Tabular Expressions for Software Documentation

Dr. Ying Jin, Dr. Adam Balaban, David L. Parnas

Abstract

Conventional Mathematics is sufficient for abstractly defining the contents of software documents in terms of relations but the representation of those relations is often complex and hard to parse when conventional expressions are used. We present tabular expressions, explain why the work, and show how their meaning can be defined by giving the equivalent conventional expression.

How can Documents be Both Precise and Readable?

This is precise:

$$(((\exists\, i,\ B[i] = \text{`x}) \wedge (B[\text{j'}] = x) \wedge (\text{present'} = \textbf{true})) \vee ((\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \ne x)) \wedge (\text{present'} = \textbf{false}))) \wedge (\text{`x} = \text{x'} \wedge \text{`B} = \text{B'})$$

But,

- Few people want to read even simple examples like this.
- You have to parse it all and understand a lot before you can find what you want.

Lets try anyway - just once.

$$(((\exists\, i,\ B[i] = \text{`x}) \wedge (B[\text{j'}] = x) \wedge (\text{present'} = \textbf{true})) \quad \vee ((\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \ne x)) \wedge (\text{present'} = \textbf{false}))) \bigwedge (\text{`x} = \text{x'} \wedge \text{`B} = \text{B'})$$

$$(((\exists\, i,\ B[i] = \text{`x}) \wedge (B[\text{j'}] = x) \wedge (\text{present'} = \textbf{true})) \bigvee ((\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \ne x)) \wedge (\text{present'} = \textbf{false})))$$

$$(((\exists\, i,\ B[i] = \text{`x}) \bigwedge (B[\text{j'}] = x) \wedge (\text{present'} = \textbf{true}))$$

$$((\forall\, i,\ ((1 \le i \le N) \Rightarrow B[i] \ne x)) \bigwedge (\text{present'} = \textbf{false})))$$

$$(\forall\, i,\ ((1 \le i \le N) \Longrightarrow B[i] \ne x))$$

It is not more formal or difficult than a programming language but parsing is difficult.

How can Documents be Both Precise and Readable?

The following is readable:

"Set i to indicate the place in the array B where x can be found and set present to be true. Otherwise set present to be false"

but vague and unclear:

- **What do you do if the array is of zero length?**
- **What do you do if x is present more than once?**
- **Are you allowed to change B or x?**
- **What does the "otherwise" mean: Does it mean, if you don't do what you should, or if there is no place in the array where x can be found, or if there are many places where x can be found?**

We have all seen worse examples of such sentences.

How can Documents be both Precise and Readable?

This is readable and more precise than the text above:

Specification for a search program:

	x can be found in B	x can not be found in B
j' must be	a place where x can be found in B	any number at all
present' must be	**true**	**false**

How can Documents be both Precise and Readable?

This is precise and readable (by trained people).

1 The first can be input to mathematics based tools but is hard for people.

2 The second seems clear but does not answer key questions.

3 The third is clearer but does not answer one key question and cannot be input to reliable tools.

4 The fourth is complete and could be processed by tools. It is, in theory, equivalent to the first, but in practice much better.

	$(\exists\, i,\ B[i] = x)$ [a]	$(\forall\, i,\ \neg(B[i] = x))$
j' | present'=	B[j'] = x true	*true* false

$\wedge$ NC(x, B)

a. These tables depend on using a logic in which evaluating a partial function outside of its domain yields "*false*" for all built-in predicates.

The table parses the expression for the reader, but is still mathematics.

Tabular Notation Is For All Software Design Documents

We first “discovered it” in the requirements area, but it can be used for

- system design documents
- module interface specifications
- module internal design documents

These documents can be tested for completeness and consistency.

Producing these documents is part of “the discipline”.

Checking them is the remainder of the discipline.

They make the programming much faster and more reliable.

But, most important:

They can be read by human beings.

• They were read and corrected by pilots, engineers, telephone-operators, and even managers.

They help us to reduce the number of errors.

