Tabular Expressions in Software Engineering

Article · January 2003

2 authors:

Alan Wassyng
McMaster University
82 PUBLICATIONS   **614** CITATIONS

SEE PROFILE

Ryszard Janicki
McMaster University
187 PUBLICATIONS   **1,482** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Presenting a new definition within the rough set domain: Optimal Rough Approximation Set. View project

Local search for attribute reduction View project

Tabular Expressions in Software Engineering

Alan Wassyng & Ryszard Janicki
McMaster University
Department of Computing and Software
Information Technology Building
1280 Main Street West
Hamilton Ontario L8S 4K1
Canada
Tel: 905.525.9140
wassyng@cas.mcmaster.ca & janicki@cas.mcmaster.ca

Abstract: Tabular expressions (tables) have been used in the software development process for more than twenty years. In addition, research has been ongoing to develop semantics for tabular expressions.   At this stage in the history of tabular expressions we see a slight split between those working to improve the semantic understanding of tables, and those using tables to document real industrial software projects. Those conducting research in the semantics of tables have their sights set on expressive mathematical models of tables that are general enough to cope with all known forms of tabular expressions, and that can be used as the basis for software tool support of tables. The other group uses tables in real-world projects under the normal constraints of schedule, budget and the skill set of current software professionals. Thus, champions of tabular expressions for use in industry are focussed on developing notations and approaches that will be accepted and useful in an industrial setting. To bridge both worlds, we present here a brief history of tabular expressions, motivation for the development of semantics for tables, a description of a semantic definition of tables, and a discussion on notation and use of tables in practice based on many years of experience of using tables in industrial software development projects.

Key-words: condition table, function table, tabular expression, tabular semantics, formal specification, documentation

1. Introduction

Tabular expressions (tables) have been around now for many years. Decision tables [9] and state transition tables date back to early years in computer science. In the late 1970s Parnas and others at the U.S. Naval Research Laboratories used tabular representations to document requirements. Since then, a number of projects have used tables to document requirements and software design. In addition, there have been a number of papers on the semantics of tabular expressions. An obvious question is "why another paper on tabular expressions?" The answer is that after the first couple of exploratory papers, most papers have been a theoretical examination of the semantics of tabular expressions. There have been a few papers that discuss the use of tables in actual software projects, but those papers were much more general, and tables were an item in a larger picture. This paper examines (briefly) the need for semantics, presents one approach that has been used to describe the semantics of tabular expressions, and then goes on to a much more practical discussion of how tables are being used in industry, when they can be used, and what we can do to make tabular expressions more accessible to software professionals in the future.

Section 2 presents a brief history of tabular expressions to place them in context. Section 3 describes why we need semantics at all. Section 4 presents an easy to digest summary of tabular semantics as described in [14]. Section 5 shows how tables are being used in practice, illustrated by principles and examples from the Darlington Nuclear Generating Station Shutdown Systems, in Ontario Canada [28]. Finally, section 6 presents conclusions and an indication of what would be fruitful future research.

2. History of tabular expressions

Although Parnas does not want tabular expressions to be referred to as "Parnas tables" (private communication), they often are - for a very good reason. Parnas and others introduced tabular expressions to the world in the requirements for the A-7E aircraft [8], [7], [3], [21]. More than any other person, Parnas has championed the use of tabular expressions in documenting software [19], [20], [24], [23], [13]. He also suggested the first semantic analysis of tabular expressions [22], and has encouraged the development of software tools for tabular expressions [26].

Since those early days there has been continuous activity regarding tabular expressions.

At least two organisations, Ontario Hydro / Ontario Power Generation and U.S. Naval Research Laboratories have fashioned their approach to software development around the use of tabular expressions [5], [6], [2], [18] and [28]. There have been a number of software projects that used tables, for example [27], as well as software engineering publications that champion tables [16], and advocate the use of tables for real-time monitoring systems [25].

Janicki has been a driving force in developing semantics for tabular expressions over the past decade [10], [11], [13], [12] and [14], and a number of others have contributed by adding their own view of semantics [4] and [15].

Abraham [1] described tabular expression semantics as developed by Janicki, showed how the semantics were being used in the development of software tools, and included a survey of tables in use at that time.

There has also been work on the mathematical transformations required to convert normal into inverted tables and vice versa [30].

3. The need for semantics

In the early days of tabular expressions, informal semantics were given so that authors and readers would have the same understanding of what the tables meant. It is interesting to note that these early, informal semantics were written in terms of the contents of the tables. For example, given the table in Figure 1

Condition	Result
	f_name
Condition 1	res 1
Condition 2	res 2
... ...	...
Condition n	res n

Figure 1 - Simple function table

we could say that the table is equivalent to

        if Condition 1 then f_name = res 1
        elseif Condition 2 then f_name = res 2
        elseif …
        elseif Condition n then f_name = res n

we would also require that

        Condition i $\wedge$ Condition j $\Leftrightarrow$ FALSE $\forall$i,j=1,..,n i $\neq$ j   (disjointness)
and   Condition 1 $\vee$ Condition 2 $\vee$ … $\vee$ Condition n $\Leftrightarrow$ TRUE   (completeness)

We will see that the above informal semantics looks nothing like the semantics that have been developed for general tabular expressions over the past decade. In fact, most formal semantics of tabular expressions do not include the disjointness and completeness requirements. These requirements come about because in practice we typically need the tables to describe functions in an unambiguous way. The formal semantics of tables is usually concerned (first and sometimes exclusively) with the structure of the table, not the contents.

