Design and Modularization Of A Hybrid Vehicle Control System

Design and Modularization Of A Hybrid Vehicle Control System

By Augustino Fella Pellegrino, B.Eng., (Electrical Engineering)

A Thesis Submitted to the School of Graduate Studies in the Partial Fulfillment of the Requirements for the Degree Master of Applied Science

McMaster University
Master of Applied Science (2021)
Hamilton, Ontario (Department of Computing and Software)

TITLE: Design and Modularization Of A Hybrid Vehicle Control System
AUTHOR: Augustino Fella Pellegrino B.Eng., (Electrical Engineering), (McMaster University)
SUPERVISOR: Dr. Mark Lawford
NUMBER OF PAGES: xvii, 206

Lay Abstract

The complexity of automotive software has increased dramatically in recent years. New technological advances as well as increasing market competitiveness create a high cost-pressure environment. As a result, improving the development of automotive software and its maintainability has become an increasingly critical issue to solve. This thesis uses a Hybrid Vehicle Controller Model developed within MATLAB Simulink to investigate the possible improvements that can be made to software modularity. The system decomposition is modified using the Simulink Module Tool, and is analyzed regarding improvements to information hiding, interface complexity, and specifically minimizing change propagation. The modular improvements made to the Simulink Model resulted in significant improvements in system changeability and information hiding, providing a useful framework for future EcoCAR students.

Abstract

The complexity of automotive software has increased dramatically in recent years. New technological advances as well as increasing market competitiveness create a high cost-pressure environment. This thesis seeks to apply established modular principles to a Simulink Model to increase information hiding to improve the maintainability of controls software. A Hybrid Supervisory Controller (HSC) model, developed as part of the McMaster EcoCAR Competition, is used throughout this thesis. The software design process followed during the HSC model development is detailed, as well as providing an example of the application of the Simulink Module Tool, a Simulink add-on developed by Jaskolka et. al. The HSC System decomposition was restructured based on an analysis of the likely changes to the vehicle software, as well the system secrets contained within the model.

This thesis also presents an analysis of the original and modular system decompositions, comparing several common software indicators of information hiding, coupling, cohesion, complexity, and testability. The modular decomposition led to a significant improvement in information hiding, both in system changeability and internal implementation. Likely changes to the system propagate to fewer modules and components within the new decomposition, with hardware data separated from behavioral algorithms, and all modules grouped based on shared secrets. The redistribution of algorithms based on separation of concern also led to improvements in coupling, cohesion, and interface complexity. The resulting software design process and modular system decomposition provides a framework for future EcoCAR students to focus on correct design and implementation of hybrid vehicle software. The benefits provided by the application of the Simulink Module Tool also contributes additional data and supporting evidence to the improvements that can be realized within Simulink Models by introducing the concepts of information hiding and modularity.

Acknowledgements

I would like to thank Dr.Lawford for your help and guidance during my graduate studies, and Dr.Emadi and Dr.Wassyng for your continuous guidance with the Mcmaster EcoCAR Team. I would also like to thank everyone from the former and current McMaster EcoCAR Teams for their guidance, cooperation, and friendship. Most importantly, I would like to thank all my friends and family, specifically Sabina Fella, and Frank, Marlena, Poppi, and Lucia Pellegrino. Without all your support, completing this long and arduous journey would not have been possible.

Contents

Lay Abstract **iii**

Abstract **iv**

Acknowledgements **v**

List of Abbreviations **xv**

Declaration of Authorship **xvii**

1 Introduction **1**

1.1 Trends in Automotive Software 1
1.2 Software Principles within Model Based Development 2
1.3 Introduction to the EcoCAR Mobility Challenge 3
1.4 Research Objective and Scope 4
1.5 Thesis Outline 4

2 Background Information and Literature Review **6**

2.1 Model-Based Design 6
2.2 Simulink and Stateflow Modeling Basics 10
2.2.1 Simulink Constructs and Overview 10
2.2.2 Stateflow Constructs and Overview 20
2.3 Modular Software Principles 21
2.3.1 Information Hiding 23
2.3.2 Coupling and Cohesion 23
2.3.3 Interface Complexity 23
2.4 Simulink Construct Comparison 24
2.5 Modularity within Simulink 25
2.5.1 Definition of a Simulink Module 27
2.5.2 Definition of a Simulink Module Interface 27
2.5.3 Guidelines for Modularization 29
2.5.4 Simulink Module Tool 32
2.6 Simulink Design Verification 32
2.6.1 Model equivalence using SDV Property Proving 32
2.6.2 Model checks using Simulink Model Advisor 33
2.6.3 SDV Test Generation and Coverage Analyzer 33
2.7 Summary 34

3 Overview of EcoCAR Competition 35
3.1 Vehicle Architecture 35
3.2 Vehicle Technical Specifications 36
3.3 Competition Sponsored Hybrid Vehicle Model 36
3.3.1 Driver Plant Model 37
3.3.2 Vehicle Controller Model 39
3.3.3 Vehicle Plant Model 39
3.3.4 Visualization and Data Logging 41

4 Model Development Process 43
4.1 Requirements Development Process 43
4.1.1 Requirement Categorization 44
4.1.2 Functional Requirements Set 45
4.1.3 Requirements Specification Document 45
4.1.4 Example: Accelerator Pedal Processing 48
4.2 Software Design and Implementation 51
4.2.1 Controller and Serial Data Architecture 51
4.2.2 Hybrid Supervisory Controller Architecture 51
4.2.3 Software Interface Development 56
4.2.4 Example: Power Moding Algorithm and Implementation 56
4.2.5 Model Version Control 59
4.2.6 Software Version Control 60
4.3 Conventional Model Modularization 63
4.3.1 Software Partitioning via Model References 64
4.3.2 XIL Functionality via Model Variants 66
4.3.3 Program State Design via Data Dictionaries 66
4.3.4 HSC Internal Modularization 69
4.3.5 Interface Implementation via Bus Objects and Signal Tags . . . 72
4.4 Model and Software Testing 74
4.4.1 Requirements Validation via Simulink Requirements 74
4.4.2 Accelerator Pedal Processing: Fault Analysis and Test Bench . 80
4.4.3 Accelerator Pedal Processing: Test Results 81
4.4.4 Test Plan Documentation 88
4.5 Summary 88

5 Application of the Simulink Module Tool 92
5.1 Review of Simulink Modular Guidelines 92
5.2 Overview of Likely Changes and System Secrets 92
5.2.1 System Module Secrets 93
5.3 System Decomposition 93
5.3.1 Original Model Decomposition 93
5.3.2 Modular Decomposition 99
5.4 Introduction to Model Changes 99
5.4.1 Limitations for Simulink Modularization 99
5.4.2 Data Storage Modifications 105
5.5 Input Conversion Layer 105
5.5.1 Data Interface Module 105

5.5.2 Input Accelerator Pedal Processing 105
5.5.3 Range and Curvature Determination 109
5.5.4 Max Vehicle Torque Calculation 110
5.5.5 CAVs Module 113
5.5.6 Gear Ratio Conversion 113
5.5.7 Hardware Limit Calculation 113
5.6 Operation Mode Loop 119
5.6.1 Propulsion Control Strategy Module 119
5.6.2 Power Moding Module 122
5.6.3 Regen Power Module 122
5.7 Component Control Layer 127
5.7.1 BAS Control Module 127
5.7.2 Motor Control Module 129
5.7.3 Battery System Manager Module 134
5.7.4 Engine Hardware Module 134
5.8 Output Conversion Layer 137
5.8.1 Motor Torque Arbitration and Power Management 137
5.8.2 Power Management Subsystem Decomposition 137
5.8.3 Motor Torque Arbitration Decomposition 139
5.8.4 Clutch Control Module 147
5.8.5 Output Accelerator Pedal Processing 147
5.9 Summary 150

6 Model Comparison 153
6.1 Model Equivalence Verification 154
6.2 Information Hiding 154
6.2.1 System Changeability 154
6.2.2 Hidden Internal Implementation 160
6.3 Interface Complexity 167
6.4 Coupling and Cohesion 169
6.4.1 Coupling 171
6.4.2 Cohesion 171
6.5 Cyclomatic Complexity 173
6.6 Testability 178
6.7 Summary 185

7 Conclusions and Future Work 187
7.1 Summary of Thesis Content 187
7.2 Future Work 187
7.3 Conclusion 188

A Vehicle Power Moding 189

B Likely System Changes 196

C System Changeability 199

Bibliography 203

List of Figures

2.1 V-cycle Model . . . 8
2.2 Rational Design Process . . . 9
2.3 Simulink Model Hierarchy . . . 11
2.4 Simulink Model Reference . . . 11
2.5 Simulink Model Variant . . . 12
2.6 Simulink Virtual Subsystem . . . 13
2.7 Removing Virtual Subsystem . . . 14
2.8 Simulink Functions . . . 15
2.9 Non-virtual Bus Object . . . 17
2.10 Bus Selector Block Parameters . . . 18
2.11 Simulink Bus Editor . . . 19
2.12 Goto/From Tags . . . 20
2.13 Power Moding Stateflow . . . 22
2.14 Implicit Input Test . . . 25
2.15 Implicit Output Test . . . 26
2.16 Simulink and C module comparison . . . 28
2.17 Simulink Module Interface . . . 30
2.18 EcoCAR Module Interface . . . 31
2.19 Model Advisor Property Window . . . 33
2.20 Test Coverage Model Configuration Parameters . . . 34

3.1 McMaster EcoCAR Vehicle Architecture . . . 36
3.2 EcoCAR Mobility Challenge Simulink Model . . . 38
3.3 Driver Plant Model . . . 39
3.4 Drive Cycle Source Block . . . 40
3.5 HWFET Drive Cycle . . . 40
3.6 Mapped Motor Block . . . 41
3.7 Vehicle Model Simulation Output . . . 42

4.1 Requirements Categorization . . . 44
4.2 HSC System Categorization . . . 46
4.3 Component Categorization . . . 47
4.4 E-GAS Standard . . . 50
4.5 Serial Data Architecture Diagram . . . 53
4.6 FSC Diagram . . . 55
4.7 Interface Diagram . . . 57
4.8 Power Moding Diagram . . . 60
4.9 Vehicle Power Moding Algorithm . . . 61

4.10 Vehicle Power Moding Stateflow 62
4.11 HSC Model Reference 67
4.12 XIL Interfaces using Model Variants 68
4.13 XIL Input Model Variant 69
4.14 XIL Output Model Variant 70
4.15 Simulink Data Dictionary 71
4.16 ControllerToHSC Subsystem 73
4.17 HSC Model Interface 74
4.18 Root HSC Model 75
4.19 Major Module 76
4.20 Minor Module 76
4.21 Controller To HSC 77
4.22 PCSM Data Flow 78
4.23 Component Control Data Flow 78
4.24 Output Conversion Data Flow 79
4.25 Simulink Requirements Editor 80
4.26 Pedal Test Harness 81
4.27 Fault Tree Analysis 83
4.28 Pedal Test Bench 84
4.29 Test Input 85
4.30 Test Output 86
4.31 Verification of Requirement *III.4.1.3* 86
4.32 Test Input 2 87
4.33 Test Output 2 87
4.34 Verification of Requirement III.4.1.4 88
4.35 HIL Test Plan Part 1 89
4.36 HIL Test Plan Part 2 90
4.37 HIL Test Plan Part 3 91

5.1 HSC root level system decomposition 96
5.2 System decomposition of the Input Conversion Layer (original) 97
5.3 System decomposition of the Output conversion layer (original) 97
5.4 System decomposition of Operation Mode and Component Control Loops (original) 98
5.5 HSC root level system decomposition (modular) 100
5.6 DIM, PPM, CAVM System decomposition (modular) 101
5.7 EMM System decomposition (modular) 101
5.8 PSCM System decomposition (modular) 102
5.9 FHM, HLCM System Decomposition (modular) 102
5.10 Hardware Modules System Decomposition (modular) 103
5.11 Input Data Processor subsystem in original model 106
5.12 Output Data Processor Subsystem in original model 107
5.13 Input and Output Simulink functions within Data Interface Module . 107
5.14 Data Interface Module as a model reference, with Input function caller block 108
5.15 Output function caller block at HSC model root level 108
5.16 Input Pedal Processing Algorithm 109

5.17 *Get_EngAxleTrq()* Simulink function 110
5.18 Input Pedal Processing Internals 111
5.19 Input Pedal Processing Simulink Function 112
5.20 *Calc_Pedal_In()* function caller block 112
5.21 Range and Curvature Subsystem located in root model 112
5.22 Subsystem Converted to Local Simulink Function within PCSM Model 113
5.23 Max Vehicle Torque Subsystem 113
5.24 Original look of Max_EngWhlTrq Function 114
5.25 Algorithms within Max Vehicle Torque Subsystem 114
5.26 Max Motor Torque function defined inside MHM Model 114
5.27 Engine Hardware functions within EHM Model 115
5.28 Gear Ratio Conversion Simulink Function defined within THM 115
5.29 Calc_MaxVehTrq() Simulink function defined within the HLCM . . . 116
5.30 Calc_MaxVehTrq() Simulink function Internals 116
5.31 Calc_MaxVehTrq() function caller block at root level of HSC Model . 117
5.32 CAVs Exported Simulink Functions 117
5.33 Top level HSC layer containing CAVs Model Reference and function caller blocks 118
5.34 Gear Ratio Conversion Subsystem in Original Model 118
5.35 Internals of Gear Ratio Simulink Function 118
5.36 Original Battery Management System virtual subsystem 119
5.37 Internals of Battery Management Simulink Function 119
5.38 Battery Management Simulink Function 120
5.39 Battery Management Simulink Function caller at HSC root level . . . 120
5.40 Battery Management Simulink Function caller at HSC root level . . . 120
5.41 EMM Model Reference at HSC root level 121
5.42 PCSR Stateflow Chart within the PCSM Model 123
5.43 PCSR Stateflow Chart Internal Structure 124
5.44 State Object implementing the Functionality for the HEV Drive Mode 125
5.45 Stateflow Box containing all local Simulink functions 125
5.46 Example of function call within HEV state using Stateflow Box Identifier 125
5.47 Stateflow Box containing graphical function 126
5.48 The PMSR Stateflow chart located within the PMM 127
5.49 Original Regen Torque Calculation Subsystem 128
5.50 Original Regen Torque Calculation Algorithm 128
5.51 Modified Regen Torque Calculation Algorithm 128
5.52 Scoped Simulink function within Regen Power Module 129
5.53 *Calc_RegenTrq()* function caller defined in EMM 130
5.54 Propulsion System Control Module within HSC 131
5.55 Top level view of BAS Control Ring 132
5.56 Improved BAS Control Ring Internals 133
5.57 Original Motor Control Model Layout 135
5.58 Fault Detection function caller blocks and data flow to MHM 136
5.59 Fault Handler Simulink functions 136
5.60 *MotFault_Handler()* function caller 136
5.61 MCM Top Level View 137
5.62 set_Voltage() Simulink function defined at root level of Stateflow Chart 138

5.63 set_Voltages() Simulink function within modified Stateflow Chart . . 139
5.64 Battery Control Graphical function within modified Stateflow Chart . 140
5.65 Modularized Battery Control Module 141
5.66 Function callers Calc_Eng_Axle_Trq() and Calc_Pedal_Out(), located in output layer 142
5.67 Original subsystem layout within output conversion layer 143
5.68 Internals of original subsystem containing two distinct algorithms . . . 143
5.69 Output conversion layer view of the separated virtual subsystems . . . 143
5.70 Power Management Algorithm in original virtual subsystem 144
5.71 Motor Torque to Electrical Algorithm in original virtual subsystem . . 144
5.72 Exported Simulink functions located in the MHM Model 145
5.73 *Batt_Lmt_Check()* Simulink function 145
5.74 *Limit_Check()* Internal Algorithm 146
5.75 *Limit_Check()* Simulink function 147
5.76 Regen Torque Arbitration Simulink Function 148
5.77 Clutch Speed Matching Subsystem in original model 148
5.78 Clutch Speed Matching Subsystem Algorithm 148
5.79 Clutch Simulink functions located in CCM Model 149
5.80 Clutch function caller blocks 149
5.81 Input Pedal Processing Subsystem 150
5.82 *Calc_Pedal_Out()* Simulink function 150
5.83 *Calc_Pdl_Out()* Simulink function Internals 151
5.84 Output Pedal Processing Function caller 152

6.1 Design Verification Model used to prove functional equivalence 155
6.2 Root level components requiring changes within EcoCAR Model . . . 156
6.3 Changes required within the Input Conversion Module 157
6.4 Changes in RPM and Output Conversion Module 157
6.5 Imported Module Change Propagation 159
6.6 MHM Change Propagation 159
6.7 Implicit Input passed to Max Vehicle Torque Calculation within EcoCAR Model 161
6.8 Implicit Input received within Max Engine Torque Calculation 161
6.9 Implicit Input passed to Simulink Function within Modularized Model 162
6.10 Implicit Input does not pass-through Simulink Function boundaries . . 162
6.11 Error received when attempted to pass implicit input to a Simulink Function 163
6.12 Implicit Output passed from Max Engine Torque Calculation Subsystem within EcoCAR Model 164
6.13 Implicit Output received from Max Vehicle Torque Virtual Subsystem 164
6.14 Implicit Output passed from Simulink Function within Modularized Model 165
6.15 Implicit Output does not pass-through Simulink Function boundaries . 165
6.16 Error received when attempted to pass implicit output from a Simulink Function 166
6.17 Engine Lookup Table containing Parameter defined in Data Dictionary 167

6.18 Engine Lookup Table containing Parameter defined in Model workspace of EHM 167
6.19 Internal Data accessible outside of virtual subsystem within EcoCAR Model 168
6.20 Internal Data inaccessible outside of EHM within Modularized Model 168
6.21 Error received when referencing parameter defined within EHM Model Workspace 169
6.22 Interface and Dependencies Before and After Modularization 170
6.23 Motor Interface Before and After Modularization 170
6.24 Model References with high data coupling to Data Dictionaries 171
6.25 Reduced data coupling after Module Data defined in Model workspaces 172
6.26 Interdependent Functions placed within singular Module to reduce coupling 173
6.27 Legend for Figures 6.28 - 6.33 174
6.28 Low Cohesion among virtual subsystems in Input Conversion Layer . . 174
6.29 Low Cohesion among virtual subsystems in Input Conversion Layer continued 175
6.30 Low Cohesion among virtual subsystems in Output Conversion Layer 176
6.31 Input Subsystems replaced by Simulink function calls at HSC Root level177
6.32 Range Curvature function moved to PCSM, improving cohesion 178
6.33 Brake Calculation function moved to RPM, improving cohesion 179
6.34 Output Subsystems replaced by Simulink function calls at HSC Root level 180
6.35 Modularized Model with High cohesion among Simulink Modules . . . 181
6.36 Modularized Model with High cohesion among Simulink Modules . . . 181
6.37 Model Advisor showing checks run on both HSC Models 182
6.38 Test Harness created for coverage analysis 184

A1.1 BAS Power Moding Algorithm 190
A1.2 Battery Power Moding Algorithm 191
A1.3 Motor Power Moding Algorithm 192
A1.4 Engine Power Moding Algorithm 193
A1.5 Vehicle Shutdown Stateflow 194
A1.6 Motor Control Ring Stateflow 195

List of Tables

2.1 A comparison of Simulink constructs, derived from [18] 26
2.2 Simulink and C construct Comparison, taken from [36] 27

3.1 McMaster EcoCAR VTS Goals 37

4.1 Pedal Requirements 49
4.2 HSC Serial Data Interface 51
4.3 EcoCAR Controller Specifications 52
4.4 CAVS-PCM Interface 58
4.5 Model Version Control Process 63
4.6 Software Version Control Document 64
4.7 Software Version Control Process 65
4.8 Accelerator Pedal Interface 82
4.9 Requirement *III.4.1.3* expected results 85
4.10 Requirement *III.4.1.4* expected results 85

5.1 System Module Secrets Part 1 94
5.2 System Module Secrets Part 2 95
5.3 Module Changes 104

6.1 Motor Hardware System Changeability 158
6.2 Change Propagation before and after Modularization 158
6.3 Model Advisor Results for Original EcoCAR Model 178
6.4 Model Advisor Results for Modularized Model 180
6.5 Comparison of Block count and Cyclomatic Complexity before and after changes 182
6.6 SDV Test Generation Analysis Results Before and After Modularization 183
6.7 Original Model Coverage Analysis 184
6.8 Modularized Model Coverage Analysis 185

A2.1 List of Likely Changes, part 1 197
A2.2 List of Likely Changes, part 2 198

A3.1 System Changeability Part 1 200
A3.2 System Changeability Part 2 201
A3.3 System Changeability Part 3 202

List of Abbreviations

ACC	**A**daptive **C**ruise **C**ontrol
AEB	**A**utomative **E**mergency **B**raking
APP	**A**ccelerator **P**edal **P**rocessing
AVTC	**A**dvanced **V**ehicle **T**echnology **C**ompetitions
AWD	**A**ll **W**heel **D**rive
BAS	**B**elted **A**lternator **S**tarter
BASCM	**B**elted **A**lternator **S**tarter
BCM	**B**ody **C**ontrol **M**odule
BHM	**B**attery **H**ardware **M**odule
BSM	**B**attery **S**ystem **M**anager
BSMM	**B**attery **S**ystem **M**anager **M**odule
CAD	**C**omputer **A**ided **D**esign
CAN	**C**ontrol **A**rea **N**etwork
CAVM	**C**onnected **A**utonomous **V**ehicle **M**odule
CAVS	**C**onnected **A**utonomous **V**ehicle **S**ystems
CCC	**C**AVs **C**omputation **C**ontroller
CCM	**C**lutch **C**ontrol **M**odule
CSC	**C**AVs **S**afety **C**ontroller
CSF	**C**AVs **S**ensor **F**usion
DIM	**D**ata **I**nterface **M**odule
EBCM	**E**lectronic **B**rake **C**ontrol **M**odule
ECM	**E**ngine **C**ontrol **M**odule
ECU	**E**lectronic **C**ontrol **U**nit
EM	**E**lectric **M**otor
EMM	**E**nergy **M**anagement **M**odule
EMSM	**E**nergy **M**anagement **S**trategy **M**odule
EHM	**E**ngine **H**ardware **M**odule
ESS	**E**nergy **S**torage **S**ystem
EV	**E**lectric **V**ehicle
FCM	**F**ront **C**ontrol **M**odule
FHM	**F**ault **H**andling **M**odule
GM	**G**eneral **M**otors
HEV	**H**ybrid**E**lectric **V**ehicle
HIL	**H**ardware - **I**n - the - **L**oop
HLCM	**H**ardware **L**imit **C**ontrol **M**odule
HS	**H**igh **S**peed
HSC	**H**ybrid **S**upervisory **C**ontroller
HV	**H**igh **V**oltage
HVAC	**H**eating **V**entilation and **A**ir **C**onditioning

HWFET	**H**igh**W**ay **F**uel **E**conomy **T**est
ICE	**I**nternal **C**ombustion **E**ngine
I/O	**I**nputs / **O**utputs
LAN	**L**ocal **A**rea **N**etwork
LS	**L**ow **S**peed
LV	**L**ow **V**oltage
MAB	**M**athworks **A**dvisory **B**oard
MABXII	**M**icro **A**uto **B**o**XII**
MBD	**M**odel **B**ased **D**evelopment
MCDC	**M**odel **C**ondition **D**ecision **C**overage
MCM	**M**otor **C**ontrol **M**odule
MHM	**M**otor **H**ardware **M**odule
MIL	**M**odel - **I**n - the - **L**oop
MPG	**M**iles **P**er **G**allon
NDA	**N**on **D**isclosure **A**greement
OEM	**O**riginal **E**quipment **M**anufacturer
PCM	**P**ropulsion **C**ontrols **M**odeling
PCSM	**P**ropulsion **C**ontrols **S**trategy **M**odule
PCSM	**P**ropulsion **C**ontrols **S**trategy **R**ing
PCCM	**P**ropulsion **C**omponent **C**ontrols **M**odule
PID	**P**roportional **I**ntegral **D**erivative
PMM	**P**ower **M**oding **M**odule
PMSR	**P**ower **M**oding **S**election **R**ing
PPM	**P**edal **P**rocessing **M**odule
PRNDL	**P**ark - **R**everse - **N**eutral - **D**rive - **L**ow
PSCM	**P**ower**S**teering **C**ontrol **M**odule
PVS1	**P**edal **V**alue **S**ensor1
PVS2	**P**edal **V**alue **S**ensor2
RCM	**R**ear **C**ontrol **M**odule
RCM	**R**egenerative **P**ower **M**odule
RSD	**R**equirement **S**pecification **D**ocument
RTI	**R**eal **T**ime **I**nterface
SDD	**S**oftware **D**esign **D**ocument
SDD	**S**imulink **D**esign **V**erifier
SIL	**S**oftware - **I**n - the - **L**oop
SOC	**S**tate **O**f **C**harge
SRS	**S**oftware **R**equirement **S**pecification
TCM	**T**ransmission **C**ontrol **M**odule
THM	**T**ransmission **H**ardware **M**odule
US	**U**nited **S**tates
VIL	**V**ehicle - **I**n - the - **L**oop
VTS	**V**ehicle **T**echnical **S**pecification
XIL	**X**nvironment - **I**n - the - **L**oop

Declaration of Authorship

I, Augustino Fella Pellegrino, declare that this thesis titled, "Design and Modularization Of A Hybrid Vehicle Control System" and the work presented in it are my own. I confirm that:

- Chapter 1: Provided an introduction to the thesis, presented motivation for research and research objectives.
- Chapter 2: Presented a literature review and background information related to Simulink fundamentals and modularity.
- Chapter 3: Provided background information on the EcoCAR vehicle architecture, VTS goals, and Competition vehicle model.
- Chapter 4: Overview of the software design process developed during the EcoCAR competition including requirements development, software architecture design, software implementation, and testing.
- Chapter 5: Contains the analysis of the original system decomposition, as well as the likely changes and module secrets resulting in a new improved decomposition. Details the Simulink model changes resulting from an application of the Simulink Module Tool.
- Chapter 6: Performed a comparison of the original and modular decompositions. Analyzes changes in information hiding, coupling, cohesion, cyclomatic complexity, and testabillity.
- Chapter 7: Summarizes thesis work and provides a brief description of future work planned.

Chapter 1

Introduction

In recent years, the extent to which software design has propagated to other technical domains is increasing rapidly. In particular, the automotive industry has seen a dramatic increase in features and complexities, with modern cars often having up to 80 separate embedded controllers requiring programming [5], [1, page 2], [41, page 3]. New technological innovations such as Advanced Driver Assistance, Advanced Energy Management, as well as increased safety protocols have resulted in a drastic increase in vehicle controls complexity [6], with up to 2000 software-based functions deployed on premium cars in 2007 [6, page 9]. This increased controls complexity has a direct impact on the amount of software developed and deployed on embedded hardware within the vehicles. Current challenges faced in the automotive software industry include an insufficient level of software quality, as well as low reusability of vehicle controls code. With most functionality within a vehicle implemented as proprietary software, a lot of functionality is not reused, as there is no established standard software structure and design process which can be implemented within multiple vehicles across all manufacturers [5],[17].

1.1 Trends in Automotive Software

The increased complexity in vehicle software can be attributed to several factors, including the increased size of overall software, as well as the increased complexity of the internal functions and algorithms [5], [1], [14]. The expansion of software within vehicles was expected to increase for another two decades in 2007 [6, page 8], and continues in that trend to this day. Complexity coming from advanced vehicle functionality includes increased crash prevention and crash safety, primarily relying on software diagnostics and mitigation to reduce the frequency and severity of vehicle malfunctions and accidents [6, page 8]. The advancement of hybrid and electric vehicle architectures are also a large contributing factor to the increase in automotive software complexity [14, page 2], [42]. As the methods of energy management and powertrain electrification become more advanced, a larger reliance has been placed on software to facilitate these complex relationships [6, page 8]. Finally, advances in Driver Assistance such as lane keep assist, cruise control and even self-driving capabilities have drastically increased the complexity of software, and have added new challenges in torque and power arbitration [6, page 8].

