A Methodology for the Simplification of Tabular Designs in Model-Based Development

Monika Bialy, Mark Lawford, Vera Pantelic, and Alan Wassyng
McMaster Centre for Software Certification
Department of Computing and Software, McMaster University, Hamilton, Ontario L8S 4L8, Canada
Email: {bialym2, lawford, pantelv, wassyng}@mcmaster.ca

***Abstract*—Model-based development (MBD) is increasingly being used to develop embedded control software, with Matlab Simulink/Stateflow being the most widely used MBD language in the automotive industry. Stateflow *truth tables*, more traditionally known as *decision tables*, are often used for implementing complex decision-making logic. As the subsystems utilizing Stateflow truth tables evolve, they often grow more complex and become difficult to maintain and test. It is in part due to the nature of decision tables that makes them more difficult to check for desirable properties such as disjointness and completeness, resulting in reduced readability and scalability. *Tabular expressions* provide an alternative representation which does not suffer from many of the same problems. With the safety-critical nature of the automotive domain, as well as the continuous growth in both size and complexity of models, well-defined and principled methodologies are required for maintaining and refactoring tables. This paper presents a refactoring methodology for simplifying decision tables through the use of tabular expressions to facilitate testing, traceability and readability to help companies comply with ISO 26262. An automotive industrial case study is used to motivate the work and demonstrate the proposed methodology.**

I. Introduction

Model-based development (MBD) has become an increasingly prevalent paradigm in numerous domains. This is indeed the case for the automotive industry, with Matlab Simulink/Stateflow being the most widely used MBD platform for embedded control software. Complex decision logic in Simulink is often implemented using *Stateflow truth tables* (traditionally known as *decision tables*[1]). In general, tabular formats provide a way of presenting complex information in a concise, well-organized manner. However, Stateflow truth tables present several issues: lack of disjointness, allowance of indirect satisfaction of completeness through the use of the else case, ambiguous interpretation as a result of left-to-right semantics, and decreased readability due to ineffective syntax and non-Boolean type condition representation. The properties of disjointness (ensuring determinism through non-overlapping input cases) and completeness (requiring consideration of all possible inputs) are integral to safety-critical systems, as they raise the confidence in correct system performance in all conditions, and also aid in detecting gaps for the input cases considered. Furthermore, the readability of tables impacts traceability to requirements, while also facilitating understandability by those reviewing requirements. In the automotive industry, with the increasing reliance on software for implementing vehicle functions, models continually grow in size and complexity, and become increasingly difficult to understand, maintain, refactor, and test. Our industrial partner, an automotive OEM, has identified some designs containing Stateflow truth tables as particularly problematic for developers. Therefore, we believe more effective tabular constructs and reliable refactoring techniques for tables are becoming an important need for automotive OEMs.

[1]Stateflow truth tables are not truth tables in the classical sense, but are, in fact, decision tables.

Moreover, the ISO 26262 standard [1] for safety-related software development in the automotive domain, mandates the avoidance of constructs with possibly ambiguous semantics, and stresses the need for low complexity of designs. Additionally, it is expected for safety-critical software that complex Boolean expressions be tested extensively with Modified Condition/Decision Coverage (MC/DC). Not surprisingly, our automotive partner aims to reduce time and effort spent on testing while retaining or enhancing coverage. As a result, reducing efforts for achieving good coverage and creation of test cases while increasing MC/DC coverage is a priority.

In this paper, we propose a methodology for the transformation and simplification of decision tables in a guided refactoring approach. Within this methodology, we leverage the use of *tabular expressions*, which strive to express complex logic in a precise, yet concise, manner, and have been successfully used in industry [2], [3]. They overcome the problems inherent in decision tables by enforcing disjointness and explicit completeness. As we will see in Section II, Simulink's diagnostic tools do not enforce disjointness, and although completeness is checked, it is a property easily forced. Furthermore, tabular expressions are easily readable and understandable by both software developers and domain experts, and, therefore, used in software documentation. Translating decision tables to tabular expressions remedies the deficiencies of disjointness and completeness, while significantly increasing the readability and understandability of implemented decision logic. Consequently, the refactoring of such tables is facilitated as logical simplifications are easier to detect and apply. Once the simplifications have been applied to tabular expressions, then they can be translated back to decision tables for code generation. The methodology has been applied in real industrial case studies on current production vehicle models. These case studies demonstrated the effectiveness of the methodology, and as a result it has been adopted by our industry partner, with

978-1-4673-7043-1/15 $31.00 © 2015 IEEE
DOI 10.1109/FormaliSE.2015.15

refactored designs incorporated into production code.

Refactoring general decision tables has been studied [4], [5]; however, comparable simplifications on tabular expressions have not been explored. Work has been done on transforming tables between various forms (e.g. inverted, normal, etc.) [6], [7], and simplifications via table restructuring are available in [8]. However, there are no simplifications defined on the actual logic, with [9] being the closest with simple domain restrictions. Additionally, a simple heuristic for transforming between decision tables and horizontal condition tables (HCT) specifically, is needed. A recent tabular construct for reactive system requirements, EDT [10], gives similar motivations towards understandable and testable tables in automotive.

