Tabular Representations in Relational Documents

Ryszard Janicki

David Lorge Parnas

Jeffery Zucker

McMaster University, Hamilton, Ontario, Canada

ABSTRACT

The use of relations, represented as tables, for documenting the requirements and behaviour of software is motivated and explained. A formal model of tabular expressions, defining the meaning of a large class of tabular forms, is presented. Finally, we discuss the transformation of tabular expressions from one form to another, and illustrate some useful transformations.

1 A Relational model of documentation

More than 30 years ago, managers of large software projects began to understand the importance of having precise documentation for software products. The industry was experiencing the frustration of trying to get software to work; most of the many “bugs” that delayed completion and led to unreliable products were caused by misunderstandings that would have been alleviated by better documentation. Since that time, hundreds of “standards” have been proposed; each was intended to improve the consistency, precision and completeness of natural language documents. In spite of these efforts, documentation is still inadequate. Because of the vagueness and imprecision of natural languages, even the best software documentation is unclear. Because informal documentation cannot be analysed systematically, it is usually inconsistent and incomplete as well.

Software Engineering, like other forms of Engineering, can benefit from the use of mathematics. Mathematical notation is commonly used in engineering documents. Only through the use of mathematics, can we obtain the precision that we need. In Computer Science, there has also been a great deal of discussion about the use of mathematics to verify the correctness of software. Before program verification becomes practical, the use of mathematics in documentation must be well-established. Specifications and design documents state the theorems that should be proven about programs.

The state of the art among software developers is such that there is no agreement on the contents of the various documents. It is quite common to hear developers arguing about whether or not a certain fact should be included in some given document. Often information is included in several documents or not found in any. The first step in using mathematics in this context is to find mathematical definitions of the contents of the documents.

In [13] the contents of a number of standard documents are defined by stating that each document must contain a representation of one or more binary relations. Each relation is a set of pairs. If the document contains enough information to determine whether or not any pair is included in the specified relation, it is complete. No additional information should be included. Below we give two examples of such document definitions, one for a system requirements document, the other for a program specification.

1.1 The System Requirements Document

The first step in documenting the requirements of a computer system is the identification of the environmental quantities to be measured or controlled and the association of those quantities with mathematical variables. The environmental quantities include: physical properties (such as temperatures and pressures), the readings on user-visible displays, etc. The association of these quantities with mathematical variables must be carefully defined, and coordinate systems, signs etc. must be unambiguously stated.

It is useful to characterise each environmental quantity as monitored, controlled, or both. *Monitored* quantities are those that the user wants the system to measure. *Controlled* quantities are those whose values the system is intended to control. For real-time systems, time can be treated as a monitored quantity. We will use $m_1, m_2, \ldots, m_p$ to denote the monitored quantities, and $c_1, c_2, \ldots, c_q$ to denote the controlled ones.

Each of these environmental quantities has a value that can be recorded as a function of time. When we denote a given environmental quantity by ν, we will denote the time-function describing its value by ν^t. Note that ν^t is a mathematical function whose domain consists of real numbers; its value at time τ is denoted by "$\nu^t(\tau)$".

The vector of time-functions $(m_1^t, m_2^t, ..., m_p^t)$ containing one element for each of the monitored quantities, will be denoted by $\underset{\sim}{m}^t$; similarly $(c_1^t, c_2^t, ..., c_q^t)$ will be denoted by "$\underset{\sim}{c}^t$".

A *systems requirements document* should contain representations of two relations. NAT describes the *environment*. REQ describes the effect of the system when it is installed.

1.1.1 The Relation NAT

— domain(NAT) is a set of vectors of time-functions containing exactly the instances of $\underset{\sim}{m}^t$ allowed by the environmental constraints,

— range(NAT) is a set of vectors of time-functions containing exactly the instances of $\underset{\sim}{c}^t$ allowed by the environmental constraints,

— $(\underset{\sim}{m}^t, \underset{\sim}{c}^t) \in$ NAT if and only if the environmental constraints allow the controlled quantities to take on the values described by $\underset{\sim}{c}^t$, if the values of the monitored quantities are described by $\underset{\sim}{m}^t$.

1.1.2 The Relation REQ

— domain(REQ) is a set of vectors of time-functions containing those instances of $\underset{\sim}{m}^t$ allowed by environmental constraints,

— range(REQ) is a set of vectors of time-functions containing only those instances of $\underset{\sim}{c}^t$ considered permissible, i.e. values that would be allowed by a correctly functioning system.

— $(\underset{\sim}{m}^t, \underset{\sim}{c}^t) \in$ REQ if and only if the computer system should permit the controlled quantities to take on the values described by $\underset{\sim}{c}^t$ when the values of the monitored quantities are described by $\underset{\sim}{m}^t$.

1.2 Program Descriptions

We use the term *program* to denote a text describing a set of state sequences in a digital (finite state) machine. Each of those state sequences will be called an *execution* of the program. Often, we do not want to document the intermediate states in the sequence. For each starting state, *s*, we want to know only:

(1) is termination possible, i.e. are there finite executions beginning in *s*?

(2) is termination guaranteed, i.e. are all executions that begin in *s* finite?

(3) if termination is possible, what are the possible final states?

This information can be described by an LD-relation [8,11]. An LD-relation comprises a relation and a subset of the domain of that relation, called the *competence set*. In a description of a program, the set of starting states for which termination is guaranteed is the *competence set*. The set of starting states for which termination is possible is the domain of the relation. An ordered pair (*x*,*y*) is in the relation if it is possible that the program's execution would terminate in state *y* after being started in state *x*.

