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

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

By

Monika Bialy, B.Co.Sc.(Hon)

A Thesis

Submitted to the School of Graduate Studies

in Partial Fulfillment of the Requirements for the Degree

Master of Applied Science

McMaster University

Master of Applied Science (2014)	McMaster University
(Software Engineering)	Hamilton, Ontario

Title:	A Methodology for the Simplification of Tabular Designs in Model-Based Development
Author:	Monika Bialy, B.Co.Sc.(Hon) (Laurentian University)
Supervisors:	Dr. Mark Lawford, Dr. Alan Wassyng
Number of Pages:	vii, 182

Abstract

Model-based development (MBD) is an increasingly used approach for the development of embedded control software, with Matlab Simulink/Stateflow as the widely accepted language. The adoption of this development paradigm is prevalent in many safety-critical domains, including the automotive industry. With an increasing reliance on software for controlling vehicle functionality and the yearly advent of new vehicle features, automotive models have been growing in size and complexity, causing them to become increasingly difficult to maintain, refactor, and test. Given the centrality of models in MBD, it is a requisite that they be maintained under well-defined and principled software development processes that use precise notation to document system requirements and behavioural design description.

Tabular methods have long been used for defining decision-making logic in software, due to their concise and precise manner of communicating complex behaviour, so it is not surprising that they are finding increased use in automotive software models. Thus their presence in Simulink models is increasingly prominent in the implementation of complex behaviour in production code. As a result of the safety-critical nature of the automotive industry, as well as the increasing size and complexity of its models, reliable refactoring and simplification techniques for tabular expressions are becoming an important need for automotive companies. To address this need, this thesis presents a methodology for refactoring complex tabular designs to improve requirements traceability with a focus on Matlab Simulink/Stateflow and the MBD approach.

A case study of industrial examples from an automotive partner are used to motivate the work and demonstrate the proposed methodology's effectiveness in reducing design size and complexity, while also increasing testability and requirements traceability.

Acknowledgments

I am particularly grateful to my supervisors, Drs. Mark Lawford and Alan Wassyng, for their insightful direction and instruction throughout my graduate work. I am equally indebted to Dr. Vera Pantelic for her mentorship and sound guidance.

Many thanks also to the entirety of my academic and industrial colleagues associated with the Automotive Partnership Canada project, as well as the McMaster Centre for Software Certification.

Finally, I wish to abundantly thank my family and friends for their unending love and support.

Contents

List of Figures

List of Tables

List of Acronyms

ASIL Automotive Safety Integrity Level

CSV comma-separated values

ECU Electrical Control Unit

ISO International Standards Association

IBV Instrumentation Based Verification

LHS left-hand-side

MBD Model-Based Development

MC/DC Modified Condition/Decision Coverage

OEM Original Equipment Manufacturer

RHS right-hand-side

SDV Simulink Design Verifier

TET Tabular Expression Toolbox

Chapter 1

Introduction

In this chapter, we introduce the main motivating factors driving the body of this work. These are explored in the general context of software engineering in Section 1.1.1, as well as with a narrowed focus on the the automotive industry in Section 1.1.2. Each perspective presents arguments for the justification and necessity of this work, and at times are complementary in nature. Contributions of this thesis are described concisely in Section 1.2, while Section 1.3 goes on to give the remaining structure of the thesis.

1.1 Motivation

The majority of today's complex systems, from medical devices to nuclear power generating stations, rely heavily on software to implement complex and safety-critical functionality. The ease of modifying software, in comparison to hardware, lends itself to its constantly changing and evolving nature. As a consequence, significant effort is devoted to defining well-principled and scalable processes for the design, development, and maintenance of these systems.

In keeping with this sentiment, the following work strives to address several deficiencies of said processes.

1.1.1 General Context

In the subsequent sections, we investigate the challenges of engineering embedded software systems through the use of Model-Based Development (MBD), and further strive to understand the unique challenges found in the automotive industry, specifically through the collaboration with an industry OEM partner.

1.1.1.1 Model-Based Development

Model-Based Development has become an increasingly prevalent paradigm, dominating such domains as nuclear, aerospace, and automotive. A dataflow programming approach, MBD employs the use of software models to describe the behaviour of embedded software systems. Models are used for simulation, code generation, test generation, formal verification, and numerous other purposes. Initially, the intent behind the use of models was to facilitate rapid prototyping, and code generation of small systems. However, today's software systems are substantially large and complex, shifting the use of the models to be long-term implementations, maintained and evolved over the span of years and multiple product lines. Therefore, the overarching intent of MBD has migrated to the production of reusable software models with a high degree of component flexibility, while still accommodating rapid time-to-market demands. Given the centrality of models in MBD, it is a requisite that they be developed under well-defined and principled software development processes.

An ever-present problem which permeates the applications of MBD is the

prodigious size and complexity of models. With the continual, long-term evolution of software models, the inevitable augmentation and modification of designs brings about steady growth in model size and complexity. For this reason, strategies for addressing overly-complex components is a necessity.

In the embedded software domain, instead of developing systems in the classical sense of writing code, models are instead used as the basis for code generation. Some of these models are implemented using domain-specific dataflow languages, and unlike textual programming languages, make use of control block diagrams for specification. For programming languages in general, guidelines exist which define methods of properly managing software throughout the development process, however considerable less direction is available for MBD approaches. Therefore, there is a gap in terms of processes dedicated for these domain specific tools. MBD employs the use of domain-specific languages and environments, necessitating their own individual consideration and study. The most widely used language is the data flow graphical language Matlab Simulink, along with Stateflow, its supplementary state chart notation (Weeks and Moskwa 1995).

1.1.1.2 Software Requirements

Requirements specification is a critical component of the development of software systems. The majority of software failures can be traced to poor software requirements (Leveson 2004). Although the importance of software requirements is widely acknowledged, rigourous requirements engineering is an area that has largely been neglected by software developers in general.

Requirements specification and maintenance are often neglected and not given the importance they deserve. As a result, it is commonly the case that

requirements solicitation occurs during other stages of the software development process. Retroactively extracting requirements from existing, already developed software generally proves to be an arduous process, and specifying them precisely yet effectively is also an objective of substantial importance.

Software requirements are a crucial component to ensure the longevity of the system that contains that software. Within the automotive domain, entire systems are seldom constructed from the ground up. They are the result of years of work. It is also common for software to be purchased and acquired from other Original Equipment Manufacturers (OEMs). At times, models or some small subsets of the entire system are developed and added. In both cases, the software has had, and will continue to have, a long lifespan. Thus, software requirements need to be maintained and updated continuously.

Furthermore, models are increasingly complex, but the development lifecycle does not reflect this and is actually becoming faster in reaction to market pressures. Hence maintaining software requirements represents a considerable problem. Additionally, requirements must be specified in a precise format, while remaining readable for domain experts. Although mathematical formulae capture the precision required for a software implementation, they are seldom practical for domain experts to parse and easily understand.

Requirements traceability is the ability to correlate a software implementation to its origin in the requirements documentation. Traceability is now widely recognized as a crucial property of complex software systems, and is important to the maintainability of the software. Owing to the non-textual specification of MBD languages, traceability to and from requirements is more difficult. In general, it has become extremely difficult to visually follow requirements traces in the MBD world. The importance of traceability is stressed in

safety-critical domains, especially when it comes to software verification and validation. For safety-critical applications we must be cognizant of the added obligation to make software safe. With respect to models specifically, this means traceability and thorough testing. Furthermore, current automotive safety-critical applications are highly complex in order to accommodate the complex control of critical functions, while also mandating traceability. This complexity is also reflected in the model size.

Therefore, exploring new methods of extracting and requirements from existing designs is necessary, while also taking into consideration the traceability of refactored designs.

1.1.1.3 Tabular Representations of Logic for Software

Multi-dimensional tabular notations have long been used as a structure for organizing and representing data in a simple and readable format. Dating back to the early years of the computer science field, *decision tables* emerged as successor formalism for other notations, namely flowcharts and narratives. Decision tables boasted many advantages over these approaches, firstly, as an aid in software documentation. An effective means of standardizing communication in general, they reduced ambiguity in interpretation and extricated superfluous information. Additionally, their superior organization and concision in expressing complex decision-making logic made them an effective approach for initial system description and design. Companies actively using decision tables observed a decrease in time required to formulate software solutions (Pollack, Hicks, and Harrison 1971; Hurley 1983).

Able to effectively convey the logic and instructions required of a computer program, their use was naturally extended to the implementation of decision

logic directly in software (Kirk 1965). Processors for converting decision tables to source code, as well as dedicated decision table languages, were swiftly developed by various organizations (Pollack, Hicks, and Harrison 1971; Hedayah 1974).

Decision tables remain a prominent structure in both software documentation as well as implementation in computer programs. However, software systems have experienced an exponential growth in both complexity and size since the inception of decision tables. In these situations, decision tables are no longer the best suited notation for representing exceptionally complex formulae. A real-world industrial example of this problem is demonstrated in Table 1.1.

In examining Table 1.1, it is no surprise that the maintenance of similar tables is prohibitively difficult and requires extra time on the part of the developer. Moreover, due to the excessive size of decision tables, it is all the more cumbersome to check crucial properties such as completeness and disjointness. The introduction of human error when performing changes is likely a event. Further exploration of decision tables and their inherent issues is presented in Section 2.3.1. It is evident that an alternative tabular formalism is needed for cases where decision tables are not scalable.

Overcoming these obstacles, *tabular expressions* (Jin and Parnas 2010) have presented a viable alternative for large and complex formulae. They are of particular use in formalizing long mathematical expressions describing system functionality, and remain humanly readably as they scale up. With this respect, as well as several others, tabular expressions have been shown to be beneficial in numerous software engineering domains. The semantics and advantages of tabular expressions are further expounded in Section 2.3.2.

Conditions	Decisions																	
ePowerSource == NotConnected	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F
ePowerSource == Connected_NotOpr	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F
ePowerSource == Connected_Opr	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T
eComponent == NotPluggedIn	T	T	T	F	F	F	F	F	F	T	T	T	F	F	F	F	F	F
eComponent == PluggedIn_NotOpr	F	F	F	T	T	T	F	F	F	F	F	F	T	T	T	F	F	F
eComponent == PluggedIn	F	F	F	F	F	F	T	T	T	F	F	F	F	F	F	T	T	T
bSysEnabled	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
bPeriodicOpr	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
bCompartmentOpen	F	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T	T
Actions:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Conditions	Decisions																	
ePowerSource == NotConnected	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F
ePowerSource == Connected_NotOpr	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F
ePowerSource == Connected_Opr	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T
eComponent == NotPluggedIn	T	T	T	F	F	F	F	F	F	T	T	T	F	F	F	F	F	F
eComponent == PluggedIn_NotOpr	F	F	F	T	T	T	F	F	F	F	F	F	T	T	T	F	F	F
eComponent == PluggedIn	F	F	F	F	F	F	T	T	T	F	F	F	F	F	F	T	T	T
bSysEnabled	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
bPeriodicOpr	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
bCompartmentOpen	F	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T	T
Actions:	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36

Conditions	Decisions																	
ePowerSource == NotConnected	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F
ePowerSource == Connected_NotOpr	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F
ePowerSource == Connected_Opr	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T
eComponent == NotPluggedIn	T	T	T	F	F	F	F	F	F	T	T	T	F	F	F	F	F	F
eComponent == PluggedIn_NotOpr	F	F	F	T	T	T	F	F	F	F	F	F	T	T	T	F	F	F
eComponent == PluggedIn	F	F	F	F	F	F	T	T	T	F	F	F	F	F	F	T	T	T
bSysEnabled	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T
bPeriodicOpr	F	F	F	F	F	F	F	F	F	T	T	T	T	T	F	F	F	F
bCompartmentOpen	F	F	F	F	F	F	F	F	F	F	F	F	F	F	T	T	T	T
Actions:	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54

Conditions	Decisions																		
ePowerSource == NotConnected	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	-
ePowerSource == Connected_NotOpr	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	-
ePowerSource == Connected_Opr	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	F	F	T	-
eComponent == NotPluggedIn	T	T	T	F	F	F	F	F	F	T	T	T	F	F	F	F	F	F	-
eComponent == PluggedIn_NotOpr	F	F	F	T	T	T	F	F	F	F	F	F	T	T	T	F	F	F	-
eComponent == PluggedIn	F	F	F	F	F	F	T	T	T	F	F	F	F	F	F	T	T	T	-
bSysEnabled	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	-
bPeriodicOpr	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	-
bCompartmentOpen	F	F	F	F	F	F	F	F	F	T	T	T	T	T	T	T	T	T	-
Actions:	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73

#	Action
1	$Action_{1}$
2	$Action_{2}$
3	$Action_{3}$
4	$Action_{4}$
5	$Action_{5}$
6	$Action_{6}$
7	$Action_{7}$
8	$Action_{8}$
9	$Action_{9}$
10	$Action_{10}$
11	$Action_{11}$
12	$Action_{12}$
13	$Action_{13}$
14	$Action_{14}$
15	$Action_{15}$
16	$Action_{16}$
17	$Action_{17}$
18	$Action_{18}$
19	$Action_{19}$
20	$Action_{20}$
21	$Action_{21}$
22	$Action_{22}$
23	$Action_{23}$
24	$Action_{24}$
25	$Action_{25}$
⋮	⋮
72	$Action_{72}$
73	$Action_{73}$

Table 1.1: Example of a large table found n automotive

The next evolution in tabular structures, the transition from decision tables to tabular expressions is a beneficial and needed process.

1.1.2 Automotive Domain

The automotive domain has evolved to a point where road vehicles rely heavily on software. Currently, embedded Electrical Control Units in automotive systems run software controlling everything from the engine to the ignition systems. Next generation drive-by-wire automotive systems will continue to introduce architectures relying solely on electronic systems, while the dependence on software will further grow by orders of magnitude for the Hybrid Electric Vehicles (HEVs) and Battery Electric Vehicles (BEVs). With this continual advent of newer and more sophisticated vehicle functions, the number of Electrical Control Unit (ECU) present in vehicles is constantly growing. As a result, modern automobiles contain numerous software systems, potentially comprising tens of millions of lines of code (Charette 2009). Consequently, today's vehicular control software is amongst the largest and most complex software systems in existence today.

The automotive sector presents its own unique and formidable challenges that must be overcome during the software development process. Most significant is the safety-critical nature of vehicles. Humans rely on passenger vehicles to perform safely and reliably, while also providing a high level of comfort. In turn, passenger vehicles are dependent on software components implementing features which deal directly with safety-critical functions. Ensuring human and environmental safety is an integral concern in the development and deployment processes of any vehicle design. To standardize safety-critical software develop-

ment and its practices in a systematic and regulated manner, the International Standards Association (ISO) 26262 standard was created with the objective of mitigating potential risks, increasing confidence in vehicle performance, and ultimately elevating the quality of software in vehicles.

1.1.2.1 ISO 26262

Passenger safety is a crucial aspect of vehicular design. Modern vehicles must be highly dependable and safe for both those who operate (i.e., drivers/passengers) as well as non-operators (e.g., pedestrians). Therefore, an international functional safety standard defined by the International Standards Association, ISO 26262 entitled, "Road Vehicles – Functional Safety" was developed to specifically address the methods and practices of safety-related electronic, electrical, and software components in series production passenger vehicles.

ISO 26262 applies to software that implements safety-related features. For these features, potential hazards, severity, and controllability, components are assessed, and these components are classified based on their level of criticality in Automotive Safety Integrity Levels (ASILs). A component is identified as level, A, B, C, or D, where D demands the highest intensity of rigour during development in terms of application of ISO 26262 requirements. Given some ASIL level, ISO 26262 prescribes requirements which must be accomplished in order to achieve compliance with the standard. Specifically, Part 6 of ISO 26262 focuses on development at the software level, providing guidance to avoid risks by specifying requirements and processes throughout the automotive software development lifecycle.

The culmination of several years' work, ISO 26262 represents the state-of-the-art when it comes to system and software safety in the automotive

sphere. Most automotive OEMs and suppliers are currently making the move to adopt this standard, and have begun to adapt their processes in a move towards attaining ISO 26262 compliance. In the future, adherence will become expected, and automotive companies are currently exploring the necessary steps required in order to migrate to this standard. To claim compliance with this standard, ISO 26262 defines requirements and methods which must be met, where applicable, while recommendations are strongly suggested but not necessary. Requirements and methods particularly relevant to the scope of this work are delineated in the following sections. They provide further motivation for the refactoring of complex and large vehicular designs, such that compliance may be achieved. These requirements serve as a good guide for making the systems that depend on this software, safer and more reliable.

Software Requirements Recommendations Software requirements are integral for the overall compliance of ISO 26262. Requirements are used throughout the various phases of product development. They are required prerequisites for the initiation of product development at the software level, ensuring that further objectives and subphases comply with the functional safety requirements. Specifically, they go on to serve as the basis for system design, system integration and testing, safety validation, functional safety requirement assessment, and many others (*ISO 26262-6:2011* 2011). Therefore, the existence of requirements and design specifications are of the utmost importance.

Software Design and Implementation Recommendations The increasing complexity of computerized vehicle systems is one of the motivating factors

behind the development of ISO 26262. As a result, the means by which software is designed and implemented is addressed.

General guidelines for modelling and programming languages are addressed in Clause 5 of *ISO 26262-6:2011* (2011). Entitled "Initiation of product development at the software level", this chapter presents criteria for selecting quality software tools and languages for software development. Requirement 5.4.6 lists criteria for assessing suitable modelling languages. These guidelines mandate the evaluation of languages in terms of ambiguity, specifically, with the intention of avoiding ambiguous syntax and semantics. If this criteria is not met, guidelines must be in place to address this deficiency and comply with this requirement.

Additionally, Requirement 5.4.7 Method 1a stresses that accompanying coding guidelines for languages, modelling and programming alike, must address methods of enforcing low complexity of design. Ensuring the correctness of implemented designs is integral to safety-critical systems, and thus this requirement is a highly recommended item for all ASILs. Method 1b again deals with the topic of ambiguity in language subsets. If a language construct is identified as being ambiguous in nature, it is to be excluded from use within the design and implementation. A construct is flagged as being ambiguous if its syntax and semantics are inherently unclear, potentially leaving it to the interpretation of the developer. Additionally, its susceptibility to inconsistent and divergent interpretation by developers, testers, and inspectors also justifies its exclusion from use. This recommendation is prescribed for all ASILs.

Much is said describing an ideal software language. However, the fact remains that Matlab Simulink is the widely accepted environment and notation in industry when it comes to embedded software. Due to its richness in fea-

tures and tools, this is unlikely to change. As a result, allocating efforts to further cultivate its abilities of software specification, will aid in meeting the requirements of ISO 26262, specifically mitigating complexity and ambiguity. Therefore, we seek to apply these guidelines to Matlab Simulink designs by being cognizant of constructs which are ambiguous, as well as the need to minimize complexity.

Clause 8 concentrates on "Software unit design and implementation". Here, requirements concerning model and code specification are described. In the context of MBD, the model is the primary artifact specifying the software. As a result, the prescribed properties apply to the model and not its generated code. Specifically, Requirement 8.4.4 outlines several design principles models should manifest. Included are: simplicity; readability and comprehensibility; modifiability; and testability (*ISO 26262-6:2011* 2011). These software qualities are largely related, and are properties of any good software specification which seeks to mitigate software errors, and therefore risk. Simplicity and conciseness lends itself to ease of readability. As a result, models are easy to understand by developers, and thus all the more modifiable. Testing is elaborated on in the next section.

In summation, minimizing complexity of code, avoiding ambiguous language constructs, and the integration of simplicity, readability, comprehensibility, and modifiability as implementation design principles is demanded by ISO 26262 over several requirements. In general, these are principles that are beneficial for all software, however large safety-related software such as the embedded software found throughout vehicles especially needs to be designed with these requirements as objectives.

Software Testing Recommendations For safety-critical systems like vehicles, unintended functionality that deviates from the safety requirements is not acceptable. Testing can help to reveal these cases. ISO 26262 imparts requirements for the testing and verification stages of the software lifecycle, which are applicable to model-in-the-loop environments (discussed in Section 2.4).

Structural testing in particular is required to measure the degree of code coverage. In Requirement 9.4.5 of ISO 26262, structural metrics to be maximized are given as statement, branch, and MC/DC. The use of these metrics is recommended for ASILs A,B, and C, while MC/DC and branch are highly recommended for ASIL D specifically. The extent to which these metrics are exercised is left up to the software tools which are used.

Furthermore, as touched upon in the previous section, Requirement 8.4.4 outlines design principles to be achieved. The testability of software is amongst these principles, and must be considered throughout the software development process.

Calibration Recommendations Different automobile product lines require divergent software implementations to accommodate their variability. Departing from the traditional definition, in the automotive industry, calibration also refers to software which is instrumented with data after the build has taken place. Calibration enables a single software infrastructure to be used across product lines, while still permitting customization of parameters for specific products. In order to achieve this, models must be designed such that the core software structure remains constant, while other components are dynamic, and permuted between product implementations. Figure 1.1 shows how this is generally accomplished.

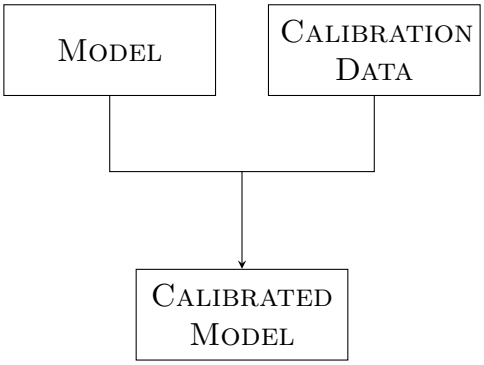

Figure 1.1: Software calibration process

ISO 26262 concerns itself with making controlled changes to software, while still facilitating the its calibration. Therefore, calibration is a property which must be considered during refactoring efforts, such that it is still supported.

1.1.2.2 Collaboration with an Automotive Partner

The undertaking of this thesis has been done under the scope of a multidisciplinary project, in collaboration with a prominent automotive OEM industrial partner. The goal of the project is to improve software development practices for innovative and leading edge vehicles, while enabling transition to industry's emerging standards. The automotive industry, in its increasing reliance on software, has experienced all of the aforementioned issues discussed in Section 1.1.1. Large and complex software systems give rise to difficulties in understandability, modifiability, testability and maintainability. One of the primary tasks of our collaboration with an automotive OEM is to improve vehicle designs in Simulink/Stateflow such that they better facilitate these properties, as well as enable the calibration of software. Additionally, the recommendations given by ISO 26262 fall in line with the priorities our automotive partner has outlined for their software initiatives. The include:

- **Complexity** Table 1.1 is a design taken from actual vehicle control software. It is overly complex to understand, and requires refactoring in order to reduce its size and complexity. The work within this thesis is demonstrated on industrial examples such as these. They are the basis for Chapter 5. Complexity of design is an issue which affects other aspects of a system.

- **Testability** A substantial amount of effort is allocated to testing. Maximizing coverage while also minimizing time and effort in creating and simulating tests is hindered by the large and complex nature of automotive software designs. Reducing the efforts in achieving good coverage and test cases is a priority.

- **Traceability to Requirements** Designs which are maintained and evolved over a long period of time are often so altered that they no longer correspond to existing software requirements. Complexity of the design exacerbates this problem by obfuscating requirement visibility even further. Simplifying and restructuring designs such that requirements are more evident is a beneficial step in refactoring.

- **Calibration** Our OEM partner does not wish to develop and maintain numerous software variants for their various vehicles, but rather employ the use of calibration data to outfit the same software with different behaviour according to the vehicle variant. This calibration data consists of some data values that change between vehicle implementations, and rather than being implemented directly in a Simulink model as a block, they are instead stored in a separate calibration file for ease of modification. Calibration data is loaded into the model according to the

vehicle design at hand, and allows for the software structure to be the same across all vehicle variants, while also facilitating customization of software behaviour in a manner that is consistent yet panoptic. Where necessary, refactored designs should facilitate ease of modification across multiple product lines. Therefore, during any refactoring of automotive designs, this property must also be taken under consideration.

1.2 Contributions

This thesis contributes to the area of formal methods and software engineering. Using basic computer science and mathematical principles, we strive to build a methods for rigorous, dependable software, which serve as an effective means of refactoring tabular designs in both industry and academia. With a focus on facilitating testing, increasing understandability, and leading to conformity to ISO 26262, the key contributions of the thesis are are follows:

- A new proposed software requirements reverse-engineering method to make use of formal property-proving in lieu of inexhaustive testing efforts.
- The creation of a methodology which transforms and simplifies conventional decision tables. Within this methodology, five simplification strategies for tabular expressions have been defined such that tabular expressions themselves can be simplified in terms of logic, resulting in smaller tables. Furthermore, this methodology permits the replication of the techniques by other researches and practitioners, both in industry and academia.

- Real industrial case studies on current vehicle production code, on which the aforementioned methodologies were applied and investigated. These case studies give testament to the practicality of the methodologies created on industrial-sized applications which are in use today, as well as demonstrate the effect of the methodology in reducing complexity and size of tables, while increasing readability and testing. The refactored designs created using the methodology have been incorporated by our automotive industry partner into production vehicular code and the methodology is being added to the company's development processes. This successful technology transfer clearly demonstrates the practicality and industrial relevance of the work.

- Another argument for the use of tabular expressions as a means for specifying software requirements with a detailed investigation as to the differences between decision tables and tabular expressions.

1.3 Outline

This thesis is structured as follows. The following section offers background information of tabular constructs and gives an analysis of their differences, with emphasis on Simulink/Stateflow implementations. It goes on to provide details of the metrics used for the analysis of the proposed methodology, as well as previous related work. Chapter 3 describes and demonstrates the application of a requirements reverse-engineering methodology. Subsequently, Chapter 4 is devoted to the introduction of a new table transformation methodology. Concrete applications on real-world automotive designs are provided in Chapter 5, along with an analysis of its impact with respect to the metrics described

earlier. The final section contains conclusions and directions for future work.

Chapter 2

Preliminaries

In this chapter, we lay the foundations for the work presented in this thesis. Here, concepts required for the comprehension of the remaining chapters are explained, while further elaborating on several ideas presented as motivation in Section 1.1.

2.1 Notation

The notation and terminology used throughout this thesis corresponds to the notation and terminology used by Matlab Simulink. We adhere to it in the interest of being consistent with respect to the accepted conventions within the scope of the MBD domain. At times the notation may not the considered the most widely accepted means of expressing concepts. For example, the assignment operator is represented as =, although := is generally favoured. Hence, == is used as the relational equality operator. Conversely, Matlab uses ∼= to denote inequality.

Another notable divergence from the norm is in the terminology used by

Simulink/Stateflow. Contrary to what the name suggests, Stateflow truth tables are not truth tables in the classical sense. In reality, Stateflow truth tables are decision tables. Thus, when we refer to Stateflow truth tables, what is actually meant is a decision table. In this work these two terms are used interchangeably, however we recognize that outside the scope of Simulink, this is not the case.

The naming convention for variables, enumerators, etc. are adopted from our industrial partner. Variable names are prefixed with their data type before the signal name, in order to help identify its role and behaviour. Enumeration types are delineated as $eName$, Boolean as $bName$, constants (e.g. enumeration tokens) as $cName$, and functions as $fName$.

2.2 Reverse-Engineering Software Requirements Methodology

Requirements specification is a critical component of the development of software systems. The majority of software failures can be traced to poor software requirements (Leveson 2004). Although there is a wide consensus and acknowledgment of the importance of software requirements, rigorous requirements engineering is an area that has largely been neglected by software developers in general. The automotive industry is no exception.

It is often the case that legacy models suffer from a lack of accompanying requirements, and even if such documentation does exist, it is rarely up-to-date and adequately maintained. As a result of this absence or neglect, the understanding of design decisions and the underlying rationale of the system are

obscured, leading to difficulties in testing, maintenance, and evolution efforts. Possessing well-specified and maintained requirements is integral to refactoring and improving software. Prior to making modifications, developers must have a notion of what the requirements of the system are, in order to ensure that any software alterations will continue to satisfy the requirements at the completion of the refactoring process. This is especially important for functional safety requirements, which specify safety-related software attributes. Consequently, reverse-engineering a requirements specification typically represents a necessary step in the refactoring process.

Retroactively extracting requirements from existing, already developed software generally proves to be an arduous process. The seemingly most intuitive and straightforward method of doing so is through the consultation of developers, that is, those experts with the greatest depth of knowledge of the system. This method, however, is time consuming on the part of developers, does not guarantee exhaustive discovery (completeness) of requirements, nor does it provide any assurance that the discovered requirements are indeed correct (accurate), i.e. implemented in the current system. Therefore, the application of an automated methodology for requirement extraction from software specification would aid in these efforts. Such a methodology for Simulink/Stateflow models was proposed by Ackermann et al. (2010). Experimental results were provided, indicating high completeness and validity of inferred requirements, as well as proving the fruitfulness of this methodology on Simulink/Stateflow automotive designs.

The primary focus of this methodology is the extraction of *invariants*. A popular concept in software verification, invariants represent statements about a system that hold true on all the possible executions of the system.