Requirements for a Keyboard Testing System

N(T)=

	¬(T = _) ∧		
T = _	1 < N(pr(T))< L	N(pr(T))= L	N(pr(T))=1
nt(T)= N(pr(T))			
¬(nt(T) = N(pr(T))) ∧ ¬(nt(T) = esc)	1	N(pr(1,T))+ 1	2
¬(nt(T)= N(pr(T))) ∧ (nt(T) = esc) ∧ ¬(nt(pr(T)) = esc)		N(pr(1,T)) - 1	N(pr(1,T)) -1	1
¬ (nt(T)= N(pr(T))) ∧ (nt(T) = esc) ∧ (nt(pr(T)) = esc)		N(pr(1,T))	N(pr(1,T))	1

Status(T)=

	¬(T = _) ∧		
T = _	1 < N(pr(T))< L	N(pr(1,T))= L	N(pr(1,T))=1
nt(T)= N(pr(T))			
¬(nt(T) = N(pr(T))) ∧ ¬(nt(T) = esc)	"incomplete"	"incomplete"	"pass"	incomplete
¬(nt(T)= N(pr(T))) ∧ (nt(T) = esc) ∧ ¬(nt(pr(T)) = esc)		"incomplete"	"incomplete"	"incomplete"
¬ (nt(T)= N(pr(T))) ∧ (nt(T) = esc) ∧ (nt(pr(T)) = esc)		"incomplete"	"incomplete"	"incomplete"
		"fail"	"fail"	"fail"

Defining the Meaning of Tabular Expressions

-not tied to physical layout, pictures, etc.

-simpler

-applicable to many more table types than the older models

-includes all previously known table types

-better suited as the basis of a new generation of tools

Previous work

- Parnas, Henninger, et. al 1977
- Parnas 92,
- Janicki 95,97,01
- Desharnais, Khedri, Mili 01

Preliminaries - indexed sets

Given a function f from a set I to a set X

-The domain of f, I, is called the index set of f;

-An element of I, is called an index.

-(X,I,f) constitutes an indexed set.

Note: the cardinality of X may be less than the cardinality of I

Expressions

Classes of Expressions:

- conventional expressions
- tabular expression

Tabular expressions

- grid (indexed set of expressions)

  G = ({ E_1, ..., E_n}, {i_1, ..., i_m}, g) where E_j- expression, i_j - index,
  g - function from {i_1, ..., i_m} to {E_1, ..., E_n}; G[i] = g[i]

- tabular expression (indexed set of grids)

  T = ({ G_1, ..., G_n}, {i_1, ..., i_m}, t) where G_j- grid, i_j - index, t -
  function from {i_1, ..., i_m} to {G_1, ..., G_n}; T[i] = t[i]

Table types

A tabular expression type can be defined by:

- a restriction predicate (to fix shape, cell types,domain conditions, others)
- an evaluation term (a conventional expression that computes the value of the tabular expression)
- a set of auxiliary function definitions

We have:

Normal Function Table, Inverted Table, Vector Table, Generalized Decision Table, Triangular Table, Circular Table, Logical Tables ….

This set is easily extended

Normal Function Table

T[2]

	x < 0	x = 0	x > 0
y < 0	$x^2 - y^2$	$x^2 + y^2$	$x^2 - y^2$
y = 0	$x + y$	$x^2 - y^2$	$x + y$
y > 0	$x^2 + y^2$	$x + y$	$x^2 + y^2$

T[1] T[0]

$$f(x,y) = \begin{cases} x^2 - y^2 & (((y<0) \wedge (x<0)) \vee ((y<0) \wedge (x>0)) \vee ((x=0) \wedge (y=0))) \\ x + y & (((y=0) \wedge (x<0)) \vee ((y=0) \wedge (x>0)) \vee ((y>0) \wedge (x=0))) \\ x^2 + y^2 & (((y<0) \wedge (x=0)) \vee ((x<0) \wedge (y>0)) \vee ((x>0) \wedge (y>0))) \end{cases}$$

Normal Function Table - cont

The definition of a normal function table T is:

(1) Restriction predicate:

$$
\begin{aligned}
&Card(T) \geq 2 \wedge \\
&IndexSet(T) = \{0,1, ..., Card(T)\text{-}1\} \wedge \\
&IndexSet(T[0]) = IndexSet(T[1]) \times ... \times IndexSet(T[Card(T)\text{-}1]) \wedge \\
&\forall i{:}1 \leq i \leq Card(T)\text{-}1 .\ IndexSet(T[i]) = \{0,1, ... , Card(T[i])\text{-}1\} \wedge \\
&\forall i{:}1 \leq i \leq Card(T)\text{-}1 .\ (TypeG(T[i]) = BOOL \wedge Proper(T[i], \underline{\textbf{\textit{true}}}))
\end{aligned}
$$

(2) An evaluation term:

$$T[0][(select(T[1]), ... , select(T[n]))]$$

(3) Definitions of auxiliary functions: *select*, *TypeG*, *Proper*, and *IndexSet*.

