

6. Conclusions

It has long been known that mathematics brings a great increase in precision. However, precision is not enough. Mathematical specifications can be precisely wrong and/or precisely incomplete. In addition to the precision that we gain by mathematics we need a disciplined systematic approach that helps us to consider all the special cases that are so easy to overlook. Tabular definitions of functions are, in this respect, better than axiomatic or equational approaches. If one has the discipline needed to either check each table for completeness and consistency or construct the table in such a way that it must be complete and consistent, one will be forced to consider the subtle cases along with the obvious ones.

Translating the natural language requirements into a mathematical notation can lead to significant improvements in the suitability, completeness, consistency, unambiguity and clarity of the requirements and the specification (as discussed in Section 3).

In addition, the program designer can benefit from a mathematical specification. Variables, conditions and the table structure in the specification can be utilized by the program designer when structuring and writing program code. Major parts of the program can even be derived mechanically from a specification in mathematical form as suggested by Dijkstra [3] and illustrated in more detail by Baber [1].

In this case example, we identified several parts of the actual program that would probably have been implemented with a single subprogram had the program designer started from a mathematical specification. We also identified several superfluous conditions in the program that probably cost the programmer unnecessary time and effort.

Similarly, a mathematical specification can benefit a reviewer of the program code and testers. For example, conditions in the specification can draw their attention to logical complexities often leading to errors in the program.

The above benefits can be realised by most disciplined attempts to represent natural language requirements in a mathematical form. However, the choice of mathematical form can lead to other benefits specific to that form. In this case study a number of mathematical forms were experimented with. In particular, readability and understandability for all stakeholders can be improved by

- structuring conditions hierarchically,
- displaying conditions on the state separately from conditions on the input,

- merging adjacent cells and
- using auxiliary predicates.

These are important considerations when communicating specifications to people with different backgrounds and different levels of mathematical knowledge.

Although it is known that the state machine model approach is theoretically equivalent in descriptive power to approaches making assertions about traces, the examples in this paper show that from a practical point of view the two are quite different. To produce the FSM descriptions, one must first design a representation for the state (unless the number of states is so small that one can simply enumerate them). The choice of the representation is arbitrary in the sense that there are many representations that will work equally well. Completing the table is then much like designing a program. In the trace approach, there is no state representation. Instead one looks at the inputs and classifies the input sequences (traces) so that each trace in a class has similar output behaviour. The behaviour can be expressed entirely in terms of externally observable quantities rather than in terms of an arbitrarily chosen state representation. The FSM method has the advantage that it is familiar to Programmers and Engineers. The trace approaches have the advantage that they are expressed directly in terms that a "user" would use.

We are not able to conclude that one method is better than another from such a limited experience. Our paper is only intended to illustrate various approaches to mathematical tabular specifications and give some practical guidance in writing such specifications.

7. References

[1] Baber, Robert L., *Mathematically Rigorous Software Design*, 2002 September, http://www.cas.mcmaster.ca/~baber/Courses/46L03/MRSDLect.pdf.

[2] Bartussek, W., Parnas, D.L., "Using Assertions About Traces to Write Abstract Specifications for Software Modules", *Lecture Notes in Computer Science* (65), Information Systems Methodology, Proceedings ICS, Venice, 1978, Springer Verlag, pp. 211-236. Reprinted as Chapter 1 in Hoffman, D.M., Weiss, D.M. (eds.), *Software Fundamentals: Collected Papers by David L. Parnas*, Addison-Wesley, 2001, 664 pgs., ISBN 0-201-70369-6

[3] Dijkstra, E. W., *A Discipline of Programming*, Prentice-Hall, 1976

[4] Heitmeyer, C., "Software Cost Reduction", in *Encyclopedia of Software Engineering*, John J. Marciniak, editor, ISBN: 0-471-02895-9, January 2002

[5] Heninger, K.L., Kallander, J.W., Shore, J.E., Parnas, D.L., "Software Requirements for the A-7E Aircraft", *NRL Report 3876*, November 1978, 523 pgs

[6] Janicki, R., Parnas, D.L., Zucker, J., "Tabular Representations in Relational Documents". In *Relational Methods in Computer Science*, Chapter 12, Ed. C. Brink and G. Schmidt, Springer Verlag, 1997, pp. 184-196. Reprinted as Chapter 4 in Hoffman, D.M., Weiss, D.M. (eds.), *Software Fundamentals: Collected Papers by David L. Parnas*, Addison-Wesley, 2001, 664 pgs., ISBN 0-201-70369-6

[7] Janicki, R.; Sekerinski, E. "Foundations of the trace assertion method of module interface specification", *IEEE Transactions on Software Engineering*, Volume: 27, Issue: 7, July 2001, Pages: 577 – 598

[8] Mealy, G. H., "*A Method for Synthesizing Sequential Circuits", Bell System Tech. J.*, Vol 34, September 1955, pp. 1045–1079

[9] Moore, E. F. "Gedanken-experiments on Sequential Machines", *Automata Studies, Annals of Mathematical Studies*, no. 34, Princeton University Press, Princeton, N. J., 1956, pp 129 – 153

[10] Parnas, D.L., *Tabular Representation of Relations*, CRL Report 260, McMaster University, Communications Research Laboratory, TRIO (Telecommunications Research Institute of Ontario), October 1992, 17 pgs.

[11] Parnas, D.L., "Predicate Logic for Software Engineering", *IEEE Transactions on Software Engineering*, Vol. 19, No. 9, September 1993, pp. 856 - 862 Reprinted as Chapter 3 in Hoffman, D.M., Weiss, D.M. (eds.), "*Software Fundamentals: Collected Papers by David L. Parnas*", Addison-Wesley, 2001, 664 pgs., ISBN 0-201-70369-6

[12] Wang, Y., *Specifying and Simulating The Externally Observable Behaviour of Modules*, CRL Report 292, McMaster University, Canada, August 1994.

[13] Wang, Y., Parnas D.L., "Simulating the Behaviour of Software Modules by Trace Rewriting", *IEEE Transactions of Software Engineering*, Vol. 20, No. 10, October 1994, pp. 750 - 759