2 Industrial Experience with Relational Documentation

The need for relational documentation, and tabular representations, became apparent in attempts to apply the ideas above to describe programs that were in use in military and civilian applications. This section describes three of those experiences.

2.1 The A-7 experience

An early version of this relational requirements model was used at the U. S. Naval Research Laboratory in 1977 to write a software requirements document for the Onboard Flight Program used in the U.S. Navy's carrier based attack aircraft, the A-7E. The software was being redesigned as an experiment on programming methods, but the design team was inexperienced in that application area. They set out to produce a document that could be used as the exclusive source of information for programmers. At the same time, it was essential that the document could be carefully reviewed by people who understood the requirements, i.e. pilots. The resulting document, [4], is described in [3]. This document was reviewed by pilots (who found hundreds of detail errors in the first versions) and then guided the programmers for several years. A description of the A-7 requirements document, and samples from it, can be found in [3]; the complete document was published as [4]. It has been used as a model for many other requirements documents. (cf. [5, 12]). Further discussion of requirements documents can be found in [17].

2.2 The Darlington Experience

The relational requirements model and the program documentation model were used in a multi-million dollar effort to inspect safety-critical programs for the Darlington Nuclear Power

Generating Station in Ontario, Canada. A relational requirements document, modelled on the A-7 document [4] was written by one group after the English document was shown to have dangerous ambiguities. A second group wrote relational descriptions of the programs. The third group compared the two descriptions. A fourth group audited the process. In all, some 60 people were involved. Although the code had been under test for six years, numerous discrepancies were found and many had to be corrected. This experience is discussed in more detail in [12,10].

2.3 The Bell Labs (Columbus Ohio) Experience

An earlier experience with relational documentation is reported in [5]. In this experience, the time invested in producing a precise statement of the requirements was paid back when the system received its first on-site test. The test period was the shortest in the lab's history because most of the misunderstandings that usually become apparent during testing had been eliminated when the relational requirements document was reviewed. Another, unique, characteristic of this project was the use of the formal mathematical documentation in preparing informal user documentation. It was found that the informal documentation could imitate the structure of the formal documentation and that its quality was greatly enhanced by basing it on the formal document.

3 Why use tabular representations of relations?

In all of these experiences, we found that conventional mathematical expressions describing the relations were too complex and hard to parse to be really useful. Instead, we began to use two-dimensional expressions, which we call *tables*. Each table is a set of cells that can contain other expressions, including tabular ones.

No single notation is well suited for describing all mathematical functions and relations. The history of mathematics and engineering shows clearly that when we become interested in a new class of functions, we also invent new notations that are well-suited for describing functions in that class. The functions that arise in the description of computer systems have two important characteristics. First, digital technology allows us to implement functions that have many discontinuities, which can occur at arbitrary points in the domain of the function. Unlike the designers of analogue systems, we are not constrained to implement functions that are either continuous or exhibit exploitable regularity in their discontinuities. Second, the range and domain of these functions are often tuples whose elements are of distinct types; the values cannot be described in terms of a typical element. The use of traditional mathematical notation, developed for functions that do not have these characteristics, often results in function descriptions that are complex, lengthy, and hard to read. As a consequence of this complexity, mathematical specifications are often incorrect, and, even more often, misunderstood. This has led many people to conclude that it is not practical to provide precise mathematical descriptions of computer systems. However, our experience shows that the use of tabular representations makes the use of mathematics in these applications practical.

3.1 Discovering the first tables

The need for new notation for software documentation first became apparent when producing the requirements document for the A-7. Attempts to write the descriptions in English were soon abandoned because we realised that our turgid prose was no better than that of others. Attempts to write mathematical formulae were abandoned because they quickly developed into parentheses-counting nightmares. We tried using the notations that are commonly used in hardware design, but found that they would be too large in this application. The present table types evolved to meet specific needs. In this work, the tables were used in an *ad hoc* manner, i.e., without formal definition.

3.2 Use of Program Function Tables at Darlington

In the Darlington experience, the requirements document was prepared using the notation developed in the A-7 project. However, for this experience, it was necessary to describe the code. New table formats were introduced for this purpose. Once more the tables were used without formal definition. This time, the lack of formal definition caused some problems. During the inspection process heated discussions showed that some of the tables had more than one possible interpretation. This led to a decision to introduce formal definitions for the tables, but this work had to be postponed until the inspection was completed. The first attempt to formalise the meaning of tabular expressions was [9].

3.3 Why we need more than one type of table

The initial experience with tables on the A-7 went smoothly for a few weeks. Then we encountered an example that would not fit on one piece of paper. In fact, there were so many different cases that we used 4 large pieces of paper, taped to four physical tables, to represent one table. In examining this monster, we found that although there were many cases to consider, there were only a few distinct expressions that appeared in the table. This led us to invent a new form, of table (“inverted tables”) that represented the same information more compactly. The various types of tables will be discussed more extensively in later sections.