With respect to the methodology of Ackermann et al. (2010), invariants are inferred via association rule mining on a set of test cases. This produces rules representing relationships, (i.e. invariants), between attributes of the system (i.e. itemsets), which hold for the system and provide insight into system behaviour. The Apriori algorithm generates these rules to satisfy two quality measurements, minimum support and confidence. Support of an itemset is the proportion of entries in the test suite which contain that specific set of items/attributes. For a rule X $\Rightarrow$ Y where X and Y are non-empty itemsets,

$$support(X \Rightarrow Y) = support(X \cup Y)$$

Confidence, sometimes called the accuracy ratio, is defined at the proportion of the instances covered by the premise that are also covered by the consequent.

$$confidence(X \Rightarrow Y) = \frac{support(X \cup Y)}{support(Y)}$$

That is, the confidence is the ratio of the number of instances containing both X and Y to the number of those containing Y. We use these two metrics to generate strong association rules which meet or exceed the minimum thresholds defining quality measurements.

2.3 Tables in Software Development

Tabular formats provide a means of describing complex information, while presenting it in a concise, well-organized manner. This also holds true for the representation of logical decision-making behaviour in embedded software systems. The de facto embedded software development platform, Matlab and

its Simulink and Stateflow diagramming notations, provide two formalisms for implementing complex decision logic within a model:

1. **Stateflow chart.** Represents sequential decision logic in a finite state machine, using states and transitions with corresponding conditions. Used particularly for event-driven systems, they are also commonly used as a more visual means of specifying decision logic, similar to that of flow charts.

2. **Stateflow truth table.** Facilitates precisely stated logic, with a greater emphasis on the relationships between conditions, and in general simpler to design. Out of the numerous structures and blocks provided by Simulink and Stateflow, they are the only component which facilitates static analysis of complexness (Aberg 2004).

Both these formalisms can be used to specify control flow, procedural, and combinational logic in systems. Although regarded as particularly useful for stateless logic, truth tables are increasingly being adopted as an alternative to Stateflow charts. A frequently encountered problem that charts suffer from is that they do not scale well in terms of size and complexity. As they are modified, additional decision-making logic is added in the form of more states and transitions, while the conditions also become long and elaborate. This increase in complexity quickly diminishes a chart's understandability and maintainability, while obscuring its function. Therefore, developers are turning to Stateflow truth tables as an alternative implementation, one which is capable of capturing the same logic, but in a more manageable and clear format. The semantics and properties of Stateflow truth tables are presented in detail in Section 2.3.1. Whereas charts implement logic by way of finite state machines, truth tables

are able to represent the equivalent behaviour in a tabular format, in most situations. Employing truth tables in lieu of the other more complex diagram constructs within Simulink simplifies designs (Aberg 2004). This shift towards a heavier reliance on tabular designs is also observed in industry. Our automotive partner is actively transforming Stateflow chart designs to Stateflow truth tables.

Nevertheless, it has long been acknowledged that in complex situations, truth tables also have the possibility to become extremely large (Pooch 1974). With this increase in size, there is a further need to document and implement tables with the intention of minimizing design complexity and increasing understandably for developers. Decision tables are often used in documentation, however they fail to convincingly convey requirements when they describe tens or hundreds of cases. They quickly become difficult to parse and understand by developers. Moreover, checking for problematic situations such as redundancies and contradictions becomes a strenuous task, one which is not always supported by tools.

Tabular expressions have emerged as a superior tabular method for representing complex mathematical formula, including decision logic. They have proven themselves in documenting both software requirements and software design, and have been successfully used in the software development process in industry (e.g., Naval Research Laboratory (Heninger 1980), OPG (Heitmeyer et al. 1998; Archinoff et al. 1990; Wassyng, Lawford, and Maibaum 2011), just to name a few). Tabular expressions are particularly suitable for specifying embedded, real-time systems. They are presented in Section 2.3.2. The following sections provide an an investigation into both formalisms.

Conditions	Condition Entries
Actions	Action Entries

(a) Classical Decision Table Notation

Conditions	Condition Entries
	Action Entries
Actions	

(b) Stateflow Truth Table Notation

Table 2.1: Organizational differences between table notations

2.3.1 Decision Tables

Over the course of their history, decision tables have taken on a variety of formats and different semantics (Pooch 1974). Diverging representations have evolved with their own idiosyncrasies. For our purposes, we adhere to the widely used, classical definition, that of Limited Entry Tables (Hughes, Shank, and Stein 1968). These tables are considered to be the most precise convention of all the decision table variants, and because of their binary logic patterns, they are well suited for software implementations. It is possible to convert decision table variants to this notation, and there exist tools which facilitate this process (Fu 1999). This is also the notation used by Stateflow truth tables, however there do exist minor differences in the organization of that data, as illustrated in Table 2.1. These minor variations do not affect the logic, but rather the visual organization. As a result, we adapt/augment some of the terminology to better correlate with the Stateflow truth table representation. Furthermore, we will use the terms decision table and truth table interchangeably, as they are essentially the same within this context. Stateflow truth tables are in fact decision tables.

Decision tables are comprised of two major sections: a condition section that decides which actions are to be taken as a result of certain conditions

being met, and secondly, an action section which defines the operations that can be performed as a consequence of these decisions being satisfied. The basic structure of a decision table is shown in Table 2.2, where these two sections appear as separate tables in and of themselves, together forming a decision table.

		Decisions			
#	Conditions	D_1	D_2	...	D_{2^n}
1	$Condition_1$	T	T	...	F
2	$Condition_2$	T	T	...	F
⋮	⋮	⋮	⋮	⋮	⋮
$n-1$	$Condition_{n-1}$	T	T	...	F
n	$Condition_n$	T	F	...	F
	Actions	1	2	...	m

#	Actions
1	$Action_1$
2	$Action_2$
⋮	⋮
$m-1$	$Action_{m-1}$
m	$Action_m$

Table 2.2: Generic decision table

Decision tables consist of three core elements used for the specification and evaluation of the actual design-making behaviour: conditions, actions, and decision rules (simply called decisions). Each row of the condition table specifies a condition, and includes entries indicating the values it is evaluated to. The possible outcomes for a condition are limited to true, false or don't care, represented as T, F, or - respectively. A don't care represents all possible

values of the condition.

In the broadest definition, actions describe some operation which takes place as a consequence of combinations of conditions being satisfied. For Stateflow truth tables, they are specified using statements, function calls, and even complicated control logic (i.e. `for` loops). Most commonly, decision tables are used to implement the logic for the computation of some value, and thus actions are typically in the form of a C assignment statement, i.e. $eOutput = value_1$; representing an output to be returned.

Decision rules are represented in the vertical columns, and are a combination of conditions and actions. They define relationships between the conditions, and are interpreted as being "and"-ed together. When the conjunction of the conditions in a column is satisfied, the corresponding action at the bottom of the table is executed and any remaining decisions are not evaluated. The decision rules for a generic function, given Table 2.2, are as follows:

$$
\begin{aligned}
Decision_1 &= Condition_1 \wedge Condition_2 \wedge \ldots \wedge Condition_n \\
Decision_2 &= Condition_1 \wedge Condition_2 \wedge \ldots \wedge \neg Condition_n \\
&\vdots \\
Decision_{2^n-1} &= \neg Condition_1 \wedge \neg Condition_2 \wedge \ldots \wedge Condition_n \\
Decision_{2^n} &= \neg Condition_1 \wedge \neg Condition_2 \wedge \ldots \wedge \neg Condition_n
\end{aligned}
$$

If the decision table employs a sequential left-to-right order of evaluation of decisions, as is generally the case, the table can further be interpreted as an

`if-then-else` statement. For example, Table 2.2 can be interpreted as:

if $Decision_1$ **then**
    $Action_1$
else if $Decision_2$ **then**
    $Action_2$
    ⋮
else if $Decision_{2^n-1}$ **then**
    $Action_{m-1}$
else
    $Action_m$
end if

Additionally, when implementing these tables in Stateflow, one is able to examine the resulting code generated by Matlab. Upon simulation, Stateflow truth tables are converted into Matlab code, which takes on the structure of the above pseudocode.

2.3.2 Tabular Expressions

Like decision tables, tabular expressions strive to express complex logic in a precise, yet concise, manner. Originating out of efforts towards describing large and complex mathematical formulae in a humanly readable, very precise format, tabular expressions are an alternative and effective tabular construct for the documentation and specification of software.

Several types of tabular expressions have been introduced (Jin and Parnas 2010). For the purposes of this paper, horizontal tabular expressions will be the notation under consideration, as shown in Table 2.3. Horizontal tabular expressions, or minor variations thereof, are especially conducive to requirements

specification (Wassyng and Janicki 2003), and have proven to be invaluable in industrial applications (Wassyng and Lawford 2003).

An extensive introduction to the fundamentals of tabular expressions is given by Janicki and Wassyng (2005). Here we introduce their basic semantics. Each row represents a subexpression of the function. If a condition is evaluated to be true, the corresponding Result cell value is the returned output. Results are distinguished with a double-lined border.

	Result
Conditions	*Name*
$Condition_1$	$Result_1$
$Condition_2$	$Result_2$
⋮	⋮
$Condition_n$	$Result_n$

Table 2.3: Generic tabular expression

An invaluable merit of tabular expressions is their properties of disjointness and completeness. A tabular expressions can be visually inspect for disjointness and completeness with ease. For any table to properly define a (total) function, two conditions must be satisfied:

1. **Disjointness** Each distinct pair of conditions, $Condition_i$, $Condition_j$ is disjoint, i.e. $i \neq j \Rightarrow \neg(Condition_i \wedge Condition_j)$

2. **Completeness** The disjunction of all $Condition_i$'s is true, i.e. $(Condition_1 \vee Condition_2 \vee \ldots \vee Condition_n) \Leftrightarrow \top$

In short, disjointness means that no two rows can be simultaneously true in a single table, and for completeness there must always be one row which is

true for the given inputs. Therefore, completeness asks that all the possible inputs are considered, while disjointness ensures determinism. If both these properties are satisfied, a tabular expressions is considered to be *proper* (Jin and Parnas 2010). In practice, the facilitation of checking for completeness on the input domain of the system is a beneficial property embodied by tabular expressions, and is invaluable for safety-critical systems (Wassyng, Lawford, and Maibaum 2011). Preserving these properties raises the overall confidence in correct system performance for all conditions, and is also beneficial in the detection of gaps for the inputs considered. In terms of coverage and testing this is particularly important, and tabular expressions have been show to be useful for testing and validation (Wassyng and Lawford 2003; Wassyng and Janicki 2003).

Furthermore, conditions are typically complex, compound expressions with compound subconditions. An extension of these tables which better emphasizes the logical relationships of conditions for each subexpression is shown in Table 2.4. Conditions which do not appear in a certain row indicate that they are a don't care condition for that subexpression.

		Result
Conditions		*Name*
$Condition_{1a}$	$Condition_{2a}$	$Result_{1a}$
	$Condition_{2b}$	$Result_{1b}$
$Condition_{1b}$		$Result_2$
⋮		⋮
$Condition_{1n}$		$Result_n$

Table 2.4: Generic horizontal tabular expression

Informal semantics for interpreting Table 2.3 have been described in the

following pseudocode:

if $Condition_1 \wedge Condition_{2a}$ **then**
    $Result = Result_{1a}$
else if $Condition_1 \wedge Condition_{2b}$ **then**
    $Result = Result_{1b}$
else if $Condition_2$ **then**
    $Result = Result_2$
    $\vdots$
else if $Condition_n$ **then**
    $Result = Result_n$
end if

However, it is important to note that in fact this is not necessarily true, as tabular expressions do not have a prescribed order of interpretation, lending to their ease of comprehension due to the explicitness of semantics. In their implementation in software tools however, this is indeed the means by which they are internally interpreted, as is the case with SRI's Prototype Verification System (PVS) (Owre, Rushby, and Shankar 1992).

In general, the success of tabular expressions as a practical formalism, as is the case for formal methods in general, hinges on appropriate automated tool support. The *Tabular Expression Toolbox (TET)*, a Simulink toolbox, was developed at the McMaster Centre for Software Certification, to integrate tabular expressions into model-based development with Simulink (Eles and Lawford 2011). The toolbox:

- Provides Simulink blocks for creating and editing tabular expressions
- Checks tables for disjointness and completeness (using the CVC3 SMT solver and the PVS theorem prover)
- Can be translated into m-functions and then used for code generation and simulation.

Therefore, tabular expressions serve as a practical decision logic structure for both software documentation and implementation purposes.

2.3.3 Comparison

There are several key problems that have been widely acknowledged as existing with decision tables. They are explained in detail in this section.

Disjointness When decision tables were first introduced, the accepted method of interpretation was that decision rules were independent (i.e. disjoint) and could be tested in any order (Pooch 1974). Eventually, the else rule was introduced as a means of capturing situations which were deemed impossible or errors. This rule was located at the end of the decision rule section, and was considered as the only rule which was allowed to disobey the disjointness property. Following suit, Harrison (1971) introduced a variant of decision tables in which left-to-right execution semantics were employed, departing completely from the notion of disjointness.

An example of non-disjointness demonstrated in the context of a valid Stateflow truth table is depicted in Table 2.5. Here, overlaps are present between the following pairs of decision rules: (D_1, D_2) and (D_2, D_3). Additionally, the else case D_4 overlaps with all of the previous decisions. In examining

		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	$Action_4$

Table 2.5: Non-disjoint and ambiguous decision table

the first pair (D_1, D_2), we see that both cover the case when all the conditions are true, thus they partially overlap for the case when $Condition_1 = \text{T} \wedge Condition_2 = \text{T} \wedge Condition_3 = \text{T}$. Without knowledge of the left-to-right semantic, these cases produce non-deterministic behaviour in the logic, as both would be true given all true inputs. As a direct consequence of not enforcing disjointness on the decisions, *ambiguity* arises. Moreover, both D_1 and D_2 prescribe different actions. From a functional specification perspective, this is a *contradiction*, that is, the prescription of different actions for identical inputs. In this situation, unwanted behaviour can be unknowingly introduced. *Redundancy* is also a product of non-deterministic behaviour, however in this case the same actions are prescribed. Although this situation does not introduce unwanted behaviour, later software changes could easily introduce a contradiction.

The left-to-right semantic directly contributes to the issue of ambiguity. It is a consequence of this convention that decisions can be implemented to rely on preceding decisions evaluation, hence meaning that they are not imple-

mented disjointly. It is often the case that the left-to-right semantic is used to implicitly filter out cases prior to evaluating some decision later on. Permitting these overlaps in decisions, i.e. multiple decisions satisfied for the same input, exacerbates the problem of ambiguity and can potentially introduce contradictions in the logic, which are not readily apparent, nor do diagnostic tools flag this as a source of error.

Simulink provides diagnostic tools for identifying certain types of table defects, specifically, the *underspecification* and *overspecification* of decisions. Underspecification of a decision table occurs when there exist input cases which are not considered in the decisions of a table. In this case, the table is lacking completeness as it does not account for all possible inputs. Conversely, overspecification occurs when there are too many decisions defined in the table, such that some are never evaluated. Simulink detects these errors by ensuring that each decision is executed at some point during simulation. Overspecification implies that there must be some overlap between decisions, resulting in decisions which are not deterministic. Simulink however does not check for exclusivity between decisions, that is, disjointness. It fails to detect the case where there is ambiguity between decisions, but both decisions are still executed for different inputs. That is, it does not identify partial overlaps of the decisions. Thus, it is possible to introduce contradictory actions for overlapping decisions without detection from Simulink, or the developer's knowledge.

Correct interpretation and understanding of the logic expressed by decision tables is contingent on the use of left-to-right semantics. Although these semantics are not ambiguous per se, their perceived interpretation often is. Logic relying on this additional implicit semantic for correct evaluation is often ambiguously interpreted by reviewers and developers, as they often do not

account for the left-to-right evaluation order. In practice these tables are used to implement logic on a case-by-case basis, however, errors can arise during maintenance when new decision rules need to be inserted, deleted or rearranged due to the fact that they rely on previous decisions. Performing any of the aforementioned actions will alter the logic of the table in ways that are not readily apparent, as a result of the underlying relationships between decision rules. A simple switch of two columns can completely alter the logic of the decision table, and behaviour of the system. This is a serious concern for safety-critical systems.

Therefore, not ensuring the presence of the disjointness property can lead to a contradiction or redundancy, and because decision tables do not facilitate disjointness, they are not well-suited for applications which require clear, unambiguous specifications.

Understandability and Readability Over the course of their history, decision tables have taken on a variety of formats and different semantics. Formal and universal semantics were defined by the Standards Association (*Decision Tables* 1970). Nevertheless, decision tables diverge widely in their representation, semantics and algorithms. It is no surprise then, that Simulink also uses its own implementation, semantics and definitions of what constitutes a correct implementation of a decision table. Therefore, decision table semantics are often not well understood upon immediate inspection. In comparison, tabular expressions have consistent, and well-defined semantics. Their semantics are intuitive to the reader and are simple to read.

In terms of visual inspection of disjointness and completeness, decision tables may be less explicit than tabular expressions, and also less readable.

For tabular expressions, one must simply scan the table from left to right. Decision tables, however, require constant referral to the condition section in order to understand what the decision rules signify. This plays a significant role in the ease and correctness of table maintenance and modification.

Further exacerbating the problem of decision table interpretations and readability is that their underlying semantics are implicit. This can lead to inconsistent understanding of the knowledge being defined, and this is especially true from the perspective of an outside reviewer with no prior knowledge of a decision table's use. As a result, an ambiguous understanding of a table can occur in the order of evaluation of decision cases. When first introduced, decision table semantics were such that decisions could be tested in any order. However, with Stateflow truth tables they are implemented with a left-to-right order of evaluation. Although the use of the left-to-right semantics is common, it is not a mandatory property of decision tables in general. Moreover, it is a property that is not readily apparent simply through visual examination, nor is it intuitive in many cases. We found this especially true for larger tables with complex decisions. It is not always the natural response of the reader when quickly searching a complex table for a decision that corresponds to a specific case.

Our OEM partner's developers expressed that tabular expressions provide superior readability due to the fact that conditions are explicit in this formalism. That is, decision tables use T/F/- constants to describe the value of a condition. This requires the reader to reference the condition section of the table in order to ascertain the meaning of these values. In tabular expressions, conditions are directly placed in the decision rules, making it much simpler to parse. This ease of inspection also lends itself well to the checking of com-

pleteness and disjointness. These properties are easily visually checked with tabular expressions. Moreover, tabular expressions have been shown to be a particularly well-suited format for requirements documentation. In general, tabular expressions are indeed a superior format in terms of readability.

Completeness and the Else Case As detailed in Section 2.3.2, completeness ensures that all logically possible states/decisions are included. According to Aberg (2004), Stateflow truth tables are the only Stateflow construct which support the analysis of completeness. This is indeed the case for verifying that the decision rules are complete in terms of handling all possible combinations of conditions.

The diagnostics tools provided by Stateflow detect any departure from this property. However, the completeness detection diagnostic is weakened by the fact that Stateflow truth tables also allow for the inclusion of an else case as part of the table. The completeness property can be simply satisfied with an else case, that indiscriminately accommodates the remainder of the unspecified cases, thus forcing completeness. The else decision rule is a convenient but indirect means of enforcing completeness (Pooch 1974). Although an else case presents a good solution for catching errors and impossible situations, it introduces uncertainty as to whether or not one has indeed assessed and explicitly addressed all cases of interest. It can potentially hide errors in true completeness and logic, although the table is syntactically correct.

In comparison, tabular expressions do not have else functionality built into their definition. This promotes avoidance of mindlessly grouping together of all unconsidered decision rules as a single else. When implemented in a Stateflow decision table, else conditions are a sequence of don't cares for the

decision rules. By their nature, these else conditions inherently introduce overlaps with all other conditions present in the decision table, thus making any decision table with an else decision rule not satisfy the disjointness property once transformed into a tabular expression. Else conditions in a tabular expression must be explicitly defined, as one or more negations of conditions. This is not to say that every case, or possible combination of inputs, must be implemented individually in the table, but rather that all are represented in an explicit manner, including the else case.

Non-Boolean Conditions Decision tables boast an adeptness in expressing complex logical relationships between conditions while making relationships between variables more apparent. This is certainly true for Boolean conditions, which only every occupy a single condition entry and are implemented as conjunctions of T/F/- values. This binary convention does lend itself to Boolean conditions well, however, with increasingly complex conditions and logic, this is not sufficient to fully express relationships between conditions.

Enumeration types are prevalent in mode-driven decision systems, where it is common practice to denote each mode with an enumerated value. When implemented in a decision table, enumerated variables are implemented with each value as a separate condition in the condition column. Although they appear and are treated as distinct conditions, there is an inherent mutual exclusivity relationship between the enumerators, i.e. a single variable cannot hold the value of multiple enumerators at any one time. However, due to their implementation as separate, unrelated conditions in decision tables, this is exactly what is implied. In order to implement this inherent relationship, diagonal patterns of T in the decision rules with F otherwise are required. This imple-

mentation is comparable to that of an identity matrix, and is demonstrated in Table 2.6.

		Decisions			
#	Conditions	D_1	D_2	D_3	D_4
1	$eVar == cEnum_1$	T	F	F	-
2	$eVar == cEnum_2$	F	T	F	-
3	$eVar == cEnum_3$	F	F	T	-
	Actions	$Result_1$	$Result_2$	$Result_3$	$Result_4$

Table 2.6: Example of enumeration type representation in decision tables

For tabular expressions, this overly verbose implementation of enumerator checking conditions is not necessary due to the disjointness property which presides over conditions. The same checking logic of Table 2.6 can be more compactly expressed in the tabular expression shown in Table 2.7.

Conditions	Result *Name*
$eVar == cEnum_1$	$Result_1$
$eVar == cEnum_2$	$Result_2$
$eVar == cEnum_3$	$Result_3$
$eVar \sim= cEnum_1 \vee eVar \sim= cEnum_2 \vee eVar \sim= cEnum_3$	$Result_3$

Table 2.7: Example of enumeration type representation in tabular expressions

Under closer inspection, when expressing an enumeration checking decision rule in standard mathematical notation, we see how differently enumerated types are implemented. For example, a decision rule to check if $eVar == cEnum_1$ in a decision table must be in the form,

$$eVar == cEnum_1 \wedge \neg(eVar == cEnum_2) \wedge \ldots \wedge \neg(eVar == cEnum_n)$$

whereas in a tabular expression the equivalent rule is simply,

$$eVar == cEnum_1.$$

Furthermore, because decision tables fail to accommodate for the implicit exclusivity relationship between enumerators, an else decision rule is required to catch cases that are not covered, and potentially another action. In reality, these cases are impossible input combinations due to the nature of enumeration types, however one is forced in include them, because Simulink otherwise flags these "missing" cases as an underspecification error. For tabular expressions, this extra else case is not required, however they are included in order to accommodate for hardware errors.

This same problem exists with other types of conditions, including ranges. An example of a range checking condition in a decision tables is given in Table 2.8, while its equivalent tabular expression implementation is given in Table 2.9. Again, there is a considerable difference in table size and readability.

		Decisions			
#	Conditions	D_1	D_2	D_3	D_4
1	$x < 0$	T	F	F	-
2	$x == 0$	F	T	F	-
3	$x > 0$	F	F	T	-
	Actions	$Result_1$	$Result_2$	$Result_3$	$Result_4$

Table 2.8: Example of range implementation in decision tables

	Result
Conditions	$Name$
$x < 0$	$Result_1$
$x == 0$	$Result_2$
$x > 0$	$Result_3$

Table 2.9: Example of range implementation in tabular expressions

Clearly, tabular expressions offer superior support for non-Boolean conditions, both because they can be implemented in a more intuitive manner, as well as their ability to implement the same logic with a smaller table size.

Scalability Although the appeal of decision tables is that they provide reasoning in a compact form, in practise decision tables capturing complex logic become difficult to manage in terms of comprehensibility, maintainability, and testability as they grow in size. Large real-world applications frequently cause decision tables to become too complex for reviewers to easily and intuitively parse. As a consequence of this complexity, the implemented logic is difficult to understand and maintain, while the underlying requirements are not evident and hence can not be easily traced to requirements. Decision tables do provide a precise notation for complex systems, however, such documentation is of little value if it cannot be easily reviewed and maintained. Creating a large decision table with a considerable number of conditions inevitably becomes a time consuming and difficult task.

The popularity and prevalent use of decision tables is primarily attributed to their ability to represent complicated logic in a concise manner as compared to narratives, flowcharts and other representations. Likewise, in the case of our automotive partner, this was the rationale behind using decision tables

in designs. What was actually encountered were large and overly complex tables which captured logic that in theory were easy to understand, however the number of conditions exacerbated the complexity of its implementation. Large decision tables pose a problem for all involved: developers, testing, and inspectors alike. Considerable effort is required in order to first understand the logic that is captured. Even then, the decision table is a web of conditions that is difficult to unravel.

As previously mentioned, decision tables do not readily implement non-Boolean conditions in an intuitive manner. The result of this is an increase in conditions, and thus table size, in order to accommodate these types of conditions.

2.4 Model-in-the-Loop Metrics

In this section, we review existing software metrics for Simulink models. Model refactoring techniques seek to improve model quality by reducing some model complexity measure or by ensuring certain model properties are satisfied. Means of measuring software are defined as metrics. Testing metrics are used for dynamic verification (i.e. testing, simulation), while software metrics are a static type of verification. Both these types of metrics are beneficial when evaluating the quality of software designs during refactoring processes. Metrics enable an early estimation of the complexity of design, further predicting understandability, testability, and maintainability of software. With a focus on the simplification of designs, refactoring techniques should be applied with guidance from metrics. Our aim is to minimize these metrics on models with Stateflow truth tables specifically.

2.4.1 Test Metrics

Given a model in an open loop, that is, a system design with unconnected inputs and outputs, testing tools seek to generate inputs with which the model is then simulated rigourously. The extent to which the model and its numerous subsystems and blocks are executed is largely contingent on the type of coverage metrics used during test generation. Test generation simulates a model with inputs, while recording the outputs. Both these values are stored in test suites. Each simulation step during this process is a test step. Successive test steps are called test cases. Different coverage metrics exist for various purposes and languages.

Boundary Coverage This coverage metrics simply exercises inputs based on the boundary values of their type.

Condition Coverage A condition is an atomic Boolean expression. To achieve condition coverage, every condition must be evaluated to every one of its possible outcomes, at least once.

Decision Coverage A decision is a Boolean expression composed of several conditions separated by Boolean operators. To achieve decision coverage, every decision must be evaluated to every one of its possible outcomes, at least once. Furthermore, every entry and exit point must be taken, at least once. Decision coverage is also known as Branch coverage, as every branch must be taken to satisfy this metric.

Modified Condition/Decision Coverage MC/DC coverage necessitates Condition and Decision coverage, with the addition of the requirement that

each condition within a decision must be shown to independently incidence the outcome of the decision. In other words, from one test to another, changing the value of a single condition will also change the overall outcome of the decision, while the remaining conditions are held at a fixed value.

For structural testing of high integrity systems, MC/DC was developed in the avionics safety standard *RTCA DO-178B* (1992), specifically for use in safety-critical applications where extensive testing of complex Boolean expressions is required. This metric has also become expected in other safety-critical domains, including nuclear and automotive. Consequently, tools which support the MC/DC metric as a criterion during test generation is recommended to fully exercise the model and achieve maximal coverage.

To achieve MC/DC during testing, the following objectives must be accomplished:

1. Every condition must be evaluated to all possible outcomes (Condition Coverage)
2. Every decision must be evaluated to all possible outcomes (Decision Coverage)
3. Every entry and exit point must be taken (Decision Coverage)
4. Every condition within a decision must be shown to independently influence the outcome of the decision

The motivation for MC/DC was to make precise software with traceability to each test. Although a very thorough criterion for testing, it poses problems and difficulties in that it is time consuming and arduous to satisfy. Nevertheless, MC/DC is the best metric for testing safety-critical systems, particularly those with complex boolean expressions (Kandl and Kirner 2011). Avenues for

achieving MC/DC are of priority to OEMs striving to satisfy safety standards. Many investigate avenues for refactoring code and creating test cases which meet the demands of this metric.

2.4.2 Software Metrics

Metrics for Simulink/Stateflow models are typically general code metrics which have been adapted. Similarity, cyclomatic complexity McCabe 1976,the most widely used software complexity metric, has been adapted in a similar fashion, and is supported in the Simulink/Stateflow environment.

Cyclomatic Complexity Cyclomatic complexity is a popular software metric for static verification, and is the most widely used metric for quantifying software complexity. Software metrics (unlike testing metrics) are methods of measuring some property of the code. Cyclomatic complexity measures the amount of decision logic in a program, or more specifically, the number of linearly independent execution paths through a program (McCabe 1976). It is directly related to the number of decision points within the code. Matlab Simulink had adapted this metric for use in Simulink models. An integer representing the amount of complexity is computed on a model or subsystem using the definition, $cc = \sum_{n=1}^{N}(o_n - 1)$ where N is the number of decision points contained in the model or subsystem, and o_n is the number of outcomes for the n-th decision point.

Furthermore, the cyclomatic complexity metric can be used to impose upper limits on the complexity of software modules that will be permitted in a design. While the original suggested upper bound of cyclomatic complexity of a module was 10 (McCabe 1976), the limit of 15 was also successfully used.

The suggestion is that limits over 10 may be used in cases where a rigorous software engineering process is applied (Watson, McCabe, and Wallace 1996). These strict limits are typically not adhered to in MBD due to the size and complexity of the application domain.

Nevertheless, cyclomatic complexity metric is used to gauge refactoring effectiveness with respect to the simplification of designs implementing decision logic.

2.5 Related Works

Refactoring is a well-established restructuring process for software developed using conventional textual programming languages. However, mechanisms for refactoring Simulink designs is a relatively unexplored area. Simulink itself does not provide refactoring support in any capacity.