Our methodology, while primarily motivated and described in the scope of Simulink/Stateflow MBD, is equally applicable to decision tables in any context. The approach contains two novel components, both useful in their own right: transformations to/from decision tables and HCTs; and, five iterative logical simplification strategies for tabular expressions.

The paper is structured as follows. Section II describes tabular constructs used and their differences. Section III presents the methodology, illustrated by its application to a real-world automotive design from our industry partner. Section IV then compares the original and refactored designs from different aspects to give evidence of the value of the methodology. The final section gives conclusions and directions for future work.

This paper is based on the Master's thesis of the first author [11], and we refer the reader to it for further details.

II. Preliminaries

A. Decision Tables

Stateflow provides two formalisms for representing complex decision logic within a model: Stateflow *charts* for implementation as a finite state machine; and Stateflow truth tables as shown in Table I. In general, Stateflow truth tables offer a more straightforward means of implementing purely logical behaviour, and we have found them increasingly becoming the standardized construct for decision logic in industry.

TABLE I: Stateflow truth table

		Decisions			
#	Conditions	D$_1$	D$_2$	D$_3$	D$_4$
1	$Condition_1$	T	T	T	-
2	$Condition_2$	T	-	F	-
3	$Condition_3$	T	T	-	-
	Actions	1	2	3	4

#	Actions
1	$Action_1$
2	$Action_2$
3	$Action_3$
4	$Action_4$

Decision tables are structured in three sections. A condition section contains statements defining conditions. The possible outcomes for a condition are true, false or "don't care," represented as T, F, or - respectively. Decisions are relationships resulting from the conjunction of conditions, which are mapped to actions. The action section defines the operations to be performed as a consequence of these decisions being satisfied. Decision tables are read left-to-right, with semantics interpreted as an `if-then-else` statement. Table I corresponds to:

```
   if  $Condition_1 \wedge Condition_2 \wedge Condition_3$ then $Action_1$
elsif  $Condition_1 \wedge Condition_3$ then $Action_2$
elsif  $Condition_1 \wedge \neg Condition_2$ then $Action_3$
 else  $Action_4$
endif
```

For syntactical correctness, Simulink flags the properties:

1) **Overspecification:** Too many decisions are defined such that some are redundant and are never executed due to the left-to-right semantics of the tables
2) **Underspecification:** Not enough decisions such that some input cases are not covered

B. Tabular Expressions

Tabular expressions are an alternative and effective tabular construct for the documentation and specification of software. We provide a brief description of their semantics, however [12] gives a detailed discussion. In this work, we use tabular expressions in the form of *horizontal condition tables (HCT)*, shown in Table II. HCTs, especially those used for software requirements documentation, define a single function and are best used for describing a single behaviour [13]. Thus, we make use of HCTs which compute a single output.

TABLE II: Horizontal condition table

Conditions		Result
		Var
$Condition_1$	$Condition_2$	$Result_1$
	$\neg Condition_2$	$Result_2$
$\neg Condition_1$		$Result_3$

HCTs can also be interpreted as `if-then-else` statements, however it is important to note that there is no defined order in which the cases are evaluated:

```
   if  $Condition_1 \wedge Condition_2$ then $Var = Result_1$
elsif  $Condition_1 \wedge \neg Condition_2$ then $Var = Result_2$
elsif  $\neg Condition_1$ then $Var = Result_3$
endif
```

For any table to properly define a (total) function, two conditions must be satisfied:

1) **Disjointness:** Each distinct pair of rows, Row_i, Row_j is disjoint, i.e. $i \neq j \Rightarrow \neg(Row_i \wedge Row_j)$
2) **Completeness:** The disjunction of all Row_is is true, i.e. $(Row_1 \vee Row_2 \vee \ldots \vee Row_n) \Leftrightarrow$ true

Each row of a tabular expression can be thought of as a decision, and the results column as the action section.

C. Comparison

Although currently in abundant use, decision tables have many key shortcomings which have been identified in industry.

Disjointness is not properly enforced. Stateflow provides diagnostic tools to detect overspecified tables, where too many decisions are defined such that some are never executed. However, overspecification is a subset of non-disjointness, and thus the check fails to detect overlaps when the condition is still executed, as is the case with partially overlapping conditions. This is demonstrated in decisions (D_1, D_2) and (D_2, D_3) of Table I. Left-to-right semantics are necessary to disambiguate the apparently contradictory actions for overlapping decisions.

Completeness is properly checked, however, it is weakened by the reliance on the else case decision as a catch-all. A convenient but indirect means of ensuring completeness, it indiscriminately accommodates remaining unspecified cases. This does not lend itself well to safety-critical software, where all cases must be explicitly considered. This potentially hides errors in logic, despite the table being syntactically correct.

Implicit left-to-right semantics for the order of decision evaluation can lead to inconsistent table interpretations, especially from the perspective of an outside reviewer with no prior knowledge on the use of decision tables. A human reader may seize upon the first column that is satisfied, but miss earlier columns, therefore, misreading the table and interpreting its behaviour incorrectly. Also, a simple swap of two columns changes the semantics of the table. Historically, decision table semantics allowed for decision evaluation in any order [14], however, left-to-right semantics are now most common. Notwithstanding, it is a property not readily apparent through visual examination, nor is it intuitive in many cases.

