CONVERTING SOFTWARE COST REDUCTION TABLES INTO TABULAR EXPRESSIONS

Imen Bourguiba and Ryszard Janicki
McMaster University
Hamilton, Ontario, Canada
imenbour@gmail.com, janicki@mcmaster.ca

ABSTRACT
Tabular expressions and the Software Cost Reduction (SCR) method are table-based specification techniques. Both of them were successfully used in practice, especially to formally specify software requirements. Tabular expressions have rather precise semantics, while SCR semantics are more intuition based. In this paper, we want to improve SCR semantics. For that, we show how to convert the SCR tables into tabular expressions. The conversion that we came up with allows the SCR tables to inherit the semantics of tabular expressions. Hence, the converted tables are more readable, and more efficient.

KEY WORDS
Software specification; tabular expressions; software cost reduction; software requirements

1 Introduction
In software engineering the need to apply formal methods, especially table-based ones is constantly increasing. Table-based specification techniques are readable and efficient. They allow us to express a system specification in a very compact and precise way. Moreover, they scale nicely when applied to practical software systems [5]. Among these table-based specification techniques, we highlight Software Cost Reduction and Tabular expressions. The Software Cost Reduction (SCR) is a quite popular formal method to specify system requirements. It has been substantially improved and extended over the last decade [5]. Tabular expressions allow us to represent the specifications of systems in a rather very compact and still precise manner using a multi-dimensional syntax [10]. Moreover, their intuitive semantics make them more suitable for most applications and easier to people who do not have a solid mathematical background, or who are not much familiar with the application domain.

A comparison between the two methods can be found at [2]. Although these two methods complement each other, the semantics of the SCR method are not as precise as those of tabular expressions. The latter ones are well founded and rich.

In this paper, we show how to algorithmically convert SCR tables into tabular expressions. The transformations we use are proved to preserve the semantics of the original tables. Also, we give several examples to illustrate these transformations.

Sections 2 and 3 are devoted to brief presentations of the SCR and tabular expressions methods respectively. In Section 4, we show the transformation of SCR tables into tabular expressions, and give simple examples to illustrate the conversion. In Section 5 we present our conclusion.

2 Software Cost Reduction
The Software Cost Reduction (SCR) was originally developed in U.S. Naval Research Lab to document the requirements of the operational flight program [1]. Then, it was successively improved by C.L. Heitmeyer and her team. Its semantics has been improved in [3]. In [21] a formal definition of SCR Tables was proposed. The SCR method has been successfully used in many industrial and academic organizations [5]. It is probably the most popular formal method based on a tabular notation for specifying the requirements of software systems. The SCR requirements specification involves two main ingredients: the system behaviour and the environment. The environment has two types of variables, the controlled variables (quantities that the system controls) and monitored variables (quantities that the system monitors). The environment generates a sequence of monitored events to which the system reacts by changing its state. The system can thus be represented via a state machine $\Sigma = (S, S_0, E^m, T)$, where $S$ is the set of states, $S_0$ is the initial set of states ($S_0 \subseteq S$), $E^m$ is the set of monitored events, and $T$ is the transform function, which from a current state $s \in S$ and an event $e \in E^m$ returns the next state $s' \in S$. The SCR state machine model is a special case of Parnas’ Four Variable Model (FVM) [14]. In
[17], Parnas and Madey developed FVM as an extension of the classical Two Variable Model (input and output) found in [14]. FVM consists of four sets of variables and four relations (see Figure 1). In what follows, we denote the set of variables monitored by the system by $\mathcal{M}$. The set $\mathcal{C}$ depicts the variables controlled by the system. The set $\mathcal{I}$ gives the input variables, and the set $\mathcal{O}$ the output variables. The relation $\text{NAT}$ defines the natural constraints on the system behaviour, and the set of possible values of the monitored and controlled variables. As shown in Figure 1, the relation $\text{REQ}$ represents the required system behaviour. The relation $\text{IN}$ describes the behaviour of the input by mapping the monitored variables into input variables. The relation $\text{OUT}$ describes the behaviour of the output. It maps the output variables into the controlled variables.