As software requirements increase for vehicle development, the development time and cost increase as well. With software playing an increasingly critical role in the successful design and production of new vehicles, increased quality standards are required [14, page 2]. Furthermore, the highly competitive nature of the automotive industry means development cycles for new vehicle software are often very short, providing little time for a software design process to be followed accurately. Further exacerbating this cost and time pressure is the heterogeneity of the software developed within the vehicle ECUs [6, page 12], [15, page 1]. Much of the software deployed within a vehicle is developed by various hardware suppliers, which do not share their internal implementations with OEMs. The level to which software is developed by Automotive manufacturers in-house or contracted to external suppliers varies between company and vehicle model [14, page 3]. As the complexity of vehicle software increases, the importance of cooperation and efficient work between automotive companies and hardware suppliers becomes increasingly critical.

1.2 Software Principles within Model Based Development

Despite the increased demand for software development within the Automotive domain, the principles followed within traditional software frameworks [40], [37], [39] are not prevalent within automotive embedded design. In order to handle the increased complexity of vehicle software and to increase its reusability, a system decomposition based on separation of concerns and information hiding is needed, to create a uniform way of automotive software development [6, page 12]. Model based design is the primary method in which automotive software controls are developed and tested, using industry standard tools such as MATLAB Simulink to model components, and perform automatic code generation [6, page 11], [17], [14, page 4]. The Mathworks Advisory Board (MAB) has existing guidelines for model structuring, but fails to properly address the issues of system decomposition practices to increase modularization and improved information hiding [21]. If complexity and prevalence of software development within the automotive industry continues to increase without establishing traditional software safe practices, system design errors could increase in prevalence and impact. Propagation of errors within embedded controls systems have potential to cause major damage to both persons and property [47], establishing a need for modular guidelines and practices to follow when developing automotive vehicle software.

Work has been done in [18] to analyze the existing Simulink modeling constructs in regards to their ability to implement information hiding and increased modularity. The structure of a Simulink Module has also been developed and discussed in detail in [35], [36], and [19], with guidelines established for implemented modularization in Simulink MBD. The Simulink Module Tool was developed by Jaskolka [43]; a Simulink tool add-on that can generate interfaces, convert between constructs, as well as evaluate a model's compatibility with modular guidelines. While the need for modularization and information hiding within Simulink models is well understood, there exists a limited amount of real-world applications of this modularization process, particularly in the case of the Simulink Module Tool. In addition, these guidelines have not been applied to a large number of systems, or examined in regards to their impact on a Simulink vehicle controls model. In order to further prove the benefits of using the Simulink Module Tool to improve information hiding in automotive software, additional examples and data should be established. This demonstration of the modularization process, as well an analysis of the benefits introduced, are the primary research objectives of this thesis.

1.3 Introduction to the EcoCAR Mobility Challenge

A notable source of innovation regarding the development of hybrid vehicles comes from the Advanced Vehicle Technology Competitions (AVTC), a set of competitions sponsored by the U.S. Department of Energy in conjunction with the North American Automotive Industry. The EcoCAR Mobility Challenge is an AVTC competition organized by the Argonne National Laboratory and sponsored by major automotive industry partners including General Motors and The MathWorks [4]. McMaster University is a participant in the EcoCAR Challenge, which is a four year long competition extending from 2018 - 2022 (referred to as Year 1 - Year 4 throughout this thesis). Year 1 of the competition focused on the initial vehicle architecture design, component selection, vehicle plant modeling, as well as testing bare bones controls in a MIL environment. In Years 2 and 3, the focus was put on Vehicle Integration as well as development and testing of vehicle controls software. Testing and validation in Year 2 was primarily in the MIL, and HIL environments, with Year 3 shifting focus to VIL testing. The final Year of the competition is meant for refinement of the vehicle controls software, and implementation of additional functionality such as diagnostics and robust fault detection, CAVS integration, and increased complexity in control strategy [20].

The McMaster EcoCAR Team is comprised of six major sub-groups, namely the Propulsion, Electrical, PCM (Propulsion Controls Modeling), CAVS (Connected and Autonomous Vehicle Systems), System Safety, and Communications teams. The Propulsion and Electrical sub-teams are primarily responsible for vehicle architecture design, vehicle integration, and vehicle CAD/Circuit Modeling. The PCM team is responsible for the design, implementation, and testing of the vehicle controls software, as well as performing system simulations to evaluate the degree to which the hybrid vehicle model meets performance targets. The CAVs team develops the advanced driver assistance functionality required in the competition vehicles, including cruise control and lane keep assist, as well as sourcing and programming relevant sensors. The System Safety team is primarily concerned with the definition of safety requirements for the vehicle, as well as analysis and mitigation of potential faults and hazards. Finally, the Communications team is responsible for maintaining an active social media presence throughout the duration of the competition, as well as performing analysis and decisions related to the high level customer goals and requirements. The research performed for this thesis was in part done during Year 2 of the EcoCAR Competition (2019-2020), during the author's time as the PCM Team Lead. The Energy Management Strategy (referred to as the PCSM in future Chapters) was primarily developed by Mike Haussmann during Year 1 of the competition, and is discussed at length in his masters thesis titled *Development of a Control System for a P4 Parallel-Through-The-Road Hybrid Electric Vehicle* [16].

1.4 Research Objective and Scope

The research presented in this thesis was done, in large part, to fulfill EcoCAR Mobility Challenge deliverables and develop vehicle controls to meet competition objectives. The remaining research involved the design of a modular system decomposition for the EcoCAR Model, and the application of the Simulink Module Tool and Simulink modular guidelines to improve information hiding and system changeability. The increase in software complexity outlined in Section 1.1, as well as the lack of existing guidelines to improve fundamental software design principles within MBD provide motivation for the development of a modular automotive controls model. This thesis presents the development of a hybrid vehicle controls model within MATLAB Simulink, and applies the Simulink Module Tool in an attempt to increase modularity and information hiding. The primary contributions of this thesis and the associated work that was performed by the author from 2019 - 2021 are summarized below.

1. Development of Hybrid Vehicle Controls software for the McMaster EcoCAR competition vehicle, in joint effort with the McMaster EcoCAR PCM Team.
   (a) Design and development of vehicle component and power moding controls for the EcoCAR Hybrid Vehicle.
   (b) Deployed vehicle control software on target embedded hardware, performed HIL bench testing and VIL CAN communication testing.
   (c) Developed CAN gateway software for the HSC to GM Hardware interface, successfully validated software on target vehicle.
2. Development of a Software Design Process followed by the McMaster EcoCAR PCM Team during Years 2 and 3 of the competition (2019-2020).
3. Providing a large scale example of the application of the Simulink Module Tool on a Vehicle Controls Model. Data has been collected regarding the possible benefits provided by the adoption of this decomposition method, motivating a transition to modular Simulink Software within the embedded hardware and automotive industries.
   (a) Creation of a modular system decomposition for the EcoCAR HSC Model, providing a framework for future McMaster EcoCAR students to use in later competitions. The information covered in this thesis, as well as the finalized modular model (Section 7.2) should provide future EcoCAR PCM students with the tools they need to significantly improve the information hiding and changeability of the HSC software.
   (b) Performed an analysis on the original and modular decomposition, evaluating their ability to meet established modularity metrics including changeability, interface complexity, coupling, cohesion, and testability.

1.5 Thesis Outline

The remainder of this thesis will be structured as follows. Chapter 2 contains a review of existing literature regarding model based design and modularization, as well as important terms and constructs used in the Simulink Environment. Chapter 3 provides more context to the EcoCAR Mobility Challenge, providing a brief description of the vehicle architecture, VTS goals, and competition vehicle model. The original EcoCAR HSC model is described in Chapter 4, outlining the software design process followed and providing details on the initial system decomposition. Chapter 5 describes the changes made to the HSC system decomposition and resulting Simulink model in order increase modularity and improve information hiding. Guidelines defined in Chapter 2 were used as a primary reference, along with utilization of the Simulink Module Tool to apply modular conversions. The resulting changes to modularity and various software metrics are discussed in Chapter 6, answering the questions of whether application of the Simulink Module Tool to a Hybrid Vehicle Controller Model can result in improved information hiding and system changeability. Chapter 7 concludes this thesis, providing a summary of results as well as future work planned.

Chapter 2

Background Information and Literature Review

Chapter 2 presents a literature review and provides the reader with the necessary background information needed to understand the remaining sections of the thesis. Section 2.1 introduces the concept of Model-based Design (MBD), and discusses common development frameworks used within the automotive domain. Section 2.2 provides an overview of MATLAB Simulink, the primary application framework used throughout this thesis, including common constructs and terminology. Section 2.3 discusses fundamental modular principles and terminology, outlining the importance of information hiding within software systems. Section 2.4 summarizes existing work that has been done to analyze the Simulink constructs in regards to their ability to implement information hiding, as well as developing a modular framework for MBD within Simulink. The resulting modular guidelines and Simulink Module definition are summarized in Section 2.5, along with a description of the **Simulink Module Tool**, a MATLAB Add-on which helps facilitate modularity within the Simulink Environment.

2.1 Model-Based Design

Model Based Design (MBD) is becoming increasingly prevalent within software processes, particularly in the automotive industry. One of the most recognized and widely used software development processes within MBD is the V-Model, shown in Figure 2.1 and described in detail in [17]. It is composed of an iterative process involving Requirements Definition, System Architecture Development, Algorithm and Functional Development, Code Generation, Unit Testing, Component Testing, System Testing, and Requirements Verification and Maintenance. The solid arrows in Figure 2.1 represent the order in which steps should be completed, while the dashed arrows represent the verification of previous steps in the V-model.

The first step of the MBD V-model process is to identify and document software requirements. High-level system requirements describe what larger systems need to accomplish, and are then further broken down to as granular of a level as needed. The consolidation of these requirements are typically documented within a software requirements specification (SRS), which is maintained and referenced throughout the remaining software development process [17, page 5]. Once requirements are complete, the architecture and software design phase can begin, starting with a decomposition of the system into smaller, manageable subsystems or modules [17, page 12].

Several different methods of system decomposition exist and are followed in industry; this thesis focuses on the decomposition concepts involving design for change. This method is discussed in more detail in Section 2.3, and refers to the decomposition of a system into groupings based on the anticipated changes to the system, while seeking to minimize the propagation of changes throughout the system hierarchy [40]. Software design decisions are recorded within a software design document (SDD), acting as a translation of the requirements within the SRS into a meaningful set of software components and interfaces [17, page 13].

The next step in the V-model process is software implementation and code generation. Code generation within most MBD environments is automatic (most notably MATLAB Simulink). The graphical models describing system and component behaviour are translated into code designed to deploy directly on the target embedded hardware [17, page 16]. This allows for rapid prototyping in different XIL environments, particularly in the HIL and VIL stages. Once software is fully designed and implemented, the testing and validation stages begin, starting with unit testing [17, page 17]. For a given system component, test cases are developed and run against the target software to determine the degree to which their associated requirements are met. This individual testing process is ideally applied to all software components within the system, but is typically restricted based on availability of equipment and external dependencies. Additional forms of testing have been identified in Figure 2.1, namely integration and acceptance testing. The degree to which multiple components are tested together is entirely dependent on the system under question and what is feasible within the project timeline and equipment availability. In the context of automotive systems, integration and acceptance testing would include the combination of several major software systems (propulsion, thermal management, power moding etc) and the validation of safe and functional vehicle operation. Software testing should always seek to validate the corresponding set of requirements defined at the beginning of the V-model development process [17, page 18-19]. Software systems within the automotive industry are typically very large and complex, requiring many modifications to the functionality and structure throughout the software development life cycle. As a result, the V-model must represent an iterative process, where changes needed in the system are first analyzed in regards to the impact they will have on the SRS. Changes required are then propagated throughout the architectural design, software design, testing, and validation stages [17, page 7].

Similar to the V-model, the notion of a rational design process is discussed in [38] and outlines the need to adhere as closely as possible to this unattainable ideal when developing software. A diagram outlining the major steps of the rational design process can be seen in Figure 2.2. Creating a requirements document facilitates the translation of a user or customer's product request to a set of functionalities that can be designed and implemented in software [38, page 4-5]. Similar to the architecture and software design stages of the V-model, the rational design process has the Module Structure stage in which the larger system is decomposed into smaller, manageable modules [38, page 6]. This is followed by the design and implementation of the module interfaces as well as the uses hierarchy, which captures the dependencies and interactions between the components of the software system. Specifying the external dependencies of each module facilitates independent development with

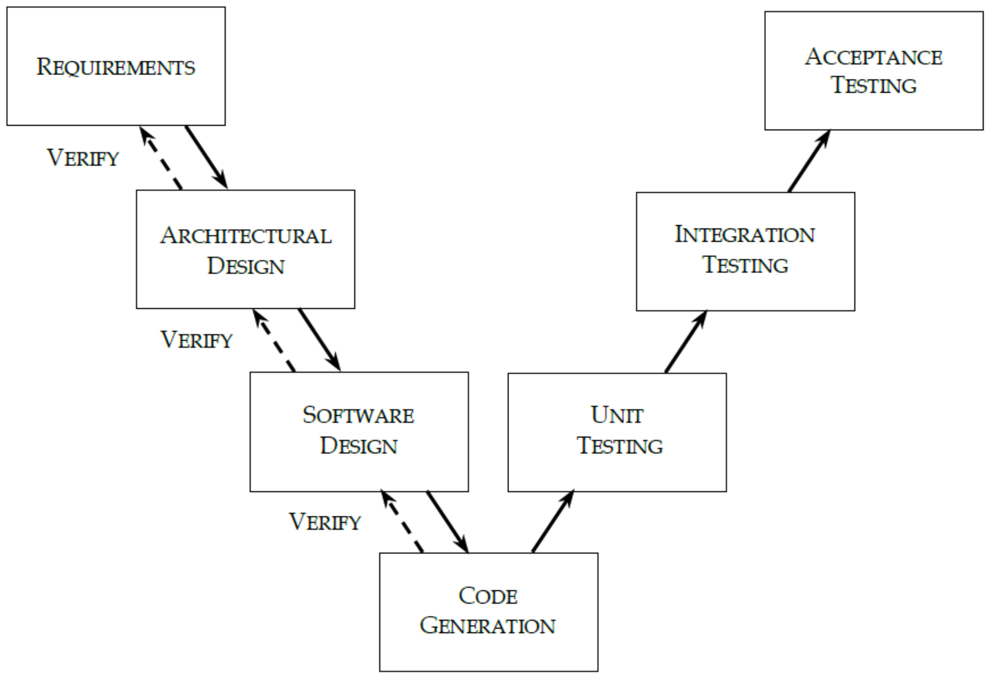


FIGURE 2.1: V-cycle Model, taken from [17, page 5]

minimal collaboration needed to implement functionality [38, page 6-7]. The uses hierarchy will capture correctness dependencies between modules (Module A depends on functionality of Module B), providing a clearer picture on potential roadblocks or issues that may come up during development.

The structures of the internal modules identified in previous stages are then designed, and captured within a module design document [38, page 7]. This documentation is followed by the software implementation stage, where the programs identified within each module are developed, tested, and verified against requirements [38, page 8]. The final stage is maintenance, a continuous process in which any changes to the system are applied to each step of the rational design process and propagated throughout, depending on the impact. It is worth noting that both the V-model and rational design process outlined in [17] and [38], respectively, are ideals that should be striven for but are not fully realizable in practical applications. Often the information needed to properly document or complete a design process stage is not available at the time, or is not yet known. As a result, designers should seek to follow the processes as closely as possible, and produce incomplete documentation if necessary. Once the missing information is found or new issues come up, it is then possible to reiterate the V-model or Rational design process, and update the corresponding documentation [38, page 2].

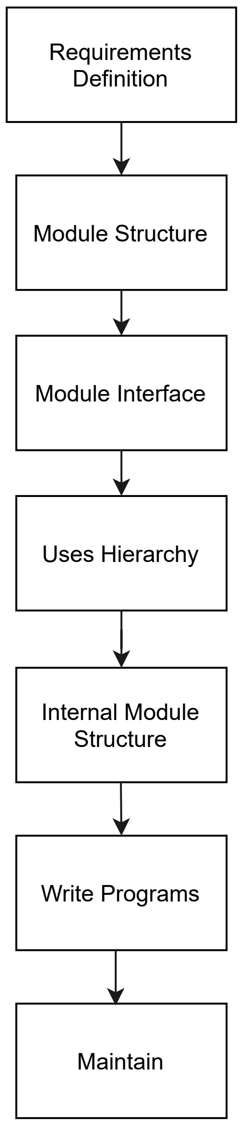


FIGURE 2.2: Rational Design Process, derived from [38]

2.2 Simulink and Stateflow Modeling Basics

MBD is applied throughout this thesis using MATLAB Simulink, a powerful modeling tool used throughout industry to develop embedded software and controls. Much of the content of this thesis relies on the user having a basic understanding of the MATLAB Simulink Environment as well as the various common constructs and processes involved in modeling. Sections 2.2.1 and 2.2.2 provide an overview of the Simulink and Stateflow constructs referenced throughout this thesis and are used extensively in the models under discussion.

2.2.1 Simulink Constructs and Overview

This section seeks to provide a brief overview of each construct used and its purpose within Simulink model development. Constructs which are leveraged to modularize and structure the model include Model References, Model Variants, Virtual Subsystems, and Simulink Functions. There are also several data and interface constructs used throughout this thesis, namely Simulink Bus objects, Goto/From Tags, and Data Dictionaries.

2.2.1.1 Project Model Root

The inclusion of multiple models within a Simulink project introduces a hierarchical structure during development and simulation [31, page 369]. Simulink Documentation gives the example shown in Figure 2.3, where the top model contains model references which can extend to lower hierarchical levels. This top model is referred to as the **root model**, or **root level** throughout this thesis. Modules lower in the model hierarchy cannot refer to models at higher levels as indicated by the red X's on arrows from lower models to higher models. The two Simulink projects that are used for analysis in Chapters 4 and 5 contain a root model with multiple model references to create a hierarchical structure.

2.2.1.2 Model References

A Model Reference block is used within Simulink to include one model in another, establishing a model hierarchy [31, page 368]. The root model containing the model reference block is referred to as the parent model, and the model associated with the reference block is the child. An example of a model reference can be seen in Figure 2.4, with the CAVS Module (child) shown defined within the HSC root level (parent). Model References have several advantages, most notably the increased modularity during software development [31, page 369]. Decomposing a large system into model reference blocks allows for independent development and testing of software. Changes to the child model are hidden from the parent, allowing for increased information hiding. Model references are used extensively throughout this thesis, both in the initial EcoCAR Model Development, as well as during the modularization process.

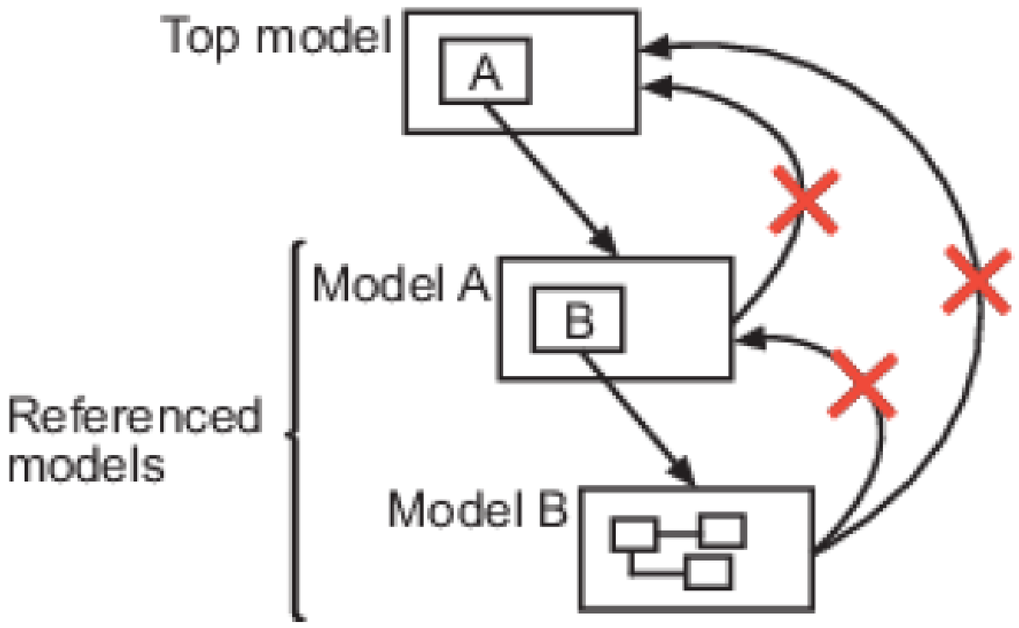


FIGURE 2.3: Simulink Model Hierarchy, taken from [31, page 369]

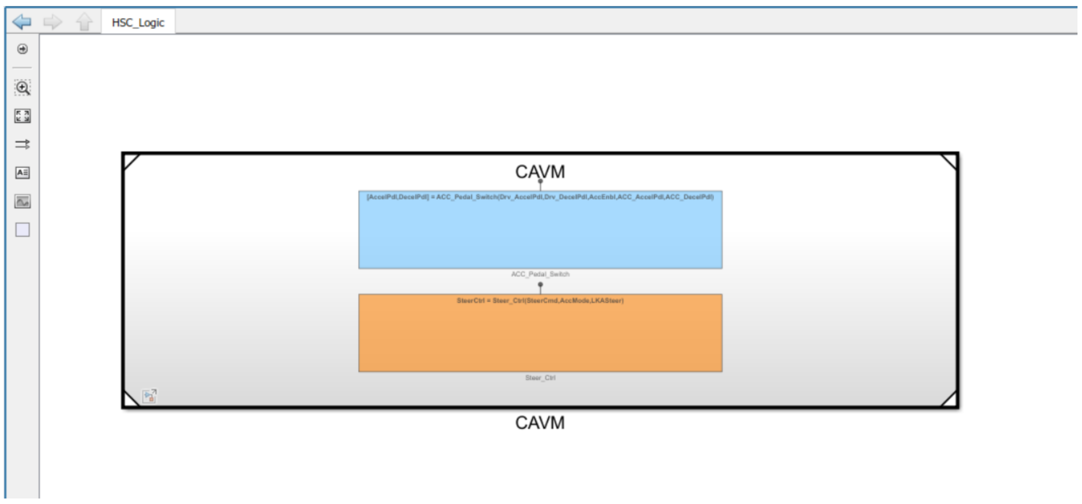
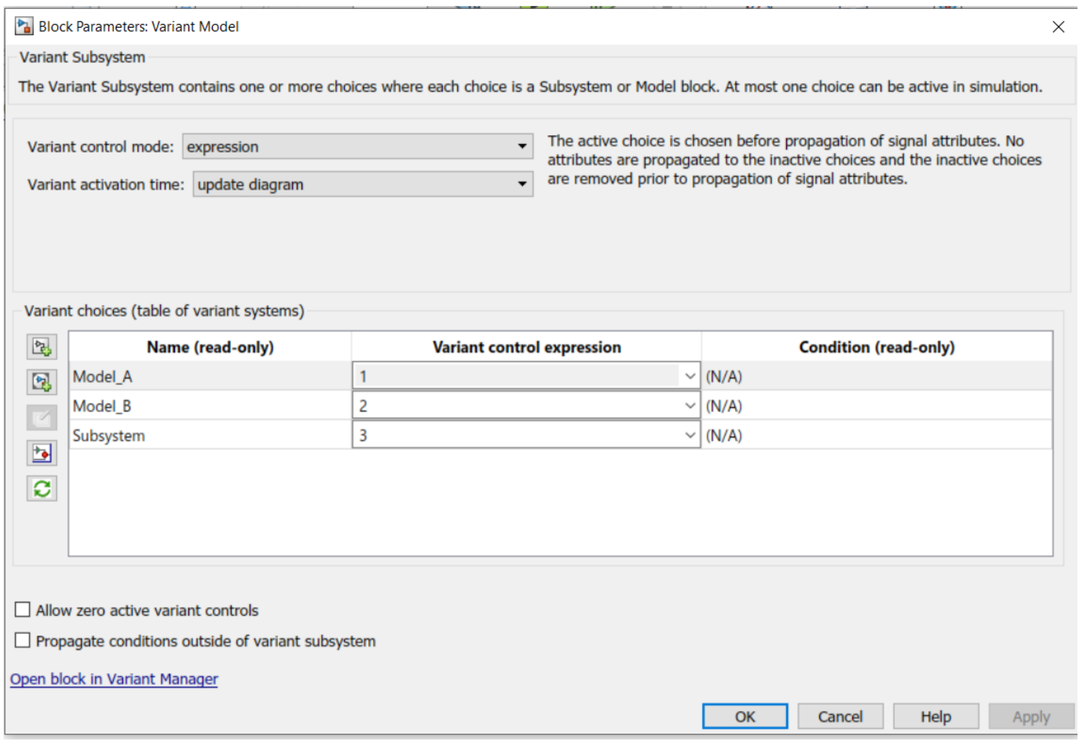

HSC_Logic
CAVM
[AccelPdl,DecelPdl] = ACC_Pedal_Switch(Drv_AccelPdl,Drv_DecelPdl,AccEnbl,ACC_AccelPdl,ACC_DecelPdl)
ACC_Pedal_Switch
SteerCtrl = Steer_Ctrl(SteerCmd,AccMode,LKASteer)
Steer_Ctrl
CAVM

FIGURE 2.4: CAVM implemented as a model reference within the HSC root model


Block Parameters: Variant Model

Variant Subsystem

The Variant Subsystem contains one or more choices where each choice is a Subsystem or Model block. At most one choice can be active in simulation.

Variant control mode: expression

Variant activation time: update diagram

The active choice is chosen before propagation of signal attributes. No attributes are propagated to the inactive choices and the inactive choices are removed prior to propagation of signal attributes.

Variant choices (table of variant systems)

Name (read-only)	Variant control expression	Condition (read-only)
Model_A	1	(N/A)
Model_B	2	(N/A)
Subsystem	3	(N/A)

☐ Allow zero active variant controls

☐ Propagate conditions outside of variant subsystem

Open block in Variant Manager

FIGURE 2.5: Simulink Model Variant

2.2.1.3 Model Variants

Another common construct used within the Simulink Environment is the Model Variant Subsystem. A model variant subsystem allows for the selection of multiple different model references or subsystem blocks. Control variables and identifiers are applied to each model variant, and determine which reference is active during simulation [31, page 779]. Several control modes are available within the block parameters dialog (shown in Figure 2.5), however the **Expression** option was used exclusively throughout this thesis. The expression control mode indicates that the active variant choice is determined prior to simulation of signal propagation within the model [31, page 799]. At run-time, only the chosen variant will be active and the remaining options will be commented out. Model variants have many advantages, including the ability to create a multitude of varying implementations of the same function, and allow for the dynamic switching between them during development and testing [31, page 779]. Figure 4.13 shows an example of a Simulink Model variant used to implement XIL testing, and is discussed in detail in Section 4.3.2.

2.2.1.4 Virtual Subsystems

Subsystems are fundamental Simulink constructs used for grouping blocks together based on functionality. Subsystems are categorized as virtual if they provide graphical organization without have an impact on model execution [31, page 246]. Figure 2.6 shows a simple example of a virtual subsystem, grouping together three input

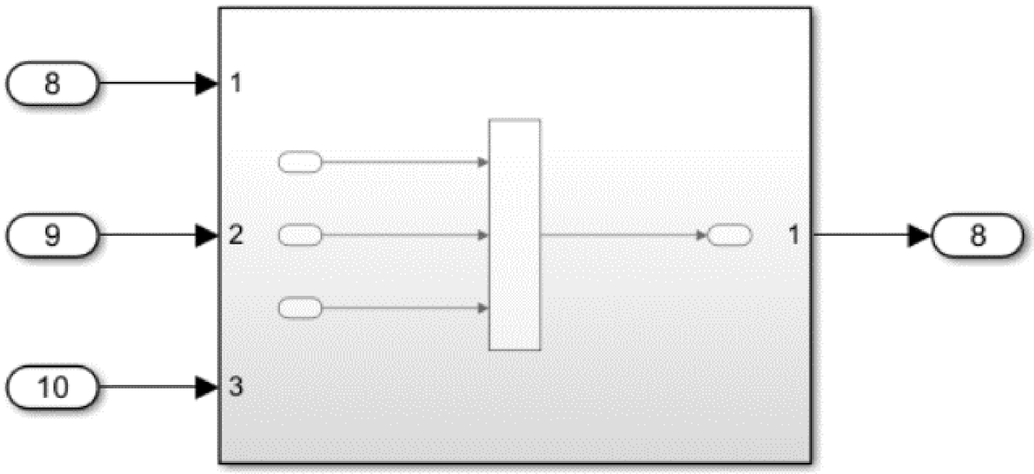


FIGURE 2.6: Example of three signal addition hidden within a virtual subsystem.

signals added together to form one output signal. Regardless of whether the virtual subsystem is present or the addition occurs at the root level, the functionality remains identical (Figure 2.7).