Non-Boolean conditions, e.g. enumeration types and numerical ranges, are not portrayed intuitively. For example, when checking an enumerated type condition, each enumerator is treated as a separate condition even if an inherent relationship between them exists. To implement these separate yet related conditions, diagonal patterns of T values are used, inevitably reducing readability. Further, because decision tables fail to accommodate the mutually exclusive nature of these conditions, an else case is needed to catch remaining input combinations, even if they may not be technically possible.

Readability is further diminished when tables describe nontrivial logic. They quickly become large and difficult to understand by developers, fail to convincingly convey requirements, and display redundant information. As a result, we have found that they are not particularly scalable. Their interpretation requires constant referral to the condition and action sections in order to understand what the decisions signify. Hence, checking for errors such as contradictions becomes a strenuous task, and is not always supported by tools.

In comparison, tabular expressions address all these identified drawbacks of decision tables. For a table to be properly defined, disjointness and completeness must be satisfied. Else conditions are explicitly implemented as negations of one or more conditions. This promotes a more careful consideration of remaining cases, as opposed to mindlessly grouping them in a catch-all else case. Further, there is no imposed order for row evaluation, and they provide intuitive syntax for related non-Boolean conditions, as well as readability. One must simply scan the table in one direction, from left-to-right, instead of continually referring to the condition and action sections.

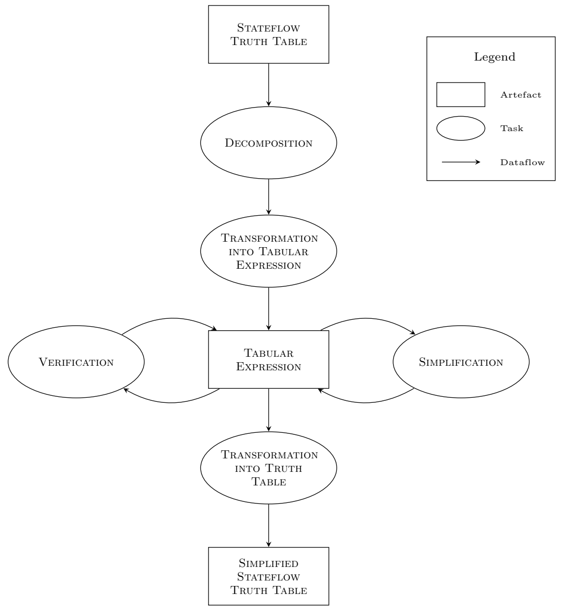

Fig. 1: Methodology for Stateflow truth table simplification

III. The Methodology

An overview of the methodology is given in Figure 1. Its strategy entails the transformation of the decision table into a tabular expression, first starting with decomposition, then subsequently transforming into a HCT. Using the HCT as the basis for refactoring, a set of simplifications are applied iteratively, which transform the table into a smaller, simplified format by way of removing/rearranging decision points. Finally, the HCT is transformed back into a Stateflow truth table. Alternatively, the *Tabular Expression Toolbox (TET)*, a Simulink toolbox developed at the McMaster Centre for Software Certification, can be used to directly implement the tabular expressions in Simulink [15]. After any given transformation, formally proving the table equivalence before and after can be done using *PVS (Prototype Verification System)* by SRI [16].

A step-by-step description of the methodology is given in the following subsections. It is also demonstrated on an industrial automotive model, as supplied to us by our industrial partner. The model contains a subsystem with four Stateflow truth tables, each of which performs arbitration of driver requests.

Due to space constraints, we present the application of our method only on one of these four tables, Table III. It is responsible for performing arbitration of driver requests to change the state of $cState_1$, using the current status of the system, i.e. the previous arbitrated status, whether or not a component is unlocked, and if the subsystem is faulty. Due to the proprietary nature of these tables, the signal/variable names have been obfuscated, but the logic remains unchanged.

A. Decomposition

If the table computes multiple outputs, it is decomposed into multiple tables, each computing a single output. This approach

TABLE III: Driver request arbitration from $cState_1$

fReqFromSt1($eDrvrRequest$:*enum*, $bFaulty$, $bCmpntUnlocked$:*bool*): *enum* =

		Decisions										
#	Conditions	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}
1	$eDrvrRequest == cState_1$	T	F	F	F	F	F	F	F	F	F	-
2	$eDrvrRequest == cState_2$	F	T	F	F	T	F	F	T	F	F	-
3	$eDrvrRequest == cState_3$	F	F	T	F	F	T	F	F	T	F	-
4	$eDrvrRequest == cState_4$	F	F	F	T	F	F	T	F	F	T	-
5	$bCmpntUnlocked$	-	T	T	T	-	-	-	-	-	-	-
6	$bFaulty$	-	F	F	F	T	T	T	-	-	-	-
	Actions	1	2	3	4	5	5	5	5	5	5	1

#	Actions
1	$eArbRequest$=$cState_1$; $bActionRequired$=$false$
2	$eArbRequest$=$cState_2$; $bActionRequired$=$false$
3	$eArbRequest$=$cState_3$; $bActionRequired$=$false$
4	$eArbRequest$=$cState_4$; $bActionRequired$=$false$
5	$eArbRequest$=$cState_1$; $bActionRequired$=$true$

introduces modularity, provides better requirements traceability, and often yields greater reductions during simplification. Should this approach be infeasible in light of actions being complex Matlab code[2], abstraction of actions can be done by implementing actions in functions outside of the table. The new action becomes a simple function call. Alternatively, the action numbers themselves can be seen as an abstraction of the action and be used to represent the output instead.