The SCR formal model uses only the relations $\text{NAT}$ and $\text{REQ}$ to define the system behaviour. To have a more concise specification, some constructs such as Mode classes and terms were added to the model. The values of mode classes are called modes. Modes are classes of system states specifying the system behaviour. With SCR, each specification is organized into dictionaries and tables. The dictionaries represent static information such as variables names and types, whereas the tables depict the changes of the variables with respect to input events. SCR uses three kinds of tables to specify a system: condition tables, event tables, and mode transition tables. These tables are better explained using the safety injections system example borrowed from [6]. They are discussed respectively in the following subsections.

2.1 SCR Condition Table

A condition table describes the changes of a variable according to a mode and a condition. A condition is a predicate defined on a system state. For example, Table 1 identifies the controlled variable SafetyInjection as a function of Pressure and the term Overridden. For instance in Table 1, the SafetyInjection is Off if the Pressure is High or Permitted, or if the Pressure is TooLow and Overridden is True.

2.2 SCR Event Table

An event table defines a variable according to a mode and an event. An event represented by $\text{@T}(c)$ means that condition $c$ changes from false to true. For example, $\text{@T}(\text{Block=On})$ when $\text{T}(\text{Reset=Off})$ means that the operator turns Block from Off to On when Reset is Off. The $\text{@T}(\text{Inmode})$ means that the system enters into the class of modes in that row. For instance, $\text{@T}(\text{Inmode})$ in the second row of Table 2 indicates the system enters in the mode TooLow or Permitted. In Table 2, the mode Pressure is defined via the current mode and the events defined on the variable WaterPress. For instance, the term overridden is True if the Pressure is TooLow or Permitted, and Block changes to On When Reset is Off.

2.3 SCR Mode Transition Table

A mode transition table specifies the changes of the modes given a mode and an event. In the first row of Table 3, we see that if the Pressure is TooLow and WaterPres is greater or equal than Low, then Pressure becomes Permitted.

3 Tabular Expressions

Tabular expressions are a generalisation of two dimensional tables as classical decision tables and state transition tables that date back to early years in computer science [8]. In the late 1970s, D.L. Parnas pioneered the use of tabular expressions to document software requirements [7]. The most known applications of this approach are descriptions
of requirements for the A-7E aircraft [1, 7] and for the Darlington Nuclear Power Plant Shutdown System [13, 16].

The formal semantics of tabular expressions evolved substantially over the last two decades. D.L. Parnas proposed the first semantics analysis of tabular expressions based on types of applications [15]. More general application independent formal semantics (in terms of relational calculus) was proposed by R. Janicki in [9] and then refined a couple of times, recently in [11]. Each class proposed in [15] could be seen as a special case of the generic model proposed by R. Janicki [9, 11]. The general model of [9, 11], although almost ‘universal’, is often too complex for simple cases, so many of special variations of it are often used [12, 18].

Both the original Parnas’ model [15] and that of [9, 11] are valid for n-dimensional tables, with an arbitrary finite n ≥ 2, however practically all applications use only the most natural 2-dimensional tables.

In brief, the concept and the semantics of tabular expressions (as in [9] and [11]) can be described as follows. An n-dimensional table is composed of n headers, H_i, i = 1, ..., n, and a grid G built from ordered cells containing some expressions. For instance, in the table drawn in Figure 2, there are two headers H_1 and H_2, and one grid G presented with a double border. The tuple T^row = (H_1, H_2, ..., H_n, G) is called the raw table skeleton. Next, the flow of information amongst the components of the table is determined to indicate “where do I start reading the table and where do I get my result?” The table cells are divided into two types: guard cells and value cells. Guard cells contain the predicates, while value cells contain the results. Value cells are often (but not always) represented with a double line border. The information flow, the guard cells, and value cells represent what is called the Cell Connection Graph (CCG) of the Tabular Expression. CCG model information flow and provide a taxonomy of Tabular Expressions. Different CCG’s correspond to different types of tables. The most popular (and natural) types are normal tables and inverted tables (see [11]). For instance, the CCG in the top left corner of Figure 2 corresponds to a Normal Table. The CCG is typically presented by an icon which resembles it and is usually placed at the top left corner of the table (see left part of Figure 2). After determining the CCG, the medium table skeleton T^med of the table is defined as the tuple T^med = (CCG, H_1, H_2, ..., H_n, G). Then, a well done table skeleton T^well of the table is defined as T^well = (P_T, r_T, C_T, CCG, H_1, ..., H_n, G), where P_T is a table predicate rule indicating how predicates are to be built from the contents of table cells, r_T is a table relation rule indicating how relations/functions are to be built from the contents of table cells, and C_T is a table composition rule which states how the global relation/function is built from local representations. The shape of C_T depends on the type of a table, i.e. CCG. For the normal table of Figure 2, we have P_T = H_1 ∧ H_2, r_T = G, while C_T is given by the relational formula \( \bigcup_{i=1}^{2} \bigcup_{j=1}^{3} R_{ij} \), where \( R_{ij} \) is a relation corresponding to the expressions in the cells \( H_1[i], H_2[j] \) and \( G[i, j] \). Finally, a Tabular Expression \( T \) is formally defined as the tuple:

\[
(P_T, r_T, C_T, CCG, H_1, ..., H_n, G, \Psi, IN, OUT)
\]

where \( \Psi \) is a mapping assigning predicate expressions to guard cells, and relation expressions to value cells, \( IN \) is the set of inputs, and \( OUT \) is the set of outputs, and the predicate expressions have variables over \( IN \), and the relation expressions have variables over \( IN \times OUT \). The meaning (semantics) of a tabular expression \( T \) is given by a relation \( R_T \subseteq IN \times OUT \), which is defined as:

\[
R_T = C_T(R_a).
\]

For the Normal Table from Figure 2,

\[
R_T = \bigcup_{i=1}^{2} \bigcup_{j=1}^{3} R_{ij},
\]

where \( R_{ij} \) are as defined above.

In [4], tabular expressions are used as a programming language. The language is built upon atomic tabular expressions and operators. As such, tabular expressions are viewed as a stack of atomic expressions and operators applied on them. The tabular operators introduced are used to compose and decompose tabular expressions in a modular way, which improves their semantics.

Due to the space limit we will not discuss the whole theory of tabular expressions, an interested reader is referred to [4].

It appears that after transformation, SCR tables correspond to the special kind of tabular expressions called Function tables that are discussed below.

3.1 Function Tables

A special kind of tabular expressions and probably the most important ones are function tables [19]. They were adopted to define requirements for the Darlington Nuclear Power Plant Shutdown System in Canada [16, 13, 19]. In principle they use functions in a tabular form to describe system’s functionality for safety critical software systems. Function tables are constantly improved to satisfy the developers
and engineers requirements. Small changes such as turning the tables aside, and removing some unnecessary symbols helped function tables to become more readable, and hence to be successfully used in practice to develop the shutdown systems for the Darlington nuclear power plant [20]. There are four kinds of function tables: vertical condition tables, horizontal condition tables, structured decision tables, and state transition tables. The tables discussed in the following are borrowed from [13].

3.1.1 Vertical Condition Tables

In vertical condition tables, the left bottom cell indicates the name of the function. For example, in Table 4, the name of the function is: f-trip. The other bottom columns indicate the value of the function when the condition of the respective columns is true. So, in this example, the function f-trip is interpreted as:

\[
\text{f-trip} = \text{e-tripped} \text{ if } ((\text{m-level} > \text{level-limit}) \text{ AND } (\text{m-enable} = \text{e-enabled}))
\]

\[
\text{f-trip} = \text{e-not-tripped} \text{ if } \text{NOT } ((\text{m-level} > \text{level-limit}) \text{ OR } \text{NOT}(\text{m-enable} = \text{e-enabled}))
\]

3.1.2 Horizontal Condition Tables

In horizontal condition tables, the right cells on the top indicate the function names. The other rows indicate the conditions and function values when the respective conditions are true.