3.4 Tables help in thinking

To people working in Theoretical Computer Science or Mathematics, the use of tabular representations seems a minor matter. It is fairly easy to see that the use of tabular representations does not extend the expressive power of the notation. Questions of decidability and computational complexity are not affected by the use of this notation. However, our experience on a variety of projects has shown that the use of this notation makes a big difference in practice. When someone sets out to document a program, particularly when one sets out to document requirements of a program that has not yet been written, they often do not know what they are about to write down. Writing, understanding, and discovery go on at the same time. Document authors must identify the cases that can arise and consider what should be done one case at a time. If the program controls the values of many variables of different types, they will want to think about each of those variables separately. They may have to consult with users, or their representatives, to find out what should be done in each case. Tabular notations are of great help in situations like this. One first determines the structure of the table, making sure that the headers cover all possible cases, then turns one’s attention to completing the individual entries in the table. The task may extend over weeks or months; the use of the tabular format helps to make sure that no cases get forgotten.

3.5 “Divide and conquer” - how tables help in inspection.

Tabular notations have also proven to be of great help in the inspection of programs. Someone reviewing a program will have to make sure that it does the right thing in a variety of situations. It is extremely easy to overlook the same cases that the designers failed to consider. Moreover, the inspection task is often conducted in open review meetings and stretch over many days. Breaks must be taken and it is easy to “lose your place” when there is a pause in the middle of considering a program.

In the Darlington inspection, the tables played a significant role in overcoming these problems. Inspectors followed a rigid procedure. First, they made sure that the set of columns was complete and that no case was included in two columns. Second, they made sure that the set of rows was complete and that there were no duplications or overlaps. Then we began to consider the table column by column proceeding sequentially down the rows. A break could be taken at the end of any entry's consideration; a simple marker told us where to begin when we returned. For a task that took many weeks, the structure provided by the tabular notation was essential.

4 Formalisation of a wide class of tables

The industrial applications of tabular expressions were conducted on an *ad hoc* basis, i.e. without formal syntax or semantics. New types of tables were invented when needed and the semantics was intuitive. The first formal syntax and semantics of tabular expressions (or simply tables) was proposed in [9]. Several different classes of tables, all invented for a specific practical application, were defined and for each class a separate semantics was provided. A different approach was proposed in [7]. Instead of many different classes and separate semantics, one general model was presented. The model covered all table classes in [9] as well as some new classes that had not been considered before. The new model followed from the topology of an abstract entity called "table", and *per se*, was application independent. In this section we will present the basic concepts of this model using very simple examples. For more interesting examples, the reader is referred to [14]. We shall not, here, make a distinction between relation and functions, but use the more general concept, relation.

Let us consider the two following definitions of functions, $f(x,y)$ and $g(x,y)$.

$$f(x,y) = \begin{cases} 0 & \text{if } x \geq 0 \wedge y = 10 \\ x & \text{if } x < 0 \wedge y = 10 \\ y^2 & \text{if } x \geq 0 \wedge y > 10 \\ -y^2 & \text{if } x \geq 0 \wedge y < 10 \\ x + y & \text{if } x < 0 \wedge y > 10 \\ x - y & \text{if } x < 0 \wedge y < 10 \end{cases} \qquad g(x,y) = \begin{cases} x + y & \text{if } (x<0 \wedge y\geq 0) \vee (x<y \wedge y<0) \\ x - y & \text{if } (0\leq x<y \wedge y\geq 0) \vee (y\leq x<0 \wedge y<0) \\ y - x & \text{if } (x\geq y \wedge y\geq 0) \vee (x\geq 0 \wedge y<0) \end{cases}$$

If we were to describe $f(x,y)$ using classical predicate logic, we would write an expression like:

$$\begin{gathered}(\forall x, (\forall y, ((x \geq 0 \wedge y = 10) \Rightarrow f(x,y) = 0) \wedge ((x < 0 \wedge y = 10) \Rightarrow f(x,y) = x) \wedge \\ ((x \geq 0 \wedge y > 10) \Rightarrow f(x,y) = y^2) \wedge ((x \geq 0 \wedge y < 10) \Rightarrow f(x,y) = -y^2) \wedge \\ ((x < 0 \wedge y > 10) \Rightarrow f(x,y) = x + y) \wedge ((x < 0 \wedge y < 10) \Rightarrow f(x,y) = x - y)))\end{gathered}$$

Such classical mathematical notation is not very readable, even though the functions are very simple ones. The description becomes much more readable when tabular notation, even without a formal semantics, is used (see Figures 1 and 2).

We must now compare these two examples. What are the differences and similarities between the two tables? They both have the same *raw skeleton*, as illustrated in Figure 3, which in both cases consists of two *headers*, H_1 and H_2, and the *grid*, G. However, in Figure 1 the formulae giving the final value is in the main grid, while in Figure 2 the final formula to be evaluated is in header H_1.

Formally, a *header* is an indexed set of cells, say $H = \{h_i \mid i \in I\}$, where $I = \{1,2, ..., k\}$ (for some k) is an index set. We treat *cell* as a primitive concept that does not need to be explained. A *grid G indexed by headers* $H_1,...,H_n$, with $H_j = \{h_i^j \mid i \in I^j\}$, $j = 1 ..., n$, is an indexed set of cells G, where $G = \{g_\alpha \mid \alpha \in I\}$, and $I = I^1 \times ... \times I^n$. The set I is the index set of G. A *raw table skeleton* is a collection of headers plus a grid indexed by this collection.