TABLE IV: Decomposition with respect to $eArbRequest$

		Decisions										
#	Conditions	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}
1	$eDrvrRequest == cState_1$	T	F	F	F	F	F	F	F	F	F	-
2	$eDrvrRequest == cState_2$	F	T	F	F	T	F	F	T	F	F	-
3	$eDrvrRequest == cState_3$	F	F	T	F	F	T	F	F	T	F	-
4	$eDrvrRequest == cState_4$	F	F	F	T	F	F	T	F	F	T	-
5	$bCmpntUnlocked$	-	T	T	T	-	-	-	-	-	-	-
6	$bFaulty$	-	F	F	F	T	T	T	-	-	-	-
	Actions	1	2	3	4	1	1	1	1	1	1	1

#	Actions
1	$eArbRequest$=$cState_1$
2	$eArbRequest$=$cState_2$
3	$eArbRequest$=$cState_3$
4	$eArbRequest$=$cState_4$

Table III was decomposed into two tables, one of which is Table IV, computing only $eArbRequest$. During this step, Actions 1 and 5 both resulted in $eArbRequest$ being assigned $cState_1$— thus, they were combined into Action 1.

B. Decision Tables To Tabular Expressions

The following transforms a Stateflow truth table to a HCT.

1) Augment Decisions with Conditions and Actions The two notations differ in their representation of decisions. Thus, for each decision, T values are replaced straightforwardly by the condition they denote as being true. For F values, the condition logic is negated. "Don't cares" do not require replacement. A similar treatment is given to actions, where actual values are put in place of indices. Afterwards, all pertinent information is contained within the decisions, and so condition and action sections are removed, as shown in Table V. For readability purposes, we denote the enumeration checking conditions as:

$$lState_i = (eDrvrRequest == cState_i), i = 1...4.$$

2) Transpose The entire table is transposed to orient the data appropriately for tabular expression form, i.e. left-to-right order of evaluation of decisions. Table VI shows the result.

[2]Departing from traditional decision tables, actions in Simulink truth tables can make use of `for` loops, `if` statements, and persistent variables

TABLE VI: Transposing

$lState_1$	$\neg lState_2$	$\neg lState_3$	$\neg lState_4$	-	-	$cState_1$
$\neg lState_1$	$lState_2$	$\neg lState_3$	$\neg lState_4$	$bCmpntUnlocked$	$\neg bFaulty$	$cState_2$
$\neg lState_1$	$\neg lState_2$	$lState_3$	$\neg lState_4$	$bCmpntUnlocked$	$\neg bFaulty$	$cState_3$
$\neg lState_1$	$\neg lState_2$	$\neg lState_3$	$lState_4$	$bCmpntUnlocked$	$\neg bFaulty$	$cState_4$
$\neg lState_1$	$lState_2$	$\neg lState_3$	$\neg lState_4$	-	$bFaulty$	$cState_1$
$\neg lState_1$	$\neg lState_2$	$lState_3$	$\neg lState_4$	-	$bFaulty$	$cState_1$
$\neg lState_1$	$\neg lState_2$	$\neg lState_3$	$lState_4$	-	$bFaulty$	$cState_1$
$\neg lState_1$	$lState_2$	$\neg lState_3$	$\neg lState_4$	-	-	$cState_1$
$\neg lState_1$	$\neg lState_2$	$lState_3$	$\neg lState_4$	-	-	$cState_1$
$\neg lState_1$	$\neg lState_2$	$\neg lState_3$	$lState_4$	-	-	$cState_1$
-	-	-	-	-	-	$cState_1$

3) Group Related Conditions Related conditions are grouped together in a single column. If the grouped conditions are mutually exclusive, only the true condition is placed in the cell. The related conditions in Table VI were the enumeration checks located in the first four columns. Thus, Table VII merges them and retains only the condition which is true.

TABLE VII: Grouping related conditions

1	$lState_1$	-	-	$cState_1$
2	$lState_2$	$bCmpntUnlocked$	$\neg bFaulty$	$cState_2$
3	$lState_3$	$bCmpntUnlocked$	$\neg bFaulty$	$cState_3$
4	$lState_4$	$bCmpntUnlocked$	$\neg bFaulty$	$cState_4$
5	$lState_2$	-	$bFaulty$	$cState_1$
6	$lState_3$	-	$bFaulty$	$cState_1$
7	$lState_4$	-	$bFaulty$	$cState_1$
8	$lState_2$	-	-	$cState_1$
9	$lState_3$	-	-	$cState_1$
10	$lState_4$	-	-	$cState_1$
11	-	-	-	$cState_1$

4) Ensure Disjointness and Completeness Firstly, the effect of the left-to-right semantics is eliminated to introduce disjointness. Decisions are sequentially inspected for overlaps in conditions with preceding decisions. If a condition is found to be overlapping, it is restricted to the remaining values not covered in previous cases. Secondly, completeness is ensured. Without considering the else case, if implemented, each condition is examined to ensure that all its possible inputs are covered. Absent cases are added, and the else case is restricted so it does not overlap with preceding decisions, i.e., it is rewritten as the negation of other conditions.