Refactoring strategies which are available are focused on model-wide refactoring efforts, and must consider effects on dataflow. Tran, Wilmes, and Dziobek (2013) introduce composite applications of transformation steps for implementing refactoring strategies. Specifically, the refactoring operations employed deal with replacing Goto/From blocks with explicit signals as well as the merging subsystems. At the time of writing, and to the best of our knowledge, other refactoring strategies for Simulink models are not readily available.

However, the scope of this thesis focuses is on small subset of the Simulink language, specifically the Stateflow truth table blocks. General table transformations are most relevant to this work. In terms of tabular expressions specifically, Shen, Zucker, and Parnas (1996) outline several transformations

for going back and forth between inverted and normal tables by changing dimensionality, inverting and normalising. Zucker (1996) goes on to define them mathematically using signatures and Boolean algebra, while discussing the relationship between them. These approaches strive to simplify tables through reorganization, not applying transformations to minimize the actual logic they specify. Likewise, Fu (1999) gave a tool for going between structured decision tables, semi-generalized decision tables, and generalized decision tables, while Shen (1995) implemented algorithms for inverting tables. These transformations do not simplify the logic, but rather strive to express them in an optimal table format.

Rastogi (1998) defines a simplification on normal, inverted, vector, predicate and inverted predicate expression tables. The simplification of the logic is done restricting the domain of a table through constraints on inputs, i.e. specialization. Cases that will never occur during normal operation due to these constraints are removed either at the cell level, or by removing rows or columns. However, this results in a loss of generality as the domain under consideration is reduced from its actual form.

For decision tables, refactoring techniques have been explored. Optimizations in terms of row ordering and execution time were addressed by Vanthienen and Wets (1994), as well as some notions of table contraction by way of column combination. These refactoring approaches were part of the process of formalizing decision tables as expert shells. Similarly, Pollack, Hicks, and Harrison (1971); Hughes, Shank, and Stein (1968) also discuss simplification by way of removing repetitive decisions, or conditions which are not required, i.e. don't care conditions. Although these simplifications were done on decision tables, they correlate to some of the simplifications defined within this thesis.

Similar simplifications on tabular expressions have not been explored as of yet. Furthermore, although it is acknowledged that several equivalent representations of a table are possible, none address this by taking into consideration the requirements of the table. Additionally, the effect that the simplifications have on tables in terms of testing or complexity is not evaluated.

Therefore, although techniques exist which can transform tables such that they are simplified in their representation, none adequately simplify the logic within the tabular expressions, nor are the effects adequately studied.

Chapter 3

Application of a Reverse-Engineering Software Requirements Methodology

Requirements specification is a critical component of the development of software systems. The majority of software failures can be traced to poor software requirements (Leveson 2004). Although there is wide acknowledgment of the importance of software requirements, requirements engineering is an area that has largely been neglected by software developers. The automotive industry is no exception.

It is often the case that legacy models suffer from a lack of accompanying requirements, and even if such documentation does exist, it is rarely up-to-date and adequately maintained. As a result of this absence or neglect, the understanding of design decisions and the underlying rationale of the system are obscured, leading to difficulties in testing, maintenance, and evolution efforts. Possessing well-specified and maintained requirements is integral to refactor-

ing and improving software. Prior to making modifications, developers must have a notion of what the requirements of the system are, in order to ensure that any software alterations will continue to satisfy the requirements at the terminus of the refactoring process. This is especially important for functional safety requirements, which specify safety-related software attributes. Consequently, reverse-engineering requirements specification represents a necessary step in the refactoring process.

Retroactively extracting requirements from existing, already developed software generally proves to be an arduous process. The seemingly most intuitive and straightforward method of doing so is through the consultation of developers, that is, those experts with the greatest depth of knowledge of the system. This method, however, is time consuming on the part of developers, does not guarantee exhaustive discovery (completeness) of requirements, nor does it provide any assurance that the discovered requirements are indeed correct (accurate), i.e. implemented in the current system. Therefore, the application of an automated methodology for requirement extraction from software specification would aid in these efforts. Such a methodology for Simulink/Stateflow models was proposed by Ackermann et al. (2010). Experimental results were provided, indicating high completeness and validity of inferred requirements, as well as proving the fruitfulness of this methodology on Simulink/Stateflow automotive designs. We go on to apply this methodology to a industrial project in partnership with an automotive OEM, and the findings are described in Section 3.1. Scripts were also created to automate the processes described in this methodology.

Additionally, we alter this methodology to maximize coverage for tabular designs, as well as produce higher quality candidate software requirements.

A new methodology is presented which makes use of formal property proving techniques as an alternative means of verifying the validity of potential requirements, whereby software properties are formally proven to hold over all behaviour of the system. Furthermore, a different tool is used as the basis for this methodology, one which provides MC/DC support for Stateflow truth tables, as well as an alternative testing approach. The proposed methodology provides more extensive test coverage, which in turn produces higher quality candidate software requirements. Additionally, this methodology can serve as a complimentary technique to that of Ackermann et al. (2010). This proposed methodology is presented in Section 3.2.

3.1 Reverse-Engineering Methodology

Here, an overview of the process of reverse-engineering requirements is described, as presented by Ackermann et al. (2010). This process is depicted in Figure 3.1.

1. **Generate Test Cases.** A test suite, i.e. collection of test cases, is automatically generated from a Simulink/Stateflow model such that model coverage is maximized according to several testing metrics.

2. **Data Mining.** Relationships between the input and output variables which remain constant over the entire test suite are identified by applying an association rule mining tool on the test data. These relationships represent invariants of the system. Since the relationships are inferred based on a test suite that exercises only a part of a model's behaviour, the identified invariants are only potential invariants, and therefore, potential

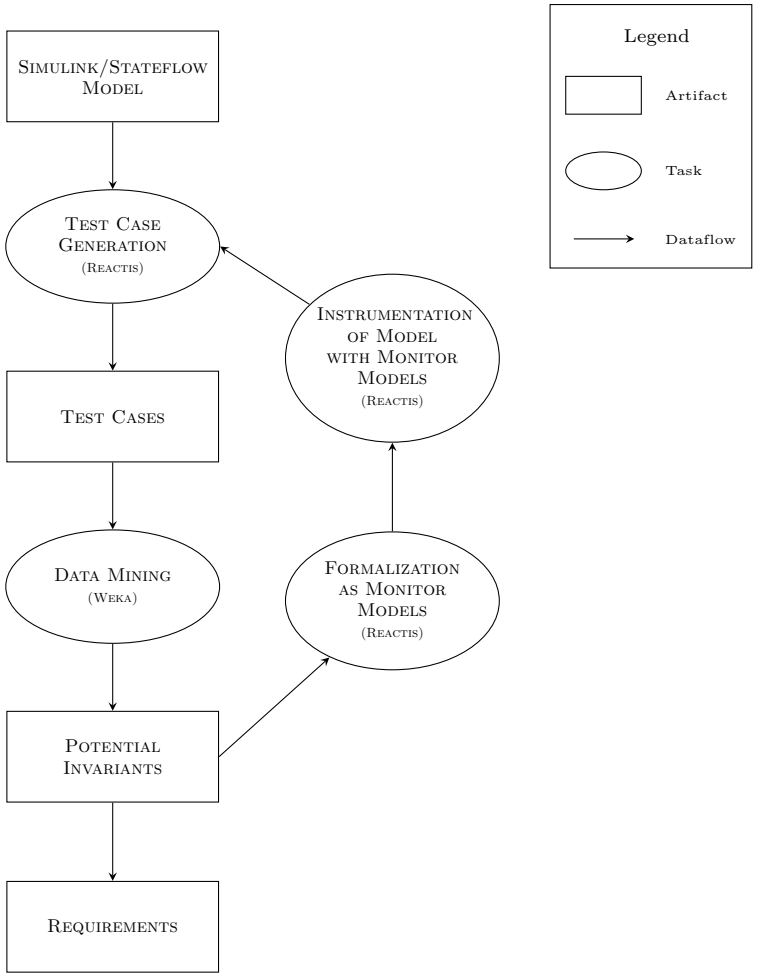

Figure 3.1: Methodology for reverse-engineering requirements from Simulink/ Stateflow models

requirements.

3. **Instrumentation-Based Verification.** Potential invariants, i.e. requirements deduced, are formalized as *monitor models*. For each potential invariant, the original Simulink design is instrumented with a corresponding monitor model, and automatic test generation is again performed to check if there are any executions of the Simulink design

that violate the monitor model. If no such execution is found, the invariant can be inferred to be a valid requirement with a high level of confidence.

3.1.1 Tools

As stated in Section 1.1.1, software models in safety-critical domains are typically large and complex. Therefore, tools are required to complement the aforementioned methodology with an effective but efficient means of automatically generating test cases for systems, and performing data mining tasks.

Reactis by Reactive Systems, Inc. is an automated testing and validation environment for Simulink/Stateflow models, and is used widely in the aerospace and automotive industries, including our automotive partner. It is integrated into their development process for other testing purposes, and so, to minimize the efforts of purchasing and acquiring new software, we extend the use of Reactis to this methodology's test generation step. Furthermore, it is the tool which is used by Ackermann et al. (2010).

Reactis' testing algorithm is a three-step process, consisting of both random and targeted phases (Cleaveland, Hansel, and Sims 2010). Reactis analyzes models and creates optimized tests that extensively exercise designs over several types of coverage metrics. These include, decision, condition, and MC/DC coverages, as well as several other structural metrics.

Additionally, Reactis provides support for Instrumentation-Based Verification (IBV). More precisely, it provides Assertion instrumentation, allowing for a requirement to be formalized as a monitor model, called an Assertion, in the system. Reactis can then perform targeted testing in an attempt to violate

the requirement, endeavouring to falsify any which do not hold true for all executions.

A beneficial feature of Reactis is its complete segregation from the original model. Any auxiliary information required for testing or other activities, for example augmenting the model with objectives or constraining inputs, never alters the model. This additional data is stored in separate files created by Reactis, thus helping to prevent accidental model modifications.

Weka, a data mining toolkit, contains a collection of data mining algorithms. Included is the well-known Apriori algorithm developed by Agrawal and Srikant (1994), which performs association rule mining on test cases. Several data mining tools do exist, implementing various algorithms, however we are interested in Apriori as it is extensively studied, and choose a simple, easy to use, open source tool to perform the data mining tasks required. Furthermore, Weka supports the comma-separated values (CSV) test data produced by Reactis, thus making for a more seamless workflow.

The most recent, stable versions of these software tools are used in our application. At the time of writing, this refers to Reactis V2014 and Weka 3.6.

3.1.2 Application and Results

The methodology outlined in Section 3.1 was applied to models provided by an automotive OEM. As a result of our collaboration with an industry partner, we have at our disposal large industrial models of vehicular control software, and have the opportunity to apply the methodology on industry code. However, due to the fact that the industry models we are working with are proprietary

information that we are unable to disclose, the following example's details have been anonymized. In general, from the numerous Simulink models and systems which compose the functionality of a vehicle, we choose one subsystem implementing some vehicle function, for which we endeavoured to extract requirements.

Figure 3.2 shows the system of interest, on which we will apply the requirements extraction methodology. This system was extracted from the rest of its model, as we wish to focus on this subsystem specifically. Models typically contain various other related systems, such as input processing, diagnostics, and output processing, which are not part of the control algorithm implementation. Thus we use Reactis to extract the subsystem for which requirements are to be reverse-engineered. At a high-level, the model presented performs arbitration of driver requests while considering the current status of the system, i.e. the previous arbitrated status, as well as other vehicle conditions. It does so through the use of four Stateflow truth tables at the core of its design. What follows is a step-by-step account of the application of the requirements extraction methodology on this system.

1. **Test Case Generation** Reactis is used to automatically generate test cases with the goal of maximizing a variety of coverage metrics. We discuss test coverage maximization strategies in greater detail in Section 3.1.4. Automatic test generation on the model in Figure 3.2 was performed using testing parameters allocating 500 tests to the random phase, 5000 execution steps per random test, and 19500 steps for the targeted phase. This test run produced 2 test cases, with a total number of 22 steps. The resulting coverages achieved are given in Table 3.1. Some

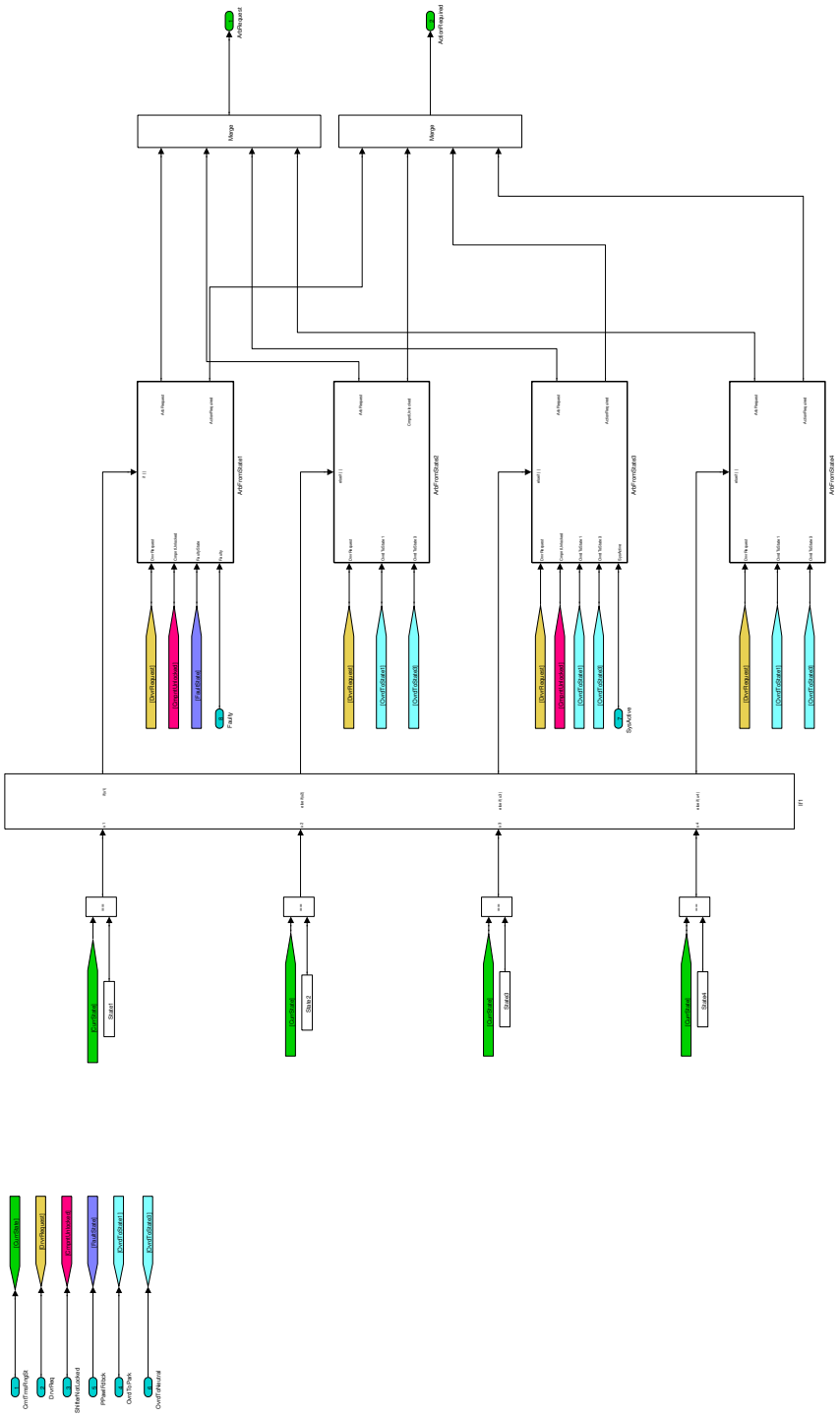

Figure 3.2: System undergoing requirements extraction

Coverage Metric	Covered	Unreachable	Uncovered	Targets Covered
Subsystem	10	0	0	100%
Branch	21	0	4	84%
Lookup Table	0	0	0	–
State	0	0	0	–
Condition Action	0	0	0	–
Transition Action	0	0	0	–
CESPT	0	0	0	–
Decision	14	0	4	78%
Condition	16	0	4	80%
MC/DC	6	0	4	60%
Boundary Value	30	0	0	100%
User-Defined Target	0	0	0	–
Assertion Violation	0	0	0	–
TOTAL	97	0	16	**86%**

Table 3.1: Testing coverage results for an automotive subsystem containing Stateflow truth tables

metrics were not applicable to the design, i.e. zero targets generated, and therefore appear with "–" coverage. For example, the design does not employ Lookup Table blocks, and therefore no tests were generated to exercise that metric.

It is important to note that Reactis relies on a random testing engine (further discussed in Section 3.2), meaning that performing test generation on the exact same model under the same conditions will result in a different test suite each time.

2. **Data Mining** After test generation, the test suite is exported and preprocessed for use with Weka. Preprocessing takes place within Weka using its various data filters. The unsupervised `NumericToNominal` filter is used to designate seemingly numeric attributes as nominal, so that the Apriori algorithm can then be applied to this data.

Section 3.1.4 discusses the Apriori algorithm's lack of support for numerical types in invariants. This necessitates additional data preparation through the removal of incompatible signal data recorded in the test suite. This can be accomplished through Weka's preprocessing utility or any other available `.csv` editor. Moreover, the inference of timing requirements is an area of research necessitating further study and is not explored at this time, so test simulation time `__t__` is also removed.

Using Weka, data mining on the test data generated in the previous step was performed. Default parameters for the Apriori algorithm were used. However, a high upper bound for the number of rules to generate was given, so that all possible invariants were generated, whereas the default is set to ten. The confidence parameter was also augmented to 100% instead of the default 90%, so that only invariants which hold over all tests are inferred. Additionally, to assist Weka in producing invariants of a specific format, specifically those with outputs as the consequent, we use *classification rule mining* to infer their class. These measures for producing more meaningful invariants are addressed in Section 3.1.4. Invariants are rules in the form of an implication, where the premise is a conjunction of variable equalities, and the consequent is a single variable equality that follows from the antecedent. In total, 496 invariants were generated during this step. A sample of the derived invariants is given in Figure 3.3, with the remaining invariants given in Appendix A.2.

3. **Instrumentation Based Verification (IBV)** The potential invariants inferred by Weka are validated using Reactis. The model is instrumented with invariants acting as monitor models, observing the model for erro-

(a) $eCurrState == cState1 \wedge eDrvrRequest == cState2 \implies eArbRequest == cState2$

(b) $bCmpntUnlocked \wedge bOvrdToState1 \wedge eFaultyState == Stage2 \wedge bSysActive \wedge bFaulty \implies eArbRequest == cState1$

(c) $eCurrState == cState3 \wedge \neg bSysActive \implies eArbRequest == cState3$

(d) $eCurrState == cState1 \wedge eDrvrRequest == cState2 \wedge bCmpntUnlocked \implies eArbRequest == State2$

Figure 3.3: Sample of generated invariants from Figure 3.2

neous values during test generation. It is through a directed testing effort that Reactis strives to violate the invariants, while also maximizing structural coverage of the design. These invariants were formalized as Reactis Assertions using a simple expression language. While this formalization and instrumentation is supported by Reactis in a manual fashion through its GUI, this process was automated through the use of Reactis' API. Custom scripts were created for the automatic processing of Weka output and subsequent addition of each potential invariant into the model's information file (`.rsi`).

After instrumenting the Simulink model with the generated assertions, assertion checking is performed. This is essentially targeted testing for the purpose of exercising assertions and monitoring their validity. Invariants which are falsified during this process are discarded, as they are do not hold over the test suite generated. This process is repeated until no further assertions are discarded. Those remaining after several iterations can be accepted as specification requirements with a high degree of confidence. Even so, these assertions, although true for some execution of the model, cannot be determined with absolute certainty that they are invariant over all behaviours of the system. Testing is rarely exhaus-

tive, and in the case of large, complex systems, exhaustivity is unattainable. Thus, drawing conclusions about whether an invariant holds for the whole system using testing is not possible. This issue is touched upon in Section 3.1.4, and an alternative verification approach is put forward. Regardless, the remaining invariants after several iterations are candidate requirements, and should be discussed with domain experts in order to ascertain their usefulness in capturing system behaviour, as not all invariants may be meaningful, and thus useful.

Returning to the example, after formalization and instrumentation, subsequent automatic test generation on the instrumented model disproved many invariants. Out of the 496 potential invariants, 481 were disproved during simulation. Upon iterating a second time, no further invariants were disproved, and so the methodology is terminated. In the end, only 15 invariants passed the IBV phase, and these can be considered as potential requirements.

Therefore, approximately 97% of initially generated candidate invariants were erroneous. With an overall test coverage of 86%, this is a substantial amount, and if it were not for the automation scripts created as a supplement to Reactis' capabilities, instrumenting the model by hand would be unrealistic. This however begs the question: Why were so many falsified?

3.1.3 Deficiencies

The primary reason for the large amount of inaccurate invariants is due to the testing capabilities provided by Reactis. This affects both the testing and IBV phases of the methodology.

Testing Tables Matlab Simulink and Statefow provide extensive block libraries, with several hundred unique blocks for the representation of a wide array of modelling concepts and equations. As a result, it is often the case that third-party software tools may only support a subset of the blocks available, or may provide superior support for some blocks over others. In particular, when it comes to Reactis, support for thorough testing of Stateflow truth tables is not provided. Although tool documentation for Reactis V2014 states that Stateflow truth tables are currently "supported", and have been since V2012 (Reactive Systems 2014), our experience has found that this simply means that Reactis is able to execute Stateflow truth table blocks and compute correct output values. However, truth tables are treated as black-boxes, with no testing of the internal logic they implement in terms of condition, decision, or MC/DC metrics.

Further investigating this deficiency, tests conducted with Reactis show that the only metric exercised on a truth table is the Boundary Value Coverage of its inports. This ensures that the input domains are exercised based on their associated type. For example, if a Stateflow truth table has two Boolean inports, Reactis will produce tests to cover the limits of these inports' domains, namely the inputs being both `true`, or both `false`. However, condition, decision, and MC/DC of the internal logic is not maximized. Thus the decisions within the table will not be tested such that the tables are thoroughly exercised, and therefore coverage of the model is ineffectively measured. Table 3.2 is taken from Reactis' test coverage report, and is the summary of local coverage for one of the tables out of the four in the system in Figure 3.2. The remaining three had identical coverages. Note the non-existent targets for decision, condition, and MC/DC coverage. In reality these three metrics are not

Coverage Metric	Covered	Unreachable	Uncovered	Targets Covered
Subsystem	1	0	0	100%
Branch	0	0	0	–
Lookup Table	0	0	0	–
State	0	0	0	–
Condition Action	0	0	0	–
Transition Action	0	0	0	–
CESPT	0	0	0	–
Decision	0	0	0	–
Condition	0	0	0	–
MC/DC	0	0	0	–
Boundary Value	0	0	0	–
User-Defined Target	0	0	0	–
Assertion Violation	0	0	0	–
TOTAL	1	0	0	100%

Table 3.2: Testing coverage results for a Stateflow truth table

applied to the internal logic that the Stateflow truth table implements. Reactis only seeks to execute the table subsystem, and thus, this is the only metric targeted.

Some literature suggests that decision tables should be tested with black-box techniques (Hass 2008). Perhaps it is for this reason that white-box testing metrics on truth tables have been neglected. Another explanation may be that due to the large library of blocks in Simulink, truth tables were simply not tested thoroughly. For safety-critical software, this is certainly not acceptable. As stated in Section 1.1.2.1, ISO 26262 requires the use of structural metrics such as MC/DC for ASIL D and also recommends MC/DC for ASILs A, B, and C. Additionally, this lack of adequate truth table testing support means a possible deficiency in test cases produced, and consequently invariants inferred. The less a test suite exercises the mode, the more likely it will be that the generated invariants will be of poorer quality. This issue was discussed with

a Reactis engineer, however there are currently no plans to expand coverage metrics to tables.

Instrumentation Based Verification The use of testing as an iterative verification step also presents deficiencies in the methodology. Testing, whether employing targeted or random testing strategies, is rarely exhaustive for non-trivial software programs. Therefore, IBV deals with an incomplete state space when it comes to validating invariants. When performing IBV using Reactis, falsified assertions can be considered as false with 100%certainty, since there exists a test case which provides a counterexample. On the other hand, it is not possible to judge with certainty the validity of an invariant should it hold for a test suite. An inexhaustive test suite encompasses a subset of a model's entire state space, and thus there is the potential for the existence of a simulation instance which proves the invariant false. However, because this simulation run was not covered during testing, the invariant was not falsified. Thus, in employing testing as a verification step, we consequently cannot draw a definitive conclusion as to the validity of an invariant for the system. Nevertheless, in iteratively verifying invariants, a high degree of invariant confidence can still be attained, although not proven formally.

These two issues are rectified in a new methodology presented in Section 3.2.

3.1.4 Issues Encountered

Several, interrelated issues were encountered in the application of this methodology on Simulink/Stateflow models, however, most were resolved when it came

to our application. This section elaborates on the additional considerations required when applying the methodology given in Section 3.1.

The most significant obstacles in applying the methodology are as follows:

- Lack of support for numerical types. In exploring this issue, we also found that enumeration types were not being handled correctly either
- Increasing test coverage of models under test for generating invariants which represent valid system behaviour
- The substantial number of invariants are generated, not necessarily in an appropriate format

Numerical Types Association rule mining is a learning scheme originally created in the context of transactional databases, for example, point-of-sale systems of a store. As a result, the Apriori algorithm works on data in the form of records where attributes are either present or not. The Apriori algorithm addresses the "Boolean Association Rules problem" where all attributes can be thought of as Boolean values that are either true or false (Agrawal and Srikant 1994). Therefore, association rule mining on numerical data is not supported (Webb 2004). If any numerical data is included, application of the algorithm cannot take place. This poses a problem because databases in most domains include both quantitative (numerical) and categorical (Boolean and enumeration) data.

In order for the Apriori algorithm to run on the test data, unsupported data must be discarded. Clearly, the disadvantage of removal is that data is lost, possibly impacting the invariants produced. If a large portion of the data is numerical, and a fair amount of attributes need to be removed, then

their absence from the data affects the results, in that removed attributes will not be present in the invariants. Although this temporary solution is not optimal, in a system where there are only a few numerical attributes removed, the results will not be altered in as significant a manner. Thus, until numerical data can be handled, these types of systems prove to be better for invariant inference. Using Weka's `NumericToNominal` filter on a numerical attribute is possible, however this wrongfully treats each individual numeric value as a valid nominal value of that attribute, discarding potentially valuable ordering information needed to handle ranges. Future work needs to be done on extending the methodology in order to accommodate numerical data, such that the learning scheme is able to create inequalities involving ranges, rather than simple equality tests.

In our presented example, no system attributes were discarded, as they were non-numerical types. On the other hand, the test suite simulation time was removed from the test suite, so timing information was lost.

Model Coverage In order to achieve better invariants which express valid system behaviour, it is necessary to extend the generated test suite in terms of coverage. Examination of the model's coverage can be done using Reactis' *Coverage-Report Browser*, which enables users to track which parts of a model have been executed and which have not. In the initial efforts of test suite generation, a lower cumulative test coverage was achieved. As a result of this poor coverage, resulting inferred invariants suggested that an input was equal to an output. It was evident that such a relationship between ports could not be valid, because it would suggest that a significant design flaw existed in the system. As anticipated, an increase of model coverage eliminated these

erroneous invariants, thus improving the quality of the invariants produced.

Test suite coverage can be maximized in a number of different ways. The most straightforward approach is adjusting the test generation parameters that are available prior to testing. Reactis performs testing by applying a three-phase algorithm on the data (Cleaveland, Hansel, and Sims 2010). These three phases of the algorithm can be adjusted to provide testing behaviour which better covers the model. The *preload phase* allows the user to specify one or more existing test suites, which the testing algorithm strives to extend. After these initial test suites are loaded, additional tests are generated to augment their tests cases, enhancing coverage. Reactis Tester also performs random testing of the model. This *random phase* of test generation can be directed by adjusting the number of different tests to generate and the upper bound on the number of steps in each of these tests. Lastly, the *targeted phase* of the algorithm aims to maximize the coverage metrics specified by the user.

An increase in coverage was also achieved through model decomposition into smaller, independently testable subsystems. In a large system with a low test coverage, subsystem extraction increased the coverage. The size of a system also effects the number of invariants which can potentially be produced. A system with many ports can potentially yield a larger set of invariants. At the completion of the reverse-engineering methodology, the inferred invariants must be manually inspected by domain experts, i.e. developers, to determine whether or not they capture valid and meaningful system properties. This becomes prohibitively difficult when the number of generated invariants is large. Decomposing the model into smaller subsystems addresses this problem since the invariants are inferred on a smaller subsystem, between a smaller set of ports, and thus produces a smaller set of system properties, which are

feasible for developers to examine.

Furthermore, increasing coverage by interactively tuning testing is also supported by Reactis, however domain expertise and knowledge of the system is required in order to provide the necessary insight for using this method. Conditions which are deemed difficult to satisfy can be identified, and the simulation steps which would cause the condition to be covered must be determined and added to the test suite. In the case that no requirements documentation detailing the behaviour of the system is available, this is not possible without intimate knowledge of the software.

For our application, parameters for testing were chosen after a couple test generation attempts, and were selected when increasing testing time or steps did not yield any gains to coverage.