From Table 5, we could see that the function f-foo is defined as follows:

\[
\text{f-foo} = \text{e-tripped} \text{ if } ((\text{m-Trip}[1] = 1) \text{ AND } (\text{m-Trip}[2] \neq 1)),
\]

\[
\text{f-foo} = \text{e-tripped} \text{ if } ((\text{m-Trip}[1] \neq 1) \text{ AND } (\text{m-Trip}[2] = 1)),
\]

\[
\text{f-foo} = \text{e-not-tripped} \text{ if } ((\text{m-Trip}[1] \neq 1) \text{ AND } (\text{m-Trip}[2] \neq 1)),
\]

\[
\text{f-foo} = \text{e-tripped} \text{ if } ((\text{m-Trip}[1] = 1) \text{ AND } (\text{m-Trip}[2] = 1)).
\]

3.1.3 Structured Decision Table

A structured decision table has conditions states, action states, and rules. Also it may have conditions macros that comes with the table to help shortening the cells content so that the table will not be cumbersome. In this example, the condition macros are:

\[
\text{w-trip-mg}[\text{m-ai, f-sp}]
\]

hitrp : \text{m-ai} \geq \text{f-sp}

ddbd : (\text{m-ai} < \text{f-sp}) \text{ AND } (\text{m-ai} \geq (\text{f-sp} - \text{k-db}))

notrp : \text{m-ai} < (\text{f-sp} - \text{k-db})

From table 6, the function f-trip is interpreted as:

\[
\text{f-trip} = \text{e-tripped} \text{ if } (\text{m-ai} \geq \text{f-sp}),
\]

\[
\text{f-trip} = \text{e-tripped} \text{ if } ((\text{m-ai} < \text{f-sp}) \text{ AND } ((\text{m-ai} \geq (\text{f-sp} - \text{k-db})) \text{ AND } (\text{f-trip-1} = \text{e-tripped})),
\]

\[
\text{f-trip} = \text{e-not-tripped} \text{ if } ((\text{m-ai} < \text{f-sp}) \text{ AND } ((\text{m-ai} \geq (\text{f-sp} - \text{k-db})) \text{ AND } (\text{f-trip-1} = \text{e-tripped})),
\]

\[
\text{f-trip} = \text{e-not-tripped} \text{ if } (\text{m-ai} < (\text{f-sp} - \text{k-db})).
\]
3.1.4 State Transition Tables

The state transition tables represent next state functions. The top row contains the transition conditions enabling the states change. The leftmost column designates the reachable states. For instance, from Table 7, we see that when the system is in the state "e-time" and the condition "m-select = e-pressed" is TRUE, then the system will be in the "e-in-time" state.

The discussed function tables can be seen as a special kind of tabular expressions. The vertical condition tables could be interpreted as a one-dimensional normal tables, horizontal condition tables could be interpreted as a 1-dimensional inverted tables, the structured decision tables correspond to a 1-dimensional decision tables, and finally the state transition tables correspond to 2-dimensional normal tables.

Table 7. State Transition Table

<table>
<thead>
<tr>
<th>Previous State</th>
<th>Transition Condition</th>
<th>m-select</th>
<th>e-running</th>
<th>e-stopped</th>
<th>e-zero</th>
</tr>
</thead>
<tbody>
<tr>
<td>e-time</td>
<td>m-select</td>
<td>= e-pressed</td>
<td>= e-unpressed</td>
<td>AND</td>
<td>(m-start-stop = e-pressed)</td>
</tr>
<tr>
<td>e-in-time</td>
<td>m-select</td>
<td>= e-pressed</td>
<td>= e-unpressed</td>
<td>AND</td>
<td>(m-start-stop = e-pressed)</td>
</tr>
<tr>
<td>e-running1</td>
<td>m-select</td>
<td>= e-pressed</td>
<td>= e-unpressed</td>
<td>AND</td>
<td>(m-start-stop = e-pressed)</td>
</tr>
<tr>
<td>e-running2</td>
<td>m-select</td>
<td>= e-pressed</td>
<td>= e-unpressed</td>
<td>AND</td>
<td>(m-start-stop = e-pressed)</td>
</tr>
<tr>
<td>e-stopped1</td>
<td>m-select</td>
<td>= e-pressed</td>
<td>= e-unpressed</td>
<td>AND</td>
<td>(m-start-stop = e-pressed)</td>
</tr>
<tr>
<td>e-stopped2</td>
<td>m-select</td>
<td>= e-pressed</td>
<td>= e-unpressed</td>
<td>AND</td>
<td>(m-start-stop = e-pressed)</td>
</tr>
<tr>
<td>e-stopwatch</td>
<td>m-select</td>
<td>= e-pressed</td>
<td>= e-unpressed</td>
<td>AND</td>
<td>(m-start-stop = e-pressed)</td>
</tr>
</tbody>
</table>