H_1

$y = 10$	$y > 10$	$y < 10$

H_2

$x \geq 0$
$x < 0$

G

0	y^2	$-y^2$
x	$x + y$	$x - y$

Figure 1: A (normal) table defining *f*

H_1

$x + y$	$x - y$	$y - x$

H_2

$y \geq 0$
$y < 0$

G

$x < 0$	$0 \leq x < y$	$x \geq y$
$x < y$	$y \leq x < 0$	$x \geq 0$

Figure 2: An (inverted) table defining *g*

$$H_1 = \{h_i^1 \mid i = 1, 2, 3\}$$

h_1^1	h_2^1	h_3^1

$$H_2 = \{h_i^2 \mid i = 1, 2, 3\}$$

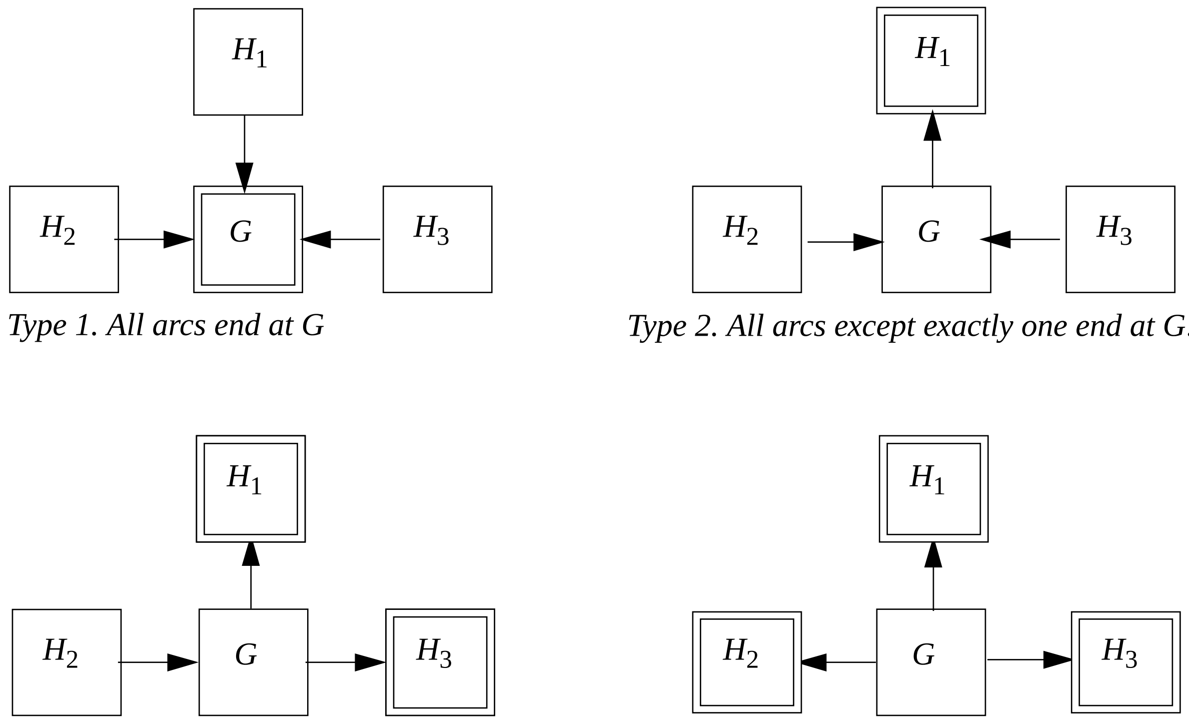
h_1^2
h_2^2

g_{11}	g_{21}	g_{31}
g_{12}	g_{22}	g_{32}

$$G = \{g_{ij} \mid i = 1, 2, 3 \quad and \quad j = 1, 2\}$$

Figure 3: A raw table skeleton of tables from Figure 1 and Figure 2

The first step in expressing the semantic difference between the two types of tables is to define the *Cell Connection Graph* (CCG in short), which characterizes *information flow* (“where do I start reading the table and where do I get my result?”). A CCG is a *relation* that could be interpreted as an *acyclic directed graph* with the grid and all headers as the nodes. The only requirement is that *each arc must either start from or end at the grid G*. There are four different types of CCGs. They are all illustrated in Figure 4 for n=3. When the number of headers is smaller than 3, type 3 disappears. Type 1 is called *normal*, and type 2 is called *inverted*. These are the two most popular types in practice. The CCG divides the cells into relation cells and predicate cells. A cell is a *relation cell* if no arc starts from it, otherwise it is a *predicate cell*. In Figure 4 (as well as in Figure 1) all relation cells are represented by double boxes. Headers and a grid, together with a CCG graph, define a *medium table skeleton*. Each predicate cell defines the domain of a subset of the relation defined; the relation cells define possible values within each domain. However, we *still* do not have enough semantics. How do we determine the domain and values? Let us take the upper table from Figure 1, and the cells h_1^1, h_1^2, g_{11}. If this table represents the function $f(x,y)$, then the CCG (together with the contents of the cells) restricted to these cells should represent the expression $(x \geq 0 \wedge y=10) \Rightarrow f(x,y)=0$. But why $(x \geq 0 \wedge y=10)$? Why not for example: $(x \geq 0 \vee y=10)$, or $(\neg(x<0 \vee y=10))$ etc.?



Type 1. All arcs end at G

Type 2. All arcs except exactly one end at G.

Type 3. At least two arcs start from G and at least one arc ends at G.

Type 4. No arc ends at G.