Constraining Inputs In order to reduce the number of possible values a test generation tool must consider, the type of an input should be constrained to the set of its possible values. In general, when the values of ports of the top-level model are specified in Simulink, Reactis is able to infer typing information, however there are potential issues when it comes to custom types or subsystem extraction, resulting in a loss of this data.

With respect to enumeration types, it was observed that several ports were inferred to be numerical `int16` type, although they were enumeration types in reality. Upon tracing the issue, it was discovered that the cause of this was due to custom handling of enumeration types in separate script files which were not picked up by Reactis. Therefore, it is imperative that inport types are inspected and constrained. During test generation Reactis will feed in ranges of integers into these enumeration types which are outside their expected scope,

resulting in two problems. Firstly, when the test suite is then opened with Weka, some attributes appear to take the set of all integers $\mathbb{Z}$ as inputs, and are classified as a numerical type, when in actuality they are enumeration types. This poses a problem in the application of the Apriori algorithm on the data, as numeric attributes are not supported, and thus must be either converted into a nominal, i.e. Boolean or enumerated, attribute or discarded.

Secondly, treatment of a data type with a small set of possible values as a larger data type encompassing a larger range of values leads to poorer test coverage. Reactis feeds in a range of integers as inputs which are beyond the expected scope of the port, and so testing unnecessarily explores state space which is unreachable. To prevent this from occurring, it is necessary to reduce the values fed into inports to a range of valid values. This is accomplished by constraining the values generated for inports during test generation, simulation, and validation through the specification of ranges and subsets within a base type. Reactis facilitates the constraining of inport ranges through its *Type Editor* utility.

There may also be potential issues with Reactis, as identified by their developers. Subsystem extraction should correctly set data types for inports and outports of the extracted subsystem, however, problems may still arise as a result of data type conversion blocks when they attempt to convert an 'auto' type, automatically inherited from a top level port, to an enumeration type. This, however, is not an issue which we are currently experiencing as these conversion blocks are not present in the industrial models used with this methodology.

Invariant Quality Although increasing coverage and model decomposition decreased the number of invariants inferred, a substantial amount are generated nonetheless. This is largely due to the nature of association rule mining. Association rule learning generates rules between two or more attributes, and unlike classification, does not seek to predict a certain set of attributes. Consequently, numerous rules are typically generated from even a very small dataset. Therefore, endeavouring to restrict rules generated to only those present in the most number of steps of the test suite, the Apriori algorithm should be applied using strict parameters with regards to rule *confidence* and *support*, as described in Section 2.2.

The Apriori algorithm was applied with the highest threshold for confidence to the test suite produced by Reactis. Only those invariants with 100% confidence were generated. Invariants not satisfying these criteria are discarded, leaving only those with the highest support of test generation data. IBV was applied iteratively to remove any which are not properties of the entire system. Furthermore, invariants need to be examined manually to determine whether they can be considered requirements. Not all observed invariants are relevant or describe meaningful system behaviours. Thus, a manageable amount of invariants must be inferred such that the quantity is feasible enough to be reviewed by domain experts. Only those rules which are deemed the strongest should be presented. Once the validity or relevance of these invariants is assessed by developers, other system invariants which may capture important but less supported behaviour may also be inferred.

Invariant Structure The Apriori algorithm's primarily purpose is for mining lists of items/attributes, typically from large databases. As such, it does

not consider how the itemsets are formed in terms of grouping. Itemsets can contain any combination of one of more attributes. For the application of Apriori on systems with data represented as signals of inputs and outputs, we wish to infer how inputs impact outputs. Naturally, attributes representing the inputs of the system should appear as the antecedents of the association rule, while outputs the consequent. Weka is unable to distinguish between inputs and outputs automatically from the test suite loaded, and there is no direct mechanism in Weka to facilitate the grouping of attributes through manual intervention. Opus Magnum, a similar tool used by Ackermann et al. (2010) has support for limiting attributes to the left-hand-side (LHS) or the right-hand-side (RHS) of an association rule, however, it is unclear if this functionality was utilized in their study.

An alternative solution is to make use of Classification Rule Mining. This type of data mining seeks to predict a specified class, and as such, will attempt to achieve it as a consequent of the rule. Therefore it is possible to force the attribute to be on the RHS of mined rules by selecting it as the attribute class to be predicted. Weka allows for this to be set by enabling the user to choose a single attribute as the consequent of rules.

Should the classification rule mining approach be unsatisfactory, logical transformations can be applied on the rules to move inputs to the antecedent, and outputs to the consequent, thereby, enabling invariants to be structured in a more telling form. This avenue is not explored here.

3.2 Proposed Methodology

During the analysis and use of the software requirements reverse-engineering methodology described in Section 3.1 deficiencies inherent in the methodology were encountered. From these findings, a revised methodology was created to address inadequacies in table support, as well as invariant confidence. To resolve these shortcomings in the methodology's tool-chain, we modify it to include an alternative model testing tool which seeks to properly maximize code coverage of truth tables. Furthermore, property proving instead of IBV is used as a means of proving that the invariants do in fact hold over all of the system's behaviours. In this new process, Simulink Design Verifier (SDV) is used to remedy Reactis' deficiencies in terms of testing Stateflow truth tables, and provideing an alternative means of verifying invariants.

For safety-critical systems, the use of separate and diverse approaches to verify data is common practice. To ensure that inferred requirements represent valid system properties, a parallel means of verification is beneficial. The use of this new methodology is complementary to the methodology given by Ackermann et al. (2010), and can be used to corroborate results using an alternative approach to IBV. The new methodology is given in Figure 3.4.

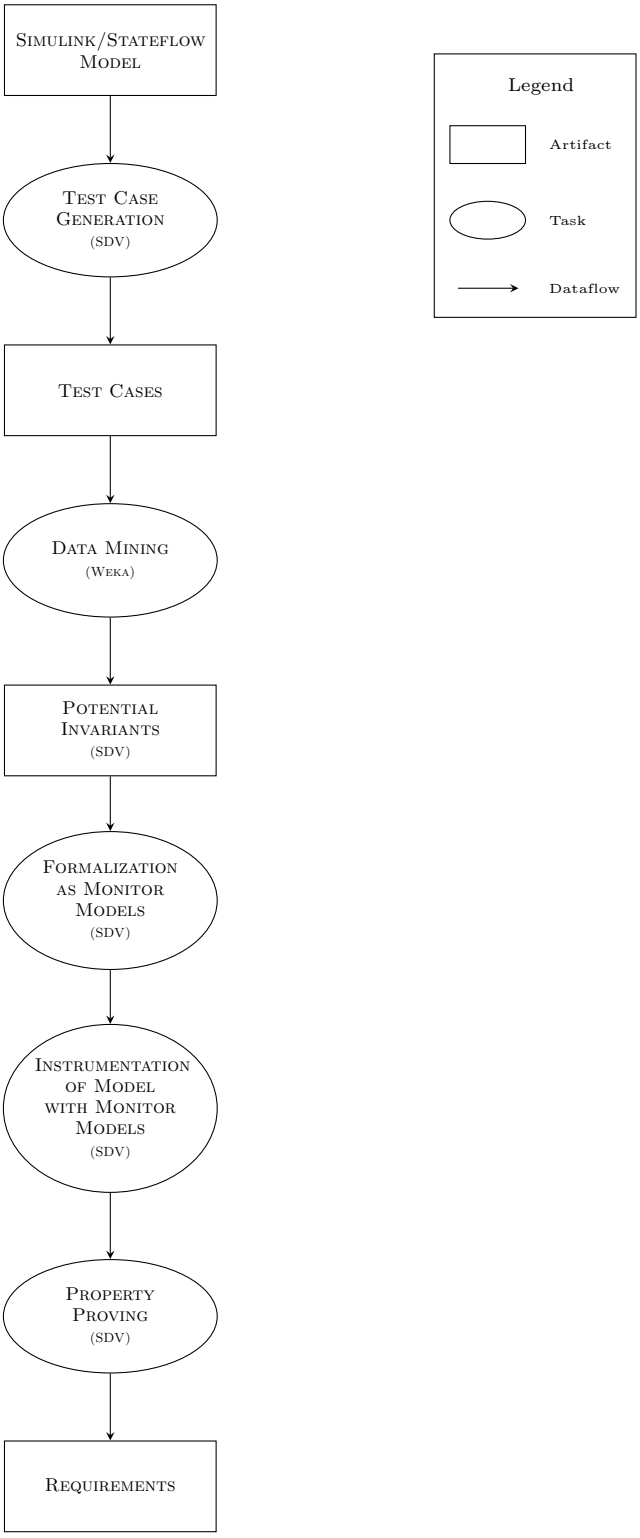

Figure 3.4: An adapted methodology for reverse-engineering requirements from Simulink/Stateflow models using property proving

3.2.1 Tools

Simulink Design Verifier by Mathworks, performs automatic test generation on Simulink/Stateflow models using a formal methods engine[1] (The MathWorks Inc. 2014). Simulink Design Verifier (SDV) uses Prover Plug-In (Andersson et al. 2002) by Prover Technology for test generation and property proving, as stated by The MathWorks Inc. (2014). The Prover Plug-in performs model-checking using symbolic manipulations of propositional logic, as developed by Sheeran and Stålmarck (1998). SDV also statically analyzes the model and creates optimized tests which exercise the design over the condition, decision, and MC/DC metrics.

The primary motivation for using SDV is largely due to the fact that other tools do not achieve satisfactory coverage in cases where designs involve Stateflow truth table blocks. SDV is superior in that it is able to perform white-box testing on truth tables using condition, decision, and MC/DC metrics on the internal logic. According to automotive standard ISO 26262, MC/DC is necessary for the testing of safety-critical software. This was discussed in Section 1.1.2.1. Thus, we select SDV over other testing tools for embedded systems.

SDV is also capable of performing the Instrumentation-Based Verification task of the original methodology, but in a different manner. It too provides mechanisms by which monitor models can be defined in the the system under test, and they are referred to as Test Objectives. Test generation, through the use of these Test Objectives, can take place in a similar fashion as the reverse-engineering methodology of Section 3.1, however there is a significant

[1] However, Simulink's language semantics are not formally defined

difference in the testing strategies utilized. This has repercussions on the methodology. For IBV, the original reverse-engineering methodology relies on a random testing strategy, allowing for the explored state space to be extended in each iteration. On the other hand, SDV relies on a formal methods verification engine, thus deterministically producing test cases.

As a result, iterating in the IBV phase does not explore additional state space of the model when using such an engine. In the original methodology, the state space was extended as a consequence of the random test generation portion of the algorithm. Therefore, it is necessary to instead use SDV's property proving capabilities for verifying that invariants hold over the system. Although deviating from the spirit of the original methodology, this alteration is advantageous in its own rite. It provides a formal check of the validity of invariants, whereas this was not previously possible. A higher degree of confidence as to the invariants' validity is ultimately attained. There is added confidence that the invariants are indeed requirements which hold over the entire system, rather than merely over a non-exhaustive test suite.

The version of SDV in use for the application of the proposed methodology is version 7.8, such that it is compatible with our automotive partner's use of Matlab 7.13 (2011b). It is fully integrated into Matlab Simulink.

Weka is used in the same capacity as outlined in Section 3.1.1. Its application in the methodology remains the same.

3.2.2 Application and Results

Application of this methodology is shown on the same system used in the demonstration of Ackermann et al.'s methodology in Section 3.1.

1. **Test Case Generation** This step differs from the original methodology due to the difference in the testing engine. With SDV's engine, testing is deterministic, and so multiple test runs on the same model will yield the same test cases and coverages each time. Before simulating the model, SDV does a static analysis and generates targets which must be satisfied in order to satisfy the coverage metric objective. The testing objectives can be set to focus on either of the three testing targets provided: decision coverage, condition coverage, and MC/DC. Maximization of the MC/DC metric is the strongest of the three, as discussed in Section 2.4, and so it is selected as the objective of the testing engine. Test conditions can also be added to constrain inputs, and are encouraged to increase model coverage. To achieve MC/DC, 1032 targets were computed and consequently attempted to be exercised. Out of these targets, 752 were satisfied, while 279 were falsified. Again, the distinction between these results and those of Reactis is that SDV can formally prove that it is impossible to satisfy these targets, whereas Reactis they may simply classify them as not covered. In total testing produced 7 test cases with approximately 73% MC/DC coverage.

2. **Data Mining** This step remains the same, however preprocessing must be done on the test data. SDV does not easily export its test data to a suitable format conducive for Weka's data mining. A manual process was used to create a compatible test suite, however this could be easily automated. 192 candidate invariants were inferred during this step.

3. **Property Proving** We use SDV to check the validity of invariants. The invariants can be formalized in SDV using monitor models in Simulink

and then model checked. SDV uses a verification engine based on formal methods. SDV can *prove* validity of a potential invariant in the case when the size of the state space of a model's behaviour is not prohibitively large. Invariants are implemented using the standard Matlab function blocks, and then their outputs are connected to SDV Proof Objective blocks. These are housed within a verification subsystem, as shown in Figure 3.5. SDV attempts to formally prove or disprove each of these specified requirements by exhaustively exploring the state space for counterexamples. If the requirement is falsified, a counterexample is provided which gives the particular simulation scenario when this transpires. The results which pass this phase are given in Appendix A.1 due to space considerations. Out of a total of 192 candidate invariants, 112 were validated, and 80 falsified.

3.2.3 Issues Encountered

Many of the points highlighted in Section 3.1.4 hold for this methodology as well. such as the Apriori algorithms lack of support for numerical data and invariant quality and structure. Test coverage is also increased when the inputs are constraining using SDV's Test Condition blocks. However, model coverage is evaluated differently in the context of SDV and cannot be maximized through the adjustment of test parameters in the same way. More time can be allocated to testing if the model is large, and the user is able choose between decision, condition, and MC/DC metrics as the targets for testing. If these targets for these metrics are unattainable, SDV can prove as much, and will terminate. The following are additional issues encountered in

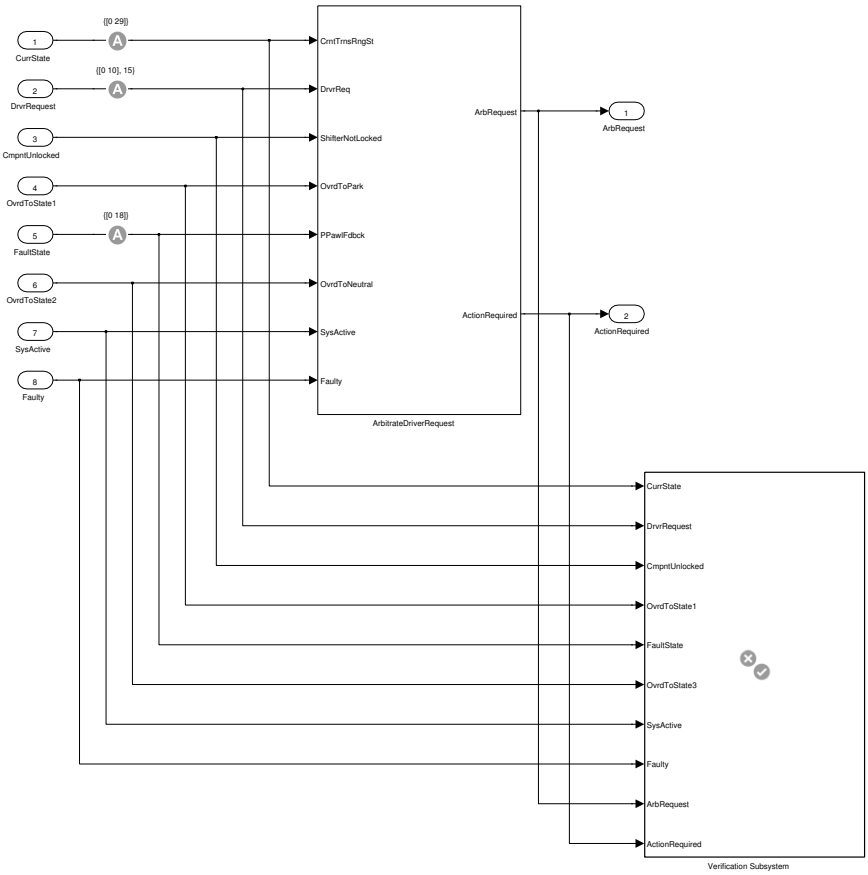

Figure 3.5: Model instrumented with a SDV Validation System

the application.

Subsystem Extraction In comparing the support of both tools' extraction capabilities, Reactis' is more robust. Reactis actually creates a new model with the extracted subsystem, while also allowing the user to specify if triggers and other artifacts should be retained. In contrast, SDV allows for the designation of certain blocks as *atomic*, and then these can be tested and proved independently from the remainder of the system. However there are limitations as to which block types this can be applied to. Issues were encountered when making `if` conditional subsystems atomic, and this is not possible for these

types of blocks. For these situations Reactis was used as a supplementary tool, providing the system extraction required.

Test Suite A drawback of using SDV in this methodology is its absence of test suite creation. While it does generate test harnesses and a custom data structure containing important test data, this information is not easily transferred as input to Weka. Extra effort was required to construct a `.csv` test suite compatible with Weka. This was a manual process, however this would be easily automated.

3.3 Comparison

Here, a comparison is done of the two reverse-engineering methodologies presented thus far: a) Ackermann et al.'s methodology using random structural testing (given in Section 3.1); and b) the property proving methodology described in Section 3.2. Analysis as to the running time, human effort, etc. are not studied here. This comparison is done simply from the perspective of producing better tests and invariants from Stateflow truth tables. Graphs are used to summarize the findings from Sections 3.1 and 3.2. These results are generated by applying the two different reverse-engineering methodologies to the same model. As the scope of this work focuses on tabular design, a model which contains Stateflow truth tables was used.

Firstly, with regards to testing systems which contain Stateflow truth tables, Figure 3.6 illustrates the amount of test data generated for each approach. In general, test cases are a means on grouping some number of test steps. Each test step records the input/output for a single simulation time step. Thus more

focus is given to the number of time steps within a test cases. The graph depicts the data produced in their respective test suites. SDV facilitates testing on Stateflow truth tables, therefore it is no surprise that its generated test suite contains more test cases and test steps than what Reactis is capable of producing. The reason for the undersized amount of test cases and steps generated by Reactis is demonstrated in Figure 5.1. Namely, Reactis fails to target the MC/DC testing metric.

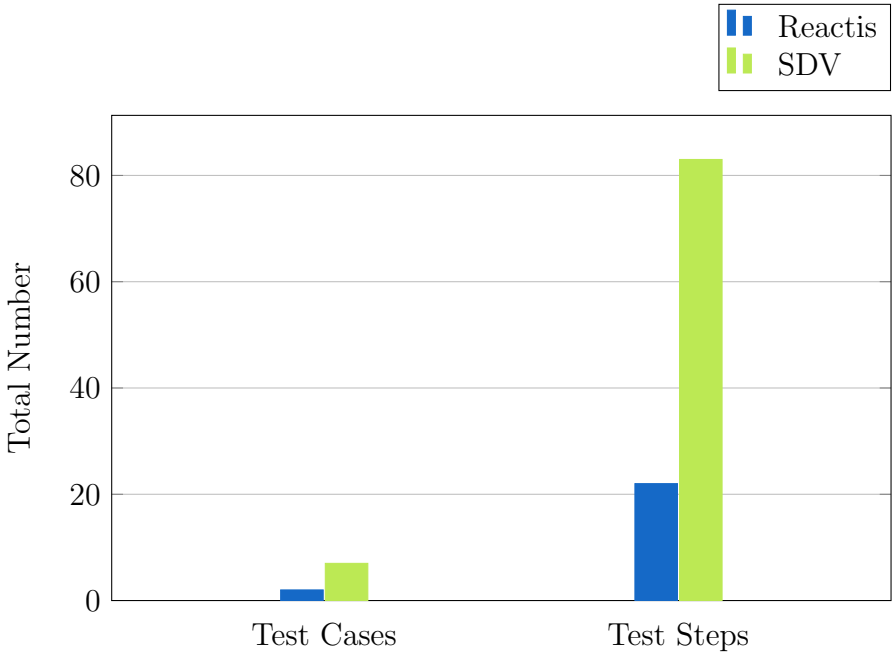

Figure 3.6: Comparison of Test Suites

In comparing the MC/DC testing capabilities, the results are stark. The total number of test objectives produced by the tools is given, as well as those which were covered and not covered during simulation. In general, test objectives represent the cases identified by the tool as necessary goals to be achieved during testing. They strive to fully exercise the model in terms of various metrics, endeavouring to explore a model's state space extensively.

In this context, attention is given in assessing MC/DC objectives generated. As evidenced in Figure 5.1, Reactis does not consider maximizing MC/DC coverage when it comes to Stateflow truth tables. In Figure 3.2, we see other blocks present within the system, which perform various decision functions. In this case, MC/DC was applied to them, and as a result a small amount of MC/DC coverage was acquired during testing with Reactis. However, the model was not fully exercised. The treatment of Stateflow truth tables as black-boxes has a significant impact as to the quality of the test suite, and is directly reflected in the number of test cases and steps. On the other hand, SDV does fully support MC/DC on Stateflow truth tables. Therefore the amount of objectives produced and achieved far outnumber that of Reactis, and we see the magnitude of Reactis' limitation in Figure 5.1. In comparison, SDV identified and achieved significantly more objectives, leaving several hundred unidentified by Reactis.

The proficiency at which a tool performs testing directly affects the quality and quantity of generated test suites. This in turn impacts the quality and quantity of inferred invariants of the system. Figure 3.8 compares the invariants which were produced by both methodologies. Poor test coverage leads to numerous invariants generated, as relationships are inferred across all possible combinations of inputs. With more data in a test suite describing the actual relationships present between system variables, fewer combinations are possible. This is exactly the case when performing invariant generation on Reactis versus SDV, and is shown in Figure 3.8. Due to poor coverage of a Reactis generated test suite, the resultant invariants are numerous, and are subsequently falsified during IBV. Much effort is wasted in formalizing and testing hundreds of invariants which are not useful in the end, and is mini-

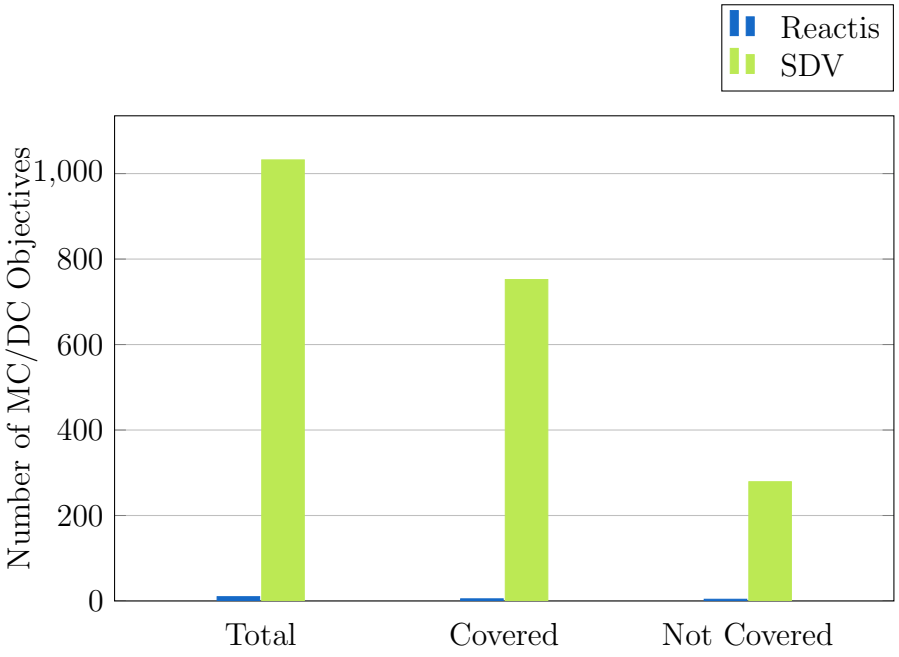

Figure 3.7: Comparison of MC/DC Testing

mally fruitful in acquiring true invariants. Out of Reactis' potential invariants initially produced, 3% were not disproven during IBV. With SDV's superior test suite, fewer potential invariants are initially produced, and the majority of these are valid over the system. SDV was 58% successful in producing valid invariants.

In conclusion, it is evident that for the success of reverse-engineering requirements from software artifacts, model testing must be done extensively and with a high degree of coverage. Capturing more test cases produces fewer, more accurate invariants. This is crucial for scaling these approaches to larger systems. Reactis does not adequately test for MC/DC for Stateflow truth tables. This directly impacts the amount the model exercised during testing, as well as the quality of the invariants inferred.

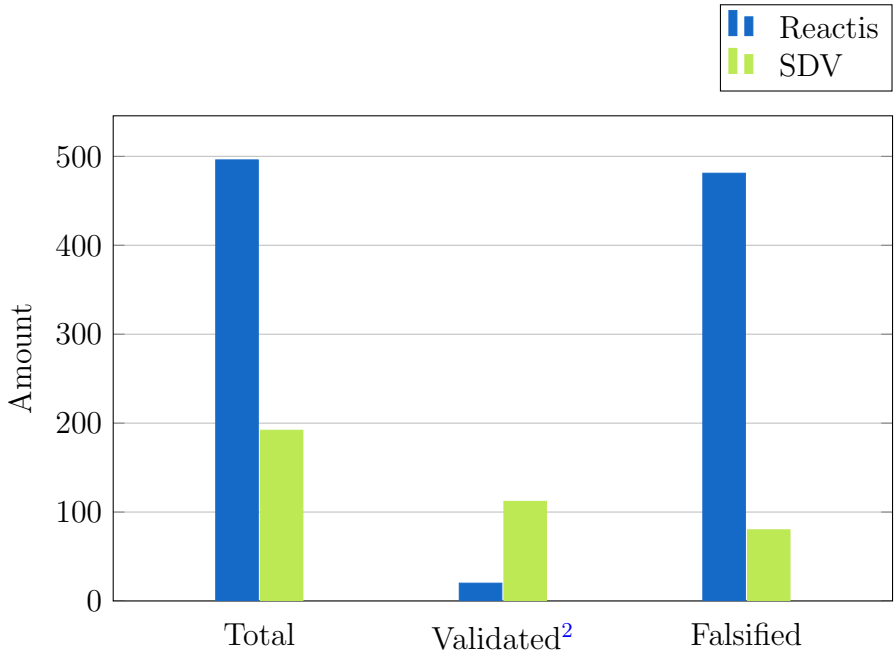

Figure 3.8: Comparison of invariant extraction

[2] i.e. Formally proven in SDV. In the context of Reactis, not disproven through testing.

3.4 Summary

In summation, this chapter presented two methodologies for the reverse-engineering of software requirements. The first, by Ackermann et al. (2010) made use of Reactis and its random and structural testing for generating and validating invariants. Our experience in using this methodology was detailed and scripts were created to facilitate its application. However, it was significantly lacking in its support for Stateflow truth tables, and also could not formally prove that candidate invariants do in fact hold over the behaviour of the system. This methodology ultimately yielded too many initial invariants, which there largely disproved through IBV. This approach proved to be inadequate for reverse-engineering software requirements from design containing the Stateflow truth

table construct.

A new methodology was created to supplement the drawbacks of the methodology given by Ackermann et al. This new approach used SDV and property proving for both test generation as well as formally verifying invariants over the behaviour of the system. SDV also provided proper testing of Stateflow truth tables, namely with MC/DC. This is particularly important as tabular constructs are increasingly being used to implement complex decision logic, including within safety-critical systems. Furthermore, the number of invariants and their quality are affected by the extensiveness of a test suite. Hence, exercising the model thoroughly is necessary to produce useful invariants, which may be used a software requirements. This methodology also provides better confidence as to the invariant nature of these requirements as they are formally proven to hold.

In comparing both approaches on an industrial model relying on Stateflow truth tables, SDV and the property proving approach was found to be superior in the number of requirements recovered from the system. However, both methodologies provide different testing approaches, each with their own benefits. Random testing provided by Reactis explores new state spaces of the model upon each test generation, whereas SDV will deterministically generates the same tests. Therefore, as future work, these two methodologies can be used in combination to maximize the amount of code exercised during testing.

Given a system with non-existent, insufficient, or out-of-date requirements documentation, software invariants can provide developers with guidance in design decisions or by providing a system overview in terms of its current functionality. Therefore, invariants extracted from Stateflow truth tables are useful for subsequent refactoring efforts. Chapter 4 presents a methodology

for the guided simplification of Stateflow truth tables, and so the invariants produced from the reverse-engineering methodologies serve as a guide during this refactoring, such that they are maintained or highlighted.

Chapter 4

Stateflow Truth Table Transformation Methodology

In this chapter, we propose a method for the refactoring of Simulink truth tables through the use of tabular expressions. Heuristics are presented, which guide refactoring in order to make tables more readable and traceable to requirements. The basic strategy of the methodology requires the transformation of the decision table into a tabular expression. The translation of decision tables to tabular expressions is employed in order to introduce disjointness into the logic, while also providing a more readable format that is conducive to software documentation. Using tabular expressions as the basis for refactoring, a set of simplifications can be applied iteratively, which reduce the size and complexity of the logic, and transform the table into a smaller, simplified format. Simplification by way of minimizing the logic of the table is achieved by the removal of decision points. The purpose of this chapter is to introduce the stepwise methodology, depicted in Figure 4.1, which can be used in the software development process.