2.2.1.5 Simulink Functions

A Simulink Function is a construct which receives a set of input signals, performs a computational task, and produces a set of output signals. Its behaviour is comparable to that of a traditional software function written in MATLAB or C++ [31, page 587]. Simulink functions can be implemented within various constructs including Simulink Function Blocks, Exported Stateflow graphical functions, Exported Stateflow MATLAB functions, and S-functions. As it relates to this thesis, only the Simulink Function block was used within the HSC Model Development, an example of which can be seen in Figure 2.8. The example *Test _Function()* is implemented as a Simulink function (highlighted orange), and is invoked via its corresponding Simulink function caller block, highlighted blue.

Each Simulink Function must have a corresponding Simulink Function caller block, which is responsible for invoking function execution wherever needed within the model hierarchy. The function prototype defined within the Simulink Function caller block specifies the external interface, including relevant signal attributes such as size, data type, complexity, and unit. The contents of the Function prototype parameter for the corresponding function caller blocks must match the interface specified in the Simulink Function Block [31, page 590]. Some of the benefits of Simulink Function Blocks include decreasing the number of routed signal lines, allowing for multiple function callers to the same function block, as well as separating the function interface from the internal implementation [31, page 592]. As will be shown in Section 2.4 and

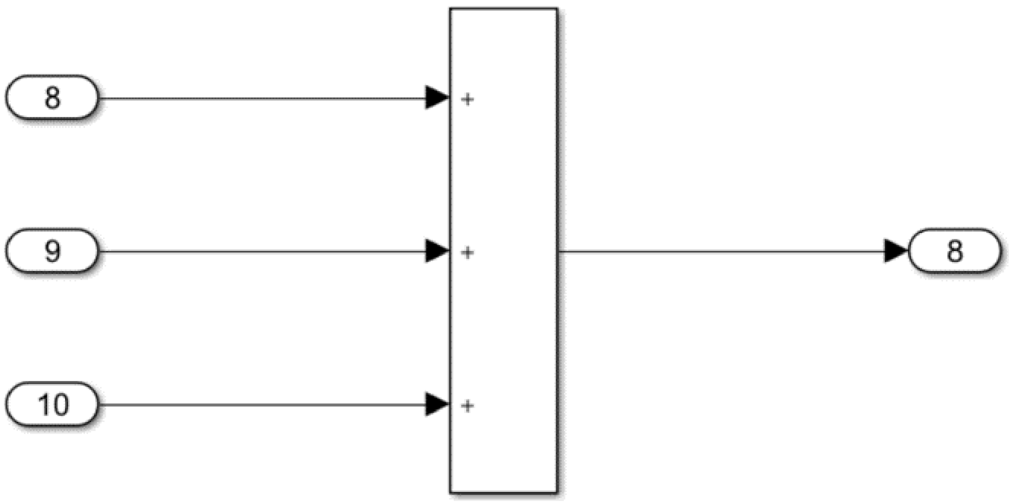


FIGURE 2.7: Functionality remains the same after virtual subsystem removal

2.5, the Simulink Function Block is particularly useful in implemented modularity within the Simulink framework.

2.2.1.6 Bus Objects

The predominant composite signal types used within the Simulink Environment include the virtual, and non-virtual Bus. A virtual Bus object is simply a visual grouping and organization of a set of Simulink signals, typically used to reduce visual complexity and increase model organization [31, page 3506] . A model or subsystem with a large number of output signals can be consolidated to a single Bus signal using a Bus Creator Block, reducing the signal routing needed to pass information within the model hierarchy. A corresponding Bus Selector block can be used to decompose a given Bus signal into its comprising elements [31, page 3512]. Despite the visual appearance, signals are not grouped in any functional sense and do not affect simulation or code generation.

In contrast, a non-virtual Bus object groups signals both visually and functionally. A Simulink Bus Object must be defined for each non-virtual bus, including a specification of each signal within the grouping. A key difference between these methods of Bus creation is that in the case of a Simulink Bus Object, all signals contained within it must explicitly define their name, data type, size, and complexity. Any Bus creator or selector block which includes a non-virtual bus must explicitly set the corresponding Simulink Bus data type, as well as ensure that all incoming and outgoing signals match the interface specifications within the Bus Object definition [31, page 3555 - 3557]. The Bus Editor within Simulink is used to create Bus Object definitions, and define the Bus Hierarchical structure. Bus Objects can be defined within the base workspace, model workspace, or within a Simulink Data Dictionary.

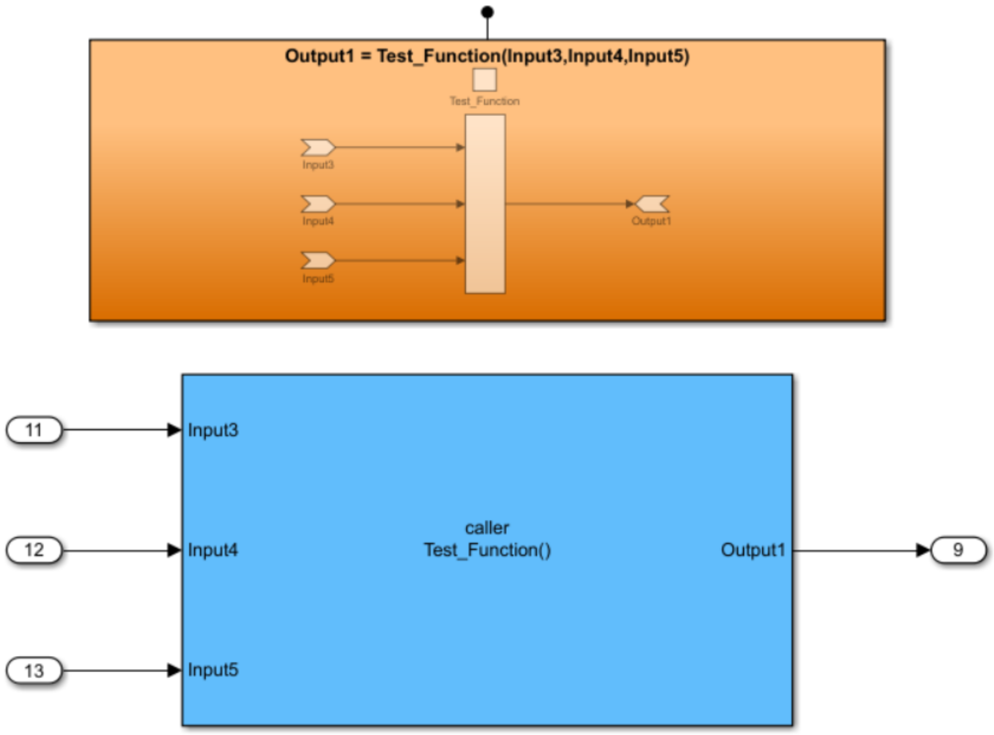


FIGURE 2.8: Example of Simulink function with corresponding function caller block

Figure 2.9 shows an example of a non-virtual bus within the EcoCAR HSC Model. The Bus Object, *HSECOLAN_In* is decomposed into its various signal elements using a Bus Creator block, with the **output as virtual bus** option disabled (shown in Figure 2.10). The definition of the *HSECOLAN_In* Bus Object can be seen within the Simulink Bus Editor, shown in Figure 2.11. The overall Bus object hierarchy shown is defined within the *Global.sldd* Simulink Data Dictionary. Details regarding the use of Simulink Bus Objects within the EcoCAR Model are discussed in Sections 4.3.5 and 5.3.2.

2.2.1.7 Goto/From Tags

One of the most common blocks used within the Simulink Environment is the Goto and From Tags. The Goto block accepts a single signal input, and must be connected to a corresponding From Tag block, which has a single output signal. Tags are primarily used as a method of signal routing that avoids direct line connections. Removing signal lines and routing can greatly increase visual organization of the model, and lead to cleaner, easily understandable model design [25]. Figure 2.12 shows an example of signal routing using Goto/From Tags. The Goto Tag (highlighted in blue) receives the input signal, *In_1* , which is passed to the corresponding From Tag (highlighted in purple). Goto/From Tag combinations can also be used to pass data implicitly within the model hierarchy, by defining a specified Tag scope. The Tag Visibility parameter can be used to limit access to a particular Goto/From Tag; the visibility is defaulted to local, meaning that for a given Goto Tag, the corresponding From Tag is only valid if it is located within the same model hierarchical level [24]. Visibility can also be set to Global, in which case a From Tag can be valid anywhere within the model hierarchy. There are limitations to the implicit passing of data using Goto/From Tags, which are further discussed in Section 2.4.

2.2.1.8 Data Dictionaries

A Simulink Data Dictionary is a repository of persistent data within the Simulink Model Environment. Relevant signals and parameters can be defined within a data dictionary and used throughout a model as a replacement for the MATLAB Base workspace. The dictionary can be linked to individual models, providing the primary data source during simulation and code generation [31, page 3368-3369]. Data dictionaries can be further broken down into sub-dictionaries through the use of Data Dictionary references. This capability introduces a data dictionary hierarchy and further refines the organization and grouping of relevant data. The use of data dictionaries within the EcoCAR model is described in detail in Section 4.3.3 and Section 5.3.2.

2.2.1.9 MATLAB Base Workspace

The MATLAB workspace contains the set of all imported variables as well as those defined through the MATLAB command line [22]. All variables defined within the base workspace have a global scope, and are accessible anywhere in a Simulink model hierarchy. The benefits provided by the base workspace are most notable in systems with a low number of models, as all relevant model data is consolidated to a single

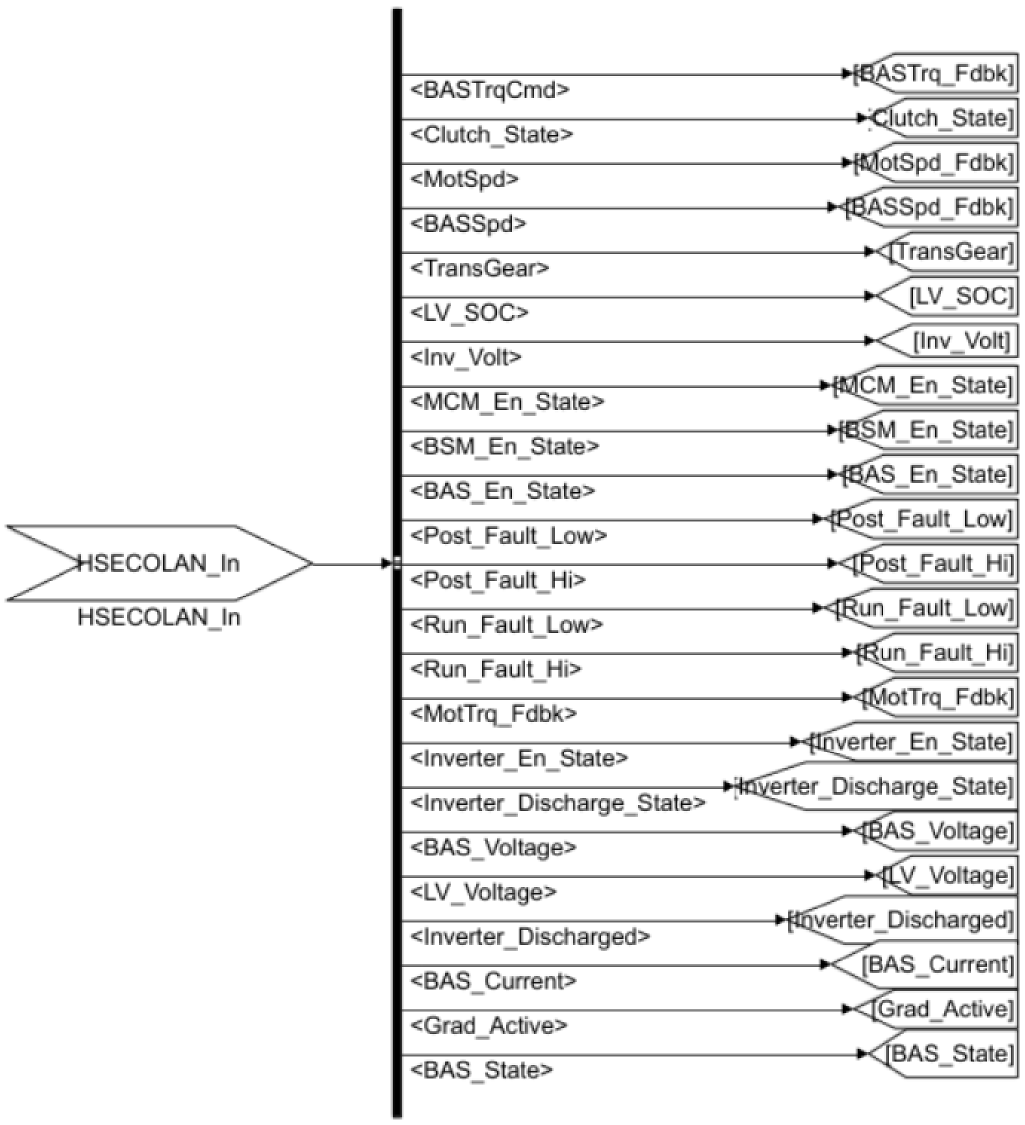
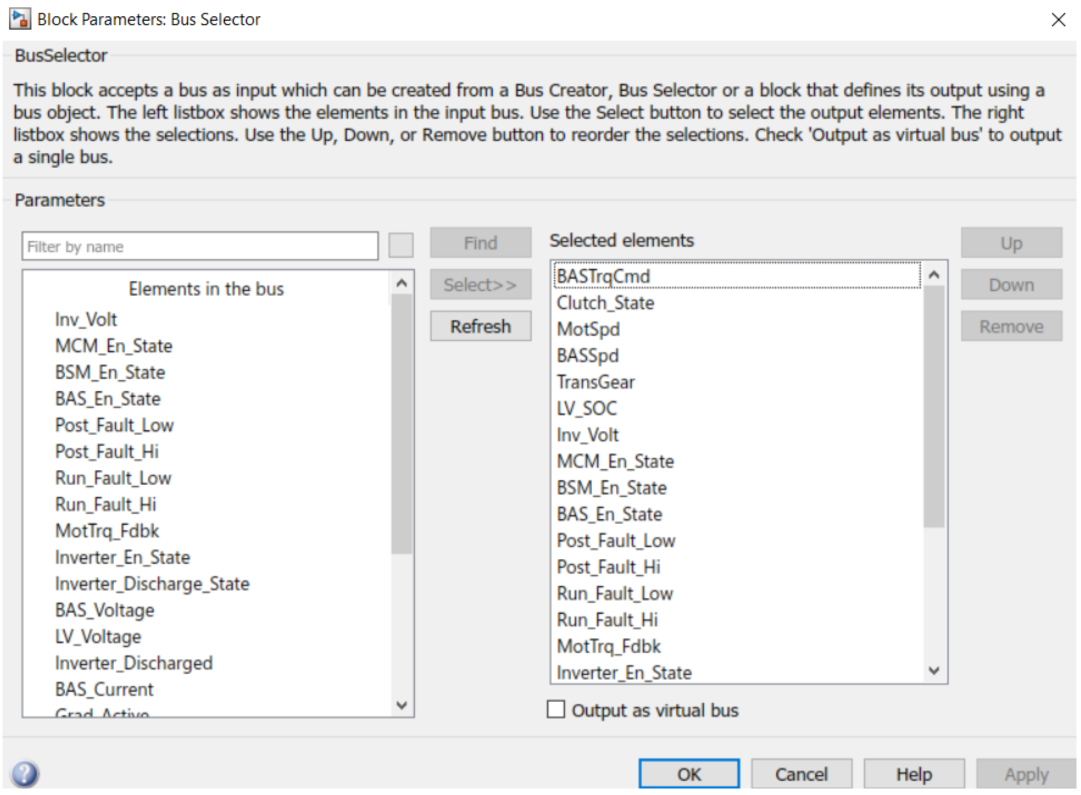


FIGURE 2.9: Non-virtual Bus Object


Block Parameters: Bus Selector

BusSelector

This block accepts a bus as input which can be created from a Bus Creator, Bus Selector or a block that defines its output using a bus object. The left listbox shows the elements in the input bus. Use the Select button to select the output elements. The right listbox shows the selections. Use the Up, Down, or Remove button to reorder the selections. Check 'Output as virtual bus' to output a single bus.

Parameters

Filter by name

Elements in the bus

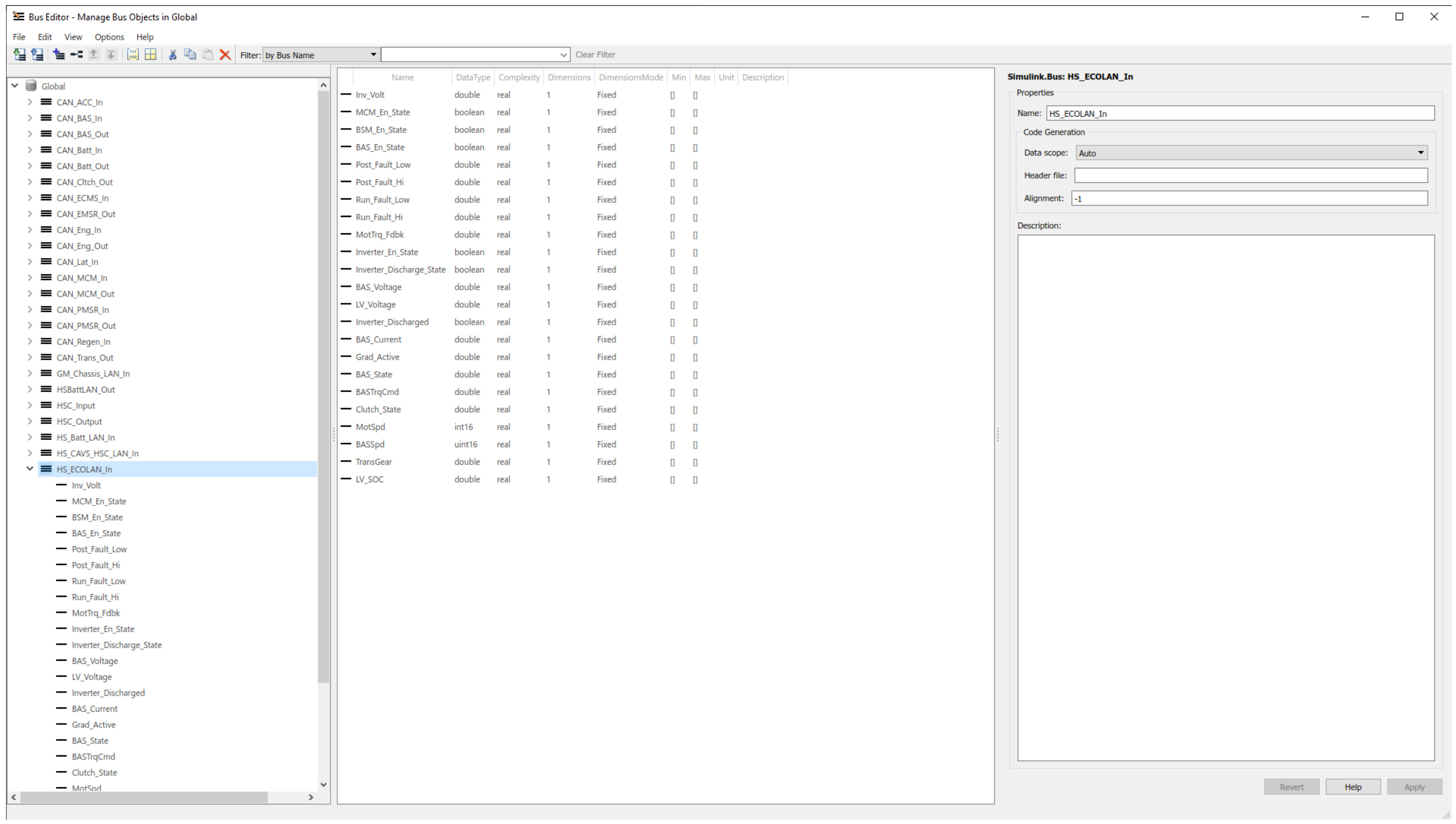
Inv_Volt
MCM_En_State
BSM_En_State
BAS_En_State
Post_Fault_Low
Post_Fault_Hi
Run_Fault_Low
Run_Fault_Hi
MotTrq_Fdbk
Inverter_En_State
Inverter_Discharge_State
BAS_Voltage
LV_Voltage
Inverter_Discharged
BAS_Current
Grad_Active

Selected elements

BASTrqCmd
Clutch_State
MotSpd
BASSpd
TransGear
LV_SOC
Inv_Volt
MCM_En_State
BSM_En_State
BAS_En_State
Post_Fault_Low
Post_Fault_Hi
Run_Fault_Low
Run_Fault_Hi
MotTrq_Fdbk
Inverter_En_State

☐ Output as virtual bus

FIGURE 2.10: Bus Selector Block Parameters



Global
- CAN_ACC_In
- CAN_BAS_In
- CAN_BAS_Out
- CAN_Batt_In
- CAN_Batt_Out
- CAN_Cltch_Out
- CAN_ECMS_In
- CAN_EMSR_Out
- CAN_Eng_In
- CAN_Eng_Out
- CAN_Lat_In
- CAN_MCM_In
- CAN_MCM_Out
- CAN_PMSR_In
- CAN_PMSR_Out
- CAN_Regen_In
- CAN_Trans_Out
- GM_Chassis_LAN_In
- HSBattLAN_Out
- HSC_Input
- HSC_Output
- HS_Batt_LAN_In
- HS_CAVS_HSC_LAN_In
- HS_ECOLAN_In
  - Inv_Volt
  - MCM_En_State
  - BSM_En_State
  - BAS_En_State
  - Post_Fault_Low
  - Post_Fault_Hi
  - Run_Fault_Low
  - Run_Fault_Hi
  - MotTrq_Fdbk
  - Inverter_En_State
  - Inverter_Discharge_State
  - BAS_Voltage
  - LV_Voltage
  - Inverter_Discharged
  - BAS_Current
  - Grad_Active
  - BAS_State
  - BASTrqCmd
  - Clutch_State
  - MotSpd

Name	DataType	Complexity	Dimensions	DimensionsMode	Min	Max	Unit	Description
Inv_Volt	double	real	1	Fixed	[]	[]		
MCM_En_State	boolean	real	1	Fixed	[]	[]		
BSM_En_State	boolean	real	1	Fixed	[]	[]		
BAS_En_State	boolean	real	1	Fixed	[]	[]		
Post_Fault_Low	double	real	1	Fixed	[]	[]		
Post_Fault_Hi	double	real	1	Fixed	[]	[]		
Run_Fault_Low	double	real	1	Fixed	[]	[]		
Run_Fault_Hi	double	real	1	Fixed	[]	[]		
MotTrq_Fdbk	double	real	1	Fixed	[]	[]		
Inverter_En_State	boolean	real	1	Fixed	[]	[]		
Inverter_Discharge_State	boolean	real	1	Fixed	[]	[]		
BAS_Voltage	double	real	1	Fixed	[]	[]		
LV_Voltage	double	real	1	Fixed	[]	[]		
Inverter_Discharged	boolean	real	1	Fixed	[]	[]		
BAS_Current	double	real	1	Fixed	[]	[]		
Grad_Active	double	real	1	Fixed	[]	[]		
BAS_State	double	real	1	Fixed	[]	[]		
BASTrqCmd	double	real	1	Fixed	[]	[]		
Clutch_State	double	real	1	Fixed	[]	[]		
MotSpd	int16	real	1	Fixed	[]	[]		
BASSpd	uint16	real	1	Fixed	[]	[]		
TransGear	double	real	1	Fixed	[]	[]		
LV_SOC	double	real	1	Fixed	[]	[]		

Simulink.Bus: HS_ECOLAN_In

Properties

Name: HS_ECOLAN_In

Code Generation

Data scope: Auto

Header file:

Alignment: -1

Description:

Figure 2.11: Simulink Bus Editor

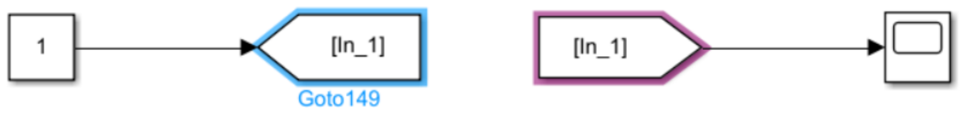


FIGURE 2.12: Goto/From Tags

location allowing for ease of access. As the number of models increases, the global scope of the base workspace poses significant issues including lack of information hiding as well as increasing the chances of errors in variable modification propagating throughout the model hierarchy.

2.2.1.10 Simulink Model Workspace

In addition to a MATLAB base workspace, each model within a Simulink Project has an individual model workspace. The signals and parameters defined within a model workspace are locally scoped within their corresponding Simulink model, and are not accessible from higher Model hierarchical levels [31, page 3025]. The model workspace provides a local name space in which data only required within a given model can be grouped together, facilitating information hiding and reducing chances of variable conflicts. The source of data for a given model workspace can be a Model file, MAT-file, MATLAB file, or MATLAB code stored within a model file [31, page 3027]. The use of model workspaces as opposed to the base workspace is critical in the process of modularity used throughout this thesis. The role of model workspaces within the Simulink Modularity Guidelines are described in Section 2.5.3, and the application of these guidelines are described in Chapter 5.

2.2.2 Stateflow Constructs and Overview

Stateflow is an environment within Simulink used primarily for timed and decision-based logic and state machine design [32]. This type of construct was particularly useful during the construction of the hybrid vehicle model as it allowed for component functionality to be modeled by state behavior and transitions.

2.2.2.1 Stateflow Charts

A Stateflow chart is a Simulink construct used to include sequential and combinatorial logic within a model [32, page 36]. The internals of a Stateflow chart can include Simulink functions, MATLAB functions, or graphical functions composed of states and transitions [34, page 37], an example of which can be seen in Figure 2.13. Stateflow charts are used to model state machines within the Simulink environment, and are particularly useful within the automotive industry for representing component and hardware operating modes.

2.2.2.2 States and Transitions

State machines can be created within a Stateflow Chart or graphical function using state and transition blocks. Each State has an associated functional output and transitions between states occur if the conditional statement for transition is true [32, page 36]. Transitions are represented by arrows connecting state blocks within a Stateflow Chart, and are typically associated with a condition that must be met in order to move from one state to another. In the case of the example given, if the condition: *BASAvail == 0* is true, the system will transition from the *Shutdown* state to the *FrontPwrtrnDisable* State. Within the context of this thesis, a state is typically used to describe an operating mode of a particular vehicle component or hardware. States can either be active or inactive at any given time step, based on whether its associated entry condition (transition) has been met [32, page 151]. Once an event drives a transition away from a particular state, it will transition from active to inactive. A hierarchical structure can also be implemented by encapsulating one state within another. The Power Moding Module shown in Figure 2.13 contains the *Conventional* State, which is a parent to the *ModeCheck*, *Shutdown*, and *FrontPwrtrnDisable* substates. Grouping states into a hierarchical structure can allow for the decomposition of state machine functionality and selectively activate or deactivate groups of states based on system events [32, page 48 - 49].

2.2.2.3 Stateflow Boxes

Boxes are primarily used for organization of Stateflow functions and state machines within a given Stateflow chart [32, page 272]. There also exists the added benefit of introducing a local namespace within each Box, allowing for the separation of functions and states. The ability of Stateflow Box objects to locally scope functions through encapsulation allows for the possibility of increased information hiding within a given Stateflow Machine [34, page 37]. The visibility of functions and states encapsulated by a Box object is restricted to that particular Stateflow chart, and cannot be exported to higher model hierarchical levels. If a particular set of functions or states need to be accessed outside of the Stateflow chart, the **Export chart level functions** and **Treat exported functions as globally visible** options must be selected within the Chart properties dialog box [32, page 293].

2.3 Modular Software Principles

Modular software principles are the design decisions which separate a larger system into independent modules [40, page 2]. Traditional software decomposition criteria separates each step in the system process as an individual module. Modeling Software such as Simulink follows a similar approach, with each model executing its blocks from left to right, following the data flow process. The primary motivation for decomposing a large system into modules is to minimize the software cost through independent development and testing [38, page 4]. Additional benefits of modular programming include shortened development time, ease of change, and comprehensibility.