Figure 4: Four different types of cell connection graphs

There is no explicit information in the table that indicates conjunction. A medium table skeleton does not provide any information on how the domain and values of the relation (function) specified are determined; such information must be added. The domain is determined by a *table predicate rule*, P_T, and the value is determined by a *table relation rule*, r_T. A medium table skeleton together with table predicate and relation rules is called a *well-done table skeleton*. Figure 5 illustrates two well-done table skeletons. The left-hand one is of type 1, the domain is defined as the conjunction of the predicates contained in appropriate cells of the headers H_1 and H_2, while the value is defined just by an expression held in the appropriate cell of the grid G. The right-hand one is of type 4, the domain is defined by the predicate held in the appropriate cell of the grid G, while the value of the defined relation is the set union of the relations defined by appropriate expressions held in the headers H_1 and H_2. To get the *full tabular* expression we need to enrich a well-done table skeleton by a mapping which assigns a predicate expression to each predicate cell, and a relation expression to each relation cell. Figures 6 and 7 show tabular expressions describing the functions *f* and *g*.

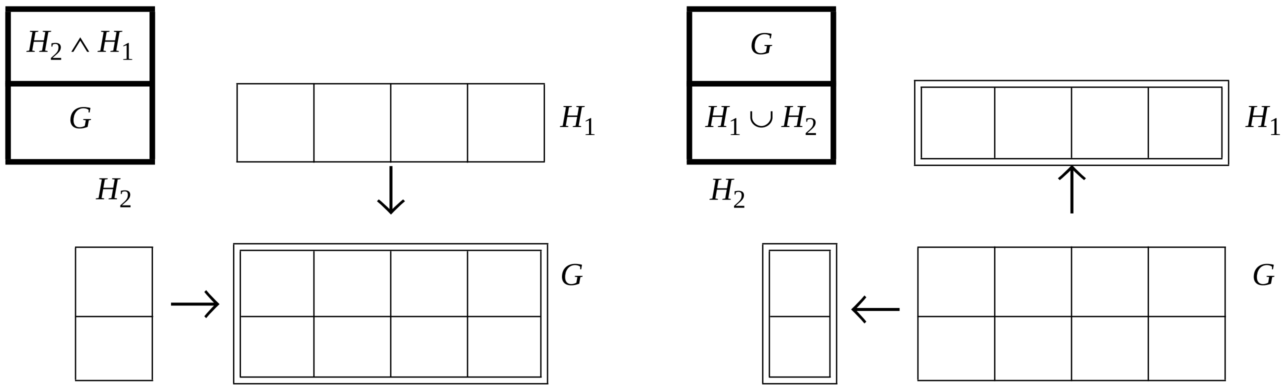


Figure 5: Two examples of well done table skeletons

$P_T(H_1, H_2) = H_1 \wedge H_2$

$r_T(G) = G$

$P_T(G) = G$

$r_T(H_1, H_2) = H_1 \cup H_2$

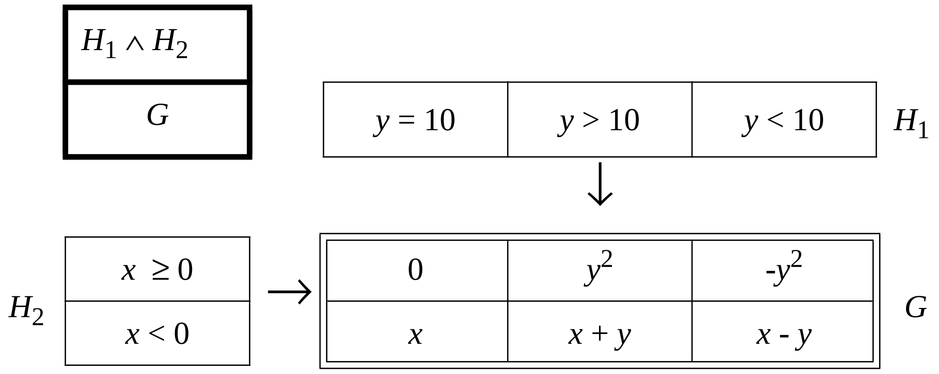


Figure 6: A tabular expressions describing the function *f*

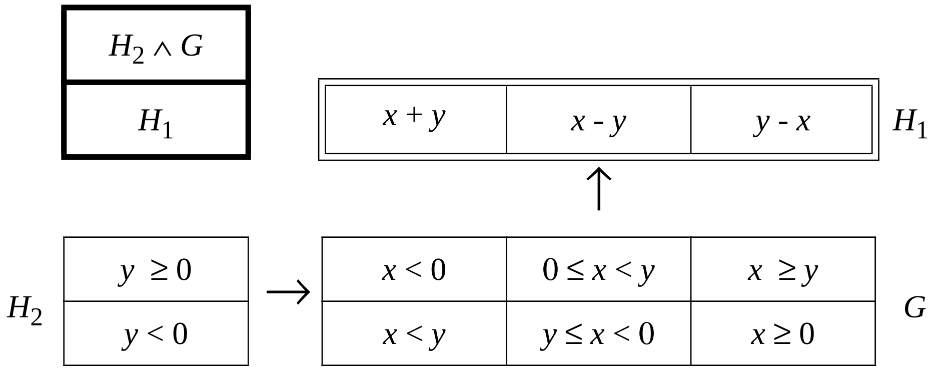


Figure 7: A tabular expression describing the function g.

Summing up, the indexing scheme determines a *raw table skeleton*, where a table is a set of sets of cells. The *cell connection graph* shows “information flow”. A *table predicate rule* and *table relation rule* provide the remaining information.