In Table VII, rows 8-10 overlap with rows 2-7. Therefore, the overlapping rows are made more specific in Table VIII. The else case is removed also, as all cases are covered. It can remain or be reinserted if needed to catch hardware errors.

5) Formatting Consecutive cells performing the exact same condition checks are merged into single cells spanning across rows. Likewise, "don't care" cells are combined across columns with previous conditions. The table is formatted so that it complies with standard notation of tabular expressions (e.g. headings and borders).

TABLE VIII: Ensuring disjointness and completeness

1	$lState_1$	-	-	$cState_1$
2	$lState_2$	$bCmpntUnlocked$	$\neg bFaulty$	$cState_2$
3	$lState_3$	$bCmpntUnlocked$	$\neg bFaulty$	$cState_3$
4	$lState_4$	$bCmpntUnlocked$	$\neg bFaulty$	$cState_4$
5	$lState_2$	-	$bFaulty$	$cState_1$
6	$lState_3$	-	$bFaulty$	$cState_1$
7	$lState_4$	-	$bFaulty$	$cState_1$
8	$lState_2$	$\neg bCmpntUnlocked$	$\neg bFaulty$	$cState_1$
9	$lState_3$	$\neg bCmpntUnlocked$	$\neg bFaulty$	$cState_1$
10	$lState_4$	$\neg bCmpntUnlocked$	$\neg bFaulty$	$cState_1$

TABLE V: Augmenting decisions with conditions and actions

$lState_1$	$\neg lState_1$	$\neg lState_1$	$\neg lState_1$	$\neg lState_1$	$\neg lState_1$	$\neg lState_1$	$\neg lState_1$	$\neg lState_1$	$\neg lState_1$	-
$\neg lState_2$	$lState_2$	$\neg lState_2$	$\neg lState_2$	$lState_2$	$\neg lState_2$	$\neg lState_2$	$lState_2$	$\neg lState_2$	$\neg lState_2$	-
$\neg lState_3$	$\neg lState_3$	$lState_3$	$\neg lState_3$	$\neg lState_3$	$lState_3$	$\neg lState_3$	$\neg lState_3$	$lState_3$	$\neg lState_3$	-
$\neg lState_4$	$\neg lState_4$	$\neg lState_4$	$lState_4$	$\neg lState_4$	$\neg lState_4$	$lState_4$	$\neg lState_4$	$\neg lState_4$	$lState_4$	-
-	$bCmpntUnlocked$	$bCmpntUnlocked$	$bCmpntUnlocked$	-	-	-	-	-	-	-
-	$\neg bFaulty$	$\neg bFaulty$	$\neg bFaulty$	$bFaulty$	$bFaulty$	$bFaulty$	-	-	-	-
$cState_1$	$cState_2$	$cState_3$	$cState_4$	$cState_1$	$cState_1$	$cState_1$	$cState_1$	$cState_1$	$cState_1$	$cState_1$

TABLE IX: Formatting

Conditions			Result $eArbRequest$
$lState_1$			$cState_1$
$lState_2$	$\neg bFaulty$	$bCmpntUnlocked$	$cState_2$
		$\neg bCmpntUnlocked$	$cState_1$
	$bFaulty$		$cState_1$
$lState_3$	$\neg bFaulty$	$bCmpntUnlocked$	$cState_3$
		$\neg bCmpntUnlocked$	$cState_1$
	$bFaulty$		$cState_1$
$lState_4$	$\neg bFaulty$	$bCmpntUnlocked$	$cState_4$
		$\neg bCmpntUnlocked$	$cState_1$
	$bFaulty$		$cState_1$

C. Simplification Transformations

Given any HCT, not necessarily as a result of the previous steps, the following transformations are used to simplify it iteratively. They can be applied in any order, based on applicability and the desired final form of the table. Using requirements as a guide may be helpful, as this refactoring approach aims to provide better requirements traceability.

1) Condition Ordering: Manipulation of the condition ordering, both vertically and horizontally, can be employed to influence the organizational and visual appearance of the tabular expression, as well as to manipulate the implementation of the tabular expression for code generation [12].

a) Vertical: Evaluation speed of the expression can be increased by forcing specific conditions to be evaluated as early as possible [12]. The vertical arrangement of conditions can be adjusted to increase efficiency in the evaluation of cases specifically by moving decisions with the most "don't cares", i.e. fewest amount of evaluations, to the upper rows, allowing for earlier evaluation. If the condition is nested, it can be moved up/down so long as it remains nested.