Table 8. Vertical Condition Table

<table>
<thead>
<tr>
<th>Expression</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pressure = High ∨ Permitted</td>
<td>(Pressure = TooLow) AND (Overridden = True)</td>
<td>f-SafetyInjection=Off</td>
</tr>
<tr>
<td>Pressure = TooLow</td>
<td>(Pressure = TooLow) AND (Overridden = False)</td>
<td>f-SafetyInjection=On</td>
</tr>
</tbody>
</table>

Table 8. Vertical Condition Table

4 Transforming SCR Tables into Tabular Expressions

In this section, we illustrate how SCR tables can be transformed into function tables, and we provide examples for that.

4.1 Transforming Condition Tables

In SCR, a condition table (CT) defines a variable according to a mode and a condition. For instance, Table 1 is interpreted as the following:

If the Pressure is High or Permitted, or if the Pressure is TooLow and Overridden is True then SafetyInjection is Off. If the Pressure is TooLow and Overridden is False then SafetyInjection is On.

In vertical condition tables (VCT), the left bottom cell indicates the name of the function. For example, in Table 8, the name of the function is: f-SafetyInjection. The other bottom columns indicate the value of the function when the condition of the respective columns is true. This implies that each value of the function can be obtained from a disjunction of the conjunction of the respective cell and its respective mode value. This justifies our conversion given in algorithm 1. Notice that in converting a CT table into a VCT table, the last row remains the same, and it contains the name of the function and its values. To have a simplified table, any expression of the form False ∨ Expression or True ∧ Expression is substituted with Expression. As a result of the conversion, the VCT presented by Table 8 is semantically equivalent to the condition Table 1. The time complexity of this algorithm is $O(nm)$.

Algorithm 1 Converting CT tables into VCT tables

1. $n$ is the number of columns
2. $m$ is the number of rows
3. for $j$ from 1 to $n - 1$
   1. $Sum ← False$
   2. for $i$ from 1 to $m - 1$
      1. $VCT[i, j] ← CT[i, j] ∧ CT[i, j + 1]$
      2. $Sum ← Sum ∨ VCT[i, j]$
   3. $VCT[i,j] ← Sum$
4. $n$ is the number of columns
5. for $j$ from 2 to $n - 1$
   1. $VCT[2, j] ← CT[m, j + 1]$
6. end for

4.2 Transforming Event Tables

An event table defines a variable according to a mode and an event. An event represented by @T(c) means that condition c changes from false to true. For example, @T(Block=On) when T(Reset=Off) means that the operator turns Block from Off to On when the Reset is Off. The @T(Inmode) means that the system enters into the class of modes in that row. In Table 2, the mode Pressure is defined via the current mode, and the events defined on the variable WaterPress. Event tables and condition tables are semantically equivalent. In fact, in both tables a variable is defined via a mode and the change of a system state. Hence, event tables will also be transformed into vertical condition tables. Therefore, slightly modified algorithm 1 can be used to convert an SCR event table into a vertical condition table. As a result, we conclude that Table 9 is semantically equivalent.
Algorithm 2 Converting MTT tables into STT tables