Auxiliary function samples

- *IndexSet(T), IndexSet(G)*

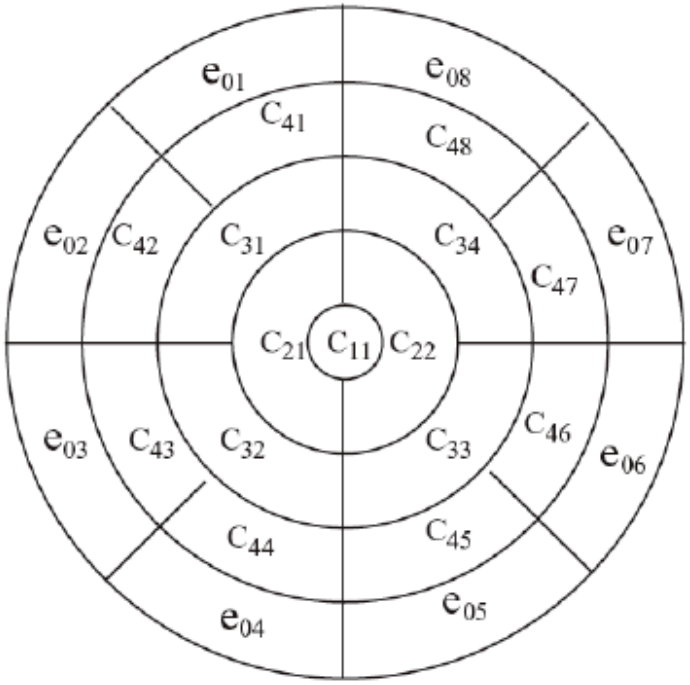
- TypeG(G) = type,
  if all the expressions in G has the same type type

- *Proper(G, C)* =

$$C \Rightarrow ((\forall j \in I.\ \forall k \in I.\ j \neq k \Rightarrow G[j] \wedge G[k] = \underline{\textbf{\textit{false}}}) \wedge (G[i_1] \vee \ldots \vee G[i_n] = \underline{\textbf{\textit{true}}}))$$

- Added by the user

Circular Table



Circular Table T:

IndexSet(T[1])={1}

IndexSet(T[2])={1, 2}

IndexSet(T[3])={1, 2, 3, 4}

IndexSet(T[4])={1,2,3,4,5,6,7,8}

IndexSet(T[0])={1,2,3,4,5,6,7,8}

$T[i][j] = C_{ij}$, where $1 \le i \le 4$, $j \in IndexSet(T[i])$

$T[0][i] = e_{0i}$, where $i \in IndexSet(T[0]$

Circular Table - cont

(1) Restriction predicate:

$$
\begin{aligned}
&Card(T)\geq 2 \wedge \\
&IndexSet(T)=\{0,1,\ldots,Card(T)-1\} \wedge \\
&IndexSet(T[0])=IndexSet(T[Card(T)-1]) \wedge \\
&\forall i{:}1\leq i\leq Card(T)-1.\ (IndexSet(T[i]) = \{1,\ldots,2^{i-1}\} \wedge TypeG(T[i])=BOOL) \wedge \\
&\forall i{:}1\leq i<Card(T)-1.\ \forall j{:}j\in IndexSet(T[i]). \\
&\quad (T[1][1]=\underline{\textbf{\textit{true}}} \wedge \ldots \wedge T[i-1][\mathrm{upb}(j/2)]=\underline{\textbf{\textit{true}}} \wedge T[i][j]=\underline{\textbf{\textit{true}}}) \Rightarrow \\
&\qquad (T[i+1][2\times j-1] \wedge T[i+1][2\times j]=\underline{\textbf{\textit{false}}} \wedge \\
&\qquad\ T[i+1][2\times j-1] \vee T[i+1][2\times j]=\underline{\textbf{\textit{true}}}\ )
\end{aligned}
$$

(2) An evaluation term: $T[0][j_n]$, where j_n is defined inductively:

$j_1 = 1$; $j_k = select(G_k)$,
where G_k is a grid whose index set is $\{2\times j_{k-1}-1, 2\times j_{k-1}\}$,
$G_k[2\times j_{k-1}-1]=T[k][2\times j_{k-1}-1]$, $G_k[2\times j_{k-1}]=T[k][2\times j_{k-1}]$.

(3) Definitions of auxiliary functions: *select*, *TypeG*, *Proper*, and *IndexSet*.