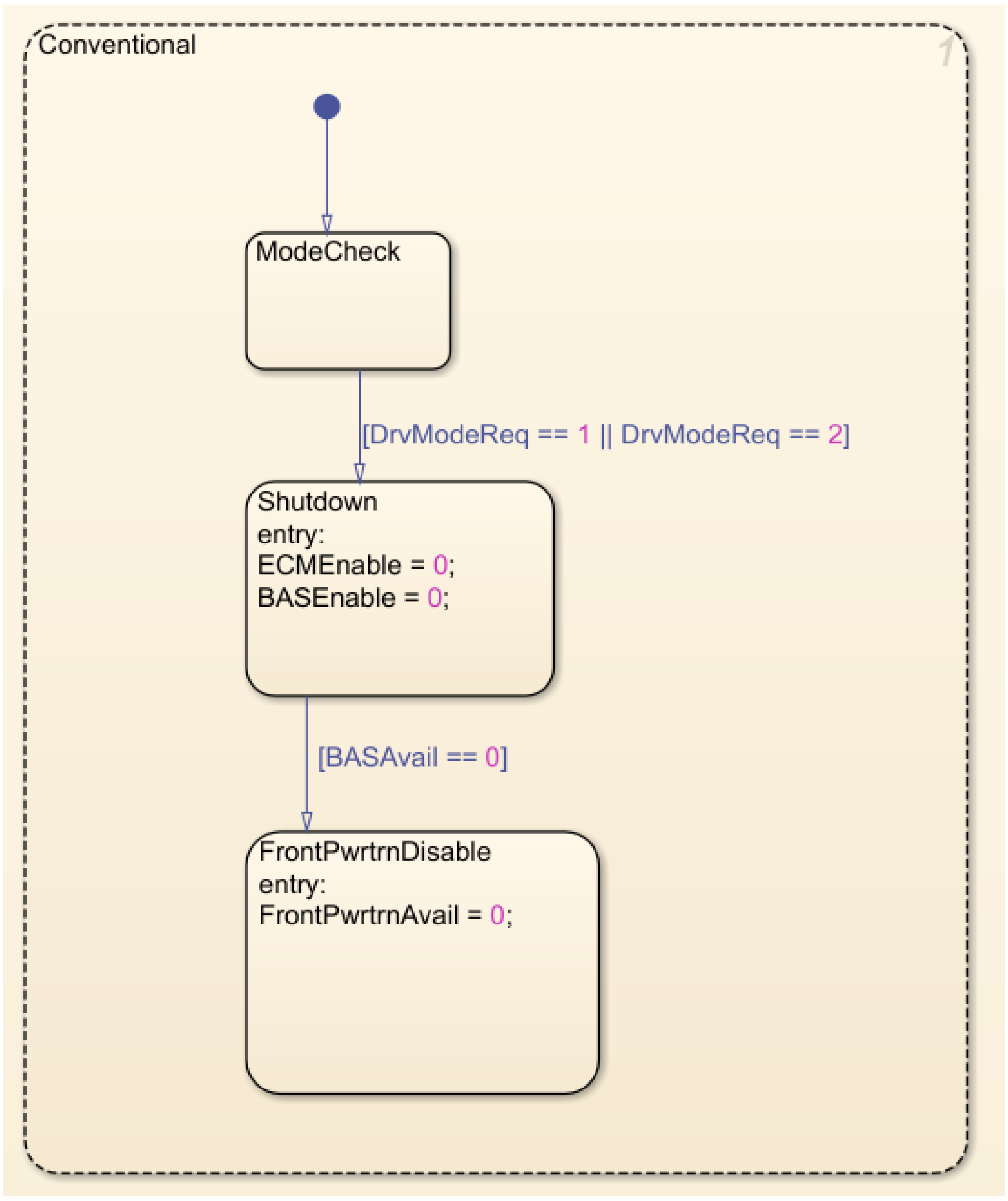


FIGURE 2.13: Power Moding Stateflow

2.3.1 Information Hiding

When information hiding is used as a criterion for decomposing software, modules are determined by the design decisions and secrets they hold. Likely changes to the system which occur independently should be grouped into separate modules [38, page 3]. Algorithms which share common data or functionality are grouped together, irrespective of their position or prevalence within the overall system process [40, page 4]. Unique data structures as well as the functions that access and modify them should be consolidated within a single module. Future changes in the data structure will be prevented from propagating to all the modules that depend on the information contained within it [40, page 4]. Process order should not dictate the grouping of algorithms within modules, but instead should make multiple calls to modules containing similar design decision secrets [40, page 4]. Modular decomposition also improves the system changeability, which refers to the level of impact that a likely change will create. It is useful to examine the extent to which a design decision change propagates outside of its given module. A successfully modularized system would minimize the change propagation of design decisions, allowing for independent development and testing of modules without external conflicts [40, page 3].

2.3.2 Coupling and Cohesion

Coupling refers to the connections between modules within a software system [44, page 2]. It is desirable to minimize the number of connections between modules, as well as the complexity of the connections themselves. Reducing connections between modules will allow for easier understanding and debugging of individual functions, as well as reducing the system propagation of module errors [44, page 3]. A primary cause for high coupling within software systems comes from the use of common data areas, or global variable namespaces. Low coupling among modules leads to a simpler, more modular system, allowing for independent development and testing. High coupling increases the chances of error propagation as well as increases the complexity of module interfaces [44, page 4]. One of the most reliable ways of reducing coupling is regrouping the elements within each module such that their internal connections, or module cohesion, is maximized [44, page 7]. Functional cohesion represents the strongest form of connection existing between two components within a module, and occurs when each module encapsulates closely related secrets and algorithms [44, page 7]. High cohesion results in likely software changes being restricted to a lower number of individual modules, as opposed to propagating throughout a low cohesion system. A primary indicator of a well modularized software system is one in which coupling is minimized, and cohesion is maximized.

2.3.3 Interface Complexity

Interface complexity refers to the number of connections and dependencies needed to fully define the external interactions involved with a given Module [44, page 5]. An interface which exposes details regarding the module internal implementation will have a higher complexity and increase the coupling unnecessarily [44, page 5]. To facilitate independent development, the interface complexity between modules should be simplified as much as possible. Designing Module interfaces with complex data structures that have a high number of dependencies will result in a large amount of development time spent on coordination between modules [40, page 4]. Proper system functionality will require a high level of collaboration between developers of each connected module and will increase overall complexity and time of development. An interface should consolidate to as abstract of a level as possible, ideally only including names of exported functions as well as input and output parameters. The interface specification should provide a user with just enough information to run the program, and an implementer just enough information to complete the program [39, page 1].

2.4 Simulink Construct Comparison

Limited work has been done to determine the extent that information hiding, reusability, and separation of concern can be achieved within Simulink. The most rigorous analysis has come from [18], which performs a Simulink Construct comparison. Jaskolka et. al notes that general decomposition guidelines exist within the Simulink User's Guide, but do not address the subject of modularity or information hiding directly [31, pages 1047-1089]. A more detailed and modularity focused analysis was performed in this paper and provided the framework for development of modularity guidelines and the Simulink Module Tool [36]. This paper provides a unique analysis of Simulink constructs, and seeks to determine the extent to which each facilitates modularity, encapsulation, and information hiding. The analysis is comprised of five different Simulink constructs: the virtual subsystem, the atomic subsystem, Simulink function, Library, and Model Reference. These commonly used constructs were compared according to four distinct categorizations: reusability, sharing of program state, encapsulation, and code generation [18, page 3-4]. A brief description of Simulink constructs relevant to this thesis can be found in Section 2.2. The results of analysis performed in [18] can be seen in Table 2.1.

Constructs which are not reusable include Virtual subsystems and Atomic subsystems. Virtual subsystems are merely graphical encapsulations to increase model aesthetics and organization, and therefore cannot be reused throughout the model without duplicating all the blocks and signals contained within it. The virtual subsystem example from Figure 2.6 cannot be reused multiple times without the three way addition block also being duplicated. This does not represent reuse, as copying subsystems would greatly increase the duplicated code within the system [18, page 6]. Model references can be used multiple times throughout a system and therefore are reusable. Similarly, blocks or functions created within a library can be reused throughout multiple models. Functionality provided by a Simulink Library construct can be included in any model that imports the related block library. Finally, a Simulink Function shares similar reusability to a C function. In the Simulink function example shown in Figure 2.8, *Test_Function()* can be invoked as many times as needed, simply by created multiple different Simulink function caller blocks. This allows for an algorithm defined within a Simulink function to be used throughout a system without introducing duplicate code [18, page 6].

The constructs were also compared in regards to their sharing of program state, where multiple instances of a construct sharing the same set of data may increase

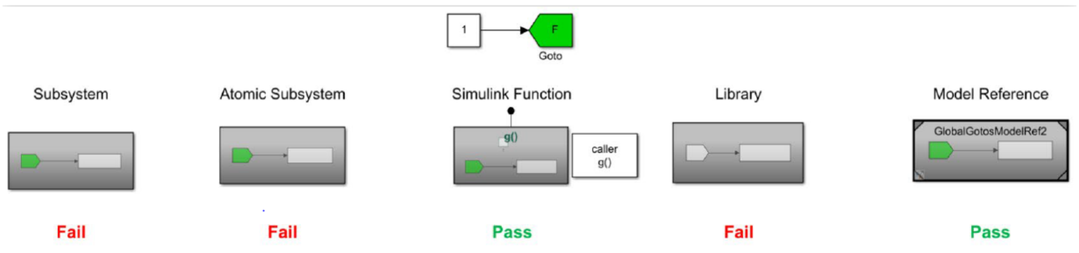


FIGURE 2.14: Implicit Goto/From Tag Input test results, taken from [18]

complexity and likelihood of unexpected changes propagating throughout the model hierarchy. Libraries, virtual, and atomic subsystems are not reusable and therefore do not share program state between duplicate copies [18, page 7]. Both Simulink Functions and Model references do not share program state across instances. However, program state sharing can be implemented under certain parameter configurations [18, page 7].

In order to analyze the level of information hiding that can be created with each construct, a series of tests were created in [18] to analyze both limitation of use, as well as restriction of data flow. The restriction of data flow was analyzed using both Goto/From Tags, as well as Data store Read/Write blocks. The results of the implicit input and output tests for Goto/From Tags performed in [18] can be seen in Figures 2.14 and 2.15, respectively. In the case of implicit input using Goto/From tags with a global scope, only Simulink functions, and model references successfully restricted access to internal data within the constructs. Implicit tag output testing resulted in Simulink Functions, Model References, and atomic subsystems successfully restricting data flow [18, page 8]. The implicit passing of data store Read and Write blocks was also investigated, with Model References being the only construct that can restrict data store reading and writing through its interface [18, page 8]. Looking at the analysis results from Table 2.1, the Simulink Function construct was identified as the best suited for implementation of information hiding within Simulink Models. The ability to scope a Simulink Function and export it to higher model hierarchical levels introduces the ability to limit access to the secrets contained within modules. The results of the Simulink construct comparison described in [18] is used extensively throughout this thesis, and provides the foundation for further Simulink Module development outlined in Section 2.5.

2.5 Modularity within Simulink

The construct comparison results outlined in Section 2.4 were a primary determinant in the modular structure designed by Jaskolka et. al in [36] and [35]. This structure along with the accompanying modularity guidelines are outlined in Sections 2.5.1 - 2.5.3. The MATLAB tool developed by Jaskolka to implement these modular guidelines is outlined in Section 2.5.4.

Table 2.1: A comparison of Simulink constructs, derived from [18]

Construct	Reusable	Shared State	Limitation of Use	Restriction of Data Flow		Code Generation
				Goto/From	Data Store	
Subsystem	No	No	No	No	No	Inlined Code
Atomic Subsystem	No	No	No	No	No	Separate function if used multiple times; otherwise inlined code
Simulink Function	Yes	Yes, by default. Can reset using other blocks	Yes	Yes	No	Separate function (If exported, in separate module also)
Library	Yes	No	No	No	No	Separate function if nonvirtual and used multiple times; otherwise inlined code
Model Reference	Yes	Data Store only	No	Yes	Local Only	Separate function, in separate module

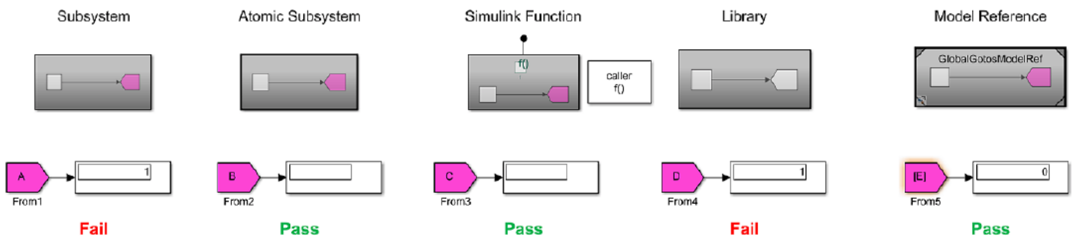
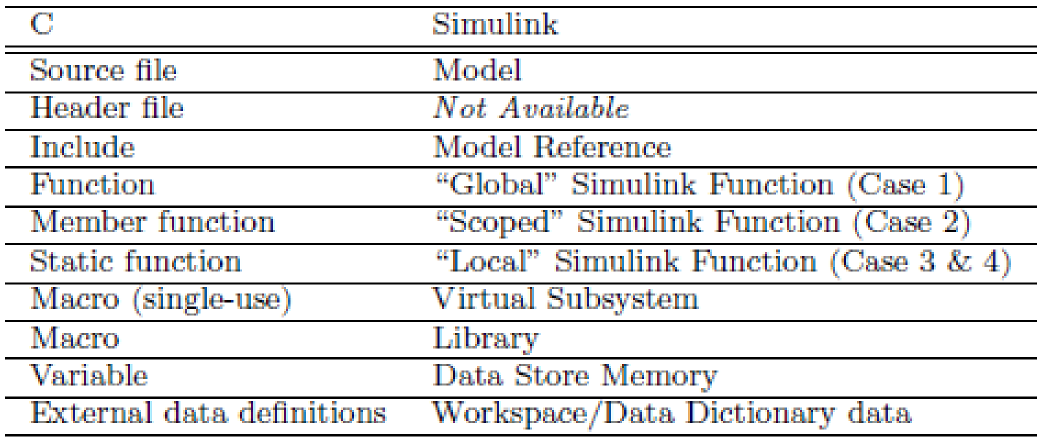


Figure 2.15: Implicit Goto/From Tag Output test results, taken from [18]

Table 2.2: Simulink and C construct Comparison, taken from [36]



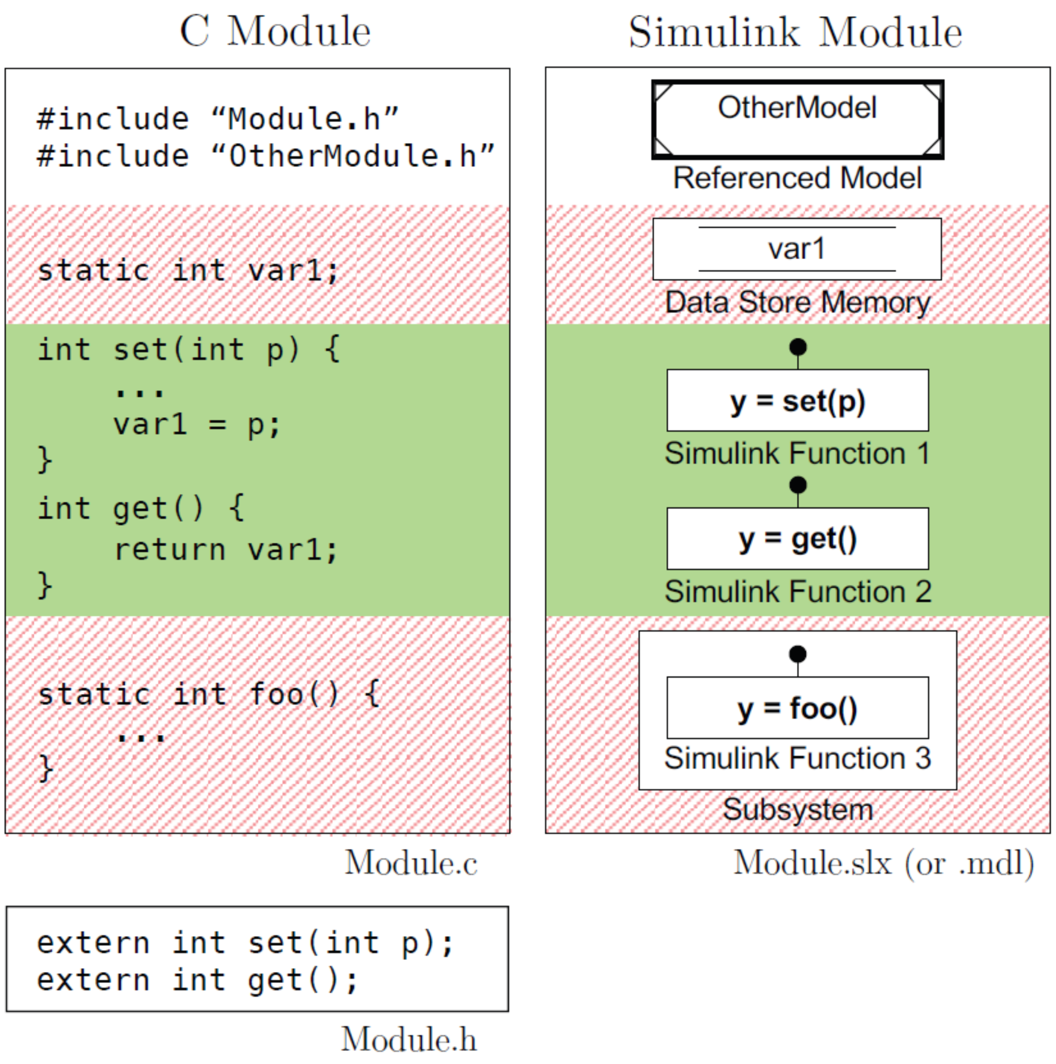
C	Simulink
Source file	Model
Header file	*Not Available*
Include	Model Reference
Function	"Global" Simulink Function (Case 1)
Member function	"Scoped" Simulink Function (Case 2)
Static function	"Local" Simulink Function (Case 3 & 4)
Macro (single-use)	Virtual Subsystem
Macro	Library
Variable	Data Store Memory
External data definitions	Workspace/Data Dictionary data

2.5.1 Definition of a Simulink Module

A process for developing a Modular structure within Simulink is defined in [19], [35], and [36]. The process of defining modules within Simulink began with a comparison between Simulink and C Constructs [36, page 9]. An excerpt of the resulting similarities can be seen in Table 2.2, and shown graphically in Figure 2.16. The structure of a software module within the C environment is shown on the left, and was used as a framework for developing a Simulink Module, shown on the right [36, page 11]. Other modules can be imported using Model References, which can act similar to an *#include* statement within C. The Simulink Function construct has the ability to be scoped both locally and globally, thus allowing for precise control over access to its internals. A similarity can be drawn to the public and static functions within the C language. A static function is accessible only within the module it is defined, and cannot be exported or accessed externally. Similarly, by encapsulating a Simulink Function within a virtual subsystem, its scope remains local to the Model in which it is defined and cannot be accessed at higher hierarchical levels. Much like a public function, a Simulink function defined at the root level of a model can be exported and used by other modules within the model hierarchy.

2.5.2 Definition of a Simulink Module Interface

A module interface consists of inputs, outputs, and exports. A well designed interface will fully describe a modules external dependencies, while minimizing access to internal module implementation [36, page 12]. Traditional Simulink Interfaces as defined by the Simulink Interface Display view only shows the combination of inport and outport blocks defined within the model [33]. A critical deficiency of this tool is its inability to display implicit communication constructs used within the model. Most notably, the Interface Display View fails to show any Simulink Function exports, which leads to an inaccurate interface definition. The Simulink Interface developed in [36, page 12] contributes additional information and is shown in Figure 2.17. The



```
#include "Module.h"
#include "OtherModule.h"

static int var1;

int set(int p) {
    ...
    var1 = p;
}
int get() {
    return var1;
}

static int foo() {
    ...
}
```

```
extern int set(int p);
extern int get();
```

Figure 2.16: Simulink and C module comparison, taken from [36]

addition of exports to the interface definition allows for the representation of Simulink functions being used by other modules within the Simulink project. The constructs highlighted in gray are recommended for finalized models ready for code generation and hardware deployment. Constructs crossed out in red are often useful and valid during development and testing stages, but are not recommended or supported for inclusion in Simulink Modules that will undergo code generation or hardware deployment [36, page 21].

As it relates to this thesis, the constructs that were primarily used include inports, outports, exported Simulink functions, Model Workspaces, and Data Dictionaries. The remaining constructs outlined in Figure 2.17, and discussed in detail in [36] were not particularly relevant to the Models discussed in Chapters 4 - 6. An example of a module interface within the EcoCAR Model is the *MCM.slx* (Motor Control Module) model, consisting of 0 inports, 0 outports, and 6 exports. If the Simulink Interface display view was used for analysis, no information would be able to be extracted due to the absence of inports or outports. Figure 2.18 shows the interface generated by the **Simulink Module Tool** (discussed in Section 2.5.4), using the definition from [36]. The 6 exported Simulink Functions are correctly represented within the Module Interface, while restricting access to the function internals. It is clear that the interface definition from [36] provides more information than just inport and outport signals, and defines the dependencies related to a given module while not exposing the internal implementations.

2.5.3 Guidelines for Modularization

Jaskolka et. al continues work on development of a Simulink Module by defining a set of modeling guidelines to follow during the modularization process [36, page 17 - 19]. The development of these guidelines was the result of a thorough analysis of existing modeling standards, including those defined by The MathWorks Advisory Boards [21]. The four primary guidelines for development of modules using Simulink functions from [36, page 18] are shown below, and are used extensively throughout the modularization of the EcoCAR Model discussed in Chapter 5. In addition to the four explicitly defined guidelines, additional rules concerning modularization of Stateflow charts were inferred from the work done by Jaskolka in [34, page 37 - 39], as well as the Stateflow Documentation [31], and are represented by Guidelines 5 - 7.

Guideline 1 (Simulink Function Placement): Place the Simulink Function block in the lowest common parent of its corresponding Function Caller blocks. Do not position the Simulink Function in the top layer for no reason. Avoid placing Simulink Function blocks below their corresponding Function Caller blocks.

Guideline 2 (Simulink Function Visibility): Limit the *Function Visibility* parameter of the Simulink Function block's trigger port to scoped if possible.

Guideline 3 (Simulink Function Shadowing): Do not place Simulink Functions with the same name and input/output arguments within each others scope.

Guideline 4 (Use of the Base Workspace): Do not use the base workspace for storing, reading, or writing data that a module is dependent on. Instead, place data in either the model workspace, if it is used in a single module, or a data dictionary if it is shared across modules.

Stateflow Guideline 5: Group all algorithms within a Stateflow Chart as either

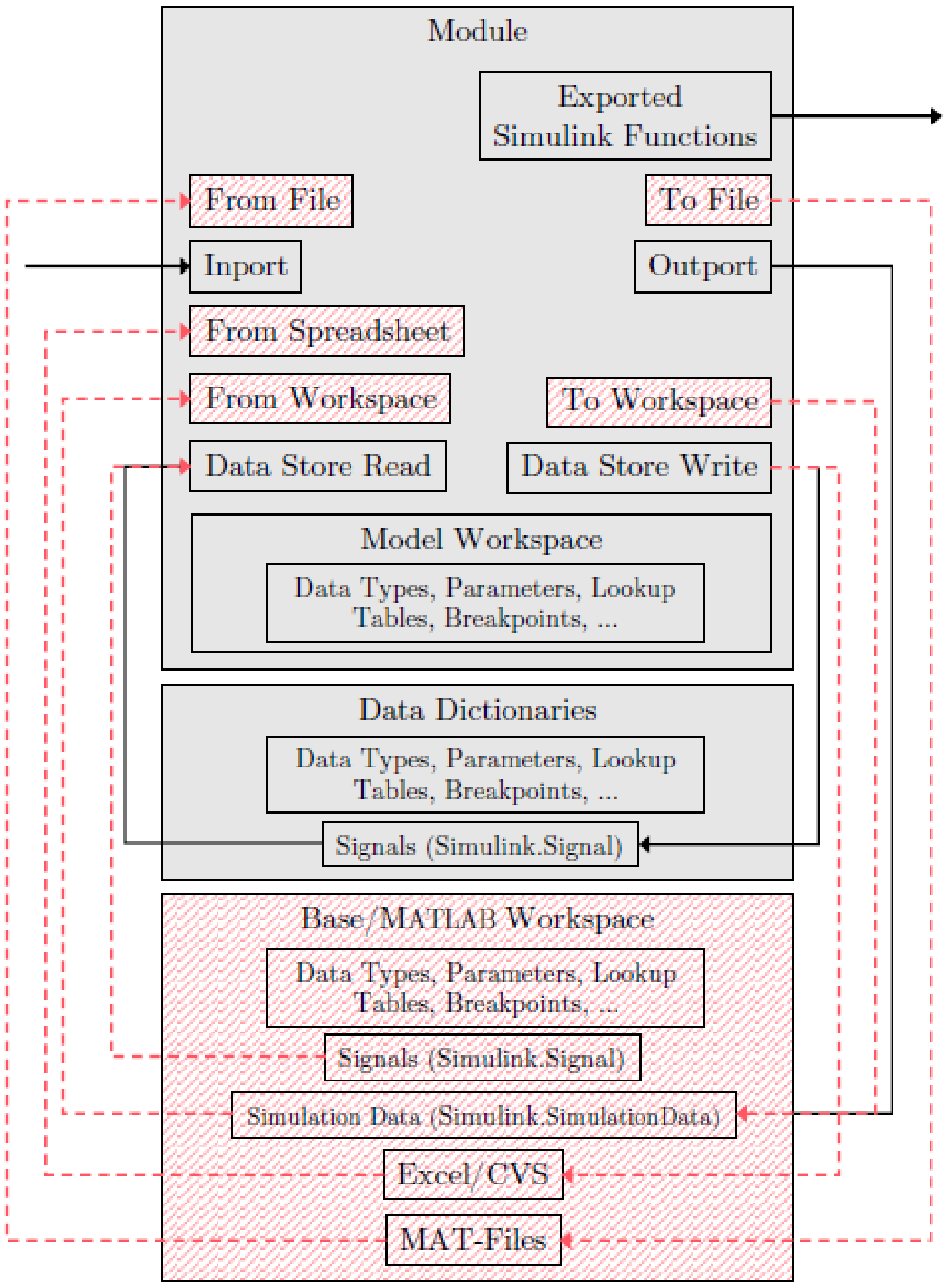

Module
Exported Simulink Functions
From File
To File
Inport
Outport
From Spreadsheet
From Workspace
To Workspace
Data Store Read
Data Store Write
Model Workspace
Data Types, Parameters, Lookup Tables, Breakpoints, ...
Data Dictionaries
Data Types, Parameters, Lookup Tables, Breakpoints, ...
Signals (Simulink.Signal)
Base/MATLAB Workspace
Data Types, Parameters, Lookup Tables, Breakpoints, ...
Signals (Simulink.Signal)
Simulation Data (Simulink.SimulationData)
Excel/CVS
MAT-Files

FIGURE 2.17: Simulink Module Interface, taken from [36]

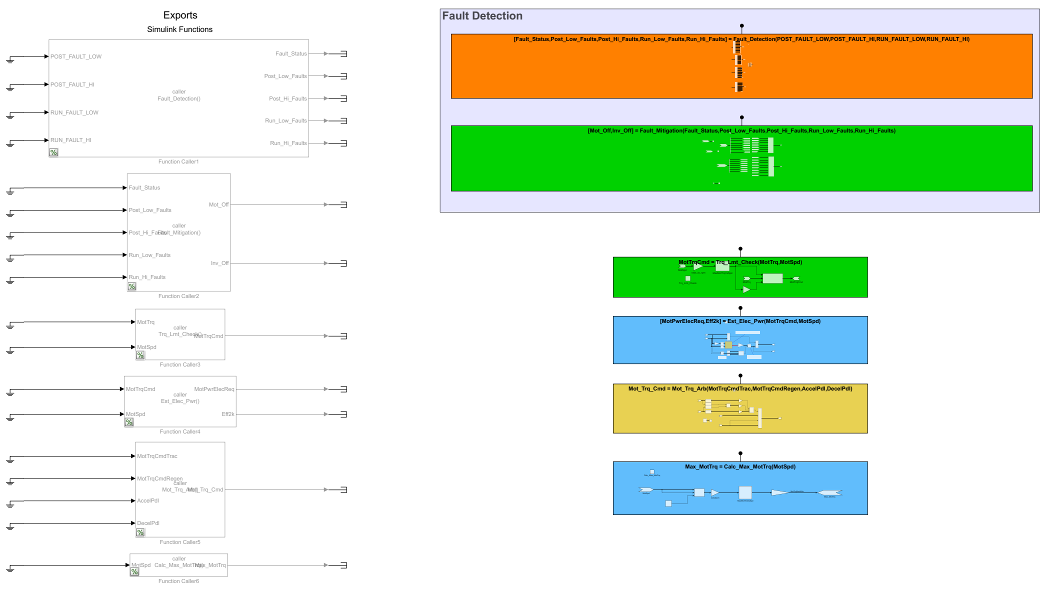


FIGURE 2.18: EcoCAR Module Interface using Simulink Module Tool

graphical state machines, MATLAB Functions, or Simulink Functions.
Stateflow Guideline 6: Make a private subset of graphical, MATLAB, or Simulink functions by encapsulating their blocks within a Stateflow Box object.
Stateflow Guideline 7: Make an exported subset of graphical, MATLAB, or Simulink functions by placing them outside of any Stateflow Box objects at the root level of the chart, and set the *export chart level functions, and treat exported functions as globally visible* property to be enabled.