5 Transformations of tables of one kind to another

As we have seen above, many kinds of tables have been found to be useful. Many questions arise naturally: Given a function, what is the best kind of table to represent it? Given two tables (of the same or different kinds), can we see whether they define the same function? Can we (or under what conditions can we) perform certain useful manipulations on tables, such as transforming a table to a “simplest” form of the same kind, or transforming one kind of table to another?

The present section focuses on the last question. We consider two of the kinds of tables discussed above, namely *normal function tables* and *inverted function tables*, with conjunction as the predicate rule. (We also revert from the relational to the functional formalism.)

We study various methods for effectively transforming one kind of table to the other. This section gives an informal overview; details can be found in [19].

We are interested in transforming tables to other, semantically equivalent tables, which may be easier to work with. We will consider transformations ϕ: of tables from one kind to another, which satisfy the following two properties: (*i*) ϕ is semantics preserving, and (*ii*) ϕ is effective or computable.

We will consider three examples of such transformations: *changing the dimensionality* of a table, *inverting* a normal table, and *normalising* an inverted table.

5.1 Changing the dimensionality of a table

Note first that any n-dimensional (normal or inverted) table can be trivially transformed to an (n+1)-dimensional table, by adding an $(n+1)^{\text{th}}$ coordinate header with a single entry, ‘**true**’.

More interestingly: given a normal n-dimensional table, we can transform it to an (n-1)-dimensional table, by “combining” two of the dimensions, i.e., combining two of the headers into a single header.

By iterating this procedure, we can transform any normal table T to a 1-dimensional normal table — albeit very long (with length equal to the number of cells in the original table), so that the value of such a transformation in general is unclear.

5.2 Inverting a normal table

We illustrate inversion with a simple example. Consider the case of a 2-dimensional 3×3 normal table, as in Figure 8. (The header entries C^i_j are conditions, and the grid entries t_{ij} are terms.)

This can be “inverted along dimension 2” (say), to produce the table in Figure 9.

In general, a normal table can be inverted along any dimension k to produce an inverted table with value header $\tilde{H}_k$, and the other headers unchanged. The practical value of this transformation is, however, dubious, since the new table is much bigger than the original. (The length of the value header in the new table is equal to the number of cells in the original table!)

In [19] we also consider a second method for inversion, which leads to better (i.e., smaller) inverted tables assuming there are not many distinct terms.