The most important reason for developing semantics is to ensure that authors and readers of the tables have the same unambiguous understanding of the meaning of the tables. In simple cases such as for the table above, the semantics in terms of the if … elseif structure may suffice. However, as soon as the tables get more complex, or we need the semantics for different reasons as discussed below, those rudimentary semantics are no longer useful.

If tabular expressions are to be truly useful in software practice, they will have to be supported by software tools. So, another high priority reason for creating general semantics for tables, is to enable us to develop software tools for creating, editing and transforming tables.

Tabular expression semantics are useful/necessary in other situations as well. For instance, tables and sub-tables are an indispensable notation device in partitioning the behaviour in a requirements document. It is also often useful to link tables of different types. In both cases, without semantics we are not sure if the composition of the tables is unambiguously defined.

As an example, let us look at the combination of a state transition table and a typical function table. The tables are modified/simplified versions of tables used in a requirements document for SDS1 at Darlington.

The state transition table is shown in Figure 2.

		Receive State				
		Start	**Receive Idle**	**Wait for Msg**	**Wait for Rest**	**Process Msg**
NOT [Timeout]	Init	Receive Idle {1} see 2.4	-	-	-	-
	Link On	-	Wait for Msg {2} see 2.4	-	-	-
	Link Off	-	-	Receive Idle {1} see 2.4	Receive Idle {1} see 2.4	Receive Idle {1} see 2.4
	Msg Rcvd	-	-	Process Msg {3} see 2.4	Process Msg {3} see 2.4	-
	Part Msg	-	-	Wait for Rest {2} see 2.4	Wait for Msg {2} see 2.4	-
	Processed & NOT [Link Off]	-	-	-	-	Wait for Msg {2} see 2.4
Timeout		-	-	Wait for Msg {2} see 2.4	Wait for Msg {2} see 2.4	-

Figure 2 - State Transition Table with links to a Function Table

In the above table, - indicates no action, and the events Link On and Link Off are defined by

Link On = Enable requested & NOT PRIOR (Enable requested)
Link Off = NOT (Enable requested) & PRIOR (Enable requested)

Other events are defined by references to other functions. The state transition table above, shows the new state in each cell, and also shows the output on entering that new state by a tag inside curly brackets as well as a reference to a function table.

The function table 2.4 is shown in Figure 3.

			Result				
	Condition		**M_CPPF**	**M_Type**	**f_Feedbk**	**M_Gain**	**M_NOP**
{3}	Enable requested & M_Comm exists	Good header	see 2.5	see 2.6	see 2.6	see 2.9	see 2.10
		NOT [Good header]	DNE	DNE	e_Invalid	DNE	DNE
{2}	Enable requested & NOT [M_Comm exists]	NOT [non-fatal error] & NOT [not in 2 passes]	DNE	DNE	e_Undef	DNE	DNE
		non-fatal error	DNE	DNE	e_Invalid	DNE	DNE
		NOT [non-fatal error] & not in 2 passes	DNE	DNE	e_2pass	DNE	DNE
{1}	NOT [Enable requested]		DNE	DNE	e_Undef	DNE	DNE

Figure 3 - Function Table referenced by a State Transition Table

In the above table, DNE means Does Not Exist.

In earlier versions of these tables the tags in curly brackets were not included. Readers were therefore told to look at function table 2.4 to determine the required behaviour on entering a particular state, without being told exactly where in the table to continue reading. Readers who understood the behaviour on which the events were based for instance, had no difficulty in determining exactly where to look in function table 2.4. However, readers without that knowledge had some difficulty in understanding the required behaviour. For example, the only obvious link between the events in the state transition table and predicates in function table 2.4 is the one between "Link Off" and "NOT [Enable requested]" represented by {1}. Even "Link On" presents a problem since it could apply to either {2} or {3} if we look only at its connection with "Enable requested". To help alleviate this problem, we added the tags in curly brackets. Now the links to particular rows in 2.4 are explicit. The tags also raise some questions. In general, do we want to reference just specific rows or columns in a table? How does a software tool interpret these tags? What implications are there for the verification process in cases like this? Do we have an unambiguous description of behaviour when we link different kinds of tables in this way? Does it make sense to say "see 2.4" in the cells in the state transition table? For that matter, does it make sense to reference sub-tables in this way even if the table and sub-tables are of the same type (as in function table 2.4 itself)? Without semantics these questions are sometimes surprisingly difficult to answer. The problem seems to be that any one person may think the answers are obvious – unfortunately, they seem obvious but still differ from person to person.

For many years we have worked on making tabular expressions easier to read and comprehend so that they will be used in industry. One of our ideas was to create what we called *natural language expressions* (NLEs), that we will discuss in some detail in section 5.1. These NLEs are natural language phrases that have specific meaning to domain experts. For example, we may have an NLE "Low power setpoint is requested or cancelled" and when it is used in a table, we use "Low power setpoint is requested" or "Low power setpoint is cancelled". So NLEs look a little different from normal mathematical functions. They typically use "is" and "are" as assignment operators, and the range of the NLE is included explicitly in the NLE, separated by "or". Again, it is important that readers have an unambiguous understanding of NLEs (they are also defined using function tables), and we must be able to build software tools that allow us to parse and use these expressions.

4. Semantics of tabular expressions

4.1. Semantics

As discussed in section 2, semantics for tabular expressions have been developed by a number of people over the past twenty years. There are sometimes significant differences between the different approaches. The semantics we will use are those developed in [14].

To illustrate the principles behind the semantics we will show various stages in their development in the context of a normal table.