2.5.4 Simulink Module Tool

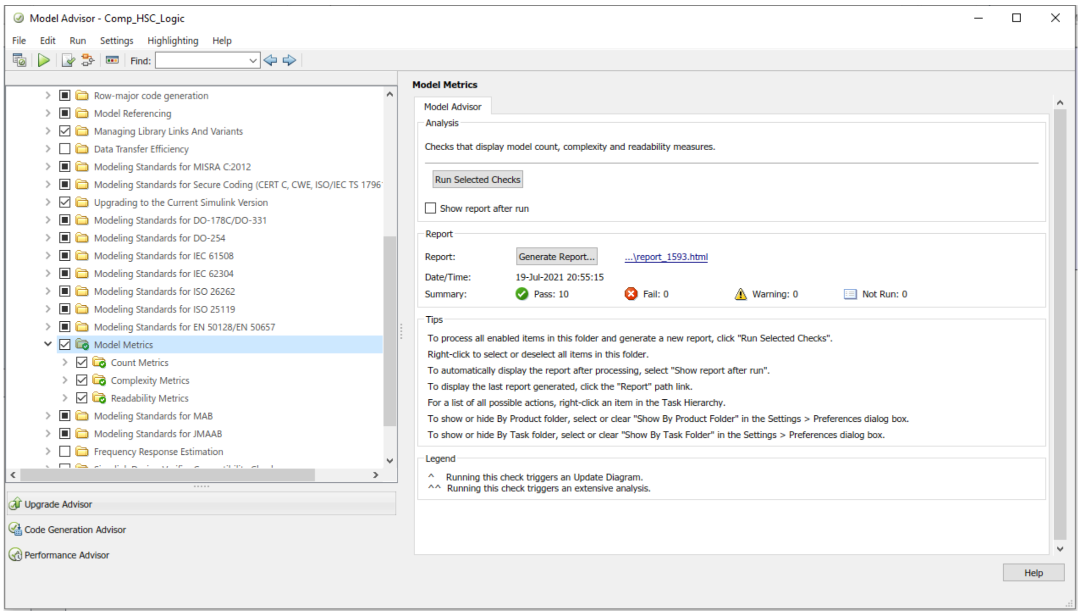
The Simulink Module Tool is an open source MATLAB add-on [7] developed by Jaskolka and described in [36, page 20 - 21]. The purpose of the tool is to provide support during the application of the Simulink Module Structure outlined in Sections 2.5.1 - 2.5.3. Conversion between subsystems and Simulink functions is easily facilitated through the menu options provided by the Simulink Module Tool. Selecting the **Convert Subsystem** menu option on a previously created subsystem will facilitate the conversion to either a global or scoped Simulink function [43, page 8]. Furthermore, a corresponding function caller block can be created by right-clicking within a given Simulink Module, and selecting the **Call function** menu option [43, page 7]. A complete list of the functions within the scope of the current model will be shown, giving the developer a clear picture of which functionality they have access to and allowing for reduced complexity in modular development. The representation of Simulink Module Interfaces as discussed in Section 2.5.2 can be generated using the Simulink Module Tool context menu. The **Show Interface** and **Print Interface** context menu options can be selected within a given module and will either print the interface to the MATLAB command window in text form, or insert the interface graphically within the model [43, page 9]. Finally, the Simulink Module Tool can check compliance to the Guidelines for Modularity developed in [36] and discussed in Section 2.5.3 using the **Check Guidelines** context menu option. A Guideline selector screen initializes, allowing for the testing of one or more of the Modular Guidelines [43, page 13].

2.6 Simulink Design Verification

The Simulink Design Verifier (SDV) Tool utilizes formal methods to perform model analysis. The tool can be run in various different modes, providing analysis of test generation, design error detection, property proving, and testing coverage [29].

2.6.1 Model equivalence using SDV Property Proving

The SDV can be leveraged to prove functional equivalence between two Simulink Models. Simulation can be run in *Property-Proving Mode*, where requirements can be specified using Proof Blocks. Functional equivalence between two models can be determined by comparing inputs using an equality logical operator fed into a Proof Objective Block [29, page 444]. If all inputs are the same the Proof Objective will be true, and the two models are functionally equivalent. This method is used in


Model Advisor - Comp_HSC_Logic
File Edit Run Settings Highlighting Help
Find:
Upgrade Advisor
Code Generation Advisor
Performance Advisor
Help

- ■ Row-major code generation
- ■ Model Referencing
- ☑ Managing Library Links And Variants
- ☐ Data Transfer Efficiency
- ■ Modeling Standards for MISRA C:2012
- ■ Modeling Standards for Secure Coding (CERT C, CWE, ISO/IEC TS 1796
- ☑ Upgrading to the Current Simulink Version
- ■ Modeling Standards for DO-178C/DO-331
- ■ Modeling Standards for DO-254
- ■ Modeling Standards for IEC 61508
- ■ Modeling Standards for IEC 62304
- ■ Modeling Standards for ISO 26262
- ■ Modeling Standards for ISO 25119
- ■ Modeling Standards for EN 50128/EN 50657
- ☑ Model Metrics
  - ☑ Count Metrics
  - ☑ Complexity Metrics
  - ☑ Readability Metrics
- ■ Modeling Standards for MAB
- ■ Modeling Standards for JMAAB
- ☐ Frequency Response Estimation

Model Metrics

Model Advisor

Analysis

Checks that display model count, complexity and readability measures.

Run Selected Checks

☐ Show report after run

Report

Report:	Generate Report...	...\report_1593.html		
Date/Time:	19-Jul-2021 20:55:15			
Summary:	✓ Pass: 10	✗ Fail: 0	⚠ Warning: 0	Not Run: 0

Tips

To process all enabled items in this folder and generate a new report, click "Run Selected Checks".
Right-click to select or deselect all items in this folder.
To automatically display the report after processing, select "Show report after run".
To display the last report generated, click the "Report" path link.
For a list of all possible actions, right-click an item in the Task Hierarchy.
To show or hide By Product folder, select or clear "Show By Product Folder" in the Settings > Preferences dialog box.
To show or hide By Task folder, select or clear "Show By Task Folder" in the Settings > Preferences dialog box.

Legend

^ Running this check triggers an Update Diagram.
^^ Running this check triggers an extensive analysis.

FIGURE 2.19: Model Advisor Property Window

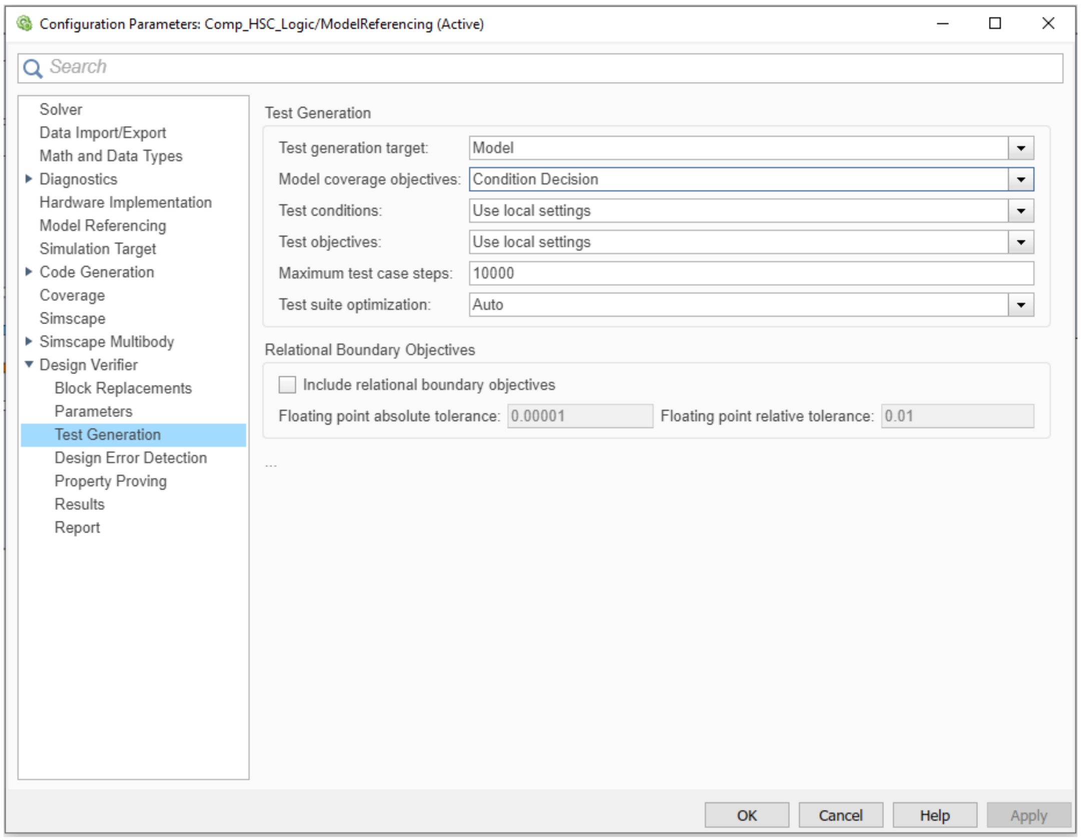
Section 6.1 to ensure that the HSC Model before and after modularization remained functionally the same.

2.6.2 Model checks using Simulink Model Advisor

The Simulink Model Advisor tool is used to check compliance to established modeling guidelines and industry standards, as well as verify model correctness [23]. The Model Advisor can be accessed through the **Modeling** tab of the Simulink editor, which launches the **System Selector - Model Advisor** dialog box [31, page 312]. Within the context of this thesis, the Simulink Model Advisor was primarily used to evaluate modeling metrics needed for modularity analysis, specifically block counts and Cyclomatic Complexity [31, page 310]. The Model Advisor check selection screen can be seen in Figure 2.19, with both the model count and cyclomatic complexity metrics enabled.

2.6.3 SDV Test Generation and Coverage Analyzer

The SDV can also automatically generate test cases to satisfy model coverage objectives when ran in **Test Generation Mode** [29, page 258]. Several different types of coverage objectives are available including decision, condition, and modified condition decision coverage (MCDC). Decision coverage objectives examine decision points within a model. Common sources of decision objectives includes switch and saturation blocks, where the input can either be greater or less than a given threshold value [29, page 281]. Condition coverage objectives analyze logical outputs coming from Simulink logic blocks or Stateflow transitions [29, page 281]. The test generation capabilities provided by the SDV are used in Section 6.6 when analyzing the changes in


Configuration Parameters: Comp_HSC_Logic/ModelReferencing (Active)

Test Generation

Test generation target: Model
Model coverage objectives: Condition Decision
Test conditions: Use local settings
Test objectives: Use local settings
Maximum test case steps: 10000
Test suite optimization: Auto

Relational Boundary Objectives

☐ Include relational boundary objectives
Floating point absolute tolerance: 0.00001 Floating point relative tolerance: 0.01

FIGURE 2.20: Test Coverage Model Configuration Parameters

model testability. An example of the test generation model configuration parameters that were set during analysis can be seen in Figure 2.20.

2.7 Summary

The preceding sections have provided the background information necessary to provide context for the remainder of this thesis. This included a discussion on the concept of MBD and various design processes mentioned in the literature. An overview of the MATLAB Simulink environment was also included, giving background information into the various common constructs and concepts used during model development in Chapters 4 and 5. The fundamental concepts of modularity were also discussed, reviewing the importance of decomposing large systems with information hiding in mind. Finally, the work done by Jaskolka et. al to apply modularity principles to the Simulink environment was summarized, with the resulting modular guidelines and Simulink Module Tool providing the basis for the EcoCAR Model changes outlined in Chapter 5, and the corresponding analysis in Chapter 6.

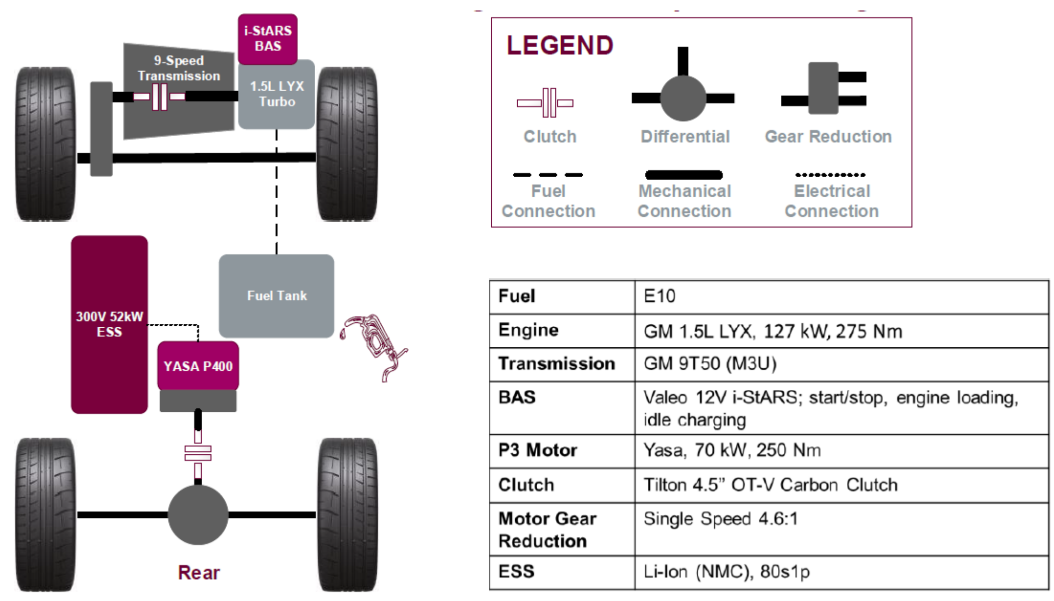
Chapter 3

Overview of EcoCAR Competition

The research presented in this thesis is primarily based on Simulink Models developed as contributions to the McMaster Engineering EcoCAR Team. McMaster is one of twelve North American universities participating in the EcoCAR Mobility Challenge, an Advanced Vehicle Competition sponsored by the U.S. Department of Energy, General Motors, and The Mathworks [8]. Prior to describing the Simulink Models and software development process, it is first necessary to introduce the EcoCAR Mobility Challenge and highlight important vehicle background information. Section 3.1 introduces the vehicle architecture used by the McMaster Team during the EcoCAR competition, as well its comprising components and hardware. Section 3.2 outlines the team VTS goals, defining the performance targets that drove both hardware and software design decisions. Finally Section 3.3 introduces the competition hybrid vehicle model developed in MATLAB Simulink, providing context for the existence and functionality of the HSC Controller model described in detail in Chapters 4 - 6.

3.1 Vehicle Architecture

During Year 1 of the EcoCAR Mobility Challenge (2018), the McMaster Team went through the architecture selection process, where different combinations of components were designed and analyzed together to determine the degree to which they met the team's end competition vehicle goals (Section 3.2). Figure 3.1 shows the vehicle architecture which was selected by the McMaster Engineering EcoCAR team, and is classified as a P4 Parallel-Through-The-Road Hybrid Electric Vehicle [16]. The vehicle powertrain has front, rear, and all wheel drive capability, depending on the engagement status of the clutch. The Front powertrain is composed of the GM 1.5L LYX Engine [2], the GM 9-speed M3U transmission [3], and the Valeo i-STARS BAS [46]. The electrical powertrain components are located on the rear drivetrain, providing torque by feeding power from the Malibu 300V Battery Pack [11] to the YASA P400 Electric Motor [49]. The final component chosen within the vehicle architecture is the Tilton Racing Clutch [45], which connects the combustion and electrical powertrains depending on what vehicle drive mode has been requested. A Table summarizing the final components chosen within the Team's vehicle architecture can also be seen in Figure 3.1. This data was collected by the McMaster EcoCAR team in 2018 during the Year 1 architecture selection process, where different components were analyzed both in terms of cost and ability to meet VTS targets.



Fuel	E10
Engine	GM 1.5L LYX, 127 kW, 275 Nm
Transmission	GM 9T50 (M3U)
BAS	Valeo 12V i-StARS; start/stop, engine loading, idle charging
P3 Motor	Yasa, 70 kW, 250 Nm
Clutch	Tilton 4.5" OT-V Carbon Clutch
Motor Gear Reduction	Single Speed 4.6:1
ESS	Li-Ion (NMC), 80s1p

FIGURE 3.1: Vehicle Architecture developed by the McMaster EcoCAR Team in 2018

3.2 Vehicle Technical Specifications

Another process completed during Year 1 of the EcoCAR competition was the definition of the McMaster Team's Vehicle Technical Specification (VTS) Goals. These goals are outlined in Table 3.1 and represent the team built vehicle performance targets that should be reached by the competition end date. The VTS goals along with the competition rule set formed the basis of the High level system requirements and resulting vehicle design process (discussed in Chapter 4). In particular, the **Fuel Economy** target played a dominant role in driving the Vehicle Architecture selection outlined in Section 3.1, as well the resulting Energy Management and Propulsion control strategies. Reaching the desired fuel economy during simulation and in vehicle led to decisions related to component selection such as the addition of the Valeo BAS, as well as changes to torque splitting strategies within the HSC [16, page 90]. The impacts that the VTS goals had on the team designed energy management strategy are discussed at length in [16], as well as the methods used for vehicle architecture selection.

3.3 Competition Sponsored Hybrid Vehicle Model

To aid in the development of vehicle controls and vehicle architecture selection, a MATLAB Simulink Vehicle Model was provided to each University participating in the EcoCAR competition. Simulink was selected as the primary modeling environment used by the McMaster PCM Team, with the abundance of software tools and libraries offered resulting in a unified development environment with simplified integration between components. Another benefit provided by MATLAB Simulink was

Table 3.1: Vehicle Technical Specifications developed by the McMaster EcoCAR Team in 2018

Specification	Competition Target	Stock AWD Blazer RS	Team Target
Acceleration, IVM-60 mph (s)	7	7.45	7
Acceleration, 50-60 mph (Passing) (s)	6.5	3.4	6.5
Braking, 60-0 mph (ft)	Stock	120	120
Cargo Capacity	Stock	Stock	Stock
Passenger Capacity (persons)	Stock	Stock	Stock
Curb Mass (kg)	N/A	2004.4	<2542
Starting Time (s)	<=2	<=2	<=2
Ground Clearance (in,)	N/A	7.87	7
Total Vehicle Range (mi.)	250	330	250
Fuel Economy (MPG)	Stock + 15%	24.9	34
Emissions	Stock	120	120
Braking, 60-0 mph (ft)	Stock	Stock	Stock

its compatibility with the dSPACE MicroAutobox II [10], the hardware that the team needs to deploy on the vehicle. This allows for the Simulink models to be code generated and deployed on embedded hardware with reduced issues, eliminating any bugs related to development framework conversion. Library add-ons such as the Powertrain Blockset [28] can be leveraged to develop model representations of physical propulsion and drivetrain hardware, providing the tools needed to develop a robust Vehicle Plant Model, improving the MIL testing procedure.

3.3.1 Driver Plant Model

The driver plant model is shown in Figure 3.3, and consists of a Drive Cycle Source block (highlighted in blue), and a longitudinal driver model variant subsystem (highlighted in green). The Drive Cycle Source block is part of the Powertrain Blockset, and generates a velocity output signal based on a specified set of input points [28, p. 133]. The drive cycle can be implemented in several file formats, including *.mat*, *.txt*, and *.csv*, and is uploaded via the drive cycle source block parameter window (shown in Figure 3.4). Figure 3.5 shows the **HWFET** drive cycle, which was used extensively by the team throughout MIL testing in Year 1. It is meant to represent typical highway driving conditions under 60 mph and was used along with other drive cycles to evaluate the vehicle control system performance [12]. The resulting speed output is received by the Longitudinal Driver Model Variant Subsystem, which is also a block within the Powertrain Blockset. It is designed to implement a speed-tracking controller which adjusts the output speed command based on both the vehicle request

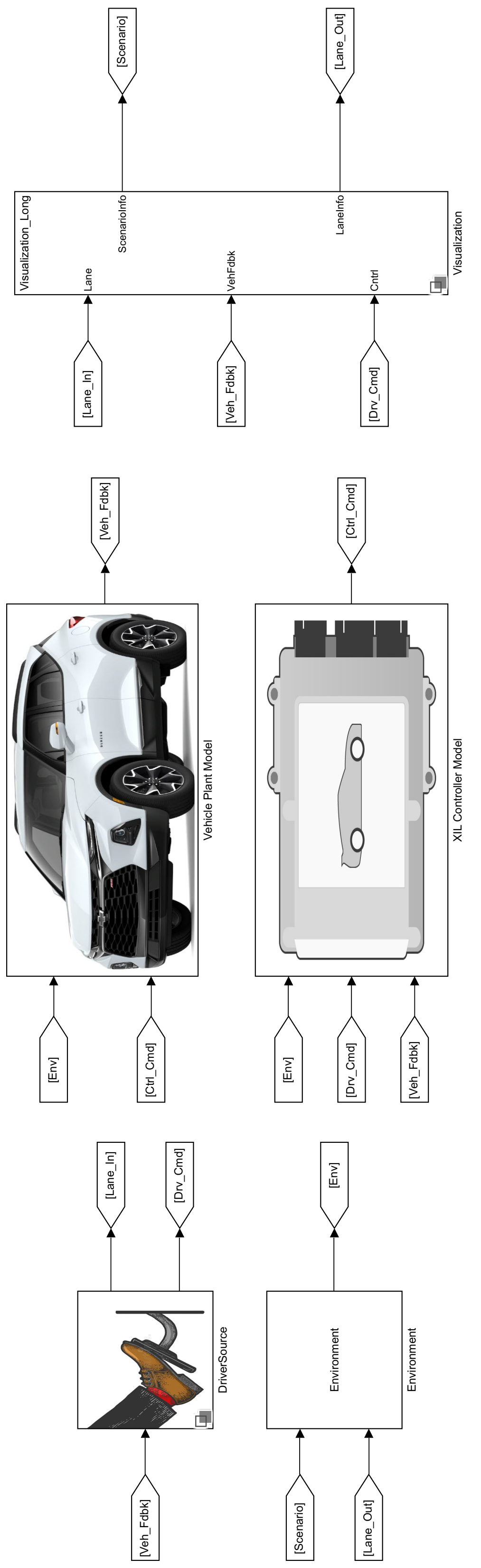


Figure 3.2: Hybrid Vehicle Model used by McMaster EcoCAR Team

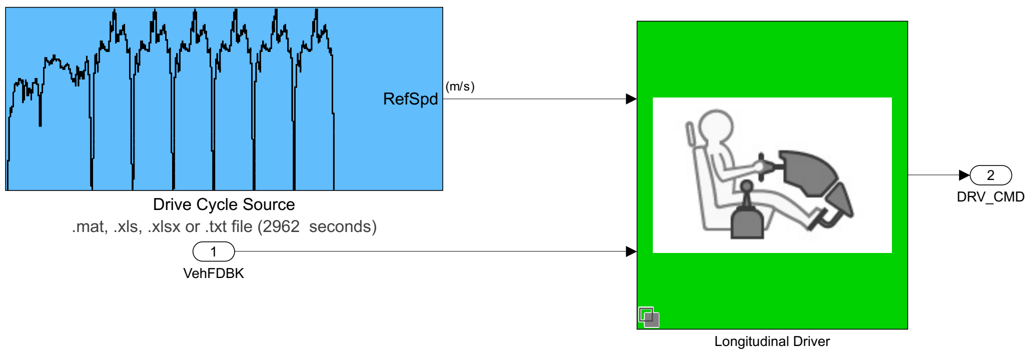


FIGURE 3.3: Driver Plant within EcoCAR Model, with Drive cycle input

generated by the Drive Cycle source block, as well as the Vehicle Speed feedback signal sent by the Vehicle Plant Model (Section 3.3.4) [26].

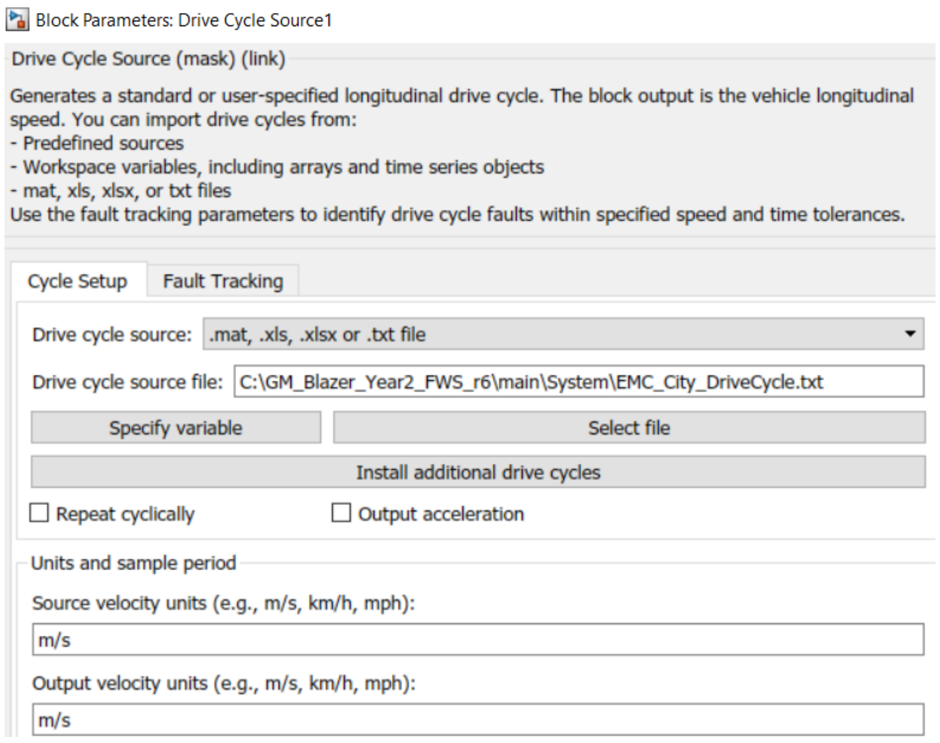
3.3.2 Vehicle Controller Model

The speed command generated by the Driver Plant model is passed to the Hybrid Supervisory Controller (HSC) Model shown in Figure 3.2. The controller model processes the speed command as well as plant model feedback, and generates command control signals for each of its comprising components. The internals and functionality of the HSC is the primary subject of Chapters 4 and 5, and therefore will not be elaborated on here further. The controller model is named **XIL Controller Model** as it contains the various Simulink models necessary to simulate and test in the MIL, HIL, and VIL environments. The development and use cases of this model variant system are described in detail in Section 4.3.2.

3.3.3 Vehicle Plant Model

The Vehicle Plant Model receives an input representing environmental conditions during simulation, *Env*, as well as the component control signals, *Ctrl_Cmd*, sent from the HSC. The component control request signals generated within the HSC Model include torque requests to the various powertrain components, as well as power management and drivetrain specific parameters necessary for the Vehicle Plant Model to function correctly during simulation. The vehicle architecture described in Section 3.1 was modeled in the Simulink Environment primarily using the Powertrain Blockset. The majority of major propulsion and drivetrain components such as the engine, battery, motor, wheels, and driveshaft all require modeling of their physical parameters within the Simulink environment to produce accurate MIL simulation results.