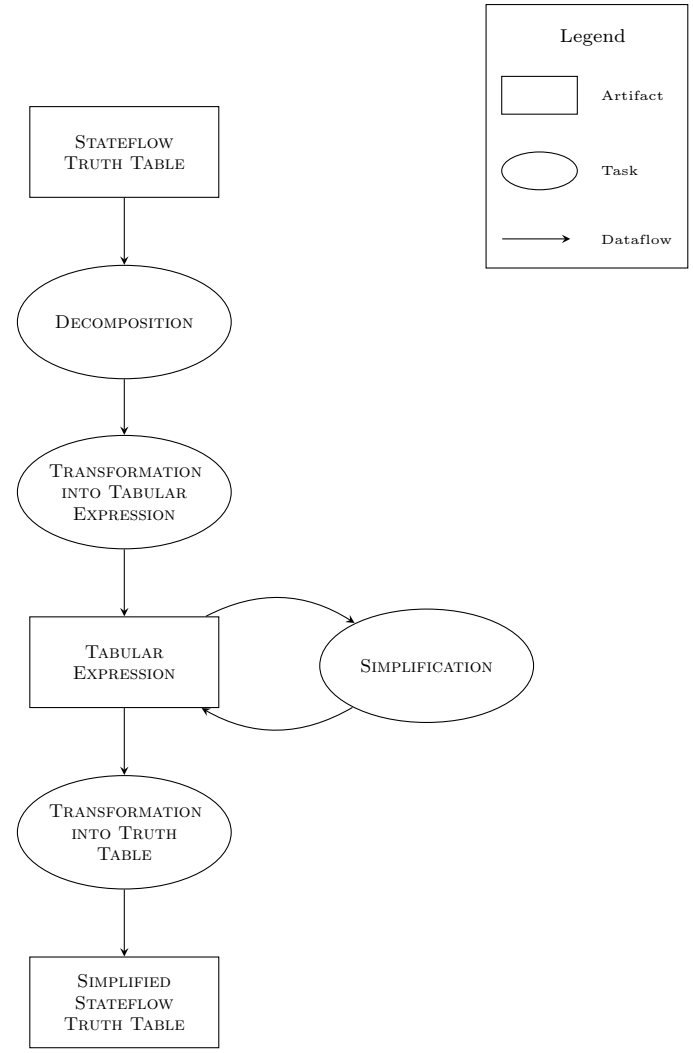

Figure 4.1: Methodology for Stateflow truth table simplification

Although this methodology uses Stateflow truth tables as the basis of its application, in general traditional decision tables can also be supported. This process is demonstrated on a generic example shown in Table 4.1. Additionally, to clearly and simply illustrate each simplification step, several discrete examples are included. The application of the methodology on real-world automotive examples is given in Chapter 5.

$fComputeFooBar(bCond_1, bCond_2 : bool, eFoo : enum) : enum, bool =$

		Decisions				
#	Conditions	D_1	D_2	D_3	D_4	D_5
1	$eFoo == cEnum_a$	T	F	F	F	-
2	$eFoo == cEunm_b$	F	T	T	F	-
3	$eFoo == cEnum_c$	F	F	F	T	-
4	$bCond_1$	-	T	-	F	-
5	$bCond_2$	-	-	F	F	-
	Actions	1	1	2	3	1

#	Actions
1	$eFoo = cEnum_a; bBar = true;$
2	$eFoo = cEnum_b; bBar = false;$
3	$eFoo = cEnum_c; bBar = true;$

Table 4.1: Example Stateflow truth table for methodology application

Figure 4.2 shows the overview of the signal flow for the table given as Table 4.1. This includes two Boolean inputs, as well as a feedback signal $eFoo$, which is both an output as well as an input for the next computation. For a given simulation time, the value of $eFoo$ used within the table is the preceding time step's value.

4.1 Decomposition

Owing to their origins as a tool for software requirements specification, by convention, tabular expressions typically describe a single mathematical expression (Jin and Parnas 2010). When evaluated, they typically compute one output for some function. This particular form of tabular expressions has been shown to be especially valuable for software documentation, as well as for the specification of software functions in general (Wassyng and Janicki 2003). For simplicity, these types of tables are used as the basis of the examples provided.

$$fComputeFooBar(bCond_1, bCond_2 : bool, eFoo : enum) : bool, enum =$$

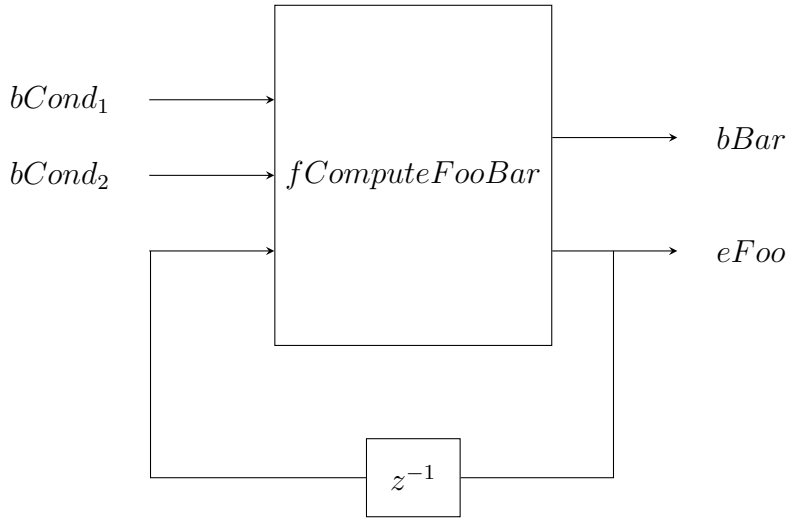

Figure 4.2: System signal flow overview of Table 4.1

Furthermore, decomposing tables to compute only a single output enforces modularization of software components and separates concerns in terms of actions. It is also often the case that separate tables yield larger reductions. We have found that when two or more actions are assigned for single decisions, there is often repetition amongst the actions. Various combinations of the output values are returned from the table, thus the same outputs are included across multiple actions. In this situation, if a well-founded enough argument can be made for the avoidance of decomposition, then this methodology can still be applied by performing the simplifications herein on the action numbers themselves, as an abstraction of the action. Notwithstanding, it is recommended to decompose a table which computes several ouputs in order to uncouple them, facilitate modularity, and provide better requirements traceability. This process necessitates the decomposition of the table in terms of its output values, and implementing the logic in multiple tables. An example

of this is demonstrated in Tables 4.2a and 4.2b, where these two tables now implement the computation of each output of Table 4.1 separately.

Nevertheless, there may exist some situations where a decomposition on the actions is more involved, or not possible. This may be the case when multiple actions are assigned per decision rule, or when the actions themselves are complex code. Structures such as `for` loops, `if` statements, and persistent variables are permissible for action specification, as Matlab code in general is supported in an action definition. This is a slight departure from traditional decision tables, where it was primarily the case that an action was simply implemented as a Boolean value indicating the presence of an action. In spite of this, complex action code can be moved to a separate function, and then replaced with a call to that function in the action table, thus avoiding operations using actions later in the methodology. Handling actions in this manner does not modify the original functionality of the design, as Matlab code generated from Stateflow truth table implements each action as a separate function regardless. Furthermore, if the Matlab code embedded in an action or condition performs supplementary decision logic, it may also be the case that it should in fact, be implemented outside of the table, and fed in as an additional input. In summation, the solutions to such cases are largely relative to the problem, thus we direct the reader to handle them according to their discretion. Although complex actions are indeed supported by Stateflow truth tables, their simplification, however cosmetic, will facilitate the processes prescribed later in the methodology.

The remainder of the steps in the methodology are shown using $fFoo$, implemented in Table 4.2a, as the basis for demonstration. These steps could equally have been performed on $fBar$ in Tables 4.2b.

$fFoo(bCond_1, bCond_2 : bool, eFoo : enum) : enum =$

		Decisions				
#	Conditions	D_1	D_2	D_3	D_4	D_5
1	$eFoo == cEnum_a$	T	F	F	F	-
2	$eFoo == cEnum_b$	F	T	T	F	-
3	$eFoo == cEnum_c$	F	F	F	T	-
4	$bCond_1$	-	T	-	F	-
5	$bCond_2$	-	-	F	F	-
	Actions	1	1	2	3	1

#	Actions
1	$eFoo = cEnum_a;$
2	$eFoo = cEnum_b;$
3	$eFoo = cEnum_c;$

(a) Truth table resulting from decomposition w.r.t. output $eFoo$

$fBar(bCond_1, bCond_2 : bool, eFoo : enum) : bool =$

		Decisions				
#	Conditions	D_1	D_2	D_3	D_4	D_5
1	$eFoo == cEnum_a$	T	F	F	F	-
2	$eFoo == cEnum_b$	F	T	T	F	-
3	$eFoo == cEnum_c$	F	F	F	T	-
4	$bCond_1$	-	T	-	F	-
5	$bCond_2$	-	-	F	F	-
	Actions	1	1	2	3	1

#	Actions
1	$eBar = true;$
2	$eBar = false;$
3	$eBar = true;$

(b) Truth table resulting from decomposition w.r.t. output $bBar$

Table 4.2: Decomposition of a truth table into multiple tables

4.2 Transformation Into Tabular Expression

As part of the methodology, we define a technique for transforming Stateflow truth tables into their equivalent tabular expressions. As a result of the differences between these two formalisms, namely disjointness, unique challenges arise which prevent this from being a straightforward process. The following steps must be taken in order to convert from the truth table notation to tabular expression notation:

1. **Insert Conditions and Actions** A visually significant difference between the two notations is their representation of conditions. As introduced in Section 2.3.1, classical decision tables (as well as Stateflow truth tables) place conditions in a separate section of the table with the values that they take on delineated by T, F and - within the decision rules. On the other hand, in tabular expressions conditions are referred to by name without the use of Boolean constants. Therefore, we substitute conditions directly into the decision rules. For each decision rule, T values are replaced straightforwardly by the condition they denote as being true. As for F values, the condition logic is negated. When conditions are don't cares, this signifies that their evaluation is not necessary in that circumstance, thus, no replacement is necessary. The same treatment is given to actions. Instead of referring to them using indices, actual values are introduced into the table.

   Due to spacing constraints, let us define labels l_cEnum_a, l_cEnum_b, and l_cEnum_c where,

   $$l\_cEnum_a \implies eFoo == cEnum_a$$

		Decisions				
#	Conditions	D_1	D_2	D_3	D_4	D_5
1	$eFoo == cEnum_a$	l_cEnum_a	$\neg l\_cEnum_a$	$\neg l\_cEnum_a$	$\neg l\_cEnum_a$	-
2	$eFoo == cEnum_b$	$\neg l\_cEnum_b$	l_cEnum_b	l_cEnum_b	$\neg l\_cEnum_b$	-
3	$eFoo == cEnum_c$	$\neg l\_cEnum_c$	$\neg l\_cEnum_c$	$\neg l\_cEnum_c$	l_cEnum_c	-
4	$bCond_1$	-	$bCond_1$	-	$\neg bCond_1$	-
5	$bCond_2$	-	-	$\neg bCond_2$	$\neg bCond_2$	-
	Actions	$cEnum_a$	$cEnum_a$	$cEnum_b$	$cEnum_c$	$cEnum_a$

#	Actions
1	$eFoo = enum_a;$
2	$eFoo = enum_b;$
3	$eFoo = enum_c;$

Table 4.3: After substitution of conditions and actions

$$l\_cEnum_b \implies eFoo == cEnum_b$$
$$l\_cEnum_c \implies eFoo == cEnum_c$$

For any label l_cEnum_i, negation is applied as $\neg l\_cEnum_i \implies eVar \sim= cEnum_i$.

All pertinent information is condensed into the decision rule section of the decision table. Therefore, the action table and condition section are no longer required for interpreting the logic, and are stripped away for ease of manipulation in upcoming steps. Similarly, headers as well as all other formatting are also removed, as tabular expressions have their individual visual organization and conventions. This results in Table 4.4

2. **Transpose** Making use of the simplistic, intermediary table implementation given in Table 4.4, we straightforwardly transpose the entire table, as one would with a matrix, i.e., rows are changed into columns. This orients the data such that decision rules are now positioned horizontally, as in tabular expression notation. The result of this step is shown in

l_cEnum_a	$\neg l\_cEnum_a$	$\neg l\_cEnum_a$	$\neg l\_cEnum_a$	-
$\neg l\_cEnum_b$	l_cEnum_b	l_cEnum_b	$\neg l\_cEnum_b$	-
$\neg l\_cEnum_c$	$\neg l\_cEnum_c$	$\neg l\_cEnum_c$	l_cEnum_c	-
-	$bCond_1$	-	$\neg bCond_1$	-
-	-	$\neg bCond_2$	$\neg bCond_2$	-
$cEnum_a$	$cEnum_a$	$cEnum_b$	$cEnum_c$	$cEnum_a$

Table 4.4: After removal of action table, condition section, and formatting details

l_cEnum_a	$\neg l\_cEnum_b$	l_cEnum_c	-	-	$cEnum_a$
$\neg l\_cEnum_a$	l_cEnum_b	$\neg l\_cEnum_c$	$bCond_1$	-	$cEnum_a$
$\neg l\_cEnum_a$	l_cEnum_b	$\neg l\_cEnum_c$	-	$\neg bCond_2$	$cEnum_b$
$\neg l\_cEnum_a$	$\neg l\_cEnum_b$	l_cEnum_c	$\neg bCond_1$	$\neg bCond_2$	$cEnum_c$
-	-	-	-	-	$cEnum_a$

Table 4.5: Transposing to re-orient decision rules

Table 4.5.

3. **Group Related Conditions** As explained in Section 2.3.3, decision tables are not well suited for expressing relationships between conditions. Certain types of non-Boolean conditions are implemented through the use of several conditions. As a result, transforming a decision table to a tabular expression requires that these related conditions are grouped together in a single column. Negated conditions can be omitted entily because they are implied due to the nature of the relationship. Table 4.6 shows the result of grouping together condition checks for a single enumeration type. These conditions are placed under a single column as they are in fact related, and are mutually exclusive.

4. **Ensure Disjointness and Completeness** The table derived from the previous step cannot yet be considered as a tabular expression due to no-

l_cEnum_a	-	-	$cEnum_a$
l_cEnum_b	$bCond_1$	-	$cEnum_a$
l_cEnum_b	-	$\neg bCond_2$	$cEnum_b$
l_cEnum_c	$\neg bCond_1$	$\neg bCond_2$	$cEnum_c$
-	-	-	$cEnum_a$

Table 4.6: Grouping of enumeration type conditions in a single column

ticeable shortcomings. Tabular expressions satisfy two properties: completeness and disjointness, as described in Section 2.3.2. It is necessary to adjust and augment conditions in order to fulfill these requirements. The difference between decision tables and tabular expressions is the support of overlapping conditions is the decision tables, as well as their method of implementing non-Boolean types. This is a consequence of the following situations:

(a) **Else Condition** In tabular expressions, conditions must be disjoint, and the implementation of an else condition must also conform to this behaviour.

(b) **Left-to-Right Semantics** This method of interpretation allows for overlapping conditions between two or more decision rules. This occurs as a result of one or more don't care conditions being included in the original decision table.

Therefore, making use of the definition of the disjointness property given in Section 2.3.2, these situations must be identified and made to be disjoint. In tabular expressions, there is no mechanism for the implementation of catch-all else conditions. Conditions which take the place of an else case must be the negation of another condition. Thus, removal

of this rule is required, and the remaining cases which would have been covered by the else must be added as new conditions, otherwise, the table is no longer complete. These new conditions maintain the else action as their output, thereby maintaining equivalent logic, but with a more precise specification. A similar treatment is given to other overlapping rules. Those rules which overlap are expanded in terms of the don't care condition which transgresses the disjointness property. The resulting expanded rules will either describe new rules not implemented as of yet by any other rule, or will implement the overlapping portion of the rule, meaning that they are already implemented and do not require addition into the table. These are therefore discarded, while the remaining are kept and given the else condition's output.

In Table 4.6, row 2 and 3 are non-disjoint. Inspecting the original Table 4.1, we see this overlap also occurs in (D_2, D_3). In considering the left-to-right semantics, row 3 is deemed the overlapping row as it the the right-most decision rule in Table 4.1. Thus, to remedy this situation it is expanded with respect to its overlapping don't care condition. Table 4.7 gives the updated table with the addition of new, explicit rows to augment the table for fulfilment of the disjointness and completeness properties.

At the conclusion of this step, the table will satisfy both the disjointness and completeness properties, making them proper tabular expressions.

5. **Formatting** What remains of the conversion to tabular expression is the visual formatting of the data such that it complies with the standard notation set out for tabular expressions. This step does not affect the

$eFoo == cEnum_a$	-	-	$cEnum_a$
$eFoo == cEnum_b$	$bCond_1$	-	$cEnum_a$
$eFoo == cEnum_b$	$\neg bCond_1$	$bCond_2$	$cEnum_a$
$eFoo == cEnum_b$	$\neg bCond_1$	$\neg bCond_2$	$cEnum_b$
$eFoo == cEnum_c$	$bCond_1$	-	$cEnum_a$
$eFoo == cEnum_c$	$\neg bCond_1$	$bCond_2$	$cEnum_a$
$eFoo == cEnum_c$	$\neg bCond_1$	$\neg bCond_2$	$cEnum_c$

Table 4.7: Addition of rules to satisfy completeness and disjointness

$$fFoo(bCond_1, bCond_2 : bool, eFoo : enum) : enum =$$

<table>
<tr><th colspan="3" rowspan="2">Conditions</th><th>Result</th></tr>
<tr><th>$eFoo$</th></tr>
<tr><td colspan="3">$eFoo == cEnum_a$</td><td>$cEnum_a$</td></tr>
<tr><td rowspan="3">$eFoo == cEnum_b$</td><td colspan="2">$bCond_1$</td><td>$cEnum_a$</td></tr>
<tr><td rowspan="2">$\neg bCond_1$</td><td>$bCond_2$</td><td>$cEnum_a$</td></tr>
<tr><td>$\neg bCond_2$</td><td>$cEnum_b$</td></tr>
<tr><td rowspan="3">$eFoo == cEnum_c$</td><td colspan="2">$bCond_1$</td><td>$cEnum_a$</td></tr>
<tr><td rowspan="2">$\neg bCond_1$</td><td>$bCond_2$</td><td>$cEnum_a$</td></tr>
<tr><td>$\neg bCond_2$</td><td>$cEnum_c$</td></tr>
</table>

Table 4.8: Formatted tabular expression

logic of the table, but rather presents it in a more readable manner. As shown in Table 2.4, cells which perform the same condition checks, are grouped together across rows. Likewise, with don't care conditions, cells are amalgamated across columns. For clarity, result values are placed under a heading containing the output name. The final, formatted tabular expression is shown in Table 4.8.

As a result, the original decision table has been transformed into an alternative representation, that of a tabular expression.

4.3 Simplification

Equipped with a proper tabular expression, avenues to simplify the logic which they implement are investigated. Given any tabular expression, not necessarily as a result of the previous steps, these techniques can be utilized to simplify the tabular expression in terms of reducing its size and logical complexity.

Depending on the function, each of the following simplifications may not always be applicable. Those dealing with enumeration types are useful specifically with functions describing some mode-driven system behaviour. Moreover, there is no prescribed order in which to perform these simplifications, as it is largely dependant on the type of function at hand, and as such will be different for each. When multiple simplifications are present for a table, their application can be done in various orders, resulting in functionally equivalent, but different tables. Depending on the requirements one wishes to bring out in the tables, one tabular representation may be of more use than another. This is demonstrated in Table 4.13. It is exactly for this purpose that the software requirements extracted from the model through reverse-engineering, as shown in Section 3, are required. Requirements are to be used as a guide for the application of simplification techniques, as well as refactoring in general.

We go on to discuss the various simplification techniques used in our experience refactoring Stateflow truth tables. Separate examples are first given to clearly illustrate each type of simplification. An application of the simplification techniques on the example is demonstrated afterwards.

4.3.1 Removal of Don't Care Conditions

This simplification reduces the table in terms of conditions required for the computation of the output. If a condition does not have an affect in the outcome of a decision, it can be removed entirely from its computation, that is, be treated as a don't care. The simplification minimizes the table both visually and logically.

The method of identifying instances where this simplification could apply is through the inspection of distinct paths through the table which lead to the same output. It is most efficient to start by finding multiple instances of the same output in the Results column, and then moving backwards through the conditions that were required to reach these outputs. If these paths are the same, save for one condition, they can potentially be combined. This one difference between rows must take the form of same condition being checked for different values, yet still ultimately yielding the same output regardless. This is considered to be a don't care condition, as it does not affect the output. For example, Table 4.9 contains two rows that encompass the same conditions aside from $Condition_j$. Regardless of whether $Condition_j$ is true or false, the outcome is $Action_1$. Therefore, $Condition_j$ does not influence the decision result, and can be considered as a don't care condition. The two rows are merged into one, as shown in Table 4.10, and the don't care condition is removed from the row.

For example, Table 4.9 illustrated a case where the first two rows are identical, aside from $Condition_j$. Although this condition check is different for the two rows, we see that it does not affect the output, as it is evaluated to $Action_1$ in both cases. Therefore these rows can be combined to form Table 4.10.

Conditions				Result
				Output
$Condition_1$	...	$Condition_i$	$Condition_j$	$Action_1$
			$\neg Condition_j$	$Action_1$
⋮				⋮

Table 4.9: Candidate don't care simplification on a Boolean condition

Conditions			Result
			Output
$Condition_1$	...	$Condition_i$	$Action_1$
⋮			⋮

Table 4.10: Application of don't care simplification on a Boolean condition

Table 4.9 provides a simple example where a Boolean condition with a cardinality of 2 is combined. This simplification is also applicable to conditions which have a greater number of potential values. For example, enumeration types with a cardinality of 4 can also be combined, so long as 4 rows are merged to together, such that they cover the complete range of that condition's type. That is to say, for any given condition $Condition_i$, $|Condition_i|$ rows must be combined, where each row contains a distinct value of the condition and jointly they cover the range of the condition.

Depending on whether or not the condition is nested, this simplification will require a horizontal rearrangement of conditions. Tables 4.9 and 4.10 displayed a removal of a don't care condition in a trivial manner where the candidate condition for removal was situated at the end of the row. Table 4.11 presents an example where the don't care condition $Condition_j$ is nested between other conditions, and is also used in the computation of other rows. Upon removal of the don't care condition in a subset of rows, the horizontal ordering of the

conditions is rearranged to reflect the dominance of the remaining conditions that actually affect evaluation. This is shown in Tables 4.11 and 4.12.

Conditions					Result
					Output
$Condition_1$	...	$Condition_i$	$Condition_j$	$Condition_k$	$Action_1$
				$\neg Condition_k$	$Action_2$
			$\neg Condition_j$	$Condition_k$	$Action_1$
				$\neg Condition_k$	$Action_3$
⋮			⋮		⋮

Table 4.11: Candidate for nested Boolean don't care simplification

In Table 4.11 there are two paths by which one can arrive at the output $Action_1$. Inspection ascertains that the paths are indeed the same, expect for $Condition_j$, taking on true and false values in these cases. It is evident that $Condition_j$'s evaluation does not affect the outcome, and therefore it can be treated as a don't care condition. To do this, $Condition_j$ is removed from rows 1 and 3 (but not 2 and 4). At this point the two rows no longer have any distinct conditions, and can be merged into one row containing only $Condition_k$. Afterwards, $Condition_k$ is shifted one column to the left, reflecting that it is a condition which is more a dominant, in the sense that it must be evaluated in more decisions than $Condition_j$, thus serving as a more prominent decision point. More details on horizontal ordering are given in Section 4.3.4. The resulting simplified and ordered table is given in Table 4.12.

Conditions					Result
					Output
$Condition_1$	...	$Condition_i$	$Condition_k$		$Action_1$
			$\neg Condition_k$	$Condition_j$	$Action_2$
				$\neg Condition_j$	$Action_3$
⋮			⋮		⋮

Table 4.12: Application of don't care simplification on a nested Boolean condition

At times, it will be possible to perform multiple simplifications on the current form of the tabular expression. An example is show in Table 4.13.

Conditions		Result *Output*
$Condition_1$	$Condition_2$	$Action_1$
	$\neg Condition_2$	$Action_1$
$\neg Condition_1$	$Condition_2$	$Action_1$
	$\neg Condition_2$	$Action_2$

(a) Multiple simplifications present

Conditions		Result *Output*
$Condition_1$		$Action_1$
$\neg Condition_1$	$Condition_2$	$Action_1$
	$\neg Condition_2$	$Action_2$

(b) Simplification of $Condition_1$

Conditions		Result *Output*
$Condition_2$		$Action_1$
$\neg Condition_2$	$Condition_1$	$Action_1$
	$\neg Condition_1$	$Action_2$

(c) Simplification of $Condition_2$

Table 4.13: Example of a situation where multiple simplifications can take place

Two diverging refactorings for the same original table are demonstrated. Table 4.13b is refactored in order to convey the dominance of $Condition_1$ through the preservation of its horizontal ordering in the row, whereas Table 4.13c communicates that $Condition_2$ is the dominate condition on which the decision logic should more heavily rely.

4.3.2 Partitioning

The constraint on merging the complete range of a condition, given in Section 4.3.1 can be relaxed when $|Condition_i| > 2$ and when grouping a subset of the rows is desired. This is particularly useful for enumerated types 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. Instead, the combination of rows with regards to some partition may be beneficial, and will still result in a minimization. Actions are still required to be the same across the conditions to be partitioned. Choosing how to partition should be done with consideration of the requirements, in order to determine which subset of values are to be grouped. The condition partitioning is demonstrated on Table 4.14 and reflected in Table 4.15.

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

Table 4.14: Candidate for partitioning simplification

In Table 4.14 we can see that the evaluations of $Condition_1$ and $Condition_2$ are the same for both $cEnum_1$ and $cEnum_i$, and thus we can merge these two rows, albeit not with the entirety of the enumerators.

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

Table 4.15: Application of partitioning simplification

4.3.3 Generalization of States for No Change

This simplification is also particularly useful for systems where modes are implemented using enumeration types, and are fed back into the system. As with the removal of don't cares, discussed in Section 4.3.1, cells are combined when containing conditions that check enumeration variables where the same conditions must be checked for each. However, in this case, instead of the actions being the same across the rows which are to be merged, they correlate to the mode-checking condition. The actions indicate that the system is to remain in the current state/mode, and so it is possible to eliminate the condition checking of the mode. The enumeration variable name takes the place of the multiple conditions used for checking the mode.

Conditions		Result $eVar$
$eVar == cEnum_1$	$Condition_1$	$cEnum_1$
	$\neg Condition_1$	$cEnum_2$
$eVar == cEnum_2$	$Condition_1$	$cEnum_2$
	$\neg Condition_1$	$cEnum_4$
$eVar == cEnum_3$	$Condition_1$	$cEnum_3$
	$\neg Condition_1$	$cEnum_4$
$\vdots$	$\vdots$	$\vdots$
$eVar == cEnum_n$	$Condition_1$	$cEnum_n$
	$\neg Condition_1$	$cEnum_3$

Table 4.16: Candidate for no change generalization simplification

Conditions		Result $eVar$
$eVar == cEnum_1$	$Condition_1$	NC
	$\neg Condition_1$	$cEnum_2$
$eVar == cEnum_2$	$Condition_1$	NC
	$\neg Condition_1$	$cEnum_4$
$eVar == cEnum_3$	$Condition_1$	NC
	$\neg Condition_1$	$cEnum_4$
$\vdots$	$\vdots$	$\vdots$
$eVar == cEnum_n$	$Condition_1$	NC
	$\neg Condition_1$	$cEnum_3$

Table 4.17: Highlighting no change (NC) cases in Table 4.16

As seen in Table 4.16, rows $1, 3, 5, \ldots, n-1$ make use of the enumeration checking condition for the output of the same enumerator. As highlighted in Table 4.17, this implements a no change (NC) in output when $Condition_1$ is true. Additionally, other conditions to check are the same between the rows. Thus we can eliminate the check on the variable $eVar$ when $Condition_1$ is true because it does not factor into the decision process, and then the no change action is implemented by moving the variable name into the results column.

This is done for those rows which were using enumeration checking for a simple pass-through to implement the no change action. The remaining rows which prescribe some action other than the no change, remain the same, however a horizontal reordering of conditions may be necessary, as is the case in this example. The resulting table of this simplification is given as Table 4.18.

Conditions		Result $eVar$
$Condition_1$		$eVar$
$\neg Condition_1$	$eVar == cEnum_1$	$cEnum_2$
	$eVar == cEnum_2$	$cEnum_4$
	$eVar == cEnum_3$	$cEnum_4$
	⋮	⋮
	$eVar == cEnum_n$	$cEnum_3$

Table 4.18: Application of no change generalization simplification

4.3.4 Condition Ordering

Manipulation of the condition orderings, both vertical 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 in software. Although no order is implied by tabular expressions, their implementation in the Tabular Expression Toolbox (TET) does enforce an ordering when it comes to code generation. Moreover, in Section 4.4 a method of translating tabular expressions back into Stateflow truth tables is described. Condition ordering of the decisions can be also use to manipulate the generated code of these tables. For this we can take advantage of Stateflow's capability of allowing for a preview of generated content of a given decision table. Additionally, when performing simplifications, altering these orderings

proves beneficial in the identification of repetitive conditions. Conditions can be rearranged in order to better distinguish patterns amongst rows.

Vertical Evaluation speed of the expression can be increased by forcing specific conditions to be evaluated as early as possible (Jin and Parnas 2010). The vertical arrangement of conditions can be adjusted to increase efficiency in the evaluation of cases. Moving decisions with the most don't cares, or fewest amount of condition evaluations, to the upper rows allows these cases to be evaluated earlier on. This allows for a gradual evaluation of conditions, such that the smallest subset of conditions required to be evaluated are checked first. This method of ordering the conditions is perhaps easier to understand in terms of the conditions of `if`-statements.