	$y = 10$	$y > 10$	$y < 10$
$x \geq 0$	0	y^2	$-y^2$
$x < 0$	x	$x + y$	$x - y$

Figure 4 - Stage 0. A normal table

We need to be able to describe the tabular expression structure. To this end, we can decompose tables into *headers* (H_1, H_2, …, H_n) and a single *grid* (G).
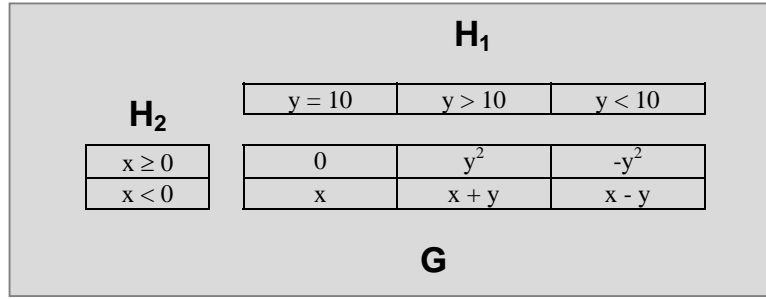

Figure 5 - Stage 1. Assigning headers and a grid

We are now able to define the *raw table skeleton* of a table, T, written as T^{raw}. T^{raw} is a tuple, defined such that $T^{raw} = (H_1, H_2, …, H_n, G)$, where G is the grid indexed by the headers H_1, H_2, …, H_n. The elements of the set $\{H_1, H_2, …, H_n, G\}$ are known as the *components* of the table T, written as Comp(T).

The next step is to identify the flow of information amongst the components of the table. For instance, to understand the meaning of the tabular expression, where should we start reading the table, and how does this lead to a result?
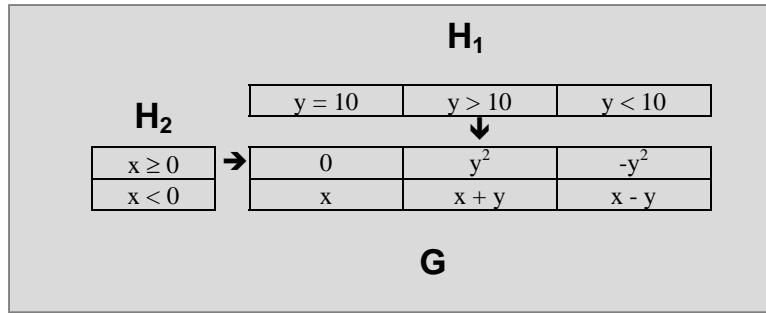

Figure 6 - Stage 2. Adding information flow

Together with the information flow, we need to identify whether individual headers and the grid represent *guard cells* or *value cells*. Guard cells contain the predicates while the value cells contain the results.
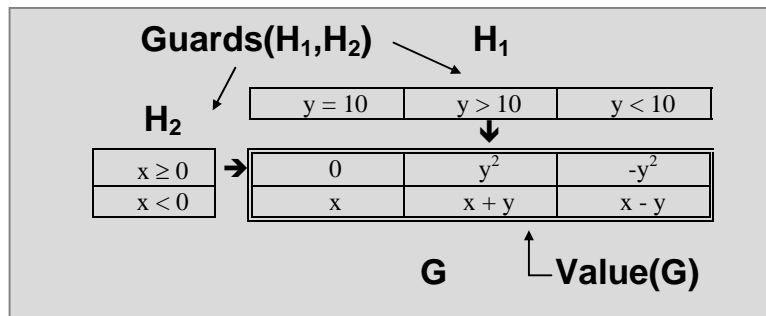

Figure 7 - Stage 3. Identifying guard and value cells

From here on, we will identify value cells in a table by using double line borders for those cells.

The information flow together with the identification of guard and value cells represents the *Cell Connection Graph* (CCG) of the tabular expression. We describe the CCG graphically as follows.
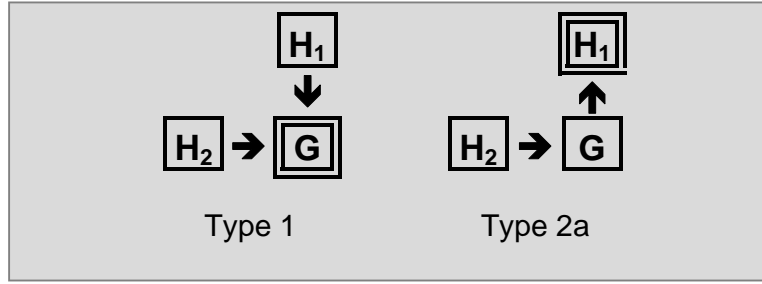

Figure 8 – Stage 4. Examples of Cell Connection Graphs

The "type" classification is taken from [14]. Type 1 corresponds to a normal table, while Type 2a corresponds to a number of tables, including an inverted table.