The YASA P400 Electric Motor was modeled within the Vehicle Plant using the Mapped Motor Block, shown in Figure 3.6. The Mapped motor block receives Battery Voltage, Motor Speed Feedback, and a Torque Command, and produces a corresponding Motor torque output, motor current, and power loss data [27]. The generic structure of the mapped motor block can be further parameterized to more accurately



Block Parameters: Drive Cycle Source1

Drive Cycle Source (mask) (link)

Generates a standard or user-specified longitudinal drive cycle. The block output is the vehicle longitudinal speed. You can import drive cycles from:
- Predefined sources
- Workspace variables, including arrays and time series objects
- mat, xls, xlsx, or txt files

Use the fault tracking parameters to identify drive cycle faults within specified speed and time tolerances.

Cycle Setup | Fault Tracking

Drive cycle source: .mat, .xls, .xlsx or .txt file

Drive cycle source file: C:\GM_Blazer_Year2_FWS_r6\main\System\EMC_City_DriveCycle.txt

Specify variable | Select file

Install additional drive cycles

☐ Repeat cyclically ☐ Output acceleration

Units and sample period

Source velocity units (e.g., m/s, km/h, mph): m/s

Output velocity units (e.g., m/s, km/h, mph): m/s

FIGURE 3.4: Drive Cycle Source Block used to import drive cycle files passed to Driver Plant Model

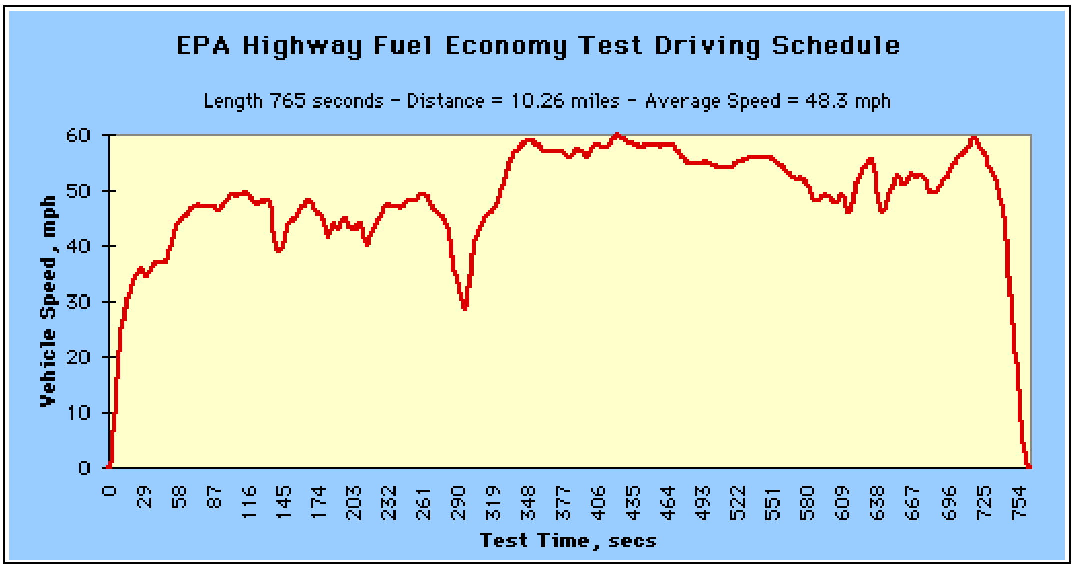


FIGURE 3.5: HWFET Drive Cycle taken from EPA website [12]

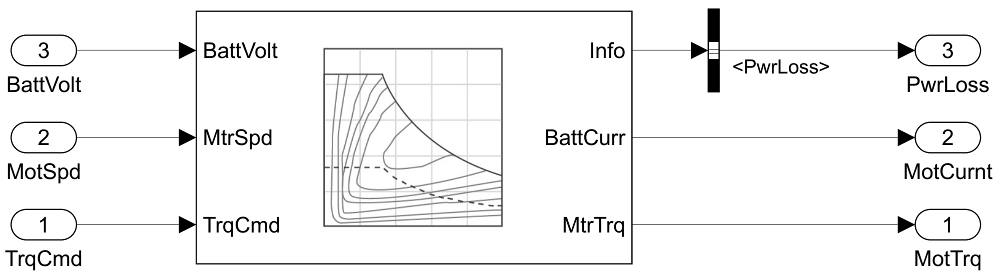


Figure 3.6: Example of Mapped Motor Block used from Powertrain Blockset to simulate motor functionality.

model a specific motor hardware component by providing a unique torque-speed envelope. This provided a significant benefit during the architecture selection process, where several motors which were being considered could be modeled using the same block, simply by changing the torque-speed envelope parameter, and retrying the simulation.

3.3.4 Visualization and Data Logging

The final major component within the competition vehicle model is the visualization and data logging block, seen in Figure 3.2. The inputs to the system include the Vehicle Plant Feedback signals, Diagnostic signals generated by the HSC, as well as the initial Driver speed command. These signal bundles are decomposed into various scopes and measurement blocks in order to provide readable and meaningful simulation output results. An example of simulation output results can be seen in Figure 3.7, using a competition provided highway drive cycle as input to the Driver Plant Model. The simulation of the competition vehicle model involved the analysis of how closely the vehicle plant speed feedback (highlighted in blue) matched the generated trace velocity target (highlighted yellow).

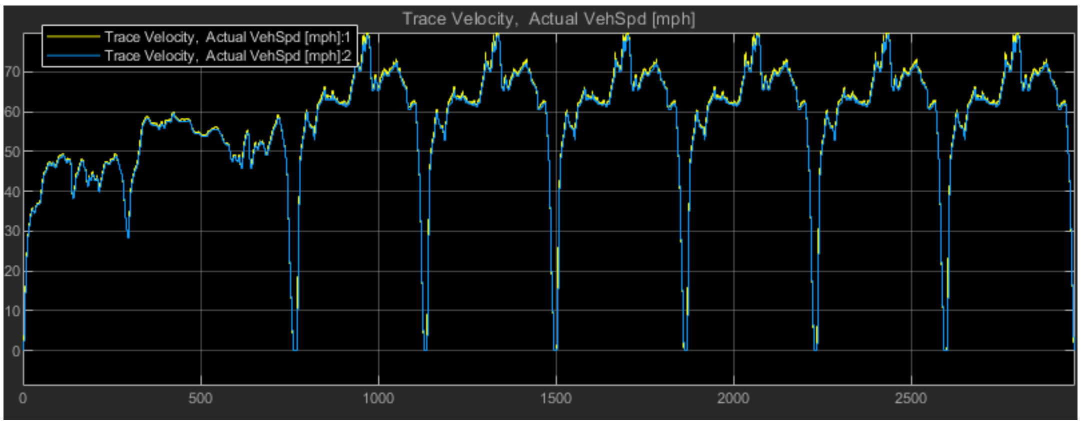


FIGURE 3.7: Simulation output of vehicle model using competition provided drive cycle.

Chapter 4

Model Development Process

This chapter is meant to provide the reader with an understanding of the software development process followed by the author during Years 2 and 3 of the EcoCAR Mobility Challenge. Additionally, an overview of the various model components is provided, as well as their modular structure within the system decomposition. The HSC Simulink Model introduced within this chapter is the primary focus of the remainder of this thesis, including the modular changes applied in Chapter 5, and the resulting modularity analysis in Chapter 6. The rest of this chapter is as follows: Section 4.1 describes the process of requirements definition and documentation. Section 4.2 describes the HSC architecture derived from the vehicle requirements set, and demonstrates examples of software interface definition as well as implementation of control algorithms. Section 4.3 introduces the method of modularization that was initially used based on conventional Simulink methods (discussed in Chapter 2), as well as methods suggested by the EcoCAR competition (discussed in Chapter 3). Finally, Section 4.4 discusses the process of requirements verification through XIL environment testing and presents examples of test results.

There are two primary systems provided as examples throughout Chapter 4, namely the Accelerator Pedal Processing Subsystem, and the Power Moding Module. The pedal processing subsystem is a relatively small component within the HSC model, and is used to show requirement specification (Section 4.1.4), as well as validation through XIL testing (Section 4.4.3). The Power Moding Module is a relatively large and complex system, controlling the wake-up and shutdown of all external propulsion hardware. This increased complexity is used to shown an example of algorithm development, and Stateflow implementation (Section 4.2.4.1). The software design process outlined below was designed by the author during the EcoCAR Competition. The implementation of this process (resulting requirements, architecture diagrams, simulink models) were completed by the author in a joint effort with the McMaster EcoCAR Team. In cases where a figure or table was not directly developed by the author, an indication of this is present within the headings, indicating the main persons responsible.

4.1 Requirements Development Process

Similar to the V-cycle outlined in Section 2.1, the first step of the hybrid vehicle development process is to define the requirements for the control system. To organize the requirements for future changes and documentation, a hierarchical categorization was implemented. Four major abstraction levels have been defined, namely high-level

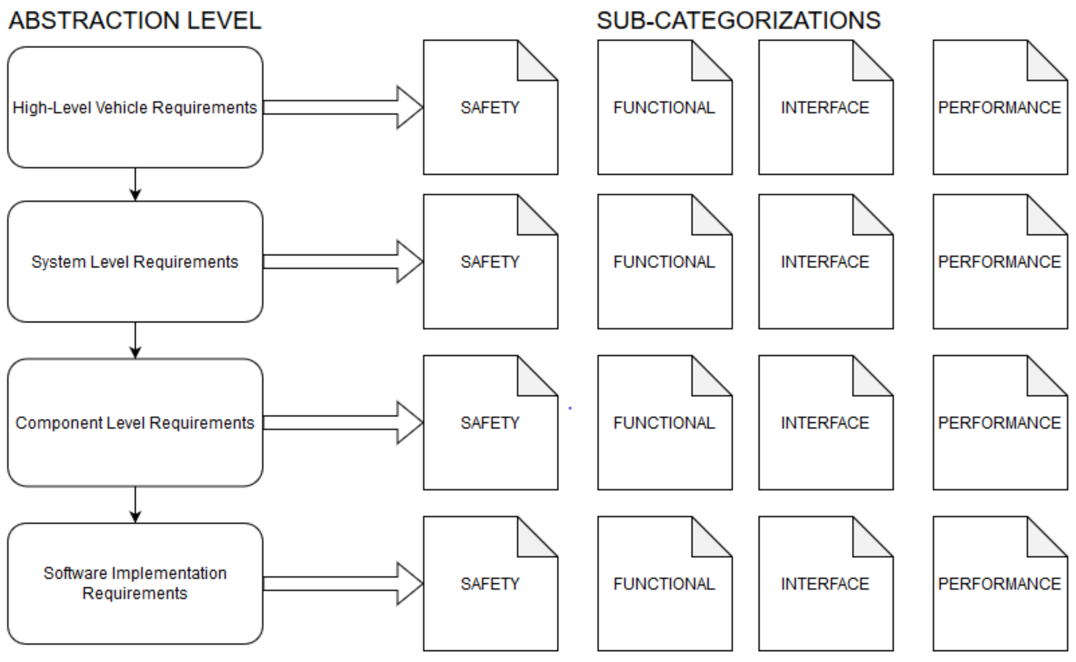

ABSTRACTION LEVEL
SUB-CATEGORIZATIONS
High-Level Vehicle Requirements
System Level Requirements
Component Level Requirements
Software Implementation Requirements
SAFETY
FUNCTIONAL
INTERFACE
PERFORMANCE

FIGURE 4.1: Requirements Categorization Diagram

vehicle, system, component, and software implementation requirements. In addition, at each level of abstraction the set of all requirements are further categorized by the specific purpose they hold within the overall vehicle design. These subcategories are safety, functional, interface, and performance requirements. Please see Figure 4.1 for a graphical representation of this categorization. The requirement subcategories are briefly described in section 4.1.1, as well as a deeper analysis of the four levels of abstraction within the functional requirements set in section 4.1.2.

4.1.1 Requirement Categorization

4.1.1.1 Safety Requirements

While safety requirements are arguably the most critical step of the vehicle software development process, they were not a focus of the EcoCAR competition during the development of the HSC model discussed in this thesis. All hazard analysis and the corresponding safety requirements were developed in later years of the competition by the McMaster EcoCAR System Safety Team, coordinating with the PCM Team when internal knowledge of the system and component functionality was required.

4.1.1.2 Functional Requirements

Functional requirements describe the necessary operations of the systems, components, and their associated software implementation within the HSC model. This requirement set is particularly relevant to the primary discussion of this paper, which is the modularization of the model, and as such they will be the primary requirement set referenced throughout the remaining sections.

4.1.1.3 Performance Requirements

The set of performance requirements form the guidelines for design such that each system, component, and component function within the HSC meets the VTS goals defined in Section 3.2. Additional performance limits on the system such as data transmission rate and function caller frequency are also defined within this requirement set.

4.1.1.4 Interface Requirements

While each system and its corresponding components can be individually analyzed in regard to functional, performance, and safety requirements, it is critical to also consider the interactions between the different subsystems and components. This interaction is captured within the interface requirements and provides the foundation for defining explicit software interfaces (see Section 4.1.3).

4.1.2 Functional Requirements Set

4.1.2.1 High-Level Vehicle Requirements

The high-level vehicle requirements are primarily derived from an analysis of the Vehicle Technical Specifications as well as the Customer or Stakeholder Requirements. In the case of the EcoCAR competition, the stakeholder requirements are defined according to the Competition Non-Year Specific Rules (omitted from this thesis due to NDA restrictions) for proper operation of team vehicles. Please see Chapter 3 for further information.

4.1.2.2 System and Component Requirements

Each high-level requirement is broken down and considered in terms of the major systems within the vehicle software. The system level requirements and their associated components were primarily defined according to the physical hardware and software components within the Hybrid vehicle powertrain. The breakdown of the HSC into systems can be seen in Figure 4.2, and the further breakdown of the Propulsion System into individual components is shown in Figure 4.3.

4.1.2.3 Software Implementation Requirements

While the component level requirements pertain to an individual feature or functionality of a given system, its software specific implementation must be further broken down and analyzed. All component level requirements are defined in terms of real software and hardware signals, as well as specific values or calibration values. The process of model design and testing begins with these software implementation requirements, and the metric for a complete implementation is their verification.

4.1.3 Requirements Specification Document

The requirement categorization described above has been defined and documented within a Requirements Specification Document. A documentation and maintenance

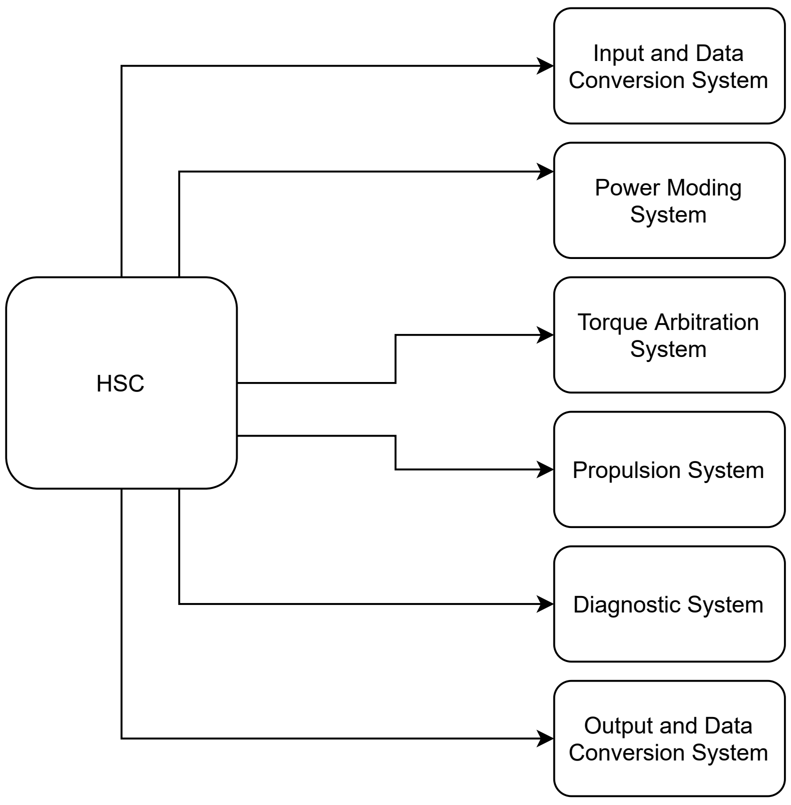

HSC
Input and Data Conversion System
Power Moding System
Torque Arbitration System
Propulsion System
Diagnostic System
Output and Data Conversion System

FIGURE 4.2: HSC System Categorization

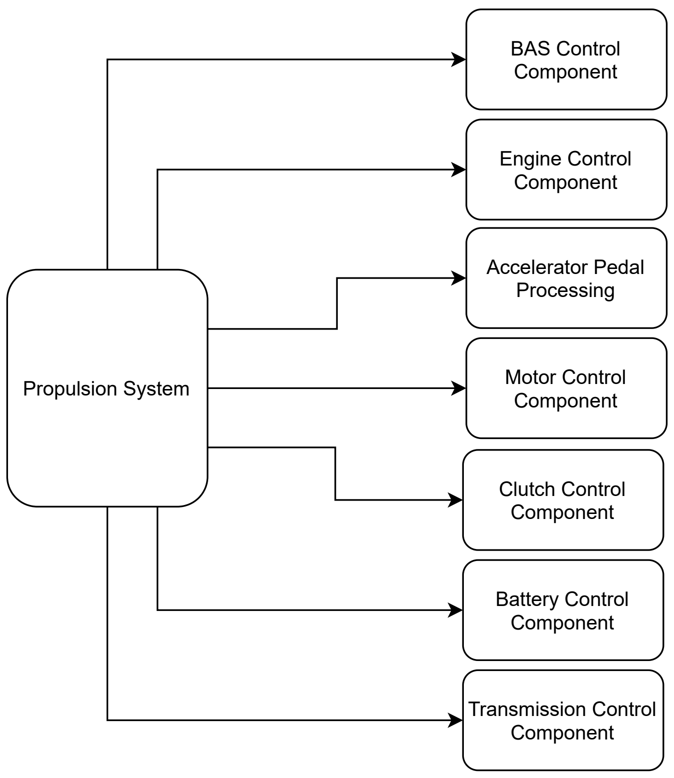


FIGURE 4.3: Propulsion System Component Categorization

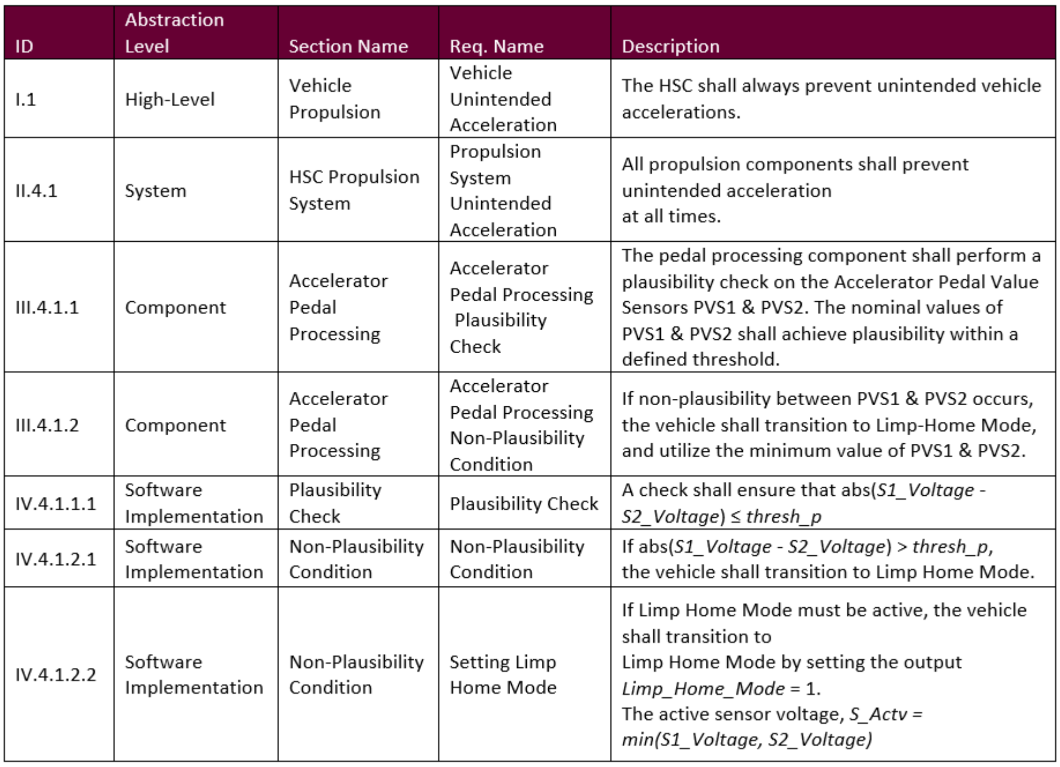
process has been developed to organize the defined requirements at the beginning of the software development cycle as well as ensuring that future changes or modifications to requirements were able to be facilitated in an efficient manner which could be tracked and maintained. The format *Abstraction.System.Component.Software* was used to assign each requirement a unique identifier, making it possible to track a low-level requirement all the way up to its high level VTS requirements. Changes to the RSD must be approved by the PCM Team Lead, and new documentation must be produced before any model or software changes are approved. Regarding document maintenance, bi-weekly meetings were established in which all members of the Controls Team who were currently making model and/or software changes would need to analyze and discuss what effects these had on the master requirements documentation. Furthermore, the requirements which require model validation have been implemented into the GM Blazer Model via the Simulink Requirements toolbox and verified via a set test case. The process of requirements verification via Simulink Requirements is discussed in Section 4.4.1.

4.1.4 Example: Accelerator Pedal Processing

An excerpt from the requirement specification document can be seen in Table 4.1, where the accelerator pedal processing system is considered. Component requirements were created based on the E-GAS standard [48], which defines the safe design and operation of an ECM (Engine control module). An overview of the system considered within the E-GAS standard can be seen in Figure 4.4. The HSC acts as the torque arbitrator within the hybrid vehicle system and must be developed to handle acceleration input and make the conversion to output torque, which is distributed to the relevant powertrain components.

Table 4.1 shows requirement *II.4.1*, which states that the HSC Propulsion System must prevent unintended acceleration at all times. In order for this higher level system requirement to be met, several component requirements must be fulfilled, including a plausibility check on the Accelerator pedal value sensors PVS1 and PVS2. This involves insuring that the difference between the two sensor voltage readings stays within a specified threshold. Another requirement of the component is that if a non-plausibility between PSV1 and PVS2 occurs, the vehicle shall transition to Limp-Home Mode, during which vehicle acceleration is restricted to a predefined threshold allowing the driver to get assistance [48, page 36]. The resulting accelerator voltage will be the minimum non-zero value of PVS1 and PVS2 [48, page 36 - 38]. These have been defined as requirements *III.4.1.1* and *III.4.1.2*, respectively. Voltage input signals *S1_Voltage* and *S2_Voltage*, threshold value *thresh_p*, as well as an output signal *Limp_Home_Mode* are explicitly defined within software implementation requirements *IV.4.1.2.1* and *IV.4.1.2.2*, respectively. Lastly, component requirements *III.4.1.3* and *III.4.1.4* describe the required system response in the event of a max signal fault on one or both sensors. A max signal fault occurs when a given sensor voltage output exceeds a predefined threshold [48, page 36 - 38]. The verification of these requirements are discussed in detail in Section 4.4.3.

Table 4.1: Accelerator Pedal Processing Requirements



ID	Abstraction Level	Section Name	Req. Name	Description
I.1	High-Level	Vehicle Propulsion	Vehicle Unintended Acceleration	The HSC shall always prevent unintended vehicle accelerations.
II.4.1	System	HSC Propulsion System	Propulsion System Unintended Acceleration	All propulsion components shall prevent unintended acceleration at all times.
III.4.1.1	Component	Accelerator Pedal Processing	Accelerator Pedal Processing Plausibility Check	The pedal processing component shall perform a plausibility check on the Accelerator Pedal Value Sensors PVS1 & PVS2. The nominal values of PVS1 & PVS2 shall achieve plausibility within a defined threshold.
III.4.1.2	Component	Accelerator Pedal Processing	Accelerator Pedal Processing Non-Plausibility Condition	If non-plausibility between PVS1 & PVS2 occurs, the vehicle shall transition to Limp-Home Mode, and utilize the minimum value of PVS1 & PVS2.
IV.4.1.1.1	Software Implementation	Plausibility Check	Plausibility Check	A check shall ensure that abs(*S1_Voltage* - *S2_Voltage*) ≤ *thresh_p*
IV.4.1.2.1	Software Implementation	Non-Plausibility Condition	Non-Plausibility Condition	If abs(*S1_Voltage* - *S2_Voltage*) > *thresh_p*, the vehicle shall transition to Limp Home Mode.
IV.4.1.2.2	Software Implementation	Non-Plausibility Condition	Setting Limp Home Mode	If Limp Home Mode must be active, the vehicle shall transition to Limp Home Mode by setting the output *Limp_Home_Mode* = 1. The active sensor voltage, *S_Actv* = *min(S1_Voltage, S2_Voltage)*

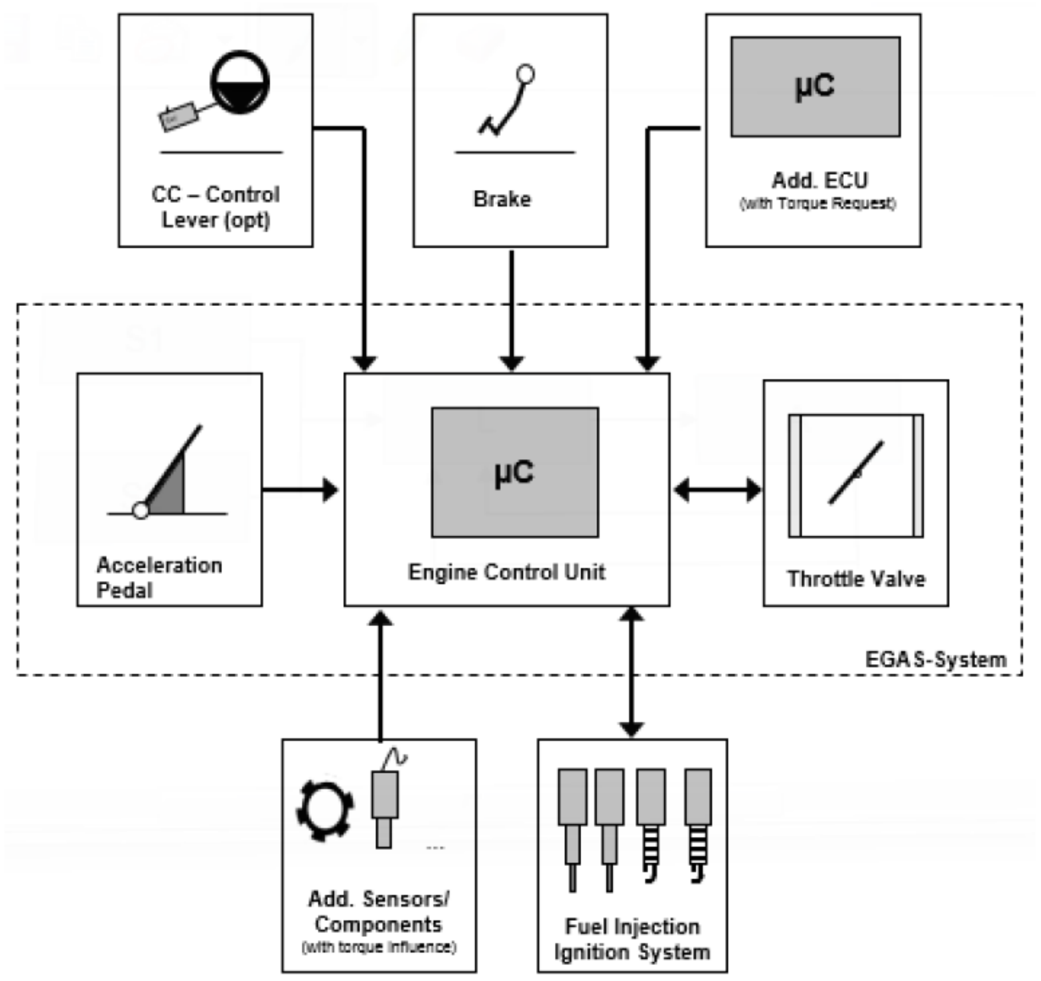


FIGURE 4.4: E-GAS Standard Component Diagram, taken from [48]