TABLE X: Vertical reordering of $bFaulty$

Conditions			Result $eArbRequest$
$eDrvrRequest == cState_4$			$cState_1$
$eDrvrRequest == cState_2$	$bFaulty$		$cState_1$
	$\neg bFaulty$	$bCmpntUnlocked$	$cState_2$
		$\neg bCmpntUnlocked$	$cState_1$
$eDrvrRequest == cState_3$	$bFaulty$		$cState_1$
	$\neg bFaulty$	$bCmpntUnlocked$	$cState_3$
		$\neg bCmpntUnlocked$	$cState_1$
$eDrvrRequest == cState_4$	$bFaulty$		$cState_1$
	$\neg bFaulty$	$bCmpntUnlocked$	$cState_4$
		$\neg bCmpntUnlocked$	$cState_1$

For visual consistency in vertical ordering, positive conditions are moved to precede negated conditions in Table X.

b) Horizontal: The horizontal ordering of conditions is used as a visual means of representing the dominance of conditions with respect to the decision. Left-most conditions are executed first, and thus moving conditions to the left signifies that they are more important as they are evaluated in more cases. Likewise, if it is advantageous to evaluate certain conditions infrequently: nesting them is a good course of action such that they are only checked after others are evaluated, narrowing the cases said conditions are needed.

TABLE XI: Horizontal reordering of $bFaulty$

Conditions			Result $eArbRequest$
$bFaulty$			$cState_1$
$\neg bFaulty$	$eDrvrRequest == cState_1$	$bCmpntUnlocked$	$cState_1$
		$\neg bCmpntUnlocked$	$cState_1$
	$eDrvrRequest == cState_2$	$bCmpntUnlocked$	$cState_2$
		$\neg bCmpntUnlocked$	$cState_1$
	$eDrvrRequest == cState_3$	$bCmpntUnlocked$	$cState_3$
		$\neg bCmpntUnlocked$	$cState_1$
	$eDrvrRequest == cState_4$	$bCmpntUnlocked$	$cState_4$
		$\neg bCmpntUnlocked$	$cState_1$

Table XI demonstrates this technique. When $bFaulty$ is true $State_1$ is always the output, no matter the value of $eDrvrRequest$, i.e., if a fault arises, the safe response is to remain in the current state. According to these requirements, we wish to treat $bFaulty$ as the more dominant condition over $eDrvrRequest$. Thus, $bFaulty$ is moved to the left, and $eDrvrRequest$ is removed when $bFaulty$ is true.

2) Granted State Simplification: This simplification is particularly useful for systems which arbitrate operational modes. Here, the conditions responsible for checking the mode are eliminated when the resulting actions simply grant/accept the change in output. The check is not required, as the condition's value is granted and passed through. Thus, we generalize this situation by placing the mode variable directly in the result column. A horizontal reordering of conditions may be necessary, as is the case in Table XII.

TABLE XII: Granted state simplification of $eDrvrRequest$

	Conditions			Result $eArbRequest$
1	$bFaulty$			$cState_1$
2	$\neg bFaulty$	$bCmpntUnlocked$		$eDrvrRequest$
3		$\neg bCmpntUnlocked$	$eDrvrRequest == cState_1$	$cState_1$
4			$eDrvrRequest == cState_2$	$cState_1$
5			$eDrvrRequest == cState_3$	$cState_1$
6			$eDrvrRequest == cState_4$	$cState_1$

In Table XI, if no faults are detected and the component is unlocked, the driver request is granted, i.e., regardless of $eDrvrRequest$, the output is $eDrvrRequest$. Thus, Table XII shows it removed as a condition and used directly as a result.

3) Removal of "Don't Care" Conditions: If a condition does not affect the outcome of a decision, it can be removed from its computation by being treated as a "don't care". This is done by identifying multiple instances of the same output in the results column, then moving backwards through the conditions required to reach these outputs. If the paths are the same, save for one condition, they can be combined.

TABLE XIII: Generic example of “don't care” simplification

	Conditions				Result
					$Output$
1	$Condition_1$	$\dots$	$Condition_i$	$Condition_j$	$Action_1$
2				$\neg Condition_j$	$Action_1$
n	$\dots$				$\dots$

Conditions			Result
			$Output$
$Condition_1$	$\dots$	$Condition_i$	$Action_1$
$\dots$			$\dots$

In Table XIII, rows 1 and 2 result in the same output. Each contains the same conditions, with the exception of row 2 negating $Condition_j$. Thus, these rows are combined. An example of this simplification on a nested condition is given in [11], with horizontal rearrangement being required after.

The removal of a Boolean condition with a cardinality of two was shown, however, this simplification also applies to conditions with a greater number of potential values. The number of rows required to be combined is equal to its type's cardinality, where jointly the rows cover the range of the type.

TABLE XIV: Simplification of $eDrvrRequest$ as “don't care”

	Conditions		Result
			$eArbRequest$
1	$bFaulty$		$cState_1$
2	$\neg bFaulty$	$bCmpntUnlocked$	$eDrvrRequest$
3		$\neg bCmpntUnlocked$	$cState_1$

In our example, it is evident in Table XII that $eDrvrRequest$ has no impact when there is a fault and the component is locked, as the output is $cState_1$ regardless. In this case, $eDrvrRequest$'s type has a cardinality of four, so in order to treat it as a “don't care”, four distinct rows which make up the complete range of the condition's type are combined, namely rows 3-6. Thus, we apply the non-nested simplification, which results in Table XIV. This concludes the simplification of the automotive example. Although rows 1 and 3 result in the same output, they cannot be combined unless they are expanded first, as described in Section III-C5. General examples are given for the remaining simplifications.