Once we have the concept of the CCG we are in a position to define the *medium table skeleton* of a table T, written as T^{med}. T^{med} is a tuple such that $T^{med} = (CCG, H_1, H_2, \ldots, H_n, G)$. CCG is represented by an icon fashioned from the shapes shown in Figure 8. For example, the icons for the above examples would be:

 and
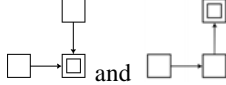
The relevant predicates that govern the tabular expression are contained within the guard cells, but so far we have not defined how the predicates are formed, nor have we yet specified how the final relation is formed. To achieve this we define a *well done table skeleton* of a table T. This is a tuple such that
$T^{well} = (P_T, r_T, C_T, CCG, H_1, H_2, \ldots, H_n, G)$, where P_T is the *table predicate rule* (how the guard cells are combined using logical operators to form the predicate), r_T is the *table relation rule* (how the predicate components are combined using set operators), and C_T is the *table composition rule* (how the component relations are combined to form the total relation).
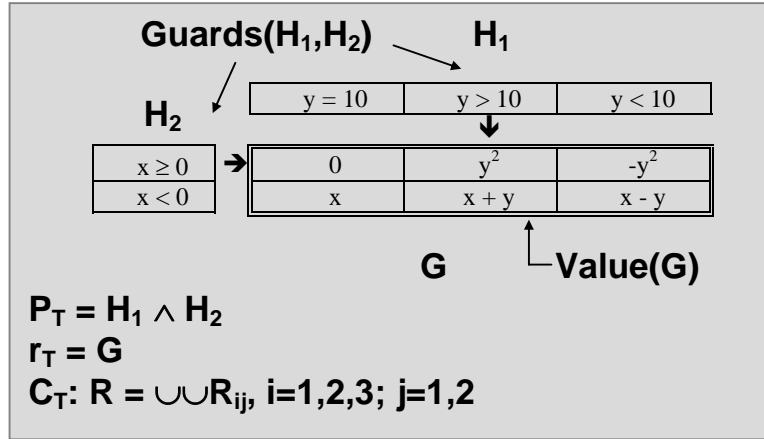
Applying this to our example we get:

Figure 9 - Stage 5. Specifying predicate, relation & composition rules

From this we eventually get a definition of a tabular expression (or table).

A tabular expression is a tuple, $T = (P_T, r_T, C_T, CCG, H_1, H_2, \ldots, H_n, G; \Psi, IN, OUT)$, where Ψ is the mapping that assigns predicate expressions to guard cells and relation expressions to value cells, IN is the set of inputs and OUT is the set of outputs, so predicate expressions have variables over IN and relation expressions has variables over IN × OUT. This adds information concerning the actual cell contents to well done table skeletons.

4.2. Classification

Tabular expressions can be classified according to many criteria. Early classifications by Parnas and others were along the lines of function tables and event tables. These classifications reflected the content of the tables. Function tables were then further sub-divided into normal, inverted and other types. Ever since Janicki proposed the concept of the CCG, classification of tabular expressions has moved closer to being dependent on semantic concepts. Normal and inverted tables can be identified through the CCG, but to distinguish between inverted and generalised decision tables for instance, we need to include C_T.

In [14] we suggested that different classifications can be made according to the semantic entities CCG, P_T, r_T, C_T and Ψ. Using Ψ will take us full circle in a way, since we will be returning to classifying tables according to their content. Using Ψ we can differentiate between function and relation tables. It also turns out that the semantics may not provide enough of a basis for a comprehensive classification scheme. For instance, semantically, there is no difference between horizontal and vertical condition tables, but it has certainly proved useful to differentiate between the two.

5. Tabular expressions in software development

Our experience is that although many different kinds of tables have now been identified, projects tend to use a very limited number of tables within the project. This makes excellent sense when you consider that familiarity with the tables is vital to their efficient and correct use. To justify the use of another kind of table, that table should have clear and compelling benefits over the set of tables already in use.

This may change in future years if the use of tabular expressions becomes as prevalent as we hope and believe it will.

In the case of the Darlington shutdown systems, SDS1 and SDS2, the SDS1 team used horizontal condition tables and state transition tables in the requirements documents, and vertical condition tables in the software design. The SDS2 team primarily used structured decision tables. In each case, the team members, even new members, became proficient in creating and reading those tables. As a future study, it would be interesting to examine those project documents to see whether the use of any different kinds of tables would have dramatically improved the documentation of any particular function.

The remainder of this section will use table formats as used in SDS1 at Darlington to illustrate a number of points.

5.1. Tables for requirements

The table format we used for SDS1 requirements documents is shown in Figure 1. This has come to be known as a *horizontal condition table* (HCT). The original table format at Ontario Hydro / Ontario Power Generation was what is now known as a *vertical condition table* (VCT, see section 5.2). (In earlier years we simply called them program function tables (PFTs.) HCTs were invented because system engineers seemed to prefer reading the tables left-to-right. Also, as mundane as it may seem, we pay quite a lot of attention to how easy it is to format tables on a page. It seems that HCTs are typically well-suited to requirements behaviour. Reasons for this may include that, at the requirements level, we tend to specify the behaviour of a single function, that there are relatively few nesting levels in the predicates for these tables, and that there is often a significant number of disjoint predicates that we have to include in each table. In the interests of using the tabular structure to highlight the structure of the predicates as much as possible, we extended the structure of tables so that adjoining cells are considered to be "anded". This is shown in Figure 10.

Figure 10 – Extended Horizontal Condition Table

It is interesting to note that when we designed these tables for SDS1, we were not aware of the decision made by those developing the semantics of tabular expressions to reserve double lines for value cells. This seems the sort of "standard" that should be promoted. With this in mind, we suggest that the tables shown in Figure 1 and Figure 10 should be re-designed as shown in Figure 11.

Figure 11 – Revised Horizontal Condition Table

We could decide whether to include the assignment operator in the cell defining the function's name depending on the specific usage of the tables in a project. For example, if all the tables define a simple = operator, then it seems that we could safely just note its implicit use.