Horizontal Similarly, the horizontal ordering of conditions can be arranged to take advantage of the nested evaluation of conditions. If it is advantageous to evaluate certain conditions infrequently, nesting them is a good course of action such that they are only checked after other conditions are evaluated, thus minimizing the cases said conditions are evaluated. Additionally, enforcing a horizontal ordering as a visual means of representing the dominance of conditions with respect to the decision aids in understanding the relationship between variables of the system. This will prove especially beneficial in the case where these tables are used for requirements and documentation purposes.

4.3.5 Compound Simplification

A compound simplification strategy is applied whereby an already simplified row is expanded, followed by some other simplification which makes use of

the newly introduced rows. An example of this is given in Tables 4.19, where it is already simplified with respect to $Condition_1$ in the first row. Further simplifying the table such that the remaining rows are also simplified with respect to $Condition_1$ is hindered due to the simplification already present in the table. As a result, the simplified row is expanded to Table 4.20 in order to facilitate a simplification to reduce the table further. Don't care conditions are straightforwardly expanded into all values of their type, with the resulting action being the same across all newly added rows. The final simplified table is shown in Table 4.21 as is a result of applying the simplification outlined in Section 4.3.1. The conditions pertaining to the checking of $eVar$ are identified as a don't care for the computation of $Action_1$, and thus is it removed and a reordering of the conditions is performed.

Conditions		Result *Output*
$eVar == cEnum_1$		$Action_1$
$eVar == cEnum_2$	$Condition_1$	$Action_1$
	$\neg Condition_1$	$Action_2$
$eVar == cEnum_3$	$Condition_1$	$Action_1$
	$\neg Condition_1$	$Action_3$

Table 4.19: Candidate for compound refactoring

Conditions		Result *Output*
$eVar == cEnum_1$	$Condition_1$	$Action_1$
	$\neg Condition_1$	$Action_1$
$eVar == cEnum_2$	$Condition_1$	$Action_1$
	$\neg Condition_1$	$Action_2$
$eVar == cEnum_2$	$Condition_1$	$Action_1$
	$\neg Condition_1$	$Action_3$

Table 4.20: Expanded table during compound refactoring

Conditions		Result *Output*
$Condition_1$		$Action_1$
$\neg Condition_1$	$eVar == cEnum_1$	$Action_1$
	$eVar == cEnum_2$	$Action_2$
	$eVar == cEnum_3$	$Action_3$

Table 4.21: Simplfied table as a result of compound refactoring

This strategy also proves useful for the general restructuring of already simplified tables, and is beneficial when altering an existing table to display a requirement in a more evident manner. Therefore, it is not necessary to use table expansion as solely a means for further simplification.

As shown in Figure 4.1, the simplification step is an iterative process. The application of multiple instances of the aforementioned simplifications can take place until the table is no longer reducible, and even then, a compound simplification strategy can take place to facilitate alternative simplifications. It is left to the reader's discretion and intuition to determine the ideal form of a table with respect to the requirements, and thus the stopping criteria is relative.

Finally, returning to the example, simplifications are performed on Table 4.8. The table's results column is examined for repetitions of actions in order to identify possibilities for simplification. Firstly, $cEnum_b$ and $cEnum_c$ are not integral to the computation of $cEnum_a$ as the output of rows 2 and 5. Both rows yield the same result, and the rows are identical save for checking of the enumeration conditions. However, $|eVar| = 3$, and so combining rows can only be accomplished by joining all 3 conditions: $cEnum_a$, $cEnum_b$, and $cEnum_c$. The first row containing $cEnum_a$ is already simplified, and does not

conform to the form required in order for it to be amalgamated with the two rows we wish to simplify. As a result, a compound refactoring technique is employed to expand the first row into the desired format. This is shown in Table 4.22. Afterwards, the simplification is able to take place, and is shown in Table 4.23.

$$fComputeFoo(bCond_1, bCond_2 : bool, eFoo : enum) : enum =$$

Conditions			Result $eFoo$
$eFoo == cEnum_a$	$bCond_1$		$cEnum_a$
	$\neg bCond_1$	$bCond_2$	$cEnum_a$
		$\neg bCond_2$	$cEnum_a$
$eFoo == cEnum_b$	$bCond_1$		$cEnum_a$
	$\neg bCond_1$	$bCond_2$	$cEnum_a$
		$\neg bCond_2$	$cEnum_b$
$eFoo == cEnum_c$	$bCond_1$		$cEnum_a$
	$\neg bCond_1$	$bCond_2$	$cEnum_a$
		$\neg bCond_2$	$cEnum_c$

Table 4.22: Expanded $cEnum_a$ for a compound refactoring approach

$$fComputeFoo(bCond_1, bCond_2 : bool, eFoo : enum) : enum =$$

Conditions			Result $eFoo$
$bCond_1$			$cEnum_a$
$\neg bCond_1$	$eFoo == cEnum_a$	$bCond_2$	$cEnum_a$
		$\neg bCond_2$	$cEnum_a$
	$eFoo == cEnum_b$	$bCond_2$	$cEnum_a$
		$\neg bCond_2$	$cEnum_b$
	$eFoo == cEnum_c$	$bCond_2$	$cEnum_a$
		$\neg bCond_2$	$cEnum_c$

Table 4.23: Removal of enumeration type don't care condition

Examining the results column of Table 4.23, it is evident that there still exists several distinct paths through the table, leading to the same output. Namely, $cEnum_a$ is computed multiple times, and has the potential to the simplified. Again, the simplification process has reached a point where two possible simplifications can take place. Most obvious is that $bCond_2$ in row 2 and 3 does not affect the logic of the computation, and can be treated as a don't care condition. Secondly, the use of $cEnum_a$, $cEnum_b$, and $cEnum_c$ in its computation is also unnecessary. Therefore, striving to refactor the table as compactly as possible, the latter simplification is applied, as it affects the greatest amount of rows. This is the second application of this technique, and displays the iterative nature of the simplification step of the methodology. The tabular expression becomes further simplified with the repetitive application of simplification strategies. This simplification is shown in Table 4.24.

$$fComputeFoo(bCond_1, bCond_2 : bool, eFoo : enum) : enum =$$

Conditions			Result $eFoo$
$bCond_1$			$cEnum_a$
$\neg bCond_1$	$bCond_2$		$cEnum_a$
	$\neg bCond_2$	$eFoo == cEnum_a$	$cEnum_a$
		$eFoo == cEnum_b$	$cEnum_b$
		$eFoo == cEnum_c$	$cEnum_c$

Table 4.24: Removal of enumeration type don't care condition during a second iteration

Incrementally, the previous simplifications have moved the enumeration conditions to the rightmost position in the tabular expression, expressing that they are less prominent conditions. Consequently, it is straightforward to recognize the simplification defined in Section 4.3.3. The enumeration values

are used as simple means of designating a no change situation. Therefore, the variable name can be used to directly implement this functionality, as shown in Table 4.25.

$$fComputeFoo(bCond_1, bCond_2 : bool, eFoo : enum) : enum =$$

Conditions		Result $eFoo$
$bCond_1$		$cEnum_a$
$\neg bCond_1$	$bCond_2$	$cEnum_a$
	$\neg bCond_2$	$eFoo$

Table 4.25: Generalization of conditions to implement a no change situation

Upon completion on these steps, a refactored tabular expression is attained that is logically equivalent to the original decision table, yet simplified. No further simplifications are applicable to further reduce the tabular expression in terms of logic or size. This form can now be implemented directly into a Simulink model using the TET. Additionally, a significant benefit of tabular expressions is their effectiveness as software documentation structures. Thus, inclusion of the simplified tabular expression into requirements documents is another use for the tabular expression in its current form. Nevertheless, our intentions are to simplify Stateflow truth tables, and so Section 4.4 goes on to construct the corresponding Stateflow truth table, such that it can be implemented in the original model.

4.4 Transformation Into Stateflow Truth Table

This section describes how a tabular expression is transformed into an equivalent Stateflow truth table, that is, a decision table. These steps serve to revert the tabular expression attained in the previous steps, back into the form in which it was originally implemented. If applied to the resultant table at the end of Section 4.2, it will reproduce the original Stateflow truth table, presented as Table 4.1. In general, this portion of the methodology can be used as a stand-alone method of converting any horizontal tabular expression into a decision table. The following are the necessary steps for the transformation:

1. **Remove Tabular Expression Formatting** Prior to performing structural transformation of decision tables to tabular expressions, tabular expression-specific formatting conventions are stripped from the table. This includes headers. Other formatting techniques such as the grouping of conditions vertically across cells are expanded. Don't care conditions grouped horizontally are made explicit for each column in which they appear.

$bCond_1$	-	$cEnum_a$
$\neg bCond_1$	$bCond_2$	$cEnum_a$
$\neg bCond_1$	$\neg bCond_2$	$eFoo$

Table 4.26: Generalization of conditions to implement a no change situation

2. **Transpose** Transposing the table once more, we utilize the mathematical property $(A^T)^T = A$, where A is a table, to arrive back to a form that

is readily implementable in a Stateflow truth table. Shown in Table 4.27. This step re-orients the decision rules such that they are compatible with decision table form, in that they are parsed in a top-down fashion.

$bCond_1$	$\neg bCond_1$	$\neg bCond_1$
-	$bCond_2$	$\neg bCond_2$
$cEnum_a$	$cEnum_a$	$eFoo$

Table 4.27: Transposing a tabular expression for Stateflow truth table notation

3. **Construct Condition and Action Sections** Condition and action sections are populated with data from the current table. This is a simple rearrangement of data into its respective section, as is required for decision table format. This information will be required to interpret the logic once the next step is performed, where the decision rules are converted into Boolean constants. Create and populate the condition section with the condition tables. Reconstruct the action table, such that it encompasses the actions used within the table. In doing so, indices for actions are assigned. Therefore, replace the action values in the action entry section with these indices. In comparison to the original Stateflow truth table, the number of actions may have experienced a reduction, specifically because of the no change generalization simplification defined in Section 4.3.3. This simplification also results in a new action being defined, which implements a no change action. Visual formatting conventions are also included at this stage of the transformation process. This step is illustrated in Table 4.28.

		Decisions		
#	Conditions	D_1	D_2	D_3
1	$bCond_1$	$bCond_1$	$\neg bCond_1$	$\neg bCond_1$
2	$bCond_2$	-	$bCond_2$	$\neg bCond_2$
	Actions	1	1	2

#	Actions
1	$eFoo = cEnum_a;$
2	$eFoo = eFoo;$

Table 4.28: Moving conditions and actions to their respective locations

$$fComputeFoo(bCond_1, bCond_2 : bool, eFoo : enum) : enum =$$

		Decisions		
#	Conditions	D_1	D_2	D_3
1	$bCond_1$	T	F	F
2	$bCond_2$	-	T	F
	Actions	1	1	2

#	Actions
1	$eFoo = cEnum_a;$
2	$eFoo = eFoo;$

Table 4.29: Condition replacement with Boolean values

4. **Replace conditions with Boolean constants** Decision rules are converted into Boolean form. Where a condition is true, it is replaced with a T. Where a condition is false, include F. Otherwise, where there is a don't care condition, insert the - don't care symbol. This is demonstrated in Table 4.29.

At the conclusion of transforming a tabular expression to a decision table, completeness and disjointness are preserved. Transforming back to Stateflow truth table allows for the comparison of the original versus refactored versions in order to observe the effects of the methodology. This final Stateflow truth

table can also now be implemented directly into Simulink, using the same Stateflow truth table construct. In comparison to the original, the final table depicted in Table 4.29 is visibly reduced in size as well as logical complexity. Proving the equivalence between the original and refactored decision table, or its equivalent tabular expression, can be accomplished using third party tools such as Prototype Verification System (PVS) by SRI. Nevertheless, the heuristics defined in this chapter were designed to provide and easy to follow process of performing guided refactoring, and the equivalence between intermediary steps is easy to see.

Chapter 5

Application of the Truth Table Transformation Methodology in the Automotive Industry

In this chapter, the Stateflow truth table transformation methodology defined in Chapter 4 is applied to real, industrial models. In our collaboration with an industrial partner, one of our primary objectives is refactoring models with the intention of increasing design maintainability, reusability and testability. Focusing specifically on tabular implementations of decision logic found in Matlab Simulink, this largely entails redesign of large, complex tables by way of simplification and the introduction of modularity. We have found through its application on production-level designs, that it effectively transforms tables to a readable format more favourable for software documentation, while also reducing the size and complexity of decision logic.

The effectiveness of this methodology is demonstrated, and its success is gauged through testing and comparison of metrics on the original and refac-

tored designs. SDV provides cyclomatic complexity, condition, decision, and MC/DC metrics for truth table analysis, while also producing detailed reports outlining coverage of the table's cells. Due to its superior Stateflow truth table support, as explored in Section 3.1.3, we employ the use of SDV to perform a comparison between original and refactored tables with respect to their test suites and model coverage achieved.

As previously mentioned, the models are proprietary information and as a result cannot be disclosed in their original form. The signal names have been obfuscated to enforce anonymity.

5.1 Example 1: Request Arbitration From $cState_1$

The first example we apply the methodology on, is the four Stateflow truth tables shown in Figure 3.2. Each of these tables is responsible for performing arbitration of driver requests from four different states. The tables consider the current status of the system, i.e. the previous arbitrated status, as well as other vehicle conditions, all of which are inputs into each table. We apply the methodology on the first of these tables, shown in Table 5.1. This table performs request arbitration from $cState_1$.

5.1.1 Application

Decomposition The two tables resulting from the decomposition are shown as Table 5.2 and Table 5.3. The remaining steps of the methodology are demonstrated on Table 5.2.

f_ArbRequestFromState1(eDrvrRequest:enum, bFaulty, eArbRequest:bool): enum
=

#	Condition	1	2	3	4	5	6	7	8	9	10	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

#	Action
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	$eArbRequest$=$cState_1$; $bActionRequired$=$true$

Table 5.1: Request Arbitration Example - Original Stateflow truth table for request arbitration from State1

f_ArbRequestFromState1(eDrvrRequest:enum, bFaulty, eArbRequest:bool): enum
=

#	Condition	1	2	3	4	5	6	7	8	9	10	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

#	Action
1	$eArbRequest$=$cState_1$
2	$eArbRequest$=$cState_2$
3	$eArbRequest$=$cState_3$
4	$eArbRequest$=$cState_4$
5	$eArbRequest$=$cState_1$

Table 5.2: Request Arbitration Example - Truth table resulting from decomposition w.r.t. *eArbRequest*

f_ActionRequiredFromState1(eDrvrRequest:enum, bFaulty, eArbRequest:bool): enum =

#	Condition	1	2	3	4	5	6	7	8	9	10	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

#	Action
1	$bActionRequired{=}false$
2	$bActionRequired{=}false$
3	$bActionRequired{=}false$
4	$bActionRequired{=}false$
5	$bActionRequired{=}true$

Table 5.3: Request Arbitration Example - Truth table resulting from decomposition w.r.t. *bActionRequired*

Transformation Into Tabular Expression Next, conditions are inserted into decision rules, and the formatting is stripped. Additionally, due to spacing constraints, we define labels l_cState_1, l_cState_2, l_cState_3, and l_cState_4 where,

$$l\_cState_1 \implies eDrvrRequest == cState_1$$

$$l\_cState_2 \implies eDrvrRequest == cState_2$$

$$l\_cState_3 \implies eDrvrRequest == cState_3$$

$$l\_cState_4 \implies eDrvrRequest == cState_4.$$

These labels simply denote the conditions with brevity. Also, for any label l_cEnum_i, negation is applied as $\neg l\_cEnum_i \implies eVar \sim= cEnum_i$. The resulting table is given in Table 5.4.

Transposing is then done straightforwardly by changing rows into columns. This is shown in Table 5.5. Table 5.6 shows how to related enumeration con-

f_ArbRequestFromState1(eDrvrRequest:enum, bFaulty, eArbRequest:bool): enum =

l_cState_1	$\neg l\_cState_1$	$\neg l\_cState_1$	$\neg l\_cState_1$	$\neg l\_cState_1$	$\neg l\_cState_1$	$\neg l\_cState_1$	$\neg l\_cState_1$	$\neg l\_cState_1$	-
$\neg l\_cState_2$	l_cState_2	$\neg l\_cState_2$	l_cState_2	$\neg l\_cState_2$	$\neg l\_cState_2$	l_cState_2	$\neg l\_cState_2$	$\neg l\_cState_2$	-
$\neg l\_cState_3$	$\neg l\_cState_3$	l_cState_3	$\neg l\_cState_3$	l_cState_3	$\neg l\_cState_3$	$\neg l\_cState_3$	l_cState_3	$\neg l\_cState_3$	-
$\neg l\_cState_4$	$\neg l\_cState_4$	l_cState_4	$\neg l\_cState_4$	$\neg l\_cState_4$	l_cState_4	$\neg l\_cState_4$	$\neg l\_cState_4$	l_cState_4	-
-	*bCmpntUnlocked*	*bCmpntUnlocked*	-	-	-	-	-	-	-
-	$\neg bFaulty$	$\neg bFaulty$	$bFaulty$	$bFaulty$	*bFaulty*	-	-	-	-
$cState_1$	$cState_2$	$cState_3$	$cState_4$	$cState_1$	$cState_1$	$cState_1$	$cState_1$	$cState_1$	$cState_1$

Table 5.4: Request Arbitration Example - Inserting conditions and removing formatting

ditions are grouped, and then Table 5.7 shows the table once disjointness and completeness are introduced.

f_ArbRequestFromState1(eDrvrRequest:enum, bFaulty, eArbRequest:bool): enum

=

l_cState_1	$\neg l\_cState_2$	$\neg l\_cState_3$	$\neg l\_cState_4$	-	-	$cState_1$
$\neg l\_cState_1$	l_cState_2	$\neg l\_cState_3$	$\neg l\_cState_4$	$bCmpntUnlocked$	$\neg bFaulty$	$cState_2$
$\neg l\_cState_1$	$\neg l\_cState_2$	l_cState_3	$\neg l\_cState_4$	$bCmpntUnlocked$	$\neg bFaulty$	$cState_3$
$\neg l\_cState_1$	$\neg l\_cState_2$	$\neg l\_cState_3$	l_cState_4	$bCmpntUnlocked$	$\neg bFaulty$	$cState_4$
$\neg l\_cState_1$	l_cState_2	$\neg l\_cState_3$	$\neg l\_cState_4$	-	$bFaulty$	$cState_1$
$\neg l\_cState_1$	$\neg l\_cState_2$	l_cState_3	$\neg l\_cState_4$	-	$bFaulty$	$cState_1$
$\neg l\_cState_1$	$\neg l\_cState_2$	$\neg l\_cState_3$	l_cState_4	-	$bFaulty$	$cState_1$
$\neg l\_cState_1$	l_cState_2	$\neg l\_cState_3$	$\neg l\_cState_4$	-	-	$cState_1$
$\neg l\_cState_1$	$\neg l\_cState_2$	l_cState_3	$\neg l\_cState_4$	-	-	$cState_1$
$\neg l\_cState_1$	$\neg l\_cState_2$	$\neg l\_cState_3$	l_cState_4	-	-	$cState_1$
-	-	-	-	-	-	$cState_1$

Table 5.5: Request Arbitration Example - Transposed table

f_ArbRequestFromState1(eDrvrRequest:enum, bFaulty, eArbRequest:bool): enum

=

l_cState_1	-	-	$cState_1$
l_cState_2	$bCmpntUnlocked$	$\neg bFaulty$	$cState_2$
l_cState_3	$bCmpntUnlocked$	$\neg bFaulty$	$cState_3$
l_cState_4	$bCmpntUnlocked$	$\neg bFaulty$	$cState_4$
l_cState_2	-	$bFaulty$	$cState_1$
l_cState_3	-	$bFaulty$	$cState_1$
l_cState_4	-	$bFaulty$	$cState_1$
l_cState_2	-	-	$cState_1$
l_cState_3	-	-	$cState_1$
l_cState_4	-	-	$cState_1$
-	-	-	$cState_1$

Table 5.6: Request Arbitration Example - Enumerations grouped

f_ArbRequestFromState1(eDrvrRequest:enum, bFaulty, eArbRequest:bool): enum

=

l_cState_1	-	-	$cState_1$
l_cState_2	$bCmpntUnlocked$	$\neg bFaulty$	$cState_2$
l_cState_2	$bCmpntUnlocked$	$bFaulty$	$cState_1$
l_cState_2	$\neg bCmpntUnlocked$	-	$cState_1$
l_cState_3	$bCmpntUnlocked$	$\neg bFaulty$	$cState_3$
l_cState_3	$bCmpntUnlocked$	$bFaulty$	$cState_1$
l_cState_3	$\neg bCmpntUnlocked$	-	$cState_1$
l_cState_4	$bCmpntUnlocked$	$\neg bFaulty$	$cState_4$
l_cState_4	$bCmpntUnlocked$	$bFaulty$	$cState_1$
l_cState_4	$\neg bCmpntUnlocked$	-	$cState_1$

Table 5.7: Request Arbitration Example - Disjoint and Complete

f_ArbRequestFromState1(eDrvrRequest:enum, bFaulty, eArbRequest:bool): enum

=

Conditions			Result
			$eArbRequest$
l_cState_1			$cState_1$
l_cState_2	$bCmpntUnlocked$	$\neg bFaulty$	$cState_2$
		$bFaulty$	$cState_1$
	$\neg bCmpntUnlocked$	-	$cState_1$
l_cState_3	$bCmpntUnlocked$	$\neg bFaulty$	$cState_3$
		$bFaulty$	$cState_1$
	$\neg bCmpntUnlocked$	-	$cState_1$
l_cState_4	$bCmpntUnlocked$	$\neg bFaulty$	$cState_4$
		$bFaulty$	$cState_1$
	$\neg bCmpntUnlocked$	-	$cState_1$

Table 5.8: Request Arbitration Example - Formatting

After formatting, the table is considered to be a valid tabular expression. Table 5.9 shows the equivalent horizontal tabular expression of Table 5.2. Labels are also removed.

f_ArbRequestFromState1(eDrvrRequest:enum, bFaulty, eArbRequest:bool): enum

=

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

Table 5.9: Request Arbitration Example - Horizontal table defining request arbitration from State1

Simplification Simplifications are now performed on Table 5.9. Firstly, $State_1$ is always the output when $bFaulty$ is true, no matter if the value of $bCmpntUnlocked$ or $eDrvrRequest$. Therefore, in Table 5.10 $bFaulty$ is rearranged horizontally such that it checked before $bCmpntUnlocked$, delineating its dominance as a condition. This is done again for $eDrvrRequest$, however a compound simplification approach is taken such that the first row, $eDrvrRequest == cState_1$, is expanded to be the same form as the other rows. This allows for another application of the reordering, and is shown in Table 5.11. Then $bFaulty$ is rearranged again, so that it is checked before $eDrvrRequest$, making it the most dominant condition. This is illustrated in Table 5.12

f_ArbRequestFromState1(eDrvrRequest:enum, bFaulty, bCmpntUnlocked: bool): enum =

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

Table 5.10: Request Arbitration Example - Reordering $bFaulty$ horizontally

fArbRequestFromState1(eDrvrRequest:enum, bFaulty, bCmpntUnlocked:bool):
enum =

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

Table 5.11: Request Arbitration Example - Row expansion to facilitate further simplification

fArbRequestFromState1(eDrvrRequest:enum, bFaulty, bCmpntUnlocked:bool):
enum =

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

Table 5.12: Request Arbitration Example - Reordering $bFaulty$ horizontally a second time

It is evident from Table 5.12 that checking $\neg bCmpntUnlocked$ always results in $cState_1$ regardless of $eDrvrRequest$. We apply a similar treatment and move the $bCmpntUnlocked$ condition such that it comes before $eDrvrRequest$, as seen in Table 5.13. Doing so clearly show that in fact, $eDrvrRequest$ is not necessary at all in rows 2-5. In all four of these cases, the output is $cState_1$, therefore it can be considered as a don't care condition. Table 5.14 demonstrates its removal, resulting in the elimination of 3 superfluous rows.

fArbRequestFromState1(eDrvrRequest:enum, bFaulty, bCmpntUnlocked:bool): enum =

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

Table 5.13: Request Arbitration Example - Reordering *bCmpntUnlocked* horizontally

fArbRequestFromState1(eDrvrRequest:enum, bFaulty, bCmpntUnlocked:bool): enum =

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

Table 5.14: Request Arbitration Example - Don't care condition removal of *eDrvrRequest*

Afterwards, there is a no change condition evident in the table's last four rows. Here, the state values are essentially passed through, and so their condition checks can be eliminated, and simply used in the output cell, as shown in Table 5.15. Additionally, the vertical ordering of the *bCmpntUnlocked* condition is manipulated such that the non-negated condition is placed first. This is simply done so that it is consistent with the notation of *bFaulty*, and is given in Table 5.16.

fArbRequestFromState1(eDrvrRequest:enum, bFaulty, bCmpntUnlocked:bool): enum =

Conditions		Result: *eArbRequest*
bFaulty		$cState_1$
$\neg bFaulty$	$\neg bCmpntUnlocked$	$cState_1$
	bCmpntUnlocked	*eDrvrRequest*

Table 5.15: Request Arbitration Example - *eDrvrRequest* no change generalization

fArbRequestFromState1(eDrvrRequest:enum, bFaulty, bCmpntUnlocked:bool): enum =

Conditions		Result: *eArbRequest*
bFaulty		$cState_1$
$\neg bFaulty$	*bCmpntUnlocked*	*eDrvrRequest*
	$\neg bCmpntUnlocked$	$cState_1$

Table 5.16: Request Arbitration Example - Reordering *bCmpntUnlocked* horizontally

Transformation into Truth Table As a result of the simplifications reducing the size of the table, this step is straightforward in comparison to Transforming the original Stateflow truth table into a tabular expression. The cells are once again transposed, and the formatting is changed to match that of Stateflow truth tables. This is illustrated in Table 5.17.

The same process was applied to the remaining three tables of the subsystem. Their steps are not outlined in detail, however the final results are included in Appendix B. Upon refactoring each table, they were implemented in the design. The resulting simplified Stateflow truth tables were shown to be equivalent to the original through the use of PVS.

fArbRequestFromState1(eDrvrRequest:enum, bFaulty, bCmpntUnlocked:bool): enum =

#	Condition	1	2	3
1	*bFaulty*	T	F	F
2	*bCmpntUnlocked*	-	T	F
	Actions	1	2	1

#	Action
1	*eArbRequest=cState*$_1$
2	*eArbRequest=eDrvrRequest*

Table 5.17: Request Arbitration Example - Equivalent Stateflow truth table

5.1.2 Results

The impact that the refactoring has had on the tables is discussed here. We investigate how refactoring and simplification affects software designs in terms of testability, complexity and requirements traceability.

Testing To compare the original and refactored tables, testing is performed using SDV, due to its superior Stateflow truth table support. All four original tables are reimplemented using the refactored tables. During testing, SDV formally analyzes the model and provides this information in coverage and analysis reports. In order to compare the two different tabular implementations with respect to their effects on testing, test case generation was performed and the coverage and analysis information were compared. Table 5.18 compares the formal analysis results of the original and refactored tables' tests, while Table 5.19 compares the achieved model coverage. SDV testing strives to maximize three types of coverage objectives: condition, decision, and MC/DC.

	Original	Refactored
Tests	7	9
Test Steps	97	48
Number of Objectives	1016	371
Objectives Satisfied	797	311
Objectives Proven Unsatisfiable	219	60
Cyclomatic Complexity	274	107

Table 5.18: Analysis of tests

	Original			Refactored		
	Satisfied	Total	Percentage	Satisfied	Total	Percentage
Condition	368	452	81%	110	140	79%
Decision	112	112	100%	95	95	100%
MCDC	141	226	62%	44	70	63%

Table 5.19: Comparison of test metrics

In examining this data, it is evident that the number of test objectives is significantly reduced by more than half. Additionally, the testing time also decreased. Although not as evident because of the overall reduction of objectives, it is also the case that the number of satisfied objectives increase from approximately 78% to 84%. The reason for this improvement is due to the simplification of the decision logic, which is confirmed by the cyclomatic complexity metric. Cyclomatic complexity decreased considerably by a factor of 2.5. In examining the Table 5.16 specifically, SDV reported a cyclomatic complexity of 10. Its accompanying table, Table **??** has a complexity of 9.

The original, Table 5.1 had a complexity of 60. Therefore, these two tables together yield a complexity of 19, which is significant lower.

No significant improvements were achieved in terms of coverages metrics, however improvements were made to testing efforts required on the part of SDV. We are primarily concerned with reducing the efforts in testing with regards to number of tests required. In Table 5.18 it appears as though the number of tests increases, and thus the refactored model is more difficult to test, however it is more important to take notice of the number of test steps required within each of these tests. Tests are a means of grouping some number of steps, and thus do not necessarily reflect more or less effort required to test a particular model. It is evident that the refactored tables require less test steps than the original. Detailed testing results describing each test are also made available in the coverage report. This data is presented in Table 5.20 and Table 5.21 respectively.