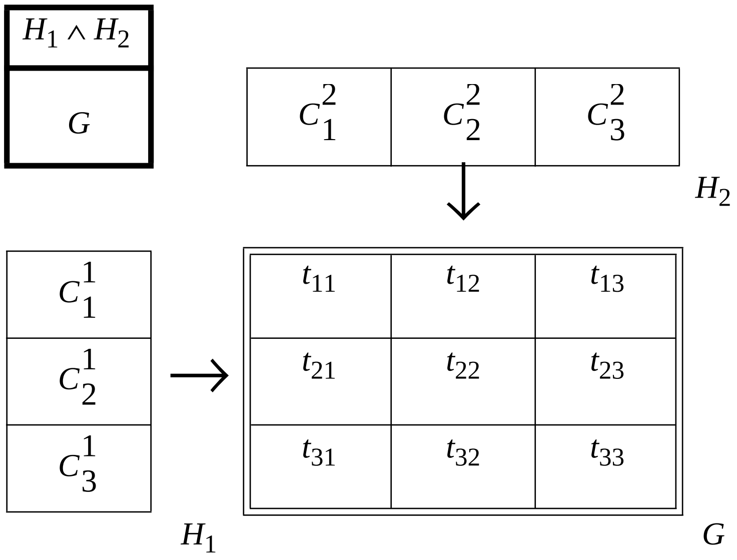


Figure 8: A two dimensional 3×3 normal table

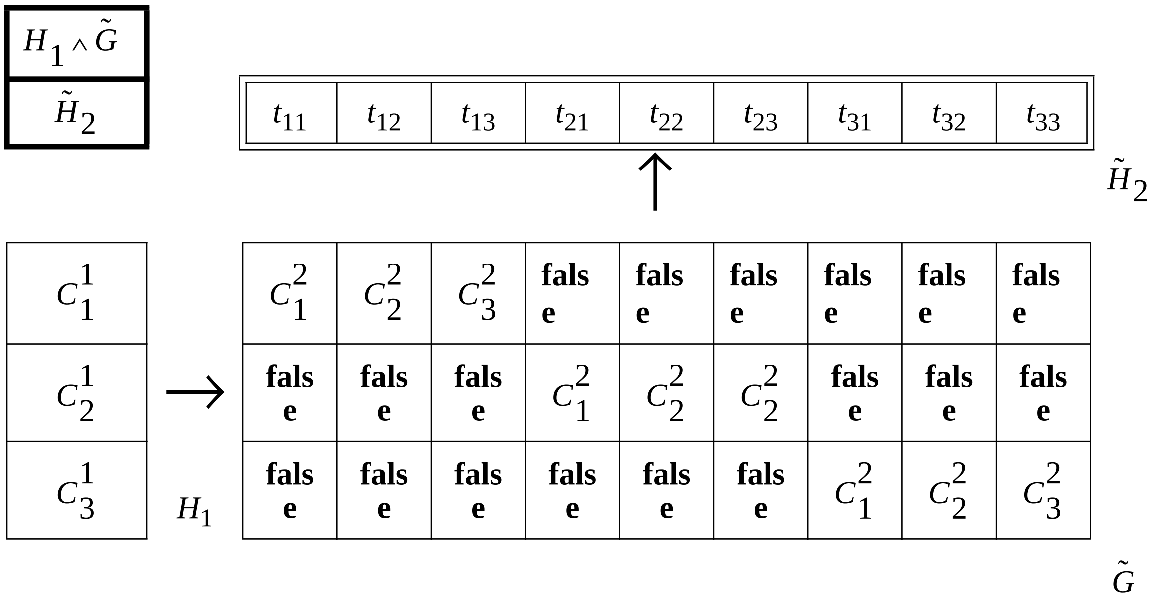

$H_1 \wedge \tilde{G}$
$\tilde{H}_2$
t_{11} t_{12} t_{13} t_{21} t_{22} t_{23} t_{31} t_{32} t_{33}
$\tilde{H}_2$
C_1^1 C_2^1 C_3^1
H_1
C_1^2 C_2^2 C_3^2 false false false false false false
false false false C_1^2 C_2^2 C_2^2 false false false
false false false false false false C_1^2 C_2^2 C_3^2
$\tilde{G}$

Figure 9: Inversion of the table in figure 8

5.3 Normalising an inverted table

Here the situation is less satisfactory. We can transform an inverted table to a 1-dimensional normal one, but not (in general) to a many-dimensional one (apart from the trivial many-dimensional version of a 1-dimensional table described in § 5.1). As a simple example, consider the 2-dimensional 2×3 inverted table T shown in Figure 10, with value header H_2.

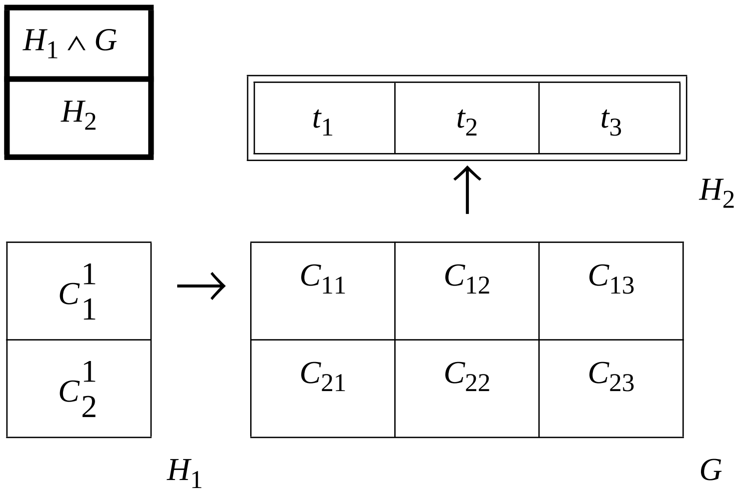


Figure 10: 2-dimensional inverted table

This can be normalised “along dimension 2” to a 1-dimensional table, as shown in Figure 10.

$(C^1_1 \wedge C_{11}) \vee (C^1_2 \wedge C_{21})$	$(C^1_1 \wedge C_{12}) \vee (C^1_2 \wedge C_{22})$	$(C^1_1 \wedge C_{13}) \vee (C^1_2 \wedge C_{23})$

$\hat{H}_2$

$$\downarrow$$

t_1	t_2	t_3

$\hat{G}$

Figure 11: Normalisation of the table in Figure 10

Note that the header $\hat{H}_2$ of the transformed table of Figure 11 has the same length as H_2. However the conditions in this header are quite complicated. We can, however, effect a trade-off between complexity of conditions and header length by “splitting” disjunctions, as in Figure 12.

$C^1_1 \wedge C_{11}$	$C^1_2 \wedge C_{21}$	$C^1_1 \wedge C_{12}$	$C^1_2 \wedge C_{22}$	$C^1_1 \wedge C_{13}$	$C^1_2 \wedge C_{23}$

$\hat{H}_2$

$$\downarrow$$

t_1	t_1	t_2	t_2	t_3	t_3

$\hat{G}$

Figure 12: Another normalisation of the table in Figure 10

Another (more complex) normalisation algorithm for inverted tables, which preserves dimensionality, was reported in [16].

5.4 Interrelationship between transformations

Given a normal table T, an *inversion* followed by a *normalisation* (by the methods considered above) is *essentially* the same as the transformation of T to 1 dimension, in the following sense.

Theorem. If T is a normal table, the result of an inversion of T (as in §5.2) followed by a normalisation (as in §5.3) can be converted to the 1-dimensional transform of T (as in § 5.1) by a sequence of *elementary transformations* of the following form: permuting components of conjunctions and disjunctions, distributing conjunctions over disjunctions, simplifying ‘$C \wedge$ **false**’ to ‘**false**’ and ‘$C \vee$ **false**’ to ‘C’, repeating rows by “splitting disjunctions”, permuting rows, and deleting rows with ‘**false**’ in the header.

6 The McMaster Table Tool Project

During the practical work discussed earlier, it became clear that working with tabular relational documents was often dull, mechanical and exacting. The work discussed in this paper provides the basis for tools to assist people when using these notations. We are now engaged in a project to produce a set of prototype tools. The kernel of our system is a “table holder” that creates objects representing uninterpreted tables. Other programs can use this kernel to store and communicate tabular expressions. The tables are stored as abstract data structures and without formatting information.