One of the reasons we cannot assume an equality operator is our use of natural language expressions. One of the major challenges facing requirements authors is to find a way of partitioning the requirements documentation so that readers can comprehend the overall behaviour through the presentation of individual function descriptions. One of the best ways of partitioning the requirements behaviour is to use NLEs. These natural language phrases are intuitively meaningful to domain experts, and should preferably represent black-box visible behaviour.

As an example, consider the following function table.

Condition	Result
	f_NOPsp
NOP Low Power setpoint is requested	k_NOPLPsp
NOP Low Power setpoint is cancelled & NOP Abnormal 2 setpoint is requested	k_NOPAbn2sp
NOP Low Power setpoint is cancelled & NOP Abnormal 2 setpoint is cancelled & NOP Abnormal 1 setpoint is requested	k_NOPAbn1sp
NOP Low Power setpoint is cancelled & NOP Abnormal 2 setpoint is cancelled & NOP Abnormal 1 setpoint is cancelled	k_NOPnormsp

Figure 12 - Function table that uses natural language expressions

The above table uses a number of NLEs to describe the required behaviour. As an example, we can look at the definition of one of those, "NOP Abnormal 1 setpoint is requested or cancelled".

Condition	Result
	NOP Abnormal 1 setpoint is requested or cancelled
$(m\_NOPspAbn1ON = e\_NotPressed)$ & $(m\_NOPspAbn1OFF = e\_NotPressed)$	No Change
$(m\_NOPspAbn1ON = e\_NotPressed)$ & $(m\_NOPspAbn1OFF = e\_Pressed)$	cancelled
$(m\_NOPspAbn1ON = e\_Pressed)$ & $(m\_NOPspAbn1OFF = e\_NotPressed)$	requested
$(m\_NOPspAbn1ON = e\_Pressed)$ & $(m\_NOPspAbn1OFF = e\_Pressed)$	requested

Figure 13 – Definition of a natural language expression

From this we can see that the NLEs help to partition the behaviour. Try for example to specify f_NOPsp without the use of NLEs. Actually, NLEs were first developed to make the requirements documents more readable for domain experts – engineers without a background in software development. The NLE's effect on partitioning the behaviour was an unexpected bonus. It is also important to note that their use does not compromise the precision of the behavioural description since each NLE is mathematically defined. When we package the function tables in the requirements documents, we present all the so-called top-level function tables first. These tables completely define the behaviour of the system, but some of the behaviour is informally defined since it relies on the intuitive understanding of the NLEs. The definitions of the NLEs provide the formal precision that is missing at that top-level. These function tables are documented in a completely different section of the requirements document – one that comes after the top-level descriptions.

It is reasonably easy to see how NLE's should be defined and used when the tabular expression in use is as simple as the function tables we used in SDS1. For more complex tabular expressions this may not be as obvious. In such cases we can use the semantics developed earlier to guide us, remembering that Ψ, IN and OUT in section 4.1 deal with the contents of the cells in a table.

An area that needs a lot more attention than it has received in general, is the choice of notation for what goes into the cells in a table. If we want tabular expressions to be used in industry, we have to be careful with the examples we use to "sell" the technology.

As an example, consider Figure 14 which shows one of the notations used when Ontario Hydro first used function tables in the late 1980s [2]. These tables were modelled on the A-7E function tables [3].

	((APPLY($\lor$, (*SGF.TRIPH(d)* $\lor$ (*SGF.DEADH(d)* $\land$ *SGF.INITHI(d)*)), d=1..5) $\lor$ APPLY($\lor$, (*SGF.TRIPL(d)* $\lor$ (*SGF.DEADL(d)* $\land$ *SGF.INITLO(d)*)), d=1..4) $\lor$ *SGF.TRIPLN* $\lor$ (*SGF.DEADLN* $\land$ *SGF.INITLON*)) $\land$ ('|p_tmr[##sgf_p##]| >= 1118 $\lor$ '|r_tmr.tmr[##sgf_r##]| >= 960) $\land$ ('|tripcond[##sgf_c##]| <> TRUE)) $\lor$ ('|mantrip[##sgf_t##]| = TRUE)	((APPLY($\land$, (*SGF.NOTRIPH(d)* $\lor$ (*SGF.DEADH(d)* $\land$ ¬*SGF.INITHI(d)*)), d=1..5) $\land$ APPLY($\land$, (*SGF.NOTRIPL(d)* $\lor$ (*SGF.DEADL(d)* $\land$ ¬*SGF.INITLO(d)*)), d=1..4) $\land$ (*SGF.NOTRIPLN* $\lor$ (*SGF.DEADLN* $\land$ ¬*SGF.INITLON*))) $\lor$ ('|p_tmr[##sgf_p##]| < 1118 $\land$ '|r_tmr.tmr[##sgf_r##]| < 960) $\lor$ ('|tripcond[##sgf_c##]| = TRUE)) $\land$ ('|mantrip[##sgf_t##]| <> TRUE)
|do_tbl.pt[##sgf_t##]|'	##t##	##u##

Figure 14 - A function table with notation that is difficult to write and read

A number of changes were made at Ontario Power Generation to make function tables easier to work with. The first change was to eliminate all delimiters, the symbols such as |, # and * that were used to give the reader information about the identifiers. Instead of these paired delimiters, we simply prefixed identifiers by a character followed by an underscore. m_name indicates a monitored variable, c_name indicates a controlled variable, f_name an internal function, k_name a constant, etc. These convey the necessary information but are easy to read. They still have the advantage that an alphabetic sort of identifiers groups identifiers by type. Another change we made was to eliminate the 'x and x' indicators. 'x represented a pre-value, and x' represented a post-value. This is not really useful since any result identifier is automatically a post-value, and any variable that appears in a predicate or in a value cell must be a pre-value. We do not use the logician's symbols for "or", "and" and "not". Rather, we simply use "OR", "&' and "NOT". Finally, we do not use $\forall$ or $\exists$. Rather we use "for all" and "there exists".