Table 4.2: HSC Serial Data Interface

CAN Network	CAN ID
HS_GM_LAN	CAN_1
HS_GMChassis_LAN	CAN_2
HS_ECO_LAN	CAN_3
HS_BATT_LAN	CAN_4
LS_GM_LAN	CAN_5
HS_CAVS_HSC_LAN	CAN_6

4.2 Software Design and Implementation

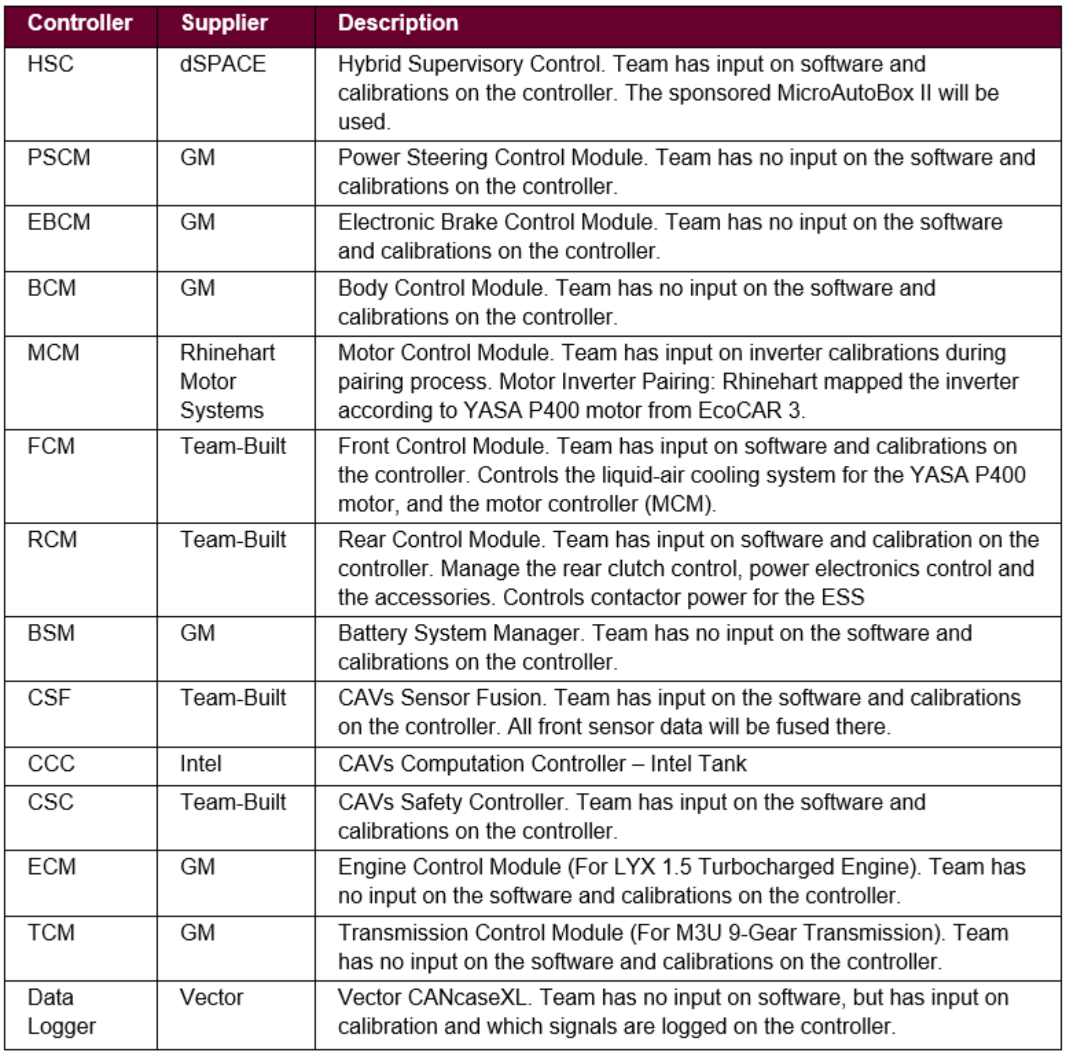
4.2.1 Controller and Serial Data Architecture

The embedded hardware used for code deployment and vehicle testing of the HSC is the dSPACE MicroAutobox II [10]. There are six available CAN channels within the hardware interface, which have been separated into the *HS GM_LAN*, *HS GMChassisLAN*, *HS ECO_LAN*, *HS BATT_LAN*, *LS GM_LAN*, and *HS CAVS_HSC_LAN*. Each network has been given a unique identifier which aids in the organization of data flow and signal routing during the software development process (Table 4.2). The controller and serial data architecture diagram can be seen in Figure 4.5. In addition, all the hardware controllers used in the vehicle architecture are summarized in Table 4.3. The stock GM network is connected to the HSC via *CAN_1*, which includes messages from the TCM, BCM, PSCM, and EBCM. Third-party and team-programmed controllers are connected on a separate high-speed CAN bus (*CAN_3*) to avoid bus overload on the main GM network, as well as avoid potential message conflicts. These controllers, namely the Rinehart Motor Controller, Front Control Module (FCM), Rear Control Module (RCM), and BAS Controller (LV BAS) will primarily communicate with each other and with the HSC. The RCM and FCM are team-built boards responsible for minor control action, including rear clutch and cooling fan actuator control. Additionally, the GM Battery System Manager (BSM) is placed on a separate LAN (*CAN_4*). The high volume of assumed CAN messages related to Battery control and diagnostics require a seperate network to avoid bus overload. The GM Chassis LAN was added into the HSC serial architecture as back-up high-speed LAN (CAN_2) in the event that the volume of CAN messages on the main GM network gets too large. The CAVs controllers use a dedicated high-speed LAN in order to separate lateral and longitudinal command signals from the rest of the propulsion system (*CAN_6*).

4.2.2 Hybrid Supervisory Controller Architecture

The Functional Supervisory Controller (FSC) or Hybrid Supervisory Controller (HSC) architecture is composed of five distinct layers, namely the input/conversion, application, and output/conversion layers. The functional supervisory controller architecture diagram is shown in Figure 4.6 and can be used as a reference for the following descriptions of module functionality and organization.

Table 4.3: EcoCAR Controller Specifications



Controller	Supplier	Description
HSC	dSPACE	Hybrid Supervisory Control. Team has input on software and calibrations on the controller. The sponsored MicroAutoBox II will be used.
PSCM	GM	Power Steering Control Module. Team has no input on the software and calibrations on the controller.
EBCM	GM	Electronic Brake Control Module. Team has no input on the software and calibrations on the controller.
BCM	GM	Body Control Module. Team has no input on the software and calibrations on the controller.
MCM	Rhinehart Motor Systems	Motor Control Module. Team has input on inverter calibrations during pairing process. Motor Inverter Pairing: Rhinehart mapped the inverter according to YASA P400 motor from EcoCAR 3.
FCM	Team-Built	Front Control Module. Team has input on software and calibrations on the controller. Controls the liquid-air cooling system for the YASA P400 motor, and the motor controller (MCM).
RCM	Team-Built	Rear Control Module. Team has input on software and calibration on the controller. Manage the rear clutch control, power electronics control and the accessories. Controls contactor power for the ESS
BSM	GM	Battery System Manager. Team has no input on the software and calibrations on the controller.
CSF	Team-Built	CAVs Sensor Fusion. Team has input on the software and calibrations on the controller. All front sensor data will be fused there.
CCC	Intel	CAVs Computation Controller – Intel Tank
CSC	Team-Built	CAVs Safety Controller. Team has input on the software and calibrations on the controller.
ECM	GM	Engine Control Module (For LYX 1.5 Turbocharged Engine). Team has no input on the software and calibrations on the controller.
TCM	GM	Transmission Control Module (For M3U 9-Gear Transmission). Team has no input on the software and calibrations on the controller.
Data Logger	Vector	Vector CANcaseXL. Team has no input on software, but has input on calibration and which signals are logged on the controller.

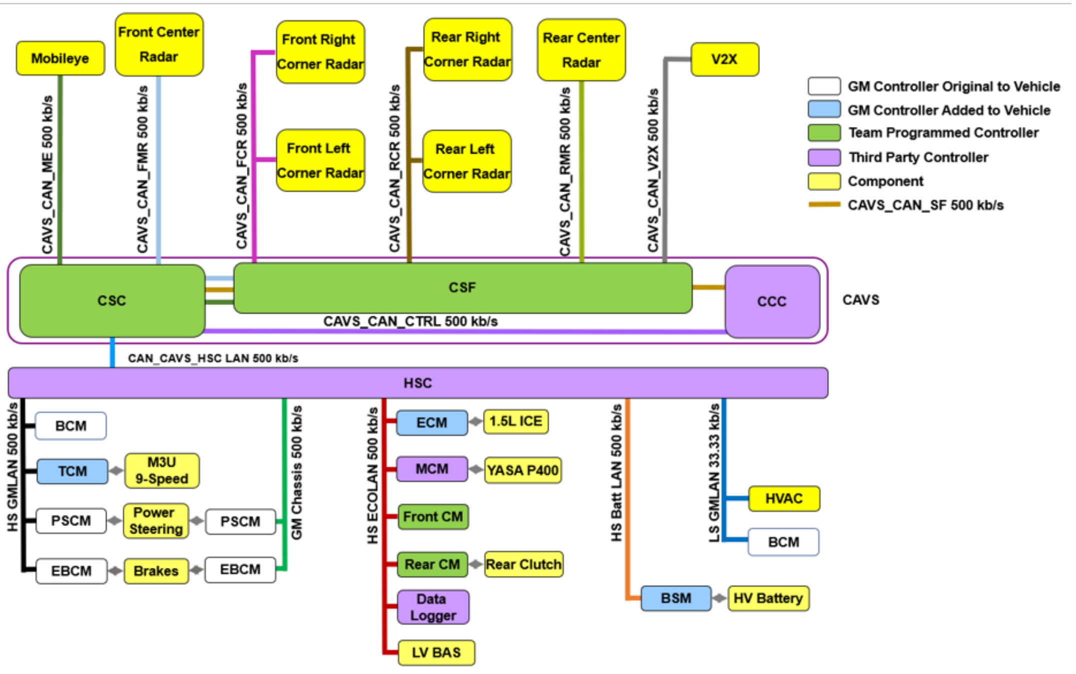


FIGURE 4.5: Serial Data Architecture Diagram created by McMaster EcoCAR Team (2019)

4.2.2.1 Input Conversion Layer

The input conversion layer receives incoming signals, decodes necessary CAN messages, and forwards them in the application layer. Contained in this layer are also any functions or modules that are needed to convert input signals from the outer HSC interface into input signals defined within the interface of the application layer modules. An input fault detection and Diagnostic Module is also contained within the input conversion layer and is designed to determine the validity of incoming data and detect corrupt or unusable signals before reaching the application layer modules.

4.2.2.2 Operation Mode Loop

The application layer contains the set of all modules within the HSC related to vehicle functionality. Systems within the application layer include Power Moding, Longitudinal and Lateral Determination, the Operation Mode Loop, and Component Control. The purpose of the operation mode loop is to receive incoming signals from the input conversion layer as well as power moding and CAVS information, and calculate the command signals necessary for each component within the propulsion system. Several modules are included in the operation mode loop, namely Drive Mode Selection, Energy Management, and Regenerative Braking Determination. Once component control signals have been generated, they are passed to various modules within the Component Control Ring, where each powertrain component is defined as a separate module with a distinct software interface.

Power Moding: This module will be discussed in more detail in Section 4.2.4.1. Its

primary purpose is to coordinate the start-up and shutdown procedures of all components within the vehicle based on external driver inputs.
Longitudinal and Lateral Determination: The design and implementation of this module is an ongoing process and not particularly relevant to this paper other than to demonstrate its location within the Functional Supervisory Controller. In future years of the competition, this module will be designed to receive inputs from the CAVS system related to vehicle propulsion, and act as a torque arbitrator in the event Longitudinal or Lateral Control is Enabled by the driver.
Energy Management Strategy Ring: The primary function of the drive mode selection module is to change active driving states based on input signals from the driver of the vehicle. These modes have been defined according to the method of propulsion that results, namely ICE (engine only), EV (electric motor only), or HEV (hybrid propulsion). As of the writing of this paper, HEV Mode is defined as the default driving state, and will likely remain the primary drive mode throughout the remainder of the vehicle development. Its purpose is to determine a power command for each powertrain component for a given driver acceleration pedal command.

4.2.2.3 Component Controls

The Components Control ring receives component control requests from the operation mode loop, and in turn calculates component control commands to send to the physical powertrain hardware. The Engine Control ring calculates the Engine Torque Command based on Engine Speed and Internal Combustion Engine (ICE) Power Command. The BAS Control ring will determine the BAS Operation mode and calculate the BAS Torque Command. The BAS Control Ring is used to either start the engine, assist the torque output, or provide a brake torque to the engine. The EM Control Ring will determine EM torque command based on EM Speed, Vehicle Speed and EM Power Command. The Transmission Control Ring will select the Transmission Mode based on PRNDL State, Operation Mode and Engine Mode. The Rear Clutch Control Ring determines rear clutch operation based on the Clutch Command signal. Lastly, the pedal position conversion is used to convert engine torque signals and brake signals into faked acceleration and brake pedal signals feeding back to the ECM.

4.2.2.4 Output Conversion Layer

The output conversion layer contains a Hardware Limit and Fault Detection Module which performs torque security checks and determines whether the torque commands generated by each component control module are safe and accurate in regard to real-time vehicle propulsion. The outgoing signals are then passed to any functions or modules that are needed to convert output signals from the application layer into output signals defined within the outer HSC interface. The final signals leaving the output layer are passed to the various vehicle controllers communicating with the HSC.

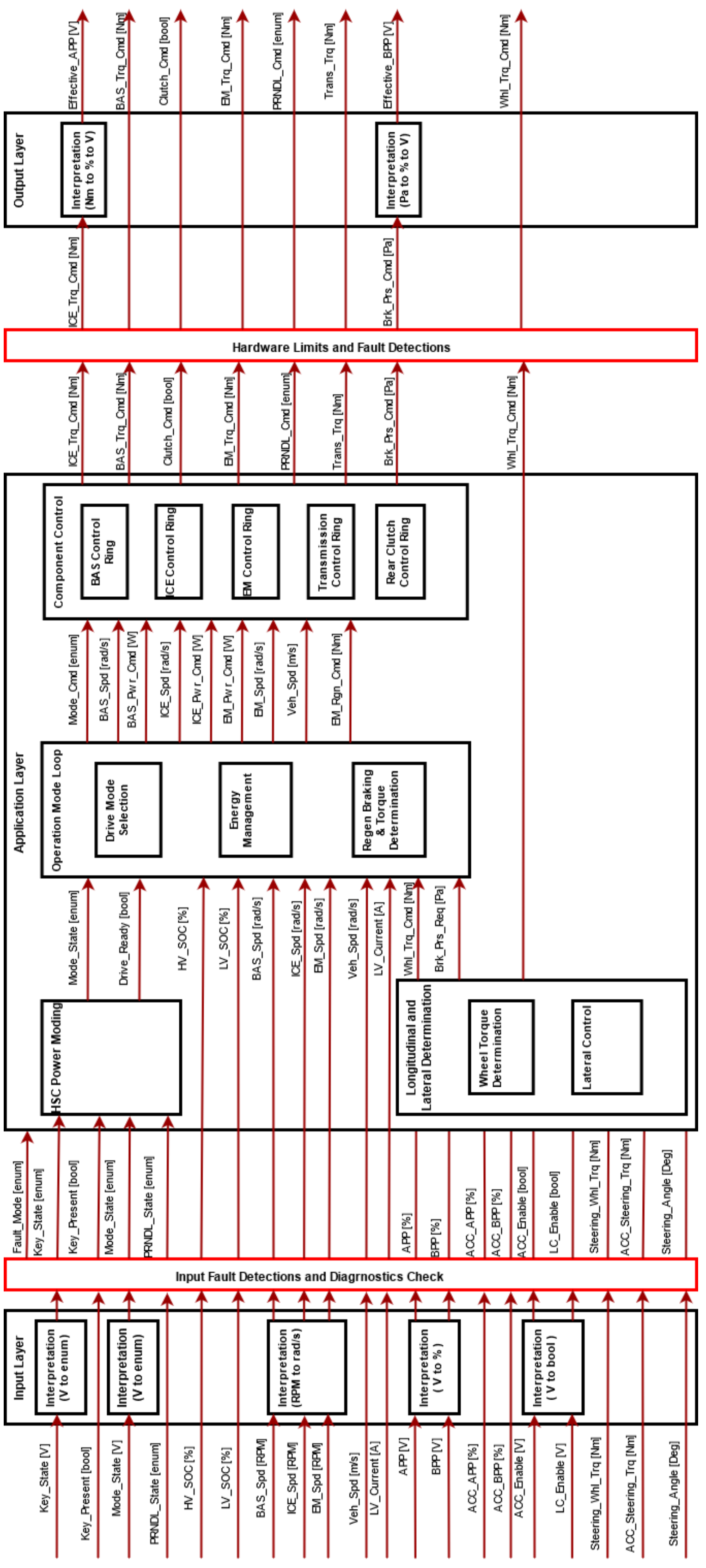


FIGURE 4.6: Functional Supervisory Controller Architecture Diagram

4.2.3 Software Interface Development

Once the software implementation requirements have been defined for a given system and its associated components, the next step in the model development process is to develop and define an explicit data interface. Performing this interface definition before any modeling or software work has begun ensures that the interaction between components and larger systems remains unchanged regardless of changes in internal functionality. To demonstrate this process of interface definition, an example is discussed below. The system under consideration is the software interface between the CAVS and HSC Hardware. This was chosen as it represents one of the most critical interface interactions and separates the two largest software systems within the vehicle model. The signal transmission diagram for the CAVS-PCM Interface is shown in Figure 4.7.

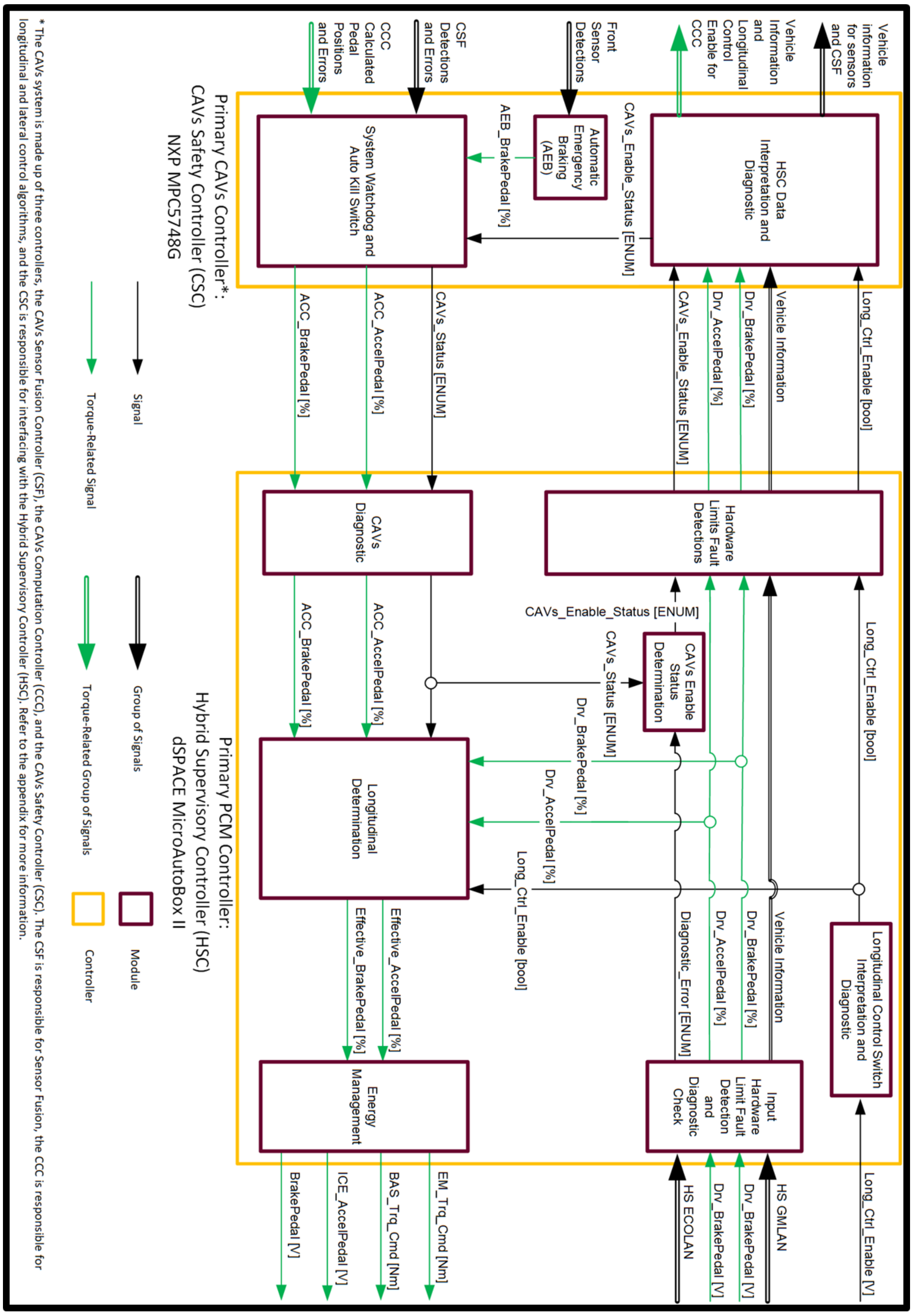
Signal identifications have been developed for all signals within a given software interface. Signals originating from a CAN Network are prefixed with the associated CAN_ID defined in Table 4.2. Signals which are commands or requests for certain functionality are suffixed with **Cmd** and **Req** respectively. Feedback signals or values measured from sensors and hardware are either suffixed with **Stat**, for status, or have a suffix omitted depending on the specific situation. The final software interface definition is explicitly defined by combining the developed high-level signal transmission diagram, as well as the signal identification system mentioned above. The software interfaces of the HSC have been defined in separate documents and contain critical information regarding the data flow of a given system or component. The final interface definition for the CAVS-PCM systems can be seen in Table 4.4, and contains the Signal ID, Signal Name, a description of the signal's purpose, the signal type (I/O), the units used inside the model, and the data type of the Simulink signal.

4.2.4 Example: Power Moding Algorithm and Implementation

For a given system component, the corresponding functional requirements set and software interface definition are used as a basis for the algorithm development and resulting software implementation. The implementation specifics of all systems and components are too extensive to include in this thesis and are not particularly relevant to the subject of model modularization. A single system is considered as an example of the vehicle control algorithm development and software implementation processes that were followed. The Power Moding System was chosen to demonstrate this process due to its relative complexity and involvement of several other software modules within the HSC.

4.2.4.1 Algorithm Development

Normal vehicle operation is only possible by reaching the Propulsion System Active state, which is dependent upon a successful wake-up and startup procedure. To achieve successful power moding functionality within the vehicle system, it was first necessary to develop the high level system power moding strategy, and identify the major functions needed. These were identified as the power moding functionalities for each powertrain component, namely the YASA Motor, GM LYX Engine, Valeo BAS, and GM Malibu Battery Pack (shown in Figure 4.8). Figure 4.9 shows the resulting



* The CAVs system is made up of three controllers, the CAVs Sensor Fusion Controller (CSF), the CAVs Computation Controller (CCC), and the CAVs Safety Controller (CSC). The CSF is responsible for Sensor Fusion, the CCC is responsible for longitudinal and lateral control algorithms, and the CSC is responsible for interfacing with the Hybrid Supervisory Controller (HSC). Refer to the appendix for more information.

FIGURE 4.7: Signal Transmission Diagram created by McMaster PCM Team, 2019

Table 4.4: CAVS-PCM Interface

Signal ID	Signal Name	Signal Description	Signal Type	Signal Units	Data Type
CAN6_CAVS_Stat	CAVS Status Signal	A status signal indicating proper functionality of CAVS hardware and software.	Input	enum	bool
CAN6_ACC_Accel_Pdl	ACC Acceleration Pedal Value	The calculated acceleration pedal command while Adaptive Cruise Control is active.	Input	%	double
CAN6_ACC_Brk_Pdl	ACC Brake Pedal Value	The calculated brake pedal command while Adaptive Cruise Control is active.	Input	%	double
Long_Ctrl_En_Switch	Longitudinal Control Enable Switch	Signal corresponding to the physical enable switch within the vehicle, determines whether longitudinal control functionality is enabled or disabled.	Input	V	double
CAN1	HS_GM_LAN Signal	Represents the set of all relevant CAN signals originating from the HS_GM_LAN Bus	Input	N/A	CAN
Drv_Accel_Pdl	Driver Accelerator Pedal Value	The accelerator pedal commands originating from the physical pedal position sensors.	Input	V	double
Drv_Brk_Pdl	Driver Brake Pedal Value	The brake pedal commands originating from the physical brake position sensors.	Input	V	double
CAN3	HS_ECO_LAN Signal	Represents the set of all relevant CAN signals originating from the ECO_LAN Bus	Input	N/A	CAN
CAN6_Long_Ctrl_En_Cmd	Longitudinal Control Enable Command	The control command sent from the HSC to CAVS, if it is determined that Longitudinal Control can be activated.	Output	Enum	bool
CAN6_Drv_Brk_Pdl	Driver Brake Pedal Value	The driver brake pedal command sent from HSC to CAVS system after Fault Detection	Output	%	double
CAN6_Drv_Accel_Pdl	Driver Acceleration Pedal Value	The accelerator pedal command sent from HSC to CAVS system after Fault Detection	Output	%	double
CAN6_CAVS_Sys_En_Cmd	CAVS System Enable Command	Command signal sent from HSC to CAVS system which enables or disables all CAVS functionality	Output	enum	bool

vehicle power moding module algorithm implemented within the HSC Model. The startup procedure for the ICE and Electric systems can be executed in parallel or individually, depending on the Drive Mode that has been selected by the driver when they initiate cranking. During the electrical powertrain startup, the motor inverter must first be initialized, and then commands are sent to the GM Battery System Manager to begin the contactor closing procedure. Once contactors have closed, the motor inverter can be fully enabled, making it ready to produce torque. The internal specifics of each component wake-up and shutdown procedure were developed within the individual component control modules (closing contactors inside the BSMM etc). The corresponding component power moding algorithms have been placed in Appendix A and show the algorithms developed for the BAS, Battery, Motor, and Engine (A1.1 - A1.4).

4.2.4.2 Stateflow Implementation

The vehicle and component power moding algorithms shown in Section 4.2.4.1 and Appendix A were used to develop the software needed to meet their associated functional requirements set. The software implementation for the Power Moding Control was performed within the Simulink environment using Stateflow, and can be seen in Figure 4.10. An introduction to Stateflow design as well as details on some commonly used blocks and concepts can be seen in section 2.2.2.

The power moding functionality was represented by a Stateflow Chart object, with states and transition objects defining the control path. The controller begins in the *VehicleOFF* state and transitions to *Vehicle Start* depending on the system power mode, determined by the driver of the vehicle. The parallel algorithm paths between ICE and Electrical startup is reflected in the parallel states *Conventional* and *Electrified*, which each handle the startup coordination of their respective propulsion components. A successful startup from the *Conventional* and *Electrified* control sequences will result in an enablement of the *FrontPwrtrnAvail* and *RearPwrtrnAvail* signals, respectively. The third control loop at the bottom of Figure 4.10, *VehiclePowerON* receives the final enablement from the Front and Rear Powertrains, and makes the determination on whether the propulsion system should be active or not.

Several component level Stateflow implementations are also provided in Appendix A, including the shutdown vehicle power moding procedure A1.5, and Motor component power moding A1.6. Certain component control software implementations were omitted from this thesis due to NDA restrictions, specifically the Battery (GM restricted), BAS (Valeo Restricted), and Engine (GM restricted). The power moding algorithm and software development was included to provide an example of the process that was followed by the author for all major vehicle functions throughout Year 2 of the EcoCAR Competition, and is not meant to be an extensive representation of the software work performed.