Separate tools will assist a user to create tables, which are stored in table-holder objects, and to print tables. The printing tool is designed for use by people who do not necessarily understand the expressions. Although the operator of this tool has control over the appearance of the table when printed, the contents cannot be changed by this tool. We are also designing tools to perform basic checks on tabular expressions.

A tool for evaluating and simplifying tabular expressions, based on the notation used in Section 4, is being designed by another student. Others are studying the problem of finding a tabular representation of the composition of two relations represented by given tables.

Dennis Peters has prepared a tool that generates test oracles from tabular program specifications [15]. The transformation algorithms discussed above have been implemented by Hong Shen [16] to provide a prototype tool that assists designers in transforming tables. Much work remains to be done in finding transformation algorithms which are simple and also produce compact transformed tables.

Related work is going on at the University of Quebec (Hull) (Iglewski) [1, 2], Warsaw University (Madey) [6]and Swansea University (Tucker and Wilder)[18].

We believe that the idea of tabular representations of relations is an area deserving attention from both theoretical and practical computer specialists.

Acknowledgements

We wish to thank our colleagues in the Software Engineering Research Group at McMaster as well as Jan Madey and Michal Iglewski for many useful discussions. Funding from TRIO and NSERC was essential.

References

[1] J. Bojanowski, M. Iglewski, J. Madey, A. Obaid, “Functional approach to Protocols Specification”, Proceedings of the 14th International IFIP Symposium on Protocol Specification, Testing and Verification, PSTV'94, Vancouver, B.C., 7-10 June 1994, pp. 3 71-378.

[2] B. Desrosiers, M. Iglewski, A. Obaid, “Utilisation de la methode de traces pour la definition formelle d'un protocole de communication", Electronic Journal on Networks and Distributed Processing, No. 2, September 1995, pp. 57-73.

[3] K.L. Heninger, “Specifying Software Requirements for Complex Systems: New Techniques and their Application”, *IEEE Transactions Software Engineering*, Vol. SE-6, No. 1, January 1980, pp. 2-13.

[4] K.L. Heninger, J. Kallander, D.L. Parnas, J.E. Shore, “Software Requirements for the A-7E Aircraft”, *NRL Memorandum Report* 3876, United States Naval Research Laboratory, Washington DC, November 1978, 523 pp.

[5] S.D. Hester, D.L. Parnas, D.F. Utter, “Using Documentation as a Software Design Medium”, *Bell System Technical Journal*, Vol. 60, No. 8, October 1981, pp. 1941-1977.

[6] M. Iglewski, J. Madey, “Software Engineering Issues Emerged from Critical Control Applications”, 2nd IFAC Workshop on Safety and Reliability in Emerging Control Technologies, Daytona Beach, Florida, USA, 1-3 November 1995, to be published in conference proceedings by PERGAMON - Elsevier Science.

[7] R. Janicki, “Towards a Formal Semantics of Parnas Tables”, *Proceedings of the 17th International Conference on Software Engineering*, Seattle WA, 1995, pp. 231-240.

[8] D.L. Parnas, “A Generalized Control Structure and Its Formal Definition”, *Communications of the ACM*, Vol. 26, No. 8, August 1983, pp. 572-581

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

[10] D.L. Parnas, “Inspection of Safety Critical Software using Function Tables”, *Proceedings of IFIP World Congress 1994, Volume III”* August 1994, pp. 270 - 277.

[11] D.L. Parnas, “Mathematical Descriptions and Specification of Software”, in: *Proceedings of IFIP World Congress 1994, Volume I”* August 1994, pp. 354-359.

[12] D.L. Parnas, G.J.K. Asmis, J. Madey, “Assessment of Safety-Critical Software in Nuclear Power Plants”, *Nuclear Safety*, Vol. 32, No. 2, 1991, pp. 189-198.

[13] D.L. Parnas, J. Madey, “Functional Documentation for Computer Systems Engineering (Version 2)”, CRL Report 237, McMaster University, TRIO (Telecommunications Research Institute of Ontario), September 1991, 14 pgs. To be published in *Science of Computer Programming* (Elsevier) 25 (1995), pp 41-61.

[14] D.L. Parnas, J. Madey, M. Iglewski, “Precise Documentation of Well-Structured Programs”, *IEEE Transactions on Software Engineering*, Vol. 20, No. 12, December 1994, pp. 948 - 976

[15] D. Peters, D. L. Parnas, “Generating a Test Oracle from Program Documentation”, *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA),* August 17-19, 1994, pp. 58 - 65.

[16] H. Shen, “Implementation of Table Inversion Algorithms”, M. Eng thesis, McMaster University, Communications Research Laboratory Dec. 1995.

[17] A.J. van Schouwen, D. L. Parnas, J. Madey, “Documentation of Requirements for Computer Systems”, Proceedings of *'93 IEEE International Symposium on Requirements Engineering*, San Diego, CA, 4 - 6 January, 1993, pp. 198 - 207.

[18] A.J. Wilder, J.V. Tucker, “System Documentation Using Tables - a short course”, CRL Report 306, McMaster University, Communications Research Laboratory, TRIO (Telecommunications Research Institute of Ontario), May 1992, 110 pgs. Also published as Report CSR 11-95, Computer Science Department, University of Wales, Swansea, 1995.

[19] J.I. Zucker, “Transformations of Normal and Inverted Function Tables”, to be published in *Formal aspects of Computing.*