These simple changes in notation together with NLEs resulted in our function tables being accepted (most important) and understood by engineers and software professionals in industry.

Finally, as far as the use of tables in documenting requirements is concerned, we should discuss domains in which we feel tables are useful and domains in which they may not be. Much of our own personal experience is in requirements for real-time / embedded systems. Typically in these systems we are primarily concerned with describing black-box behaviour of how a set of controlled environment variables should behave given a set of monitored environmental variables. These systems are ideally suited to the use of tabular expressions. If you look at the history of tables in industrial usage, they started with such a system (the A-7E) and the documented successes have been in projects such as the Darlington shutdown systems. They are clearly just as useful in many non-real time applications, particularly in the description of behaviour that is truly black-box. However, we do not yet know how to use tables effectively to document a graphical user interface for example. Even more difficult, is to use tables to define a specific algorithm. As we have seen, tabular expressions have a huge strength in that they define strictly black-box behaviour, from pre-conditions to a post-condition. They are strictly sequence independent. We do not know how they would be used to document Gauss Elimination for example.

5.2. Tables for software design

Although horizontal condition tables are well suited to requirements documentation, they are not as well suited to documenting software designs. Functions in a software design are more likely to specify a number of results dependent on the same or similar behaviour. The predicates tend to be nested to a number of levels and there are relatively few disjoint predicates at the top level. For these reasons, vertical condition tables are more appropriate.

Many of the same principles apply to VCTs as they did to HCTs, but the choice of identifier names and notation may be dictated by the programming language used to implement the design. Obviously there is a decision to be made as to how close the software design notation should be to the programming language, but it is often the case that in long-term projects in which the programming language is fixed, there may be significant advantages to tying the design to the specific language that will be used.

A software design has to serve a number of purposes. The main purpose is to decompose requirements into a form that will satisfy a number of goals. Major goals are typically that the decomposition should facilitate intellectual control of the designed behaviour, that future changes can be made safely and efficiently, and that the design provides a solid basis for the development of code. The design should also provide documentation that aids testing and verification – mathematical verification in the case of high-reliability applications.

A powerful feature of tabular expressions is that they describe behaviour without imposing a sequence of operations on that description. Tables are entirely free of any sequence dependence. In other words, a single table describes a relation strictly in terms of its black-box behaviour. Possibly the single most positive experience we had with tables in developing SDS1 and SDS2 at Darlington [28], was that the code that was implemented from tabular descriptions was extremely well-structured. Not only was the code an order of magnitude better structured than earlier versions, but it turned out to be remarkably easy to analyse the code and develop function tables from the code – without looking at the software design. In a safety-critical software environment, this was a crucial point in facilitating the verification of the code.

In a safety-critical context, it is not sufficient to code the required behaviour. The code has to implement the required behaviour in a fail-safe manner. It therefore has to cope with anticipated hardware failures. This is achieved by adding to the code certain self-check routines. These routines monitor the state of the hardware and report malfunctions. However, we cannot always detect such failures, especially when they are intermittent. In these cases, although we cannot guarantee "correct" behaviour, we need to (almost) guarantee "safe" behaviour. To do this we need to be able to convey to the coder exactly what safe behaviour is. If the software design is documented using function tables, then it turns out there is an effective way to document this information. As mentioned, SDS1 uses VCTs to document the behaviour in the software design. A simple convention of always placing the *safe state* (if there is one) in the right-most column of the table, allows us to create coding guidelines that uses this information to structure the code so that the safe state is the one that is most likely to be entered in the case of hardware failure.

FOR i= 1 TO KNUMXX	l_SVal[i] <= (l_STSP - KTHYXX)	l_SVal[i] < l_STSP AND l_SVal[i] > (l_STSP - KTHYXX)	l_SVal[i] >= l_STSP
STSXX[i]	$FALSE	NC	$TRUE
STIXX[i]	NC	NC	$TRUE

Figure 15 - Vertical condition table with safe state in right column

Figure 15 describes the behaviour for a *sensor trip* (STSXX[i]) in the shutdown system. If the signal of interest (l_SVal[i]) is above or equal to the *setpoint* (l_STSP) then the sensor is tripped ($TRUE). This is documented in the right column. The middle column covers the case when the signal is in the deadband – there is no change to the sensor trip. The left column covers the case when the signal is below the setpoint less the hysteresis. in this case the sensor is not tripped ($FALSE). The safe state in this behaviour is clearly to set the sensor trip to tripped. The coder uses this information to ensure that in the case of any of the likely CPU

malfunctions, or in the case of a RAM failure (for instance), the software will most likely return a value of $TRUE for STSXX[i]. In this particular case, the safe state is quite obvious. However, in other cases, the safe state is not as obvious. It is unlikely that someone not familiar with the requirements and their implementation in the design will be able to recognise that safe state.

Finally, we have to discuss the types of software designs that tabular expressions can document. As discussed in the previous section, tables are sequence independent and we have not yet found a way of using them to document specific algorithms. Their usage in software design is very broad however. It is also possible to sequence blocks of behaviour using tables. In the SDS1 software designs for instance, design blocks are identified. Each block can be documented by a table, but then the blocks themselves are placed in a specific sequence. The outputs of one block/table become the inputs of the next block/table.