4) Grouping: The constraint on merging the complete range of a condition, given in Section III-C3, can be relaxed when grouping a subset is desired. This is particularly useful for enumerated types, shown in Table XV, employed for the representation of modes where it is the case that removing these mode-centric conditions in their entirety is not achievable, nor necessary. A subset of conditions can be grouped, however, they must lead to the same output.

5) Compound Simplification: This strategy expands an already simplified row, enabling further table simplification through the use of the newly introduced rows. Here, conditions are straightforwardly expanded into all values of their type, with the same action carried across all newly added rows. This strategy essentially reverses Section III-C3 and III-C4. It proves useful for the restructuring of already simplified tables, and is beneficial when altering an existing table to display a design in a specific manner that corresponds to a requirement.

TABLE XV: Generic example of grouping

Conditions			Result
			$Output$
$eVar == cEnum_1$	$Condition_1$		$Action_1$
	$\neg Condition_1$	$Condition_2$	$Action_2$
		$\neg Condition_2$	$Action_3$
$\dots$	$\dots$		$\dots$
$eVar == cEnum_i$	$Condition_1$		$Action_1$
	$\neg Condition_1$	$Condition_2$	$Action_2$
		$\neg Condition_2$	$Action_3$
$\dots$	$\dots$		$\dots$
$eVar == cEnum_n$	$\dots$		$Action_n$

Conditions			Result
			$Output$
$eVar == cEnum_1 \vee \dots \vee eVar == cEnum_i$	$Condition_1$		$Action_1$
	$\neg Condition_1$	$Condition_2$	$Action_2$
		$\neg Condition_2$	$Action_3$
$\dots$	$\dots$		$\dots$
$eVar == cEnum_n$	$\dots$		$Action_n$

D. Tabular Expressions to Decision Tables

1) Remove Tabular Expression Formatting Strip tabular expression-specific formatting conventions from the table, e.g. headings. Expand each vertically merged cell, first dividing it into the number of rows it was grouped over, and then explicitly denoting the condition per cell. Horizontally merged cells are also divided, but with the condition being moved to its corresponding column, and the remaining newly added cells denoted as “don't care” conditions. The result is shown in Table XVI.

TABLE XVI: Formatting removal

$bFaulty$	-	$cState_1$
$\neg bFaulty$	$bCmpntUnlocked$	$eDrvrRequest$
$\neg bFaulty$	$\neg bCmpntUnlocked$	$cState_1$

2) Transpose Re-orient the decisions such that they are read in a top-down fashion, as shown in Table XVII.

TABLE XVII: Transposing

$bFaulty$	$\neg bFaulty$	$\neg bFaulty$
-	$bCmpntUnlocked$	$\neg bCmpntUnlocked$
$cState_1$	$eDrvrRequest$	$cState_1$

3) Construct Condition Section Add a condition section, populating each row with the corresponding condition in that row of the decision. Replace the conditions of the decisions with T/F Boolean constants, as in Table XVIII.

TABLE XVIII: Constructing condition section

		Decisions		
#	Conditions	D_1	D_2	D_3
1	$bFaulty$	T	F	F
2	$bCmpntUnlocked$	-	T	F
	Actions	$cState_1$	$eDrvrRequest$	$cState_1$

4) Construct Action Section Create the action section with a numbered entry for each unique action found in the last row. Replace actions with their corresponding index.

With this, the methodology is concluded. These heuristics were designed to be an easy-to-follow process of performing guided refactoring. The equivalence between steps is easy to see. For the example, equivalence between the original and refactored tabular expressions was proven with PVS. Verifying equivalence between other intermediate steps is also possible.

TABLE XIX: Constructing action section

		Decisions		
#	Conditions	D_1	D_2	D_3
1	$bFaulty$	T	F	F
2	$bCmpntUnlocked$	-	T	F
	Actions	1	2	1

#	Actions
1	$eArbRequest$=$cState_1$
2	$eArbRequest$=$eDrvrRequest$

IV. Automotive Case Study

In this section, we investigate how the described methodology affects designs in terms of testability, complexity and requirements traceability. The *Simulink Design Verifier (SDV)* tool by Mathworks automatically generates test cases from Simulink/Stateflow models in order to maximize a number of test coverage metrics. It also calculates cyclomatic complexity [17], a well-known metric that measures the amount of decision logic in a model. We are concerned with reducing the complexity, as well as reducing efforts in testing with regards to the number of tests required for MC/DC coverage.

Analysis was done on the arbitration subsystem described in Section III, with the refactored Stateflow truth tables taking the place of the original tables. Although we only show a single simplified table from this design, the methodology was applied to the remaining three tables of the subsystem, each of which resulted in two tables. The comparison between original and refactored designs is given in Table XX as Case Study 1. Additionally, a second case study was done on a more complex automotive system. Due to space constraints we refer the reader to [11] for the full details, however, the results are also summarized in Table XX.