Test #	Length (s)	Test Steps	Objectives Satisfied
1	1.0	6	74
2	3.4	18	205
3	0.8	5	67
4	3.2	17	196
5	5.4	28	148
6	3.0	16	98
7	1.2	7	9
Total	18	97	797

Table 5.20: Detailed test information for subsystem with original tables

Test #	Length (s)	Test Steps	Objectives Satisfied
1	2.4	13	128
2	0.6	4	41
3	1.2	7	86
4	1.6	9	15
5	2.0	11	14
6	0.0	1	9
7	0.0	1	7
8	0.0	1	7
9	0.0	1	4
Total	7.8	48	311

Table 5.21: Detailed test information for subsystem with refactored tables

Furthermore, Table 5.19 also shows how many objectives are required of the design in order to satisfy condition, decision, and MC/DC. In all three cases, the refactored table requires less objectives.

Requirements Tracablilty Tabular expressions are more readable tabular constructs than decision tables. Additionally, they allow requirement to be more evident. Therefore they are more conducive to making requirements more traceable. This is demonstrated in Table 5.22.

$$fArbRequestFromState1(eDrvrRequest{:}enum,\ bFaulty,\ bCmpntUnlocked{:}bool):$$

$$enum =$$

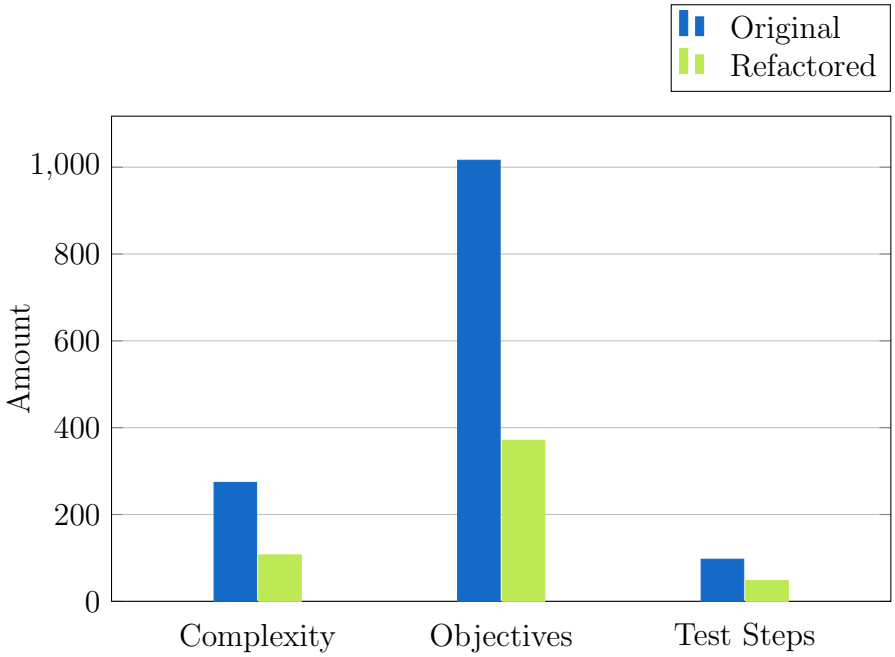

Figure 5.1: Summary of SDV analysis

Conditions		Result
		eArbRequest
bFaulty		$cState_1$
$\neg bFaulty$	*bCmpntUnlocked*	*eDrvrRequest*
	$\neg bCmpntUnlocked$	$cState_1$

Table 5.22: Visible requirement

Calibration Flexibility These tables are not in use as calibratable data. Therefore this is not considered for this system.

In summary, this methodology proves to be indeed successful at reducing cyclomatic complexity of tabular designs as well as testing efforts.

5.2 Example 2: System Status

We illustrate the aforementioned refactoring approach on another vehicle system, illustrated in Figure 5.2. This example is of considerable size,and flagged as a subsystem that is particularity odious to maintain my developers from our OEM partner. Although the logic to be simplified in this subsystem is in Stateflow truth table form, application of the proposed simplification methodology cannot be applied directly. We first describe its functionality, outline the approach by which it is to be refactored, and then demonstrate refactoring for a single table.

At a high level, the functionality of this subsystem is concerned with the computation of the state of a major vehicular system. Using several these input signals, the subsystem arbitrates the enabling of various operations and other subsystems. The overall system "state" is a combination of system and operation statuses which are either enabled or disabled. The statuses of these systems are the outputs of this subsystem, shown in Figure 5.23. In total, there are eight system/operation statuses that are computed in this subsystem.

At the core of this implementation are two tables capturing the decision logic for determining the system state. The effective purpose of these tables is to select which outputs are enabled/disabled, i.e. true/false, based on the given input. The following is a brief explanation of the process of determining the system state as it was implemented. We take interest in this computation in particular due to its heavy use of tables.

1. Two truth tables return an integer value each. The first table returns $nIndex_1$ in the range of [1,73] and the second returns an integer value $nIndex_2$ in the range [1,21]. Each output corresponds to the x and y

(row and column respectively) values of the *cEncodedState* matrix. The matrix issues system states.

2. The x and y values are fed into a 2D-selector, which returns the element of the matrix at the indices provided. This value is the system state in decimal form. Possible state values returned are {0, 128, 132, 140, 196, 204, 228, 236, 244}.

3. Allow for any overrides to take place. If there is an override, system state is 0.

4. Decode the system state value from decimal to binary. Each bit of the binary number corresponds to a output value describing the enabling/disabling of the systems/operations for that system state.

The motivations behind refactoring this system, as well as the problem formulation are as follows:

- **Design is unintuitive.** Implementation does not correlate well with the desired functionality. We see the use of three large tables: two Simulink truth tables and a 2D Matlab matrix. This approach essentially uses two tables to look up entries in a third table. From the description above, it is apparent that implementation of the functionality we desire to achieve is done so through an indirect means. Furthermore, the system states are represented numerically in an inexplicit manner, which necessitates the use of a decoder subsystem.

- **Tables are too large and complex.** The matrix is sparse, that is, largely empty. In total there are 80 non-zeros vs. 1453 zeros. Furthermore, the method of lookup is inefficient. Only rows 1, 10, 37, 46 have

non-zero values. When the first table points to an all-zero row, it is not necessary to check the second table. In this case, determining the y index is inconsequential because all columns will contain zero values regardless. Additionally, storing this large matrix as a calibration also requires a sizeable amount of memory, specifically 3000K.

- **Modifiability.** In addition to the implementation being prohibitively complex to easily facilitate changes in functionality, future changes to the system may be difficult due in part to the current implementation's reliance on the binary representation of the system state. At present, the system currently works for 8 bits, because there are 8 outputs. But in the case when another system/operation needs to be included in the arbitration logic, to accommodate the increase, more bits are required, necessitating a more intensive refactoring of the system.

- **Obfuscates vehicular requirements.** It is not readily evident from this implementation how the various vehicle systems operate. Added time and effort are required to inspect and understand how the system functions. Representing states as decimal numbers provides no information to developers in terms of what they actually represent. Lack of requirements documentation further exacerbates this problem, especially so for new developers.

All of these points lend themselves to the problems which developers have encountered: poor readability, understandability, maintainability, and testability.

5.2.1 Application

With eight systems/operations to consider as outputs, eight tables will be required. We show the refactoring of the simplest example, namely the Drive Allowed bit. This output is essentially a flag indicating whether the system is allowed to go into certain states. The remaining refactored tables are included in Appendix C.

Decomposition We start with three separate tables capturing the logic we wish to simplify, instead of a single truth table. Consequently, some additional work is required in order to apply the methodology and create tables for each output. We take a bitwise refactoring approach to decompose the system into non-coupled elements. Firstly, one of the objectives is to implement the computation of the system state in a more intuitive manner. Therefore, instead of using two tables to compute the index of a third, we represent each computation as a function which computes the status of each of the systems/operations. In the current design implementation, each system/operation status is a bit of the binary number that represents a system state. As such, we decompose the tables by isolating the logic behind each bit, and compute it separately as the output of its own table. For each output, only those inputs which affect the status of that system/operation are included, eliminating the need to include checking of conditions which are irrelevant for that system/operation. Developers will be able to calibrate functionality on a bit-by-bit basis, affording more flexibility when it comes to modifiability in the future. Furthermore, removing repetitions of conditions within tables is fundamental to the simplification methodology. Upon inspection of the two tables, we see that the outputs are unique integers and there is no repetition. Application of the methodology on

the tables as they are currently implemented will be ineffectual. Moreover, no requirements can be derived from this implementation, as the integers have no other meaning than to point to matrix entries. Therefore, a bitwise refactoring approach is employed.

		Output System/Operation Status							
		$bSystem_A$	$bSystem_B$	$bSystem_C$	$bOpr_A$	$bSystem_D$	$bSystem_E$	$bOpr_B$	$bOpr_C$
State	0	0	0	0	0	0	0	0	0
	128	1	0	0	0	0	0	0	0
	132	1	0	0	0	1	1	0	0
	196	1	1	0	0	0	1	0	0
	204	1	1	0	0	1	1	0	0
	228	1	1	1	0	0	1	0	0
	236	1	1	1	0	1	1	0	0
	244	1	1	1	1	0	1	0	0

Table 5.23: System states and their system/operation statuses

Firstly, we isolate the logic which computes this specific output. This is done by inspecting the states, and identifying those in which the system/operation is enabled (i.e. true). Table 5.23 presents the eight system states and the status of the various systems within those states.

We can see that the $bOpr_A$bit is only enabled in state 244. Consulting the matrix, we target any entries where state 244 is the prescribed output, and work backwards to the two truth tables to discover which conditions were required to arrive at these matrix entries. At this stage, it may be is possible to simplify the resulting expressions intuitively, through visual inspection. We however leave simplification to the application of the methodology. Avoiding any simplifications at this time may allow for a different set of simplification to take place during the next steps.

Derivation of Function Definition The logic derived from the matrix and tables provides the expression for each cell that is contained in the definition of the output we are examining. The following expression formalizes the logic required for the $bOpr_A$bit to be true:

$$
\begin{aligned}
bOpr_A = \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \\
&\neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge \\
&eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \\
&\neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge \\
&eSystem_D == cNotPlgdIn)
\end{aligned}
$$

With the bit now extracted and defined separately from the other logic concerning the remaining outputs, we now possess the function definition for $bOpr_A$.

Express as Tabular Expression The function definition is transformed into a tabular expression, as presented in Figure 5.24.

f_OprAState(eOprAStat, eOprBStat, $eSystem_D$, eKState:enum, bEnblCond, bProcessRun, bDrvCondMet:bool): bool =

Conditions							Result $bOpr_A$
$eOprAStat == cNotPlgIn$	$eOprBStat == cNoTools$	$\neg bEnblCond$	$\neg bProcessRun$	$bDrvCondMet$	$eSystem_D == cNotPlgdIn$	$eKState == cPos5$	$true$
						$eKState \sim= cPos5$	$false$
					$eSystem_D \sim= cNotPlgdIn$		$false$
				$bDrvCondMet$	$eSystem_D == cNotPlgdIn$	$eKState == cPos5$	$true$
						$eKState \sim= cPos5$	$false$
					$eSystem_D \sim= cNotPlgdIn$		$false$
			$bProcessRun$				$false$
		$bEnblCond$					$false$
	$eOprBStat \sim= cNoTools$						$false$
$eOprAStat \sim= cNotPlgIn$							$false$

Table 5.24: System Status Example - Extracted and isolated $bOpr_A$bit prior to simplification

Simplify Simplifications are then carried out on the table. In this simple example, there is a single simplification resulting from the identification of DrvDoorCondMet as a "don't care" condition. Therefore, it is removed from the table, and is no longer considered as a useful input for the computation of this output. The resulting table after simplification and completion of the methodology is shown in Table 5.25.

f_OprAState(eOprAStat, eOprBStat, $eSystem_D$, eKState:enum, bEnblCond, bProcessRun:bool): bool =

<table>
<tr><th colspan="6">Conditions</th><th>Result
$bOpr_A$</th></tr>
<tr><td rowspan="6">$eOprAStat == cNotPlgIn$</td><td rowspan="5">$eOprBStat == cNoTools$</td><td rowspan="4">$\neg bEnblCond$</td><td rowspan="3">$\neg bProcessRun$</td><td rowspan="2">$eSystem_D == cNotPlgdIn$</td><td>$eKState == cPos5$</td><td>$true$</td></tr>
<tr><td>$eKState \sim= cPos5$</td><td>$false$</td></tr>
<tr><td colspan="2">$eSystem_D \sim= cNotPlgdIn$</td><td>$false$</td></tr>
<tr><td colspan="3">$bProcessRun$</td><td>$false$</td></tr>
<tr><td colspan="4">$bEnblCond$</td><td>$false$</td></tr>
<tr><td colspan="5">$eOprBStat \sim= cNoTools$</td><td>$false$</td></tr>
<tr><td colspan="6">$eOprAStat \sim= cNotPlgIn$</td><td>$false$</td></tr>
</table>

Table 5.25: System Status Example - $bOpr_A$ bit simplified

Upon closer inspection of the *Result* column, it is evident that $bOpr_A$is only ever true under one specific set of conditions. This emerging property of the system can potentially be considered as a requirement. Table 5.26 provides a good demonstration of how requirements become more evident through the use of tabular expressions and our methodology for simplifying them. We amalgamate those columns which are false into a single row/cell, and similarly represent the path of conditions which lead to the true output in a single cell. This step simply emphasizes the presence of the single true case, and groups the false cases as an "else" type of condition. This form may be beneficial for use in requirements documentation.

Matlab Simulink/Stateflow currently does not have built-in support for tables in the style of tabular expressions. Nevertheless, we can simply translate back into truth table form for the purpose of implementing the refactored table.

f_OprAState(eOprAStat, eOprBStat, $eSystem_D$, eKState:enum, bEnblCond, bProcessRun:bool): bool =

Conditions	Result $bOpr_A$
$eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge eSystem_D == cNotPlgdIn \wedge eKState == cPos5$	$true$
$eOprAStat \sim= cNotPlgIn \wedge eOprBStat \sim= cNoTools \wedge bEnblCond \wedge bProcessRun \wedge eSystem_D \sim= cNotPlgdIn \wedge eKState \sim= cPos5$	$false$

Table 5.26: System Status Example - $bOpr_A$ further simplified

f_OprAState(eOprAStat, eOprBStat, $eSystem_D$, eKState:enum, bEnblCond, bProcessRun:bool): bool =

		Decisions	
#	Conditions	D_1	D_2
1	$eOprAStat == cNotPlgIn$	T	-
2	$eOprBStat == cNoTools$	T	-
3	$bEnblCond$	F	-
4	$bProcessRun$	F	-
5	$eSystem_D == cNotPlgdIn$	T	-
6	$eKState == cPos5$	T	-
	Actions	1	2

#	Action
1	$bOpr_A = true$
2	$bOpr_A = false$

Table 5.27: System Status Example - $bOpr_A$ as a Stateflow truth table

Table 5.27 gives Table 5.26 as it would be represented in Simulink/Stateflow.

5.2.2 Results

The remaining tables were simplified and implemented in Simulink/Stateflow. The refactored subsystem is shown in Figure 5.3

A comparison between this refactored version and the original design is presented here. Table 5.28 compares the tests generated for each of these implementations.

	Original	Refactored
Tests	23	6
Test Steps	1214	24
Number of Objectives	1954	498
Objectives Satisfied	1951	445
Objectives Proven Unsatisfiable	202	53
Objectives Undecided	161	0
Cyclomatic Complexity	935	248

Table 5.28: Analysis of tests

Table 5.29 illustrates the differences in testing metrics. A higher MC/DC was achieved in the refactored tables. This is also the case for Condition and Decision coverages. Additionally, the number of objectives to satisfy each was reduced significantly.

	Original			Refactored		
	Satisfied	Total	Percentage	Satisfied	Total	Percentage
Condition	1309	1672	78%	363	418	87%
Decision	297	300	99%	84	84	100%
MCDC	473	836	57%	154	209	74%

Table 5.29: Comparison of test metrics

In Tables 5.30 and 5.31, detailed information for each test step is given. Again, the refactored subsystem yields fewer tests and objectives, while also requiring less time.

Test #	Length (s)	Test Steps	Objectives Satisfied
1	43	44	1135
2	9	10	54
3	18	19	173
4	9	10	81
5	2	3	36
6	2	3	28
7	2	3	20
8	1	2	5
9	1	2	5
10	1	2	5
11	1	2	5
12	1	2	5
13	1	2	4
14	1	2	4
15	1	2	4
16	1	2	7
17	1	2	4
18	1	2	3
19	1	2	3
20	1	2	3
21	1	2	3
22	1	2	3
23	0	1	1
Total	100	1214	1591

Table 5.30: Detailed test information for subsystem with original table

Test #	Length (s)	Test Steps	Objectives Satisfied
1	2.2	12	373
2	0.8	5	62
3	0.2	2	2
4	0.2	2	2
5	0.2	2	2
6	0	1	4
Total	3.6	24	445

Table 5.31: Detailed test information for subsystem with refactored tables

Performing these analyses on the original took considerable more time to complete. First attempts timed out. Only after the analysis was allocated more time in SDV did it complete after approximately 15 minutes. Furthermore, undecided objectives were produced for the original subsystem, but not the refactored.

Calibration Flexibility In this application, the matrix we are refactoring is different for each vehicle. To accommodate flexility across all possible vehicle variants, all input combinations are allowed. However, all of these may not be relevant that for particular vehicles. Simplifying the logic so as to remove any unused inputs or impossible input combinations, although produces simpler code for that model, does not facilitate calibration across multiple vehicles. Even if this particular implementation does not use some input of the system, other vehicular software versions may require it. Moreover, implementing tables directly as Simulink truth tables negates the benefits of having separate calibration files which change depending on the vehicle design. Simulink truth

tables are in a sense, hard-coded into the implementation. Any changes must be done directly in the model, instead of a separate calibration environment.

No analysis has been done as to how much flexibility is actually needed. For this, multiple vehicular variations are needed for examination. If it was the case that all the vehicle variants were known as well as which features each enabled, then we could analyze what is really going on, such as, for example, that a vehicles charging operation is never enabled when the vehicle is not in park. On the other hand, vehicle calibrations are not always known in advance, nor can one predict what calibrations will be required for future vehicle designs as they evolve. Nevertheless, some configurations will just not be possible, and these should be identified and stripped from the implementation.

With regards to the OEM's calibration tables, they are implemented such that all inputs and all of their possible combinations are included. Naturally, this outfits the implementation with complete flexibility, allowing for the implementation of arbitrary behaviour, however, in doing so the implementation becomes completely opaque.

5.3 Summary

In this chapter we demonstrated how the methodology proposed in Chapter 4 assists in reducing design size and complexity, while also increasing testability and requirements traceability.

We have demonstrated the effectiveness of the methodology by performing an industrial case study. This refactoring methodology was applied to real, industrial designs from an automotive OEM, which demonstrated its application step-by-step. Although the methodology is not a formally defined algorithmic

approach, it is easy to follow for developers without a heavy formal methods background. This knowledge was transferred to developers from our automotive partner, who were able to understand and make of of this methodology without assistance.

Furthermore, the refactored designs created using our methodology in Section 5.1have been incorporated by our automotive industry partner into production vehicular code. This successful technology transfer clearly demonstrates the practicality and industrial relevance of this thesis.

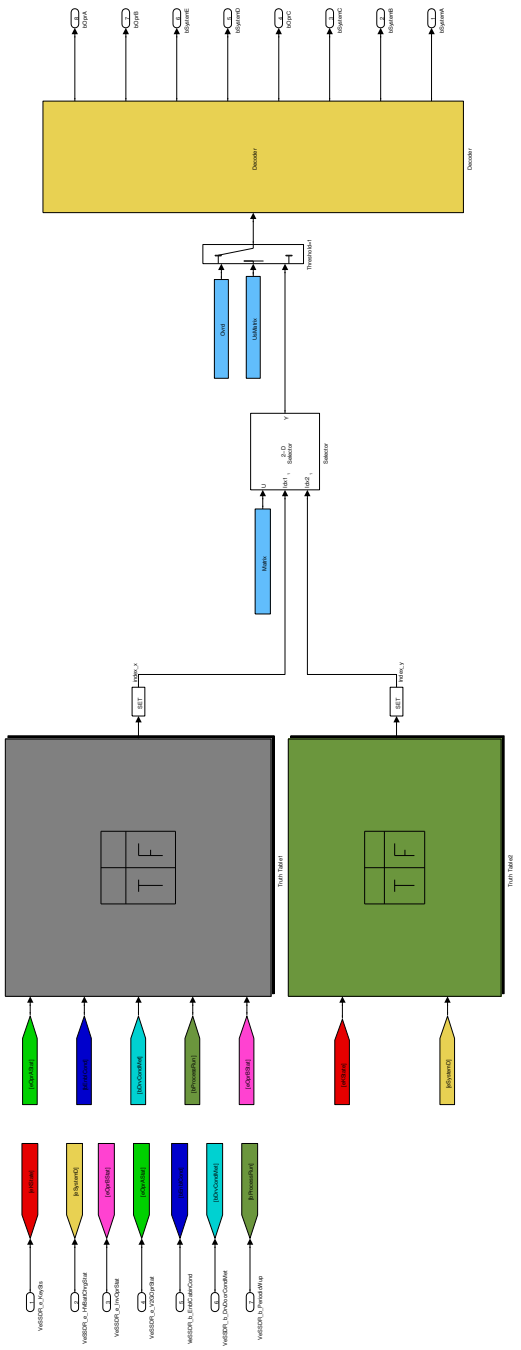

Figure 5.2: System Status Example - Subsystem undergoing requirements extraction

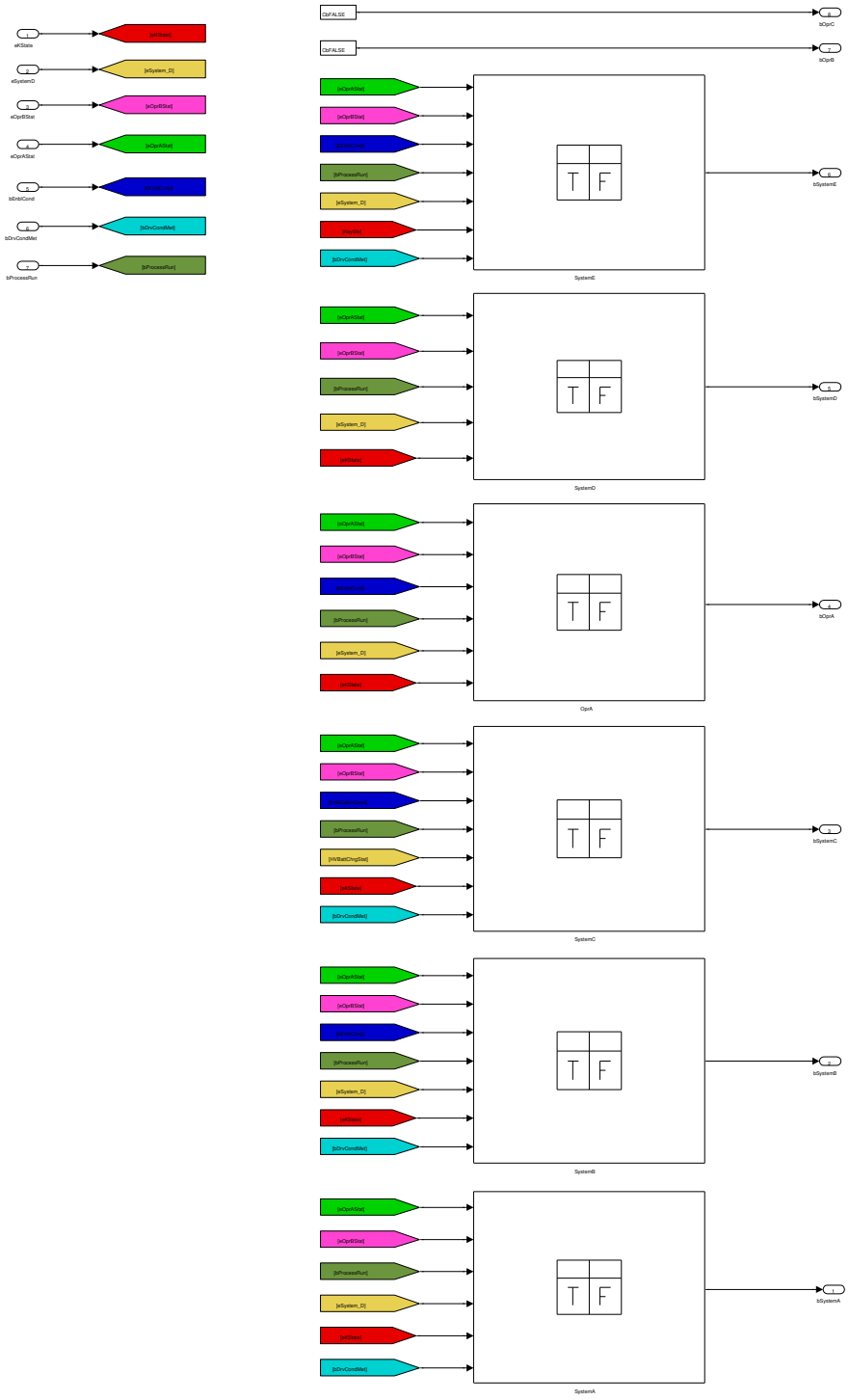

Figure 5.3: System Status Example - Refactored subsystem implementation

Chapter 6

Conclusions and Future Work

In this thesis, a novel methodology for the guided refactoring of tabular designs within the context of MBD was developed. Models are especially susceptible to becoming excessively large and complex, consequently decreasing testability and requirements traceability. Existing techniques for model refactoring techniques to address these issues are few. We examined the literature with regards to this, as well as to survey techniques for simplifying tabular constructs. Specifically, decision tables, such as those found in Simulink/Stateflow, suffer from similar problems, with the addition of non-disjointness. We then endeavoured to construct heuristics which can perform refactoring of these tabular designs, while also considering requirements. Lack of up-to-date requirements is an area of concern in the software context in general. This is especially true in MBD, and so first a methodology for reverse-engineering software requirements was applied. This resulted in a second, new methodology being created to address the deficiencies of the first. Equipped with software requirements as the guide for refactoring efforts, a methodology for simplifying tabular designs was created. Within this methodology, five transformations were proposed for

the logical simplification of tables. Furthermore, this methodology does not solely rely on Simulink/Stateflow as the basis for refactoring, and can be applied to decision tables of any origin. Likewise, the proposed simplifications can be applied to simplify tabular expressions in general. Lastly, this methodology was the basis for an industrial case study, which saw the application of this methodology and resulted in significant gains in terms of reduction of testing efforts, minimization of complexity, and requirements traceability.

6.1 Future Work

This section identifies future areas of work that are required to further the work described in this thesis. We look at the possibility for extending the work thus far presented in terms of tools, applications, and theory.

Firstly, the need for a robust toolchain is required in order to facilitate the application of this methodology, as well as create a seamless workflow for integration in the software development processes. Commercial tools and custom scripts were used throughout this work, however a seamless process is required. The methodology presented in Chapter 4 is a guided process, and so further research is needed in order to determine how much can be feasibly automated. The simplification phase of the methodology yields different, equivalent refactorings for the same table. Thus, it would be beneficial to support the viewing of multiple table forms simultaneously in order allow the user to determine the best final form. Moreover, the integration of simplification techniques in existing tools must be explored. Currently, the Matlab Simulink environments does not support any refactoring or simplification tools for models. Integrating simplification transformations in tools such as the TET would be beneficial.

Different applications of this methodology need to be pursued to further asses its robustness on industrial applications. We approached the design of this methodology with respect to the needs of the automotive domain, however MBD is an interdisciplinary approach. Other sectors such as aerospace, defence and consumer electronics also rely on models as the basis for their development process. This methodology must be applied on more models from different contexts in order to explore the robustness of this methodology, as well as accommodate the specific needs of different domains.

Refactoring Simulink models is a relatively new pursuit. Much room exists for extending the theory used as the basis for this work. Safety-critical systems are often hard real-time applications, and so more research much be done into refactoring designs which deal with such constraints. With respect to the reverse-engineering of requirements, there is a need for extending this portion of the methodology to infer timing requirements, and numerical ranges in general. Tables which implement timing behaviour also warrant further study. Furthermore, this work tackled the refactoring of Stateflow truth table blocks, as they are the major construct for capturing complex decision logic. Nevertheless, there exist other tabular constructs within Simulink/Stateflow notation which may also require a similar treatment. A survey of other constructs is required.

6.2 Closing Remarks

There is need for industrially relevant software engineering and formal methods techniques which stand the test of industry. The need for such techniques guided the work of this thesis. It was in understanding the needs of our auto-

motive OEM partner that the methodology was developed. The successful application of this methodology to their designs resulted in the refactored designs being put to use in production level vehicle control software. This technology transfer clearly demonstrates the practicality and industrial relevance of this work.

Appendices

Appendix A

Reverse-Engineered Invariants

A.1 Methodology of Ackermann et al.

1. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_4 \wedge \neg bFaulty \implies eArbRequest{==}cState_2$
2. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge bOvrdToState1 \wedge eFaultState{==}\text{cStage}_4 \wedge \neg bFaulty \implies eArbRequest{==}cState_2$
3. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_4 \wedge \neg bOvrdToState3 \wedge \neg bFaulty \implies eArbRequest{==}cState_2$
4. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_4 \wedge bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_2$
5. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge bOvrdToState1 \wedge eFaultState{==}\text{cStage}_4 \wedge \neg bOvrdToState3 \wedge \neg bFaulty \implies eArbRequest{==}cState_2$
6. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge bOvrdToState1 \wedge eFaultState{==}\text{cStage}_4 \wedge bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_2$
7. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge bOvrdToState1 \wedge eFaultState{==}\text{cStage}_4 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_2$
8. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_4 \wedge \neg bOvrdToState3 \wedge bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_2$
9. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_4 \wedge \neg bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_2$

10. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_4 \wedge bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_2$
11. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge bOvrdToState1 \wedge eFaultState{==}\text{cStage}_4 \wedge \neg bOvrdToState3 \wedge bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_2$
12. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge bOvrdToState1 \wedge eFaultState{==}\text{cStage}_4 \wedge \neg bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_2$
13. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge bOvrdToState1 \wedge eFaultState{==}\text{cStage}_4 \wedge bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_2$
14. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_4 \wedge \neg bOvrdToState3 \wedge bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_2$
15. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_2 \wedge bCmpntUnlocked \wedge bOvrdToState1 \wedge eFaultState{==}\text{cStage}_4 \wedge \neg bOvrdToState3 \wedge bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_2$

A.2 Proposed Methodology

1. $\neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
2. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bOvrdToState3 \implies eArbRequest{==}cState_1$
3. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState3 \wedge \neg bSysActive \implies eArbRequest{==}cState_1$
4. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
5. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bOvrdToState3 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
6. $\neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bSysActive \implies eArbRequest{==}cState_3$
7. $\neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_3$
8. $\neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
9. $\neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$

10. $\neg bOvrdToState1 \ \wedge \ eFaultState{==}\text{cStage}_1 \ \wedge \ bOvrdToState3 \ \wedge \ \neg bFaulty \implies eArbRequest{==}cState_3$
11. $\neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
12. $\neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
13. $\neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bSysActive \implies eArbRequest{==}cState_3$
14. $\neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
15. $\neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
16. $\neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
17. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \implies eArbRequest{==}cState_1$
18. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \implies eArbRequest{==}cState_1$
19. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bSysActive \implies eArbRequest{==}cState_1$
20. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
21. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
22. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
23. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \implies eArbRequest{==}cState_1$
24. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \implies eArbRequest{==}cState_1$
25. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState3 \implies eArbRequest{==}cState_1$
26. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bSysActive \implies eArbRequest{==}cState_1$

27. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
28. $\neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bSysActive \implies eArbRequest{==}cState_3$
29. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bOvrdToState3 \implies eArbRequest{==}cState_1$
30. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bSysActive \implies eArbRequest{==}cState_1$
31. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
32. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
33. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bOvrdToState3 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
34. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bFaulty \implies eArbRequest{==}cState_1$
35. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bFaulty \implies eArbRequest{==}cState_1$
36. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState3 \wedge \neg bFaulty \implies eArbRequest{==}cState_1$
37. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_1$
38. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
39. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bOvrdToState3 \wedge \neg bFaulty \implies eArbRequest{==}cState_1$
40. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_1$
41. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
42. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_1$
43. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$

44. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
45. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
46. $eCurrState{==}cState_1 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
47. $\neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bSysActive \implies eArbRequest{==}cState_3$
48. $\neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_3$
49. $\neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
50. $\neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
51. $eDrvrRequest{==}cState_1 \wedge bOvrdToState3 \implies eArbRequest{==}cState_3$
52. $eCurrState{==}cState_3 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \implies eArbRequest{==}cState_3$
53. $eCurrState{==}cState_3 \wedge bOvrdToState3 \wedge \neg bFaulty \implies eArbRequest{==}cState_3$
54. $eCurrState{==}cState_3 \wedge bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
55. $eCurrState{==}cState_3 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
56. $eDrvrRequest{==}cState_1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
57. $eCurrState{==}cState_3 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_3$
58. $eCurrState{==}cState_3 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
59. $eCurrState{==}cState_3 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
60. $bOvrdToState1 \wedge \neg bOvrdToState3 \implies eArbRequest{==}cState_1$
61. $bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \implies eArbRequest{==}cState_1$
62. $bOvrdToState1 \wedge \neg bOvrdToState3 \wedge \neg bFaulty \implies eArbRequest{==}cState_1$

63. $bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
64. $eCurrState{==}cState_3 \wedge \neg bOvrdToState1 \wedge bOvrdToState3 \implies eArbRequest{==}cState_3$
65. $eDrvrRequest{==}cState_1 \wedge \neg bOvrdToState1 \wedge bOvrdToState3 \implies eArbRequest{==}cState_3$
66. $eDrvrRequest{==}cState_1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \implies eArbRequest{==}cState_3$
67. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \implies eArbRequest{==}cState_1$
68. $eCurrState{==}cState_3 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bOvrdToState3 \implies eArbRequest{==}cState_3$
69. $eCurrState{==}cState_3 \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \implies eArbRequest{==}cState_3$
70. $eCurrState{==}cState_3 \wedge \neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bFaulty \implies eArbRequest{==}cState_3$
71. $eCurrState{==}cState_3 \wedge \neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
72. $eDrvrRequest{==}cState_1 \wedge \neg bCmpntUnlocked \wedge bOvrdToState3 \wedge \neg bSysActive \implies eArbRequest{==}cState_3$
73. $eDrvrRequest{==}cState_1 \wedge \neg bCmpntUnlocked \wedge bOvrdToState3 \wedge \neg bFaulty \implies eArbRequest{==}cState_3$
74. $eDrvrRequest{==}cState_1 \wedge \neg bCmpntUnlocked \wedge bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
75. $eDrvrRequest{==}cState_1 \wedge \neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bFaulty \implies eArbRequest{==}cState_3$
76. $eDrvrRequest{==}cState_1 \wedge \neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
77. $eDrvrRequest{==}cState_1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bSysActive \implies eArbRequest{==}cState_3$
78. $eDrvrRequest{==}cState_1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bFaulty \implies eArbRequest{==}cState_3$
79. $eDrvrRequest{==}cState_1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
80. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \implies eArbRequest{==}cState_1$

81. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bSysActive \implies eArbRequest{==}cState_1$
82. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
83. $eCurrState{==}cState_3 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \implies eArbRequest{==}cState_3$
84. $eCurrState{==}cState_3 \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \wedge bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
85. $eCurrState{==}cState_3 \wedge \neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
86. $eDrvrRequest{==}cState_1 \wedge \neg bCmpntUnlocked \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_3$
87. $eDrvrRequest{==}cState_1 \wedge \neg bCmpntUnlocked \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
88. $eDrvrRequest{==}cState_1 \wedge \neg bCmpntUnlocked \wedge bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
89. $eDrvrRequest{==}cState_1 \wedge \neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_3$
90. $eDrvrRequest{==}cState_1 \wedge \neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
91. $eDrvrRequest{==}cState_1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \implies eArbRequest{==}cState_3$
92. $eDrvrRequest{==}cState_1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
93. $eDrvrRequest{==}cState_1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
94. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \implies eArbRequest{==}cState_1$
95. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bSysActive \implies eArbRequest{==}cState_1$
96. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$

97. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bSysActive \implies eArbRequest{==}cState_1$
98. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
99. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
100. $eCurrState{==}cState_3 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bFaulty \implies eArbRequest{==}cState_3$
101. $eCurrState{==}cState_3 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
102. $eCurrState{==}cState_3 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge \neg bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
103. $eCurrState{==}cState_3 \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
104. $eDrvrRequest{==}cState_1 \wedge \neg bCmpntUnlocked \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
105. $eDrvrRequest{==}cState_1 \wedge \neg bOvrdToState1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
106. $eDrvrRequest{==}cState_1 \wedge eFaultState{==}\text{cStage}_1 \wedge bOvrdToState3 \wedge \neg bSysActive \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$
107. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bSysActive \implies eArbRequest{==}cState_1$
108. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
109. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
110. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$
111. $eCurrState{==}cState_3 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bFaulty \wedge \neg bActionRequired \implies eArbRequest{==}cState_3$

112. $eCurrState{==}cState_1 \wedge eDrvrRequest{==}cState_4 \wedge \neg bCmpntUnlocked \wedge \neg bOvrdToState1 \wedge eFaultState{==}\text{cStage}_1 \wedge \neg bOvrdToState3 \wedge \neg bSysActive \wedge \neg bActionRequired \implies eArbRequest{==}cState_1$

Appendix B

Driver Request Arbitration Example

B.1 Request Arbitration From $cState_2$

The original Stateflow truth table is shown in Table B.1 with its equivalent and simplified tabular expression in Table B.2. Table B.3 is the implementation of the simplified tabular expression in Stateflow truth table form.

B.2 Request Arbitration From $cState_3$

The original Stateflow truth table is shown in Table B.4 with its equivalent and simplified tabular expression in Table B.5. Table B.6 is the implementation of the simplified tabular expression in Stateflow truth table form.

fArbRequestFromState2(eDrvrRequest:enum, bOvrdToState1, bFaulty: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	T	F	F	F	F	F	-
2	$eDrvrRequest == cState_2$	F	T	F	F	F	T	F	T	F	F	-
3	$eDrvrRequest == cState_3$	F	F	T	F	F	F	F	F	T	F	-
4	$eDrvrRequest == cState_4$	F	F	F	T	F	F	T	F	F	T	-
5	$bOvrdToState1$	F	F	-	F	T	T	T	F	F	F	-
6	$bFaulty$	-	F	F	F	F	F	F	T	T	T	-
	Actions	1	2	3	4	3	3	3	5	5	5	2

#	Action
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	$eArbRequest$=$cState_1$; $bActionRequired$=$true$

Table B.1: Original Stateflow truth table for request arbitration from $cState_2$

fArbRequestFromState2(eDrvrRequest:enum, bOvrdToState1, bOvrdToState3:bool): enum =

Conditions		Result
		$eArbRequest$
$bOvrdToState3$	$bOvrdToState1$	$cState_2$
	$\neg bOvrdToState1$	$cState_3$
$\neg bOvrdToState3$	$bOvrdToState1$	$cState_1$
	$\neg bOvrdToState1$	$eDrvrRequest$

Table B.2: Simplified tabular expression for request arbitration from $cState_2$

B.3 Request Arbitration From $cState_4$

The original Stateflow truth table is shown in Table B.7 with its equivalent and simplified tabular expression in Table B.8. Table B.9 is the implementation of the simplified tabular expression in Stateflow truth table form.

*fArbRequestFromState2(eDrvrRequest:enum, bOvrdToState*1*, bOvrdToState*3*:bool): enum =*

		Decisions			
#	Conditions	D_1	D_2	D_3	D_4
1	*bOvrdToState*3	T	T	F	F
2	*bOvrdToState*1	T	F	T	F
	Actions	2	3	1	4

#	Action
1	$eArbRequest{=}cState_1$
2	$eArbRequest{=}cState_2$
3	$eArbRequest{=}cState_3$
4	*eArbRequest=eDrvrRequest*

Table B.3: Equivalent Stateflow truth table for request arbitration from $cState_2$

*fArbRequestFromState3(eDrvrRequest:enum, bSysActive, bOvrdToState*3*, bOvrdToState*1*, 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	T	F	F	-
2	$eDrvrRequest == cState_2$	F	T	F	F	T	F	F	F	T	F	-
3	$eDrvrRequest == cState_3$	F	F	T	F	F	T	F	F	F	F	-
4	$eDrvrRequest == cState_4$	F	F	F	T	F	F	T	F	F	T	-
5	*bSysActive*	-	T	-	T	-	-	-	-	-	-	-
6	*bOvrdToState*3	F	F	-	F	-	-	-	T	T	T	-
7	*bOvrdToState*1	-	F	F	F	T	T	T	F	F	F	-
8	*bCmpntUnlocked*	T	T	-	T	-	-	-	-	-	-	-
	Actions	1	2	3	4	5	5	5	5	3	3	3

#	Action
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	$eArbRequest{=}cState_1$; $bActionRequired{=}true$

Table B.4: Original Stateflow truth table for request arbitration from $cState_3$

*fArbRequestFromState3(eDrvrRequest:enum, bOvrdToState*1*, bOvrdToState*3*, bCmpntUnlocked, bSysActive:bool): enum =*

					eArbRequest
$eDrvrRequest == cState_1$	*bOvrdToState*3				$cState_3$
	$\neg$*bOvrdToState*3	*bCmpntUnlocked*			*eDrvrRequest*
		$\neg$*bCmpntUnlocked*			$cState_3$
$eDrvrRequest \sim= cState_1$	*bOvrdToState*1				$cState_1$
	$\neg$*bOvrdToState*1	*bOvrdToState*3			$cState_3$
		$\neg$*bOvrdToState*3	$\neg$*bCmpntUnlocked*		$cState_3$
			bCmpntUnlocked	*bSysActive*	*eDrvrRequest*
				$\neg$*bSysActive*	$cState_3$

Table B.5: Simplified tabular expression for request arbitration from $cState_3$

*fArbRequestFromState3(eDrvrRequest:enum, bOvrdToState*1*, bOvrdToState*3*, bCmpntUnlocked, bSysActive:bool): enum =*

		Decisions							
#	Conditions	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8
1	$eDrvrRequest == cState_1$	T	T	T	F	F	F	F	-
2	*bOvrdToState*3	T	F	F	-	T	F	F	-
3	*bOvrdToState*1	-	-	-	T	F	F	F	-
4	*bCmpntUnlocked*	-	T	F	-	-	F	T	-
5	*bSysActive*	-	-	-	-	-	-	T	-
	Actions	2	3	2	1	2	2	3	2

#	Action
1	$eArbRequest{=}cState_1$
2	$eArbRequest{=}cState_3$
3	*eArbRequest=eDrvrRequest*

Table B.6: Equivalent Stateflow truth table for request arbitration from $cState_3$

*fArbRequestFromState4(eDrvrRequest:enum, bOvrdToState*1*, bOvrdToState*3*: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	T	F	F	F	F	F	-
2	$eDrvrRequest == cState_2$	F	T	F	F	F	T	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	*bOvrdToState*3	F	F	-	F	T	T	T	F	F	F	-
6	*bOvrdToState*1	-	F	F	F	-	F	F	T	T	T	-
	Actions	1	2	3	4	3	3	3	5	5	5	4

#	Action
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	$eArbRequest$=$cState_1$; $bActionRequired$=$true$

Table B.7: Original Stateflow truth table for request arbitration from $cState_4$

*fArbRequestFromState4(eDrvrRequest:enum, bOvrdToState*1*, bOvrdToState*3*:bool): enum =*

Conditions			Result
			eArbRequest
*bOvrdToState*3	$eDrvrRequest == cState_1$		$cState_3$
	$eDrvrRequest \sim= cState_1$	*bOvrdToState*1	$cState_4$
		$\neg$*bOvrdToState*1	$cState_3$
$\neg$*bOvrdToState*3		*bOvrdToState*1	$cState_1$
		$\neg$*bOvrdToState*1	*eDrvrRequest*

Table B.8: Simplified tabular expressions for request arbitration from $cState_4$

*fArbRequestFromState4(eDrvrRequest:enum, bOvrdToState*1*, bOvrdToState*3*:bool): enum =*

		Decisions				
#	Conditions	D_1	D_2	D_3	D_4	D_5
1	*bOvrdToState*3	T	T	T	F	-
2	$eDrvrRequest == cState_1$	T	F	F	-	-
3	*bOvrdToState*1	-	T	F	T	-
	Actions	2	3	2	1	4

#	Action
1	$eArbRequest{=}cState_1$
2	$eArbRequest{=}cState_3$
3	$eArbRequest{=}cState_4$
4	*eArbRequest*=*eDrvrRequest*

Table B.9: Equivalent Stateflow truth table for request arbitration from $cState_4$

Appendix C

System Status Example

C.1 $bSystem_A$

Decision logic enables $bSystem_A$in states 128, 132, 140, 196, 204, 228, 236, and 244. Due to the excessively large nature of the logical expression, it is not included here.

C.2 $bSystem_B$

The original decision logic of the matrix enables the $bSystem_B$ in states 196, 204, 228, 236, and 244. The function definition for this behaviour is given in Figure C.1. It has been simplified to the Stateflow truth table shown as

Table C.1.

$$
\begin{aligned}
&bSystem_B = \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cPlgInNOOP \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cPlgInNOOP) \vee
\end{aligned}
\tag{C.1}
$$

$$
\begin{aligned}
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cRun)
\end{aligned}
\tag{C.2}
$$

Figure C.1: $bSystem_B$ function definition

C.3 $bSystem_C$

The original decision logic of the matrix enables the $bSystem_C$ in states 140, 204, and 236. The function definition for this behaviour is given in Figure C.2.

		Decisions						
#	Conditions	D_1	D_2	D_3	D_4	D_5	D_6	D_7
1	*eOprAStat == cNotPlgIn*	T	T	T	T	T	T	-
2	*eOprBStat == cNoTools*	T	T	T	T	T	T	-
3	*bProcessRun*	F	F	F	F	F	F	-
4	*bEnblCond*	F	F	T	T	T	T	-
5	*bDrvCondMet*	-	-	T	F	F	F	-
6	*eKState == cPos1* $\vee$ *eKState == cPos2*	T	F	-	T	T	F	-
7	$eSystem_D == cNotPlgdIn$	-	-	-	T	F	-	-
	Actions	2	1	1	2	1	1	2

#	Action
1	$bSystem_B = true$
2	$bSystem_B = false$

Table C.1: $bSystem_B$ simplified Stateflow truth table

It has been simplified to the Stateflow truth table shown as Table C.2.

$$
\begin{aligned}
&bSystem_C = \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cNotReq) \vee
\end{aligned}
\tag{C.3}
$$

$$
\begin{aligned}
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotPlgdIn) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cPlgInNOOP) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cRun)
\end{aligned}
\tag{C.4}
$$

Figure C.2: $bSystem_C$ function definition

		Decisions						
#	Conditions	D_1	D_2	D_3	D_4	D_5	D_6	D_7
1	$eOprAStat == cNotPlgIn$	T	T	T	T	T	T	-
2	$eOprBStat == cNoTools$	T	T	T	T	T	T	-
3	$bProcessRun$	F	F	F	F	F	F	-
4	$bEnblCond$	F	F	T	F	T	T	-
5	$bDrvCondMet$	-	-	T	F	F	F	-
6	$eKState == cPos4 \vee eKState == cPos5$	F	T	-	T	T	F	-
7	$eSystem_D == cNotPlgdIn$	-	-	-	-	T	F	-
	Actions	2	1	1	1	1	1	2

#	Action
1	$bSystem_C = true$
2	$bSystem_C = false$

Table C.2: $bSystem_C$ simplified Stateflow truth table

C.4 $bSystem_D$

The original decision logic of the matrix enables the $bSystem_D$ in states 140, 204, and 236. The function definition for this behaviour is given in Figure C.3. It has been simplified to the Stateflow truth table as Table C.3.

$$
\begin{aligned}
&bSystem_D = \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge \neg bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge \neg bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cNotReq) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos1 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos2 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos3 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos4 \wedge eSystem_D == cRun) \vee \\
&(eOprAStat == cNotPlgIn \wedge eOprBStat == cNoTools \wedge bEnblCond \wedge \neg bProcessRun \wedge bDrvCondMet \wedge eKState == cPos5 \wedge eSystem_D == cRun)
\end{aligned}
$$

Figure C.3: $bSystem_D$ function definition

#	Conditions	Decisions					
		D_1	D_2	D_3	D_4	D_5	D_6
1	*eOprAStat == cNotPlgIn*	T	T	T	T	T	-
2	*eOprBStat == cNoTools*	T	T	T	T	T	-
3	*bProcessRun*	F	F	F	F	F	-
4	$eSystem_D$ *== cNotPlgdIn*	T	F	F	F	T	-
5	$eSystem_D$ *== cPlgInNOOP*	-	T	F	F	F	-
6	$eSystem_D$ *== cNotReq*	-	-	T	T	F	-
7	$eSystem_D$ *== cRun*	-	-	-	-	T	-
8	*eKState == cPos3* $\vee$ *eKState == cPos4* $\vee$ *eKState == cPos5*	-	-	F	T	-	-
	Actions	2	2	2	1	1	2

#	Action
1	$bSystem_D = true$
2	$bSystem_D = false$

Table C.3: $bSystem_D$ simplified Stateflow truth table

C.5 $bSystem_E$

Decision logic enables $bSystem_E$ in states 132, 140, 196, 204, 228, 236, and 244. Due to the excessively large nature of the logical expression, it is not included here.

C.6 $bOpr_B$

$bOpr_B$ is not enabled in any state, thus always disabled for this calibration.

$$bOpr_B = false$$

C.7 $bOpr_C$

$bOpr_C$ is not enabled in any state, thus always disabled for this calibration.

$$bOpr_C = false$$

Bibliography

Aberg, Rob (June 2004). "Logic Design Using Stateflow Truth Tables". In: *MathWorks News & Notes*. URL: http://www.mathworks.com/company/newsletters/articles/logic-design-using-stateflow-truth-tables.html (cit. on pp. 23, 24, 37).

Ackermann, Chris et al. (2010). "Automatic Requirement Extraction from Test Cases". In: *Runtime Verification*. Ed. by Howard Barringer et al. Vol. 6418. Lecture Notes in Computer Science. Springer Berlin Heidelberg, pp. 1–15 (cit. on pp. 21, 22, 50, 51, 53, 70, 71, 74, 78, 82, 83, 153).

Agrawal, Rakesh and Ramakrishnan Srikant (1994). "Fast algorithms for mining association rules". In: *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases*. Santiago de Chile, Chile: Morgan Kaufmann, pp. 487–499 (cit. on pp. 54, 64).

Andersson, Gunnar et al. (2002). "A proof engine approach to solving combinational design automation problems". In: *Design Automation Conference, 2002. Proceedings. 39th*. IEEE, pp. 725–730 (cit. on p. 73).

Archinoff, GH et al. (1990). "Verification of the shutdown system software at the Darlington nuclear generating station". In: *International Conference on Control & Instrumentation in Nuclear Installations* (cit. on p. 24).

Charette, Robert N. (2009). "This Car Runs on Code". In: *IEEE Spectrum*. URL: http://spectrum.ieee.org/transportation/systems/this-car-runs-on-code (cit. on p. 8).

Cleaveland, Rance, David Hansel, and Steve T. Sims (Jan. 2010). "System and method for automatic test-case generation for software". U.S Patent. No. 7,644,398 (cit. on pp. 53, 66).

Decision Tables (1970). Standard Document No. Z243.1-1970 for Decision Tables. New York, NY, USA: Canadian Standards Association (CSA) (cit. on p. 35).

Eles, Colin and Mark Lawford (2011). "A tabular expression toolbox for Matlab/Simulink". In: *NASA Formal Methods*. Vol. 6617. Lecture Notes in Computer Science. Springer, pp. 494–499 (cit. on p. 31).

Fu, Tian (1999). "Structured decision table, Generalized decision table conversion tool". Citeseer (cit. on pp. 25, 47).

Harrison, William J. (1971). "Practically Complete Decision Tables: A Range Approach". In: *ACM SIGPLAN Notices* 6.8, pp. 89–93 (cit. on p. 32).

Hass, Anne Mette Jonassen (2008). *Guide to advanced software testing*. Artech House (cit. on p. 62).

Hedayah, Mohamed M. (1974). *An Introduction to Decision Logic Tables*. University of Calif., School of Library Science (cit. on p. 6).

Heitmeyer, Constance et al. (1998). "SCR: A toolset for specifying and analyzing software requirements". In: *Computer Aided Verification*. Springer, pp. 526–531 (cit. on p. 24).

Heninger, Kathryn L. (1980). "Specifying software requirements for complex systems: New techniques and their application". In: *Software Engineering, IEEE Transactions on* 1, pp. 2–13 (cit. on p. 24).

Hughes, Marion L., Richard M. Shank, and Elinor S. Stein (1968). *Decision Tables*. Wayne, PA, USA: MDI Publications (cit. on pp. 25, 47).

Hurley, Richard B. (1983). *Decision Tables in Software Engineering*. New York, NY, USA: John Wiley & Sons, Inc. (cit. on p. 5).

ISO 26262-6:2011 (Nov. 2011). *Road vehicles - Functional safety - Part 6: Product development at the software level*. Standard Document No. ISO 26262-6:2011. Geneva, Switzerland: International Organization for Standardization/Technical Committee 22 (ISO/TC 22) (cit. on pp. 10–12).

Janicki, Ryszard and Alan Wassyng (2005). "Tabular expressions and Their Relational Semantics". In: *Fundamenta Informaticae* 67.4, pp. 343–370 (cit. on p. 29).

Jin, Ying and David Lorge Parnas (2010). "Defining the meaning of tabular mathematical expressions". In: *Science of Computer Programming* 75.11, pp. 980–1000 (cit. on pp. 6, 28, 30, 87, 106).

Kandl, Susanne and Raimund Kirner (2011). "Error detection rate of MC/DC for a case study from the automotive domain". In: *Software Technologies for Embedded and Ubiquitous Systems*. Springer, pp. 131–142 (cit. on p. 44).

Kirk, H.W. (Jan. 1965). "Use of Decision Tables in Computer Programming". In: *Communications of the ACM* 8.1, pp. 41–43. ISSN: 0001-0782. URL: http://doi.acm.org/10.1145/363707.363725 (cit. on p. 6).

Leveson, Nancy G. (2004). "Role of software in spacecraft accidents". In: *Journal of Spacecraft and Rockets* 41.4, pp. 564–575. URL: http://sunnyday.mit.edu/papers/jsr.pdf (cit. on pp. 3, 20, 49).

McCabe, Thomas J (1976). "A complexity measure". In: *Software Engineering, IEEE Transactions on* 4, pp. 308–320 (cit. on p. 45).

Owre, Sam, John M Rushby, and Natarajan Shankar (1992). "PVS: A prototype verification system". In: *Automated Deduction?CADE-11*. Springer, pp. 748–752 (cit. on p. 31).

Pollack, Solomon L., Harry T. Hicks, and William J. Harrison (1971). *Decision Tables: Theory and Practice*. Ed. by Richard G. Canning and J. Daniel Couger. Wiley Business Data Processing Series. Wiley-Interscience (cit. on pp. 5, 6, 47).

Pooch, Udo W (1974). "Translation of decision tables". In: *ACM Computing Surveys (CSUR)* 6.2, pp. 125–151 (cit. on pp. 24, 25, 32, 37).

Rastogi, Preeti (1998). "Specialization: An approach to simplifying tables in software documentation". Citeseer (cit. on p. 47).

Reactive Systems, Inc. (2014). *Reactis: Model-Based Testing and Validation. User's Guide*. Version 2014. URL: http://www.reactive-systems.com/papers/user.pdf (cit. on p. 61).

RTCA DO-178B (Dec. 1992). *Software Considerations in Airborne Systems and Equipment Certification*. Standard Document No. DO-178B. Washington, DC, USA: Radio Technical Commission for Aeronautics (RTCA) (cit. on p. 44).

Sheeran, Mary and Gunnar Stålmarck (1998). "A tutorial on Stålmarck's proof procedure for propositional logic". In: *Formal Methods in Computer-Aided Design*. Springer, pp. 82–99 (cit. on p. 73).

Shen, Hong (1995). "Implementation of table inversion algorithms". Communications Research Laboratory, McMaster University (cit. on p. 47).

Shen, Hong, J Zucker, and David Lorge Parnas (1996). "Table transformation tools: Why and how". In: (cit. on p. 46).

The MathWorks Inc. (2014). *Simulink Design Verifier. User's Guide*. Version R2014a. URL: http://www.mathworks.com/help/releases/R2014a/pdf_doc/sldv/sldv_ug.pdf (cit. on p. 73).

Tran, Quang Minh, Benjamin Wilmes, and Christian Dziobek (2013). "Refactoring of Simulink diagrams via composition of transformation steps". In: *ICSEA 2013, The Eighth International Conference on Software Engineering Advances*, pp. 140–145 (cit. on p. 46).

Vanthienen, Jan and Geert Wets (1994). "From decision tables to expert system shells". In: *Data & Knowledge Engineering* 13.3, pp. 265–282 (cit. on p. 47).

Wassyng, Alan and Ryszard Janicki (2003). "Tabular expressions in Software Engineering". In: *Proceedings of ICSSEA*. International Conference on Software and System Engineering. Vol. 4. Paris, France, pp. 1–46 (cit. on pp. 29, 30, 87).

Wassyng, Alan and Mark Lawford (2003). "Lessons Learned from a Successful Implementation of Formal Methods in an Industrial project". In: *Proceedings of FME'03*. Formal Methods Europe. Springer, pp. 133–153 (cit. on pp. 29, 30).

Wassyng, Alan, Mark Lawford, and Thomas S.E. Maibaum (2011). "Software certification experience in the Canadian nuclear industry: Lessons for the Future". In: *Proceedings of the ninth ACM international conference on Embedded software*. ACM, pp. 219–226 (cit. on pp. 24, 30).

Watson, Arthur H, Thomas J McCabe, and Dolores R Wallace (1996). "Structured testing: A testing methodology using the cyclomatic complexity metric". In: *NIST special Publication* 500.235, pp. 1–114 (cit. on p. 46).

Webb, Geoffrey I. (2004). "Association rules". In: *The Handbook of Data Mining*, pp. 25–39 (cit. on p. 64).

Weeks, Robert W and John J Moskwa (1995). *Automotive engine modeling for real-time control using matlab/simulink*. Tech. rep. SAE Technical Paper (cit. on p. 3).

Zucker, Jeffery I (1996). "Transformations of normal and inverted function tables". In: *Formal Aspects of Computing* 8.6, pp. 679–705 (cit. on p. 47).