5.3. Designing test cases from tabular expressions of behaviour

One of the crucial tasks in designing test cases is to identify boundary cases. If we have the requirements documented using tables, we can use those tables as a basis for designing validation, random testing, and software integration test cases and oracles. The tables define black-box behaviour, but also give a clear indication of boundary cases that have to be tested. Consider Figure 13 for instance. This table clearly describes the test cases involving the NOP Abnormal 1 pushbuttons. These are digital inputs so the 4 cases were probably obvious anyway. However, in other cases the disjoint predicates are likely to involve specific values for setpoints and other black-box visible items. Sometimes it is not simply a test of all possible boundaries. For instance in Figure 12 there is a precedence described in the function table. The tables are especially useful for testing, because, as noted, they not only highlight possible test case, they also provide the oracle for those tests.

The same is true for developing unit tests from the tables in the software design. In SDS1, a huge majority of the unit test cases are developed from the software design function tables. The code sometimes includes additional predicate boundaries, created when implementing fail-safe behaviour for instance. However, these additional cases are relatively small in number.

5.4. Tables in verification and review

In many software development life-cycles, especially those for the development of highly reliable systems, the software design is reviewed and (sometimes) verified against its requirements. In a similar way, the code is reviewed and (sometimes) verified against the software design. In all these cases, having the behaviour documented in tabular expressions facilitates the process. Since the tables present the behaviour in such a structured way, comparing the behaviour in one document with that in another document is greatly facilitated. The fact that we already have, and are developing better semantics all the time, means that we can build software tools to help us with these processes. Although the tables help, the mathematical verification of a software design against its (formal) requirements is an extremely time-consuming and labour intensive task. However, progress has been made in automating the verification using tools such as PVS [17].

Because of the formal structure of the tables, they lend themselves to mathematical manipulation which we can use to "prove" that one set of tables is behaviourally equivalent to another set of tables. In general we accept the fact that the lower level process may add behaviour not already in the higher level process, but that additional behaviour can be isolated and then justified. The completeness and disjointness properties also make the review process easier. It is possible to check these properties in a mechanised way (much of the time), and then use the assumption that they are true to drive the review. One of the complicating factors in this regard is that for hard real-time systems, time itself makes some of the properties quite difficult to check.

In section 5.2 we saw that it has proved to be remarkably easy to reverse engineer the function tables from the code (it helps that we developed the coding guidelines with this in mind). We achieve this also through placing strategic comments in the code. These comments provide links to important items in the software design.

6. Conclusion

Tables have proved to be invaluable in documenting requirements and software designs. We have presented reasons why they are so useful in all the major stages of software development. In particular we have presented examples of how tables are used in requirements, software design, testing, verification and how effective they are when used as a basis to code from.

Semantics for tabular expressions are crucial to their use in practice and also to the development of software tools for creating, editing and transforming tables. An example of table semantics was presented and reasons why semantics are important were discussed.

Notations used in tables, and the actual structure of the tables are fundamental to getting tables accepted by software professionals. Some examples of how notation and table structure can be improved in very simple ways were highlighted. This is an area that seems almost completely overlooked.

A real strength of tabular expressions is their sequence independent view of behaviour. This enables us to deliver true black-box descriptions of behaviour. This strength is also a weakness. We have yet to find a way to use tabular expressions to document algorithms, simply because in an algorithm, sequence of operations is supremely important. We used this to illustrate when tables seem to be effective and when they do not.

We have not said very much about software tools for tables. This is not because we do not believe that they are extremely important. On the contrary, the long-term survival of tabular expressions as a practical notation device for documenting software projects is likely to hinge on the successful creation of tool support. However, although there have been a number of attempts to produce software tools for tables, the projects that we are familiar with can rightly be classified as "proof of concept". We are yet to see the emergence of professional quality tools. This brings us to our next topic – future work.

Top of the list of future work, in our opinion, is the creation of professional quality tool support for tabular expressions. New versions of semantics for tabular expressions are being developed on an almost continuous basis. We suggest that we now have sufficient knowledge of both semantics and table usage to be able to produce high quality software tools for tabular expressions.

There are some topics still that need further research in terms of semantics for tables.

For example, semantics that cope with arrays; semantics that deal with concurrency (see [29]) and for dealing with time are all still in the early stages, if they exist at all.

Classification of tables is going to be important if tabular expressions become as popular as they should in industry. Since classification schemes depend to a large extent on the semantics, it will be interesting to note whether we can arrive at classifications that make sense in the context of the few competing semantic approaches that have already been presented in the literature.

7. References

[1]   R. Abraham, *Evaluating Generalized Tabular Expressions in Software Documentation*, M. Eng. Thesis, Dept. of Electrical and Computer Engineering, McMaster University 1997, also CRL Report 346, McMaster University, Hamilton, Ontario, Canada, 1997.

[2]   G. H. Archinoff, R. J. Hohendorf, A. Wassyng, B. Quigley, M. R. Borsch, *Verification of the Shutdown System Software at the Darlington Nuclear Generating Station*, International Conference on Control and Instrumentation in Nuclear Installations, Glasgow, U.K., 1990, No. 4.3.

[3]   P. C. Clements, *Function Specification for the A-7E Function Driver Module*, NRL Memorandum Report 4658, U.S. Naval Research Lab., 1981.