TABLE XX: Testing and complexity analysis

	Case Study 1		Case Study 2	
	Original	Refactored	Original	Refactored
Tests	7	9	23	6
Test Steps	97	48	1214	24
Test Time (s)	18	7.8	100	3.6
Number of Objectives	1016	371	1954	498
Objectives Satisfied	797 (78%)	311 (84%)	1591 (81%)	445 (89%)
Objectives Proven Unsatisfiable	219 (22%)	60 (16%)	202 (10%)	53 (10%)
Objectives Undecided	0	0	161 (8%)	0
MC/DC Coverage	62%	63%	57%	74%
Cyclomatic Complexity	274	107	935	248

For the presented example, it is evident that the number of test objectives significantly reduced (by more than half), decreasing the number of test steps required to satisfy these objectives, and ultimately decreasing the testing time needed. MC/DC was slightly increased, and the number of satisfied objectives increased from 78% to 84%, with the reason for this improvement being the simplification of the decision logic. This is confirmed by the cyclomatic complexity, which decreased considerably (by a factor of $\sim$2.5). Case Study 2 gave the same trends in results, but boasted more significant improvements. Moreover, due to the large and complex nature of the original design, testing resulted in undecided objectives which could not be proved/disproved by the SDV engine. Application of the methodology resolved this issue.

Regarding their use as requirements, it is evident that the refactored HCT made requirements more traceable and readable. A requirement for the system was, "remain in $cState_1$ when there are no faults, but the component is locked," and it is clearly shown in the last row of Table XIV.

V. Conclusion

In this paper, a novel methodology for the guided refactoring of tabular designs in MBD was developed. This methodology was applied to an automotive industrial case study, resulting in significant gains in terms of reduction of testing efforts, minimization of complexity, and improved requirements traceability. Currently, tools in the Matlab Simulink environment for model refactoring are limited [18]; therefore, future work entails automating this methodology.

References

[1] "ISO 26262-6:2011," International Organization for Standardization/Technical Committee 22 (ISO/TC 22), Geneva, Switzerland, Standard Document No. ISO 26262-6:2011, Nov. 2011.

[2] K. L. Heninger, "Specifying software requirements for complex systems: New techniques and their application," *IEEE Transactions on Software Engineering*, no. 1, pp. 2–13, 1980.

[3] G. Archinoff, R. Hohendorf, A. Wassyng, B. Quigley, and M. Borsch, "Verification of the shutdown system software at the darlington nuclear generating station," in *International Conference on Control & Instrumentation in Nuclear Installations*, 1990.

[4] S. L. Pollack, H. T. Hicks, and W. J. Harrison, *Decision Tables: Theory and Practice*, ser. Wiley Business Data Processing Series, R. G. Canning and J. D. Couger, Eds. Wiley-Interscience, 1971.

[5] K. Shwayder, "Combining decision rules in a decision table," *Communications of the ACM*, vol. 18, no. 8, pp. 476–480, 1975.

[6] H. Shen, J. Zucker, and D. L. Parnas, "Table transformation tools: Why and how," in *COMPASS'96. Proceedings of 11th Annual Conference on Computer Assurance*. Gaithersburg, MD, USA: IEEE and National Institute of Standards and Technology, 1996, pp. 3–11.

[7] T. Fu, "Structured decision table, generalized decision table conversion tool," Master's thesis, McMaster University, 1999.

[8] J. I. Zucker, "Transformations of normal and inverted function tables," *Formal Aspects of Computing*, vol. 8, no. 6, pp. 679–705, 1996.

[9] P. Rastogi, "Specialization: An approach to simplifying tables in software documentation," Master's thesis, McMaster University, 1998.

[10] R. Venkatesh, U. Shrotri, G. M. Krishna, and S. Agrawal, "EDT: A specification notation for reactive systems," in *Proceedings of the conference on Design, Automation & Test in Europe*, European Design and Automation Association. IEEE, 2014, p. 215.

[11] M. Bialy, "A methodology for the simplification of tabular designs in model-based development," Master's thesis, McMaster University, 2014.

[12] Y. Jin and D. L. Parnas, "Defining the meaning of tabular mathematical expressions," *Science of Computer Programming*, vol. 75, no. 11, pp. 980–1000, 2010.

[13] A. Wassyng and R. Janicki, "Tabular expressions in software engineering," in *Proceedings of ICSSEA*, vol. 4, Paris, France, 2003, pp. 1–46.

[14] U. W. Pooch, "Translation of decision tables," *ACM Computing Surveys (CSUR)*, vol. 6, no. 2, pp. 125–151, 1974.

[15] C. Eles and M. Lawford, "A tabular expression toolbox for Matlab/Simulink," in *NASA Formal Methods*, ser. Lecture Notes in Computer Science. Springer, 2011, vol. 6617, pp. 494–499.

[16] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *CADE-11*. Springer, 1992, pp. 748–752.

[17] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, pp. 308–320, 1976.

[18] S. Radpour, L. Hendren, and M. Schäfer, "Refactoring MATLAB," in *Compiler Construction*. Springer, 2013, pp. 224–243.