\[
\begin{align*}
&\{\text{initialize the first row of STT to the input events starting at cell } [1, 2] \} \\
&n \text{ is the number of rows of MTT} \\
&\text{Let } R \text{ be a set} \\
&\text{for } i \text{ from } 1 \text{ to } n \text{ do} \\
&\quad \text{add MTT}[i, 1] \text{ to } R \\
&\text{end for} \\
&\text{Copy the elements of } R \text{ into the first raw of STT starting a cell } [1, 2] \\
&\text{\{initialize the first column of STT to the modes starting at cell } [2, 1] \}\}
&\text{Let } C \text{ be a set} \\
&\text{for } i \text{ from } 1 \text{ to } n \text{ do} \\
&\quad \text{add MTT}[i, 1] \text{ to } C \\
&\text{end for} \\
&\text{Copy the elements of } C \text{ into the first column of STT starting a cell } [2, 1] \\
&\text{for } i \text{ from } 1 \text{ to } n \text{ do} \\
&\quad \text{STT}[i, i+1] \leftarrow \text{MTT}[i, 2] \\
&\text{end for} \\
&\text{for } i \text{ from } 2 \text{ to } n \text{ do} \\
&\quad \text{STT} [\text{MTT}[i, 1], \text{MTT}[i, 2]] \leftarrow \text{MTT}[i, 3] \\
&\text{end for} \\
&\{\text{Handle STT’s empty cells} \}
&\text{for every mode } m \text{ and event } e \text{ do} \\
&\quad \text{if STT}[m, e] \text{ is an empty cell then} \\
&\quad \quad \text{STT}[m, e] \leftarrow m \\
&\quad \text{end if} \\
&\text{end for}
\end{align*}
\]

TABLE 9. Vertical Condition Table (Function table)

<table>
<thead>
<tr>
<th>(Pressure=TooLow)</th>
<th>(Pressure=Permitted)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(block =On) When Reset = Off)</td>
<td>(Pressure=TooLow)</td>
</tr>
<tr>
<td>1-Overridden= True</td>
<td>1-Overridden= False</td>
</tr>
</tbody>
</table>

Table 9. Vertical Condition Table (Function table)

4.3 Transforming Mode Transition Tables

A mode transition table (MTT) generates a destination mode from a mode and an event. Similarly, a state transition table (STT) represents next state functions, given a current state and a condition. Modes are seen classes of system states specifying the system behaviour. Therefore, the two tables are semantically equivalent. In the following we present an algorithm to show how Mode transition tables (MTTs) are transformed into state transition tables (STTs). In this algorithm we start by filling the rows of the STT table with modes of the MTT table, and the columns of the STT table with events of the MTT table. Since each cell in STT can be indexed by a mode \( m \) and an event \( e \) as \( T[m, e] \), we can easily compute the next state in STT as follows:

\[
\text{STT [MTT}[i, 1],\text{MTT}[i, 2]] \leftarrow \text{MTT}[i, 3]
\]

where \( \text{MTT}[i, 1] \) and \( \text{MTT}[i, 2] \) are the mode and event at row \( i \), respectively, and \( \text{MTT}[i, 3] \) is the next mode at row \( i \). Then, we have to merge the rows having identical states, and the columns having identical events to reduce the size of the table. The time complexity to look for rows with identical states, the columns having identical events, and merge them is \( O(n^2) \). However, we were able to reduce the complexity to \( O(n) \) with the indexing property of STTs tables that we proposed. In fact with the indexing property of STTs tables, we access directly to fill in the new states since they are indexed by mode and events. Finally, we fill the empty cells with the same current state to indicate that the state remains the same.

After applying algorithm 2, we found out that Table 10 is semantically equivalent to Table 3.

STT: f-Pressure

<table>
<thead>
<tr>
<th>( \text{Pressure} \geq \text{Low} )</th>
<th>( \text{Pressure} \geq \text{Permitted} )</th>
<th>( \text{Pressure} &lt; \text{Low} )</th>
<th>( \text{Pressure} &lt; \text{Permitted} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>TooLow</td>
<td>TooLow</td>
<td>TooLow</td>
<td>TooLow</td>
</tr>
<tr>
<td>Permitted</td>
<td>Permitted</td>
<td>Permitted</td>
<td>Permitted</td>
</tr>
<tr>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

Table 10. State Transition Table (Function table)

5 Conclusion

Both SCR and tabular expressions are well suited to specify software requirements. They both adopt a tabular notation which increases their readability, and, they were both successfully used in practice.

In this paper we show how SCR tables can be transformed into equivalent tabular expressions. Hence we can now use the formal apparatus of tabular expressions for SCR tables. The transformation proposed are quite efficient and easy to implement.

Acknowledgment

This research was partially supported by NSERC Grant of Canada.

References