[4]   J. Desharnais, R. Khédri, A. Mili, *Towards a Uniform Relational Semantics for Tabular Expressions*, Proc. of RELMICS 98, Warsaw 1998.

[5]   C. Heitmeyer, A. Bull, C. Gasarch, B. Labaw, *SCR: A Toolset for Specifying and Analyzing Requirements*, Proc. 9th Annual Conf. on Computer Assurance, COMPASS'95, Gaithersburg, MD, 1995.

[6]   C. Heitmeyer, *Using the SCR Toolset to Specify Software Requirements*, Proceedings, Second IEEE Workshop on Industrial Strength Formal Specification Techniques (WIFT'98), Boca Raton, FL, Oct. 19, 1998.

[7]   K. L. Heninger, *Specifying Software Requirements for Complex Systems: New Techniques and their Applications*, IEEE Transactions on Software Engineering, 6, 1, 2-13, 1980.

[8]   K. L. Heninger, J. Kallander, D. L. Parnas, J. E. Shore, *Software Requirements for the A-7E Aircraft*, NRL Memorandum Report 3876, U.S. Naval Research Lab., 1978.

[9]   R. B. Hurlay, *Decision Tables in Software Engineering*, Van Nostrand Reinhold Company, New York 1983.

[10]   R. Janicki, *Towards a Formal Semantics of Parnas Tables*, 17th International Conference on Software Engineering, ICSE'95, IEEE Computer Society, Seattle, WA, 231-240, 1995.

[11]   R. Janicki, *On Formal Semantics of Tabular Expressions*, CRL Report 355, McMaster University, Hamilton, Ontario 1997.

[12]   R. Janicki, R. Khédri, *On Formal Semantics of Tabular Expressions*, Science of Computer Programming, 189-214, 39 2001.

[13]   R. Janicki, D. L. Parnas, J. Zucker, *Tabular Representations in Relational Documents*, in C. Brink, W. Kahl, G. Schmidt (eds.): Relational Methods in Computer Science, Springer-Verlag 1997.

[14]   R. Janicki, A. Wassyng, *On Tabular Expressions*, In D.A. Stewart, ed. Proceedings of CASCON 2003, Markham, Ontario, Canada, 38-52, October 2003.

[15]   W. Kahl, *Compositional Syntax and Semantics of Tables*, SQRL Report No. 15, McMaster University, Hamilton, Ontario, Canada, 2003.

[16]   R. Khédri, *Requirements Scenarios Formalization Technique: N Versions Towards One Good Version*, in W. Kahl, D. L. Parnas, G. Schmidt (eds.), Relational Methods in Software, Elsevier, 19-37, 2001.

[17]   M. Lawford, H. Wu, *Verification of Real-Time Control Software Using PVS*, In P. Ramadge and S. Verdu, eds., *Proceedings of the 2000 Conference on Information Sciences and Systems*, vol. 2, Dept. of Electrical Engineering, Princeton University, Princeton, NJ, pp. TP1-13--TP1-17, 2000.

[18]   M. Lawford, J. McDougall, P. Froebel, G. Moum, *Practical Application of Functional and Relational Methods for the Specification and Verification of Safety Critical Software*, Proc. of AMAST'2000, Lecture Notes in Computer Science 1816, Springer, 73-88, 2000.

[19]   D. L. Parnas, G. J. K. Asmis, J. D. Kendall, *Reviewable Development of Safety Critical Software*, International Conference on Control and Instrumentation in Nuclear Installations, Glasgow, U.K., 1990, No. 4.3.

[20]   D. L. Parnas, G. L. K. Asmis, J. Madey, *Assessment of Safety-Critical Software in Nuclear Power Plants*, Nuclear Safety, 32,2, 189-198, 1991.

[21]   D. L. Parnas, *A Generalized Control Structure and Its Formal Definition*, Communications of the ACM, 26, 8, 572-581, 1983.

[22]   D. L. Parnas, *Tabular Representation of Relations*, CRL Report 260, Telecommunications Research Institute of Ontario (TRIO), McMaster University, Hamilton, Ontario, Canada, 1992.

[23]   D. L. Parnas, J. Madey, *Functional Documentation for Computer Systems Engineering*, Science of Computer Programming, 25, 1, 41-61, 1995.

[24] D. L. Parnas, J. Madey, M. Iglewski, *Precise Documentation of Well-Structured Programs*, IEEE Transactions on Software Engineering, 20, 12, 948-976, 1994.

[25] A. J. van Schouwen, *The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems*, Technical Report 90-276, Queen's University, CIS, TRIO, Kingston, Ontario, Canada, 1990.

[26] SERG - Software Engineering Group, *Table Tool System Developer's Guide*, CRL Report 339, TRIO, McMaster University, Hamilton, Ontario, Canada 1997.

[27] A. Wassyng, *Software Requirements for AECB Project 2.314.1*, GARD Research Consulting Inc., 23141-DOC-4, Revision 2, 1995.

[28] A. Wassyng, M. Lawford, *Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project*, In K. Arakai, S. Gnesi, and D. Mandrioli, eds. FME 2003: International Symposium of Formal Methods Europe Proceedings, Pisa, Italy, LNCS Vol. 2805, 133-153, Springer-Verlag, September 2003.

[29] Y. Yang, *Modelling Concurrency by Tabular Expressions*, M. Sc. Thesis, Dept. of Computing and Software, McMaster University, Hamilton, Ontario, Canada 2002.

[30] J. Zucker, *Transformations of Normal and Inverted Function Tables*, Formal Aspects of Programming, 8 679-705, 1996.