Logical Tables

T[1] T[0]

$$\begin{array}{|c|}\hline \alpha_1 \\ \hline \alpha_2 \\ \hline \end{array}\ \hat{}\ \begin{array}{|c|c|}\hline \beta_1 & \beta_2 \\ \hline \beta_3 & \beta_4 \\ \hline \end{array} \stackrel{\text{df}}{=} (\alpha_1 \Rightarrow \beta_1 \wedge \beta_2) \wedge (\alpha_2 \Rightarrow \beta_3 \wedge \beta_4)$$

$$A_1 \wedge A_2 \wedge A_3 \wedge A_4 \Leftrightarrow \hat{}\begin{array}{|c|c|c|c|}\hline A_1 & A_2 & A_3 & A_4 \\ \hline \end{array} \Leftrightarrow \hat{}\begin{array}{|c|c|}\hline A_1 & A_2 \\ \hline A_3 & A_4 \\ \hline \end{array} \Leftrightarrow \hat{}\begin{array}{|c|}\hline A_1 \\ \hline A_2 \\ \hline A_3 \\ \hline A_4 \\ \hline \end{array} \Leftrightarrow \hat{}\begin{array}{|c|}\hline A_1 \wedge A_2 \\ \hline A_3 \wedge A_4 \\ \hline \end{array}$$

$$A_1 \vee A_2 \vee A_3 \vee A_4 \Leftrightarrow \check{}\begin{array}{|c|c|c|c|}\hline A_1 & A_2 & A_3 & A_4 \\ \hline \end{array} \Leftrightarrow \check{}\begin{array}{|c|c|}\hline A_1 & A_2 \\ \hline A_3 & A_4 \\ \hline \end{array} \Leftrightarrow \check{}\begin{array}{|c|}\hline A_1 \\ \hline A_2 \\ \hline A_3 \\ \hline A_4 \\ \hline \end{array} \Leftrightarrow \check{}\begin{array}{|c|}\hline A_1 \vee A_2 \\ \hline A_3 \vee A_4 \\ \hline \end{array}$$

Expressions vs. Functions/Relations

function - an abstraction

description- something you can write down

procedure + mechanism, can compute values given a description

Example

$$F(x) = x + 1$$

$$\lambda(y)\ [y + 1]$$

$$F(x) = (x = 1 \rightarrow 2,\ x \neq 1 \rightarrow x + 1)$$

$$\{(x,y) \mid y = x + 1\}$$

Notational variations exist because they are needed.

Why we Need This New Form of Expression

Arbitrary set of discontinuities

Values are tuples, elements of different types,

---- no "typical" element

Tabular expressions are suited for such relations

Previously used on *ad hoc* basis, formal definition badly overdue

A Few More Examples

sn(n,T) (subsequent n)

(<integer> × <trace> ➛ <trace>)

sn(n,T) is a trace consisting of all of the elements of T subsequent to on(n,T) in the order in which they appear in T. It can be defined as follows.

sn(n,T) ≡

$n > L(T) \vee n < 1$		
$n \le L(T)) \wedge$	n = 1	s(T)
	n > 1	sn(n-1, s(T))

Our Little Clocks

hr(T) ≡

PGM(r(T)) = SET HR ∧		0 ≤ 'in(r(T)) < 24	'in(r(T))
		¬ (0 ≤ in(r(T)) < 24)	hr((p(T)))
PGM(r(T)) = SET MIN			hr((p(T)))
PGM(r(T)) = INC ∧	min(p(T))= 59 ∧	hr(p(T))= 23	0
		¬ hr(p(T))= 23	1+ hr((p(T)))
	¬ (min(p(T))=59)		hr((p(T)))
PGM(r(T)) = DEC ∧	¬ (min(p(T))= 0)		hr((p(T)))
	min(p(T))= 0 ∧	¬ (hr(p(T)))= 0	hr((p(T)))-1
		hr(p(T))= 0	23
T= _			0

Could you get this right as a conventional expression?

Part of a Stack Definition, Recursively Defined Function

ps(T1,T2) ≡

T2 = _			
(T2 ≠ _) ∧ noeffect(o(T2))			
(T2 ≠ _) ∧ ¬noeffect(o(T2))∧	PGM(o(T2))=PUSH ∧	full(T1)	
		¬ full(T1)	
	PGM(o(T2))=POP ∧	¬empty(T1)	
		empty(T1)	

T1
ps(T1,s(T2))
ps(T1,s(T2))
ps(T1.o(T2),s(T2))
ps(p(T1), s(T2))
ps(T1, s(T2))

strip(T) ≡ ps(_,T)

Final Remark

Any mathematical method can use tabular expressions.

Only when combined with the relational model do you get good documentation.