4.2.5 Model Version Control

GitLab was used as the primary version control tool throughout the development of the vehicle controller model [13], with the model version control process outlined in Table 4.5. Simulink has built in Git tools, which aid in the creation and merging of

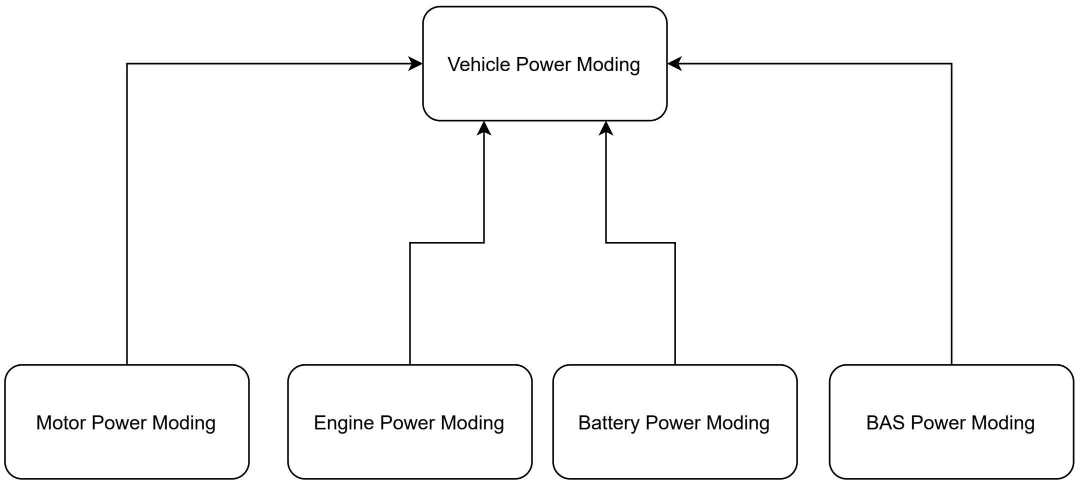


Figure 4.8: High Level Power Moding Algorithm

branches, as well as the ability to push to a remote repository. Each new feature is developed within a seperate branch created from the master model. Seperating each independent task into a branch allows multiple developers to implement changes to the vehicle plant or controller simulataneously. The changes made in the component or feature branch will be periodically committed and pushed to the Team's GitLab server repository. Each commit requires an accompanying message indicating the name of the developer, the major changes made to the branch, and the estimated time to complete. When model changes have been completed and unit tested, they are merged with the master branch and pushed to the master repository.

4.2.6 Software Version Control

The HSC model was developed by the author during Year 2 of the EcoCAR competition, which focused on the transition from developing controls software in a strictly MIL environment to developing code which was deployed on target embedded hardware. The transition to HIL and VIL environments increases risk of potential errors or incompatibilities between software releases. Mistakes made within the modeling environment pose minimal risk, typically increasing development time and complexity. In contrast, an error in software deployed on embedded hardware has potential to cause major setbacks, including damage to both equipment and persons interacting with the vehicle and/or test bench. The added software complexity as well as the increased risk provided the motivation to develop a more robust software version control and tracking system.

A software version control document was developed during Year 2 of the EcoCAR competition, shown in Table 4.6. This document contains a record of each new software release within the HSC system and tracks useful properties. The **Software Environment** column indicates the XIL environment the code was developed for, as the interfaces and configuration parameters are distinct between HIL test benches and in vehicle deployment (VIL). Each software release is assigned a version number

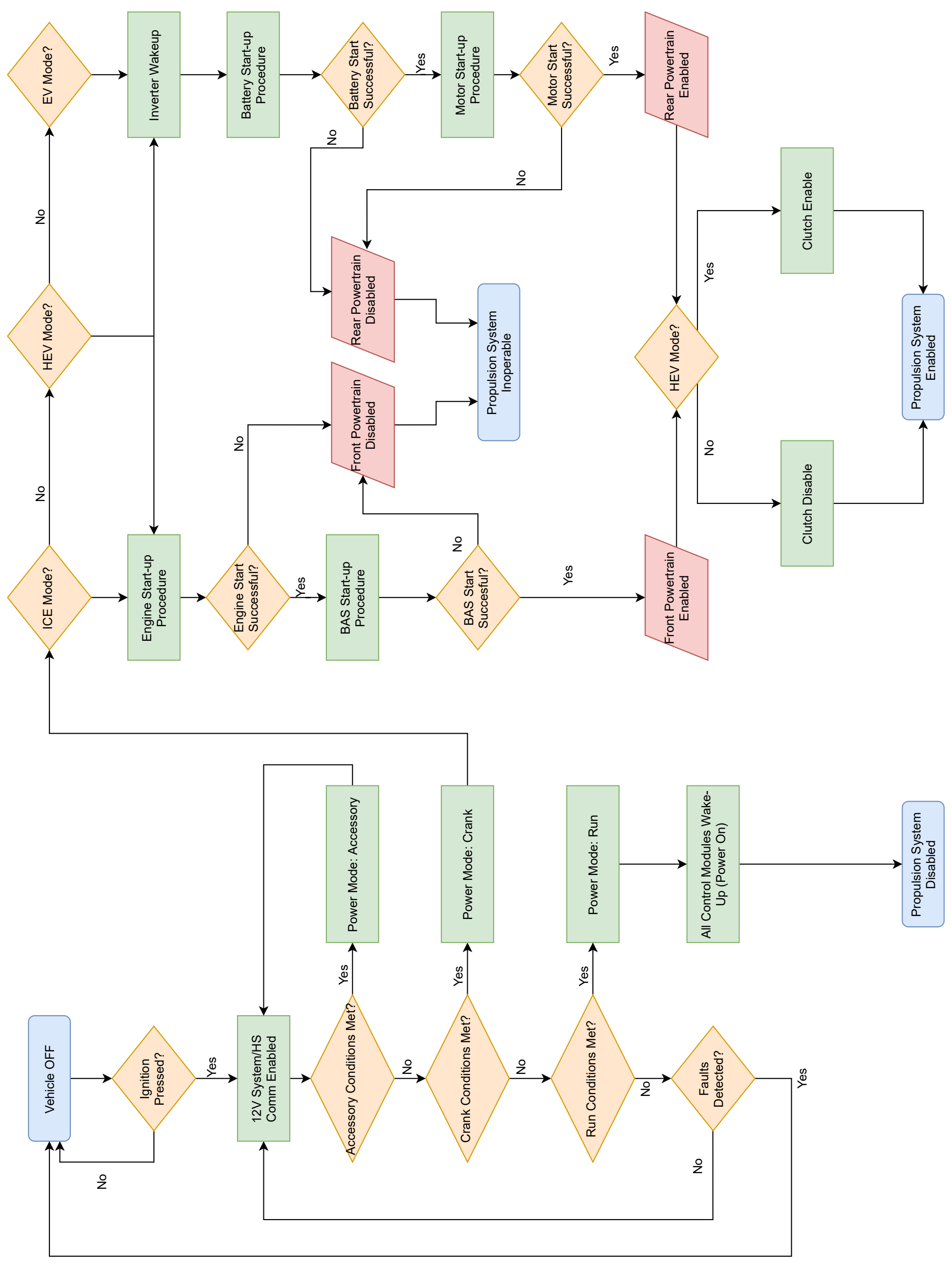


FIGURE 4.9: Vehicle Power Moding Algorithm

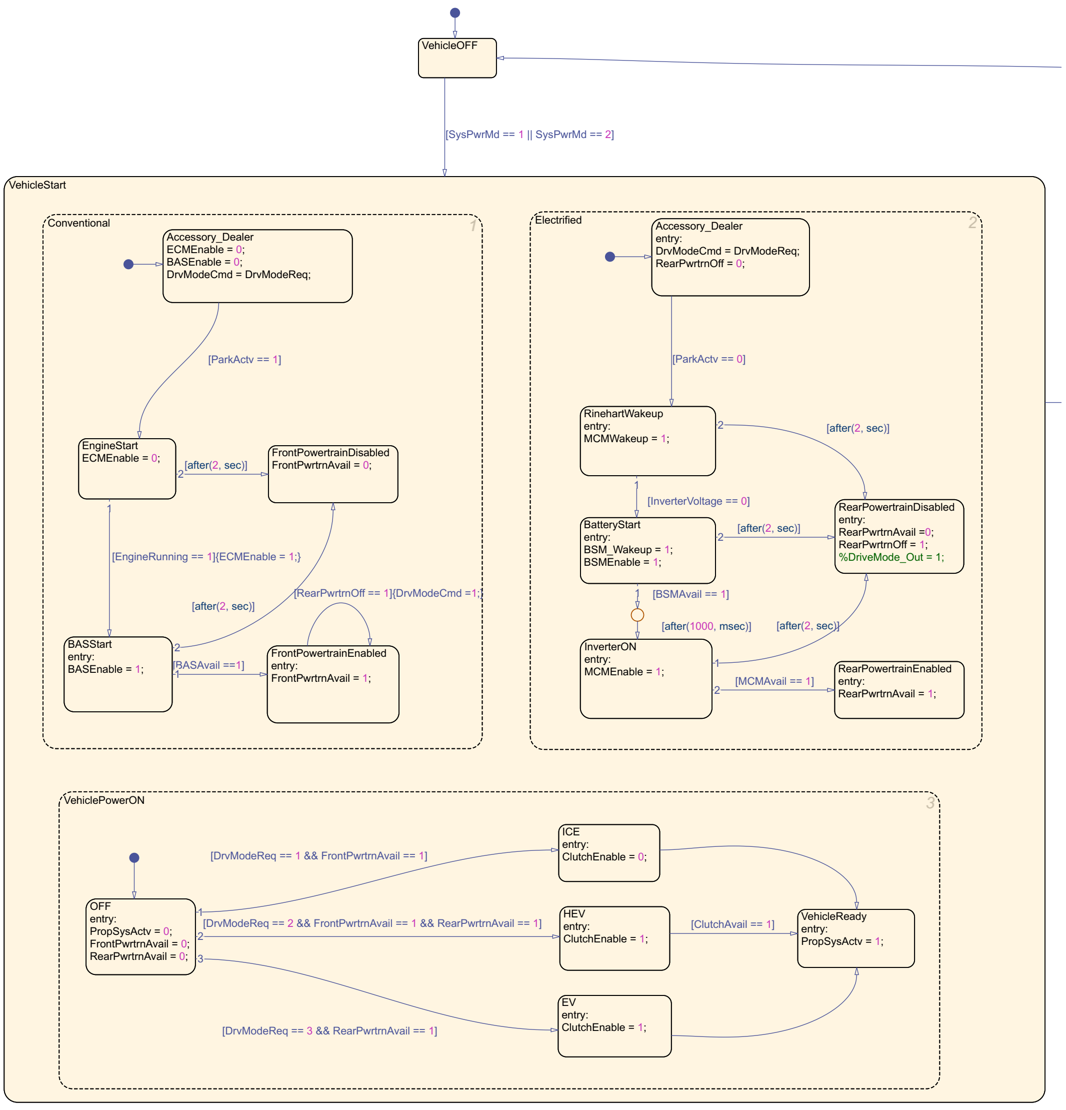


FIGURE 4.10: Vehicle Startup Stateflow Implementation

Table 4.5: Model Version Control Process

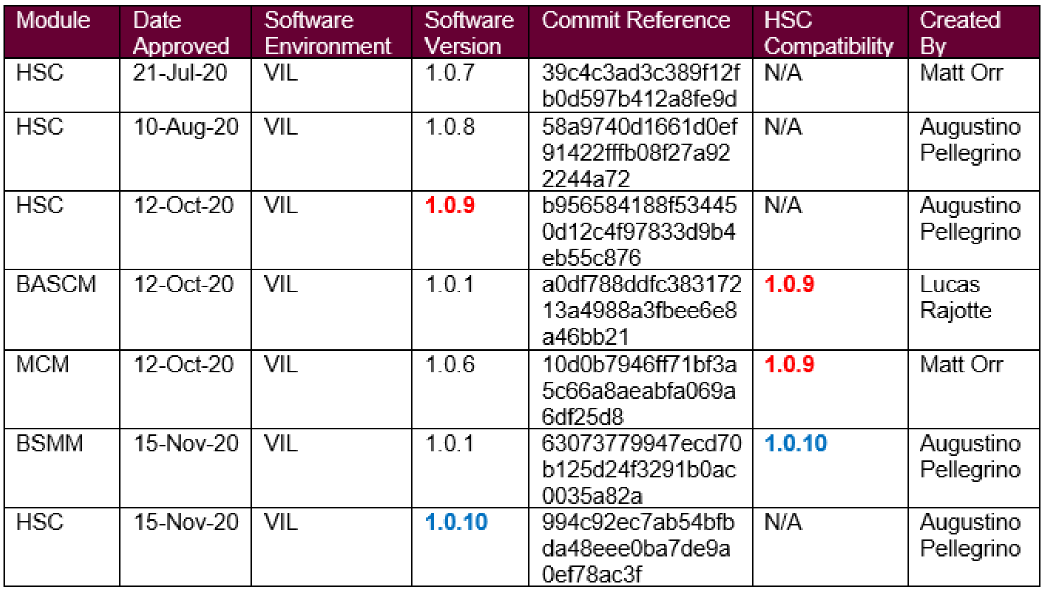
Step	Title	Description	Responsible Party
1	Branch Creation	When a model component requires changes, a new branch shall be created from the master model branch.	Developer
2	Model Development	As model changes are made on feature branch, developer will commit and push to the remote repository.	Developer
3	Developer Review	Once model is developed, Developer shall test and validate model requirements within feature branch, present branch to PCM Lead	Developer
4	PCM Lead Review	PCM Lead shall review feature branch validation results, giving approval for branch merging.	PCM Lead
5	Branch Merging	Developer and PCM Lead shall collaboratively merge feature branch with master branch.	PCM Lead/Developer
6	Merged Commit	Annotation Git Commit Tag shall be created for the software version release which contains Developer ID, changes made, date committed.	Developer
7	Push to Master	PCM Lead shall push the merged master branch to remote repository, thus finalizing the new model version.	PCM Lead

identification, using the *Major.Minor.Patch* format. Finally, the **HSC Compatibility** field indicates the HSC software release which is compatible with a given software module. For example, the HSC software release *1.0.10* is compatible with the BATTCR release *1.0.1* as its corresponding **HSC Compatibility** field matches the HSC version number. In contrast, the BASCR and MCM are only compatible with the older HSC software release *1.0.9*. Table 4.4 outlines the software version control process, including the main persons responsible for each step. Using this process allows the team to independently develop and test software modules, while ensuring that a full HSC system software build contains only those modules which are compatible with one another.

4.3 Conventional Model Modularization

Throughout the software development process, an importance has been placed on software structuring and partitioning to modularize the system for independent development, testing, and verification. The Simulink constructs that can be used to accomplish this structural decomposition are Simulink Model References, Model Variants, and virtual subsystems. The supervisory controller (HSC) Model has been implemented as a Simulink Model Reference within the hybrid vehicle model (discussed in Section 3.3). Furthermore, the HSC internal system components have also been implemented primarily with Model References, in order for the developer or small group of developers to be assigned a single component or feature, which they will independently develop and test. Once the functionality of an individual component

Table 4.6: Excerpt from Software Version Control Document



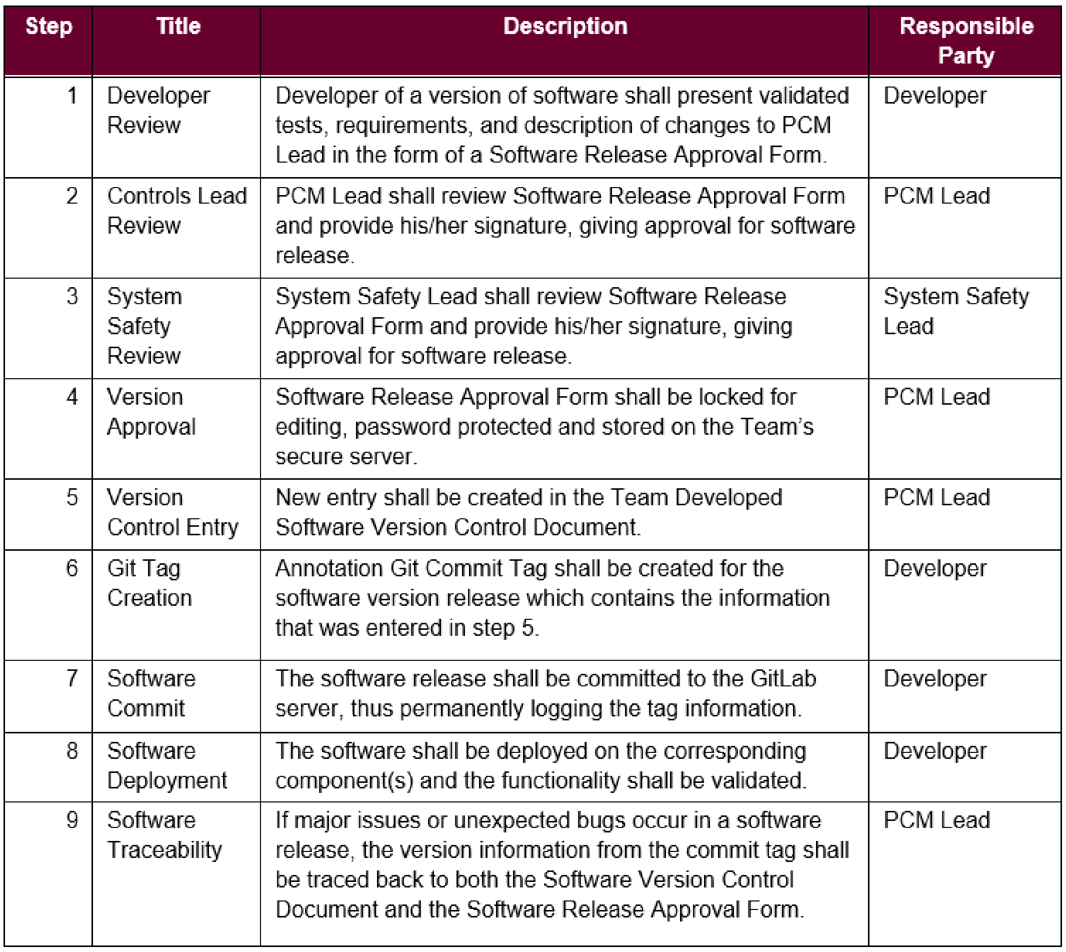
Module	Date Approved	Software Environment	Software Version	Commit Reference	HSC Compatibility	Created By
HSC	21-Jul-20	VIL	1.0.7	39c4c3ad3c389f12fb0d597b412a8fe9d	N/A	Matt Orr
HSC	10-Aug-20	VIL	1.0.8	58a9740d1661d0ef91422fffb08f27a922244a72	N/A	Augustino Pellegrino
HSC	12-Oct-20	VIL	**1.0.9**	b956584188f534450d12c4f97833d9b4eb55c876	N/A	Augustino Pellegrino
BASCM	12-Oct-20	VIL	1.0.1	a0df788ddfc38317213a4988a3fbee6e8a46bb21	**1.0.9**	Lucas Rajotte
MCM	12-Oct-20	VIL	1.0.6	10d0b7946ff71bf3a5c66a8aeabfa069a6df25d8	**1.0.9**	Matt Orr
BSMM	15-Nov-20	VIL	1.0.1	63073779947ecd70b125d24f3291b0ac0035a82a	**1.0.10**	Augustino Pellegrino
HSC	15-Nov-20	VIL	**1.0.10**	994c92ec7ab54bfbda48eee0ba7de9a0ef78ac3f	N/A	Augustino Pellegrino

has been validated, the model changes can be reflected in the overall HSC system by updating the Model Reference file path to point to the updated software component. This structure also provides improved system changeability in models with high cohesion. A likely change to the system would ideally affect only one of the various model references within the HSC, and could be modified independent from the root level software.

4.3.1 Software Partitioning via Model References

Model partitioning within the HSC model creates a modular structure encouraging independent development and testing. The Model Reference Simulink construct can be leveraged to aid in this model partitioning, simplifying system decomposition and separating areas of concern. Major software features and components can be developed, tested, and validated independently, allowing multiple developers to be productive simultaneously with minimal collaboration. The set of components and features associated with the vehicle Soft ECUs, HSC, and Vehicle Plant components were primarily implemented as seperate Simulink Models, included in the root HSC software via Model Reference Blocks. This process of partitioning via model references can continue to as granular of a level as is desired. The largest factor in determining this level of granularity is a) The size and complexity of a given component or feature, and b) the effect that changes in each component or feature would have on the other elements in its model. If the complexity is high and/or changes would have a large effect, the component or feature is moved to a separate model and pointed to in the hybrid propulsion system as a model reference. The software partitioning process outlined above has been represented graphically by Figure 4.11, showing the top level HSC System Implemented as a model reference. The model reference construct

Table 4.7: Software Version Control Process



Step	Title	Description	Responsible Party
1	Developer Review	Developer of a version of software shall present validated tests, requirements, and description of changes to PCM Lead in the form of a Software Release Approval Form.	Developer
2	Controls Lead Review	PCM Lead shall review Software Release Approval Form and provide his/her signature, giving approval for software release.	PCM Lead
3	System Safety Review	System Safety Lead shall review Software Release Approval Form and provide his/her signature, giving approval for software release.	System Safety Lead
4	Version Approval	Software Release Approval Form shall be locked for editing, password protected and stored on the Team's secure server.	PCM Lead
5	Version Control Entry	New entry shall be created in the Team Developed Software Version Control Document.	PCM Lead
6	Git Tag Creation	Annotation Git Commit Tag shall be created for the software version release which contains the information that was entered in step 5.	Developer
7	Software Commit	The software release shall be committed to the GitLab server, thus permanently logging the tag information.	Developer
8	Software Deployment	The software shall be deployed on the corresponding component(s) and the functionality shall be validated.	Developer
9	Software Traceability	If major issues or unexpected bugs occur in a software release, the version information from the commit tag shall be traced back to both the Software Version Control Document and the Software Release Approval Form.	PCM Lead

is also a useful tool for introducing additional software functionalities. Updating existing software features, or integrating new ones simply requires a modification of the interface at the top hierarchical level. The significant portion of change can occur within the given reference module and can have virtually no effect on the behaviors of the modules it interacts with.

4.3.2 XIL Functionality via Model Variants

The transition between XIL environments was facilitated with the use of Model Variant constructs. Section 2.2.1.3 provides an overview on the basic functionality of model variants, which are primarily used to develop a selectable set of model references which can only be active one at a time during simulation. This functionality was used to address the issue of XIL interface development. The transition from model development and MIL testing to software deployment and vehicle testing (HIL & VIL) requires distinct interfaces and model configuration parameters. Examples of differences between XIL interface designs include the use of dSPACE RTI CAN and I/O block sets within the HIL and VIL environment in order to transmit and receive serial data from a physical network. In the MIL environment, these CAN receivers and transceivers are not present, and are primarily represented by Simulink inports and outports. Models which utilize the dSPACE MicroAutobox II specific Simulink block libraries must also adhere to the modeling configuration limitations of the hardware. This includes a specified sample time, as well as many hardware specific code generation and simulation settings. These changes must remain unique to the HIL and VIL environment, and are not compatible with MIL simulation settings.

As a result, the three primary XIL interfaces used throughout the model development process were placed in individual Simulink Models, and created as variant choices within a Model Variant Object. The input and output XIL interfaces can be seen at a high level in Figure 4.12, while the internals of the input and output interface model variants can be seen in Figures 4.13 and 4.14, respectively. A primary advantage resulting from this model design is that the MIL, HIL, & VIL interfaces can remain separate from one another and the model configuration parameters required for hardware deployment can be internal to the corresponding Model Reference. In addition, the internal functionality of the HSC and its resulting modules remain constant and do not require additional models or code to represent all possible XIL environments. Code can be independently distributed and developed across the software team, while testing can be performed in the target XIL environment by selecting the appropriate interface and configurations (step-time, code generation etc.).

4.3.3 Program State Design via Data Dictionaries

The Simulink Data dictionary construct was utilized to define and organize program state constants for modeling and simulation. Several data dictionaries were defined for the major software and hardware components and organized in a hierarchical manner to facilitate more efficient calibrations and handling of changes to model constants. The top level of the data hierarchy contains the System Data dictionary (*SystemDD.sldd*). To add lower levels of hierarchy, referenced dictionaries were added to the system dictionary and organized according to the modeling components which

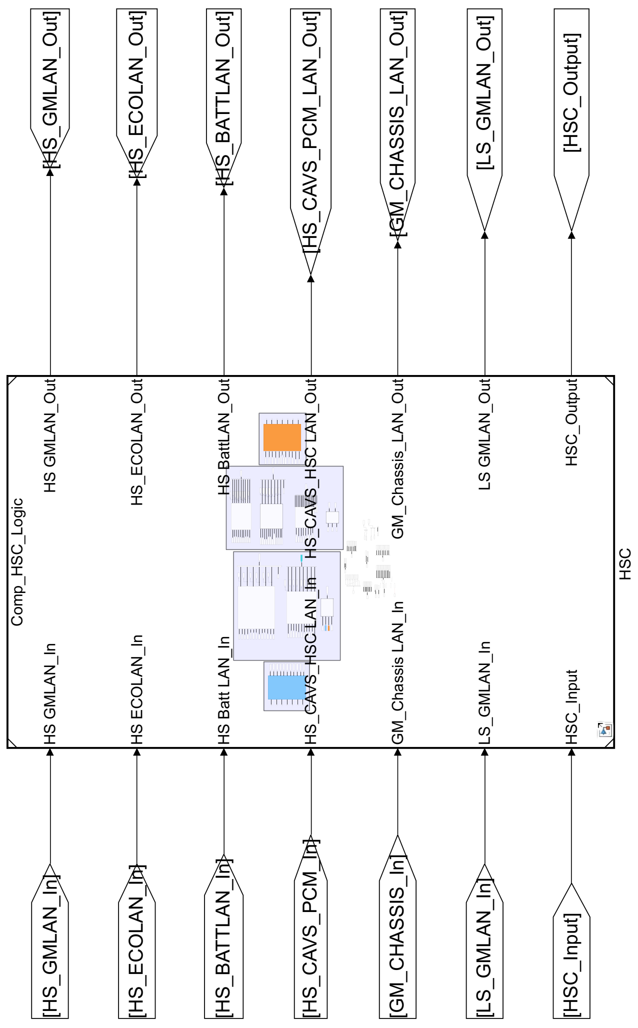


Figure 4.11: HSC Implemented using a Model Reference Block

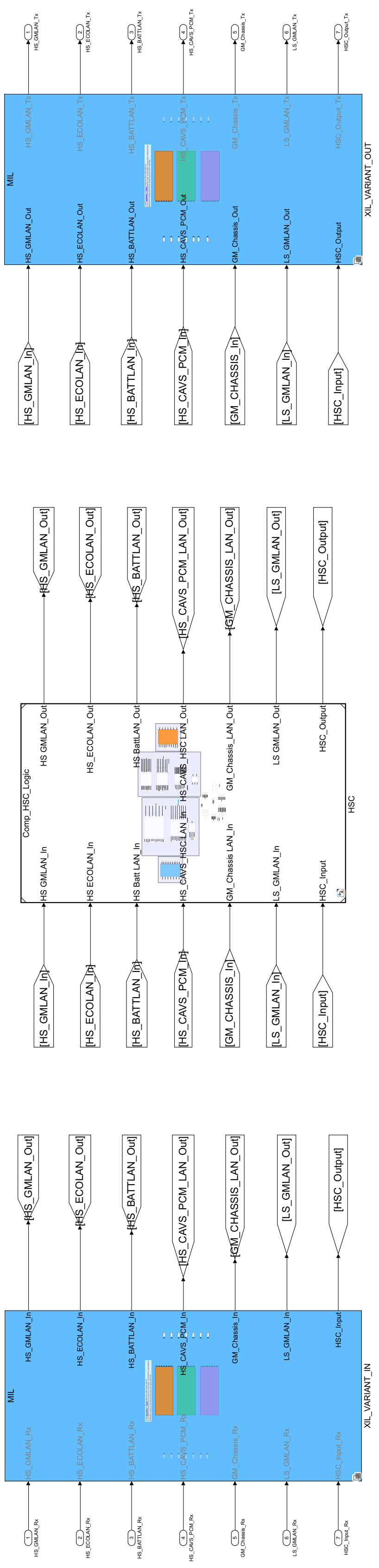


Figure 4.12: XIL Interfaces using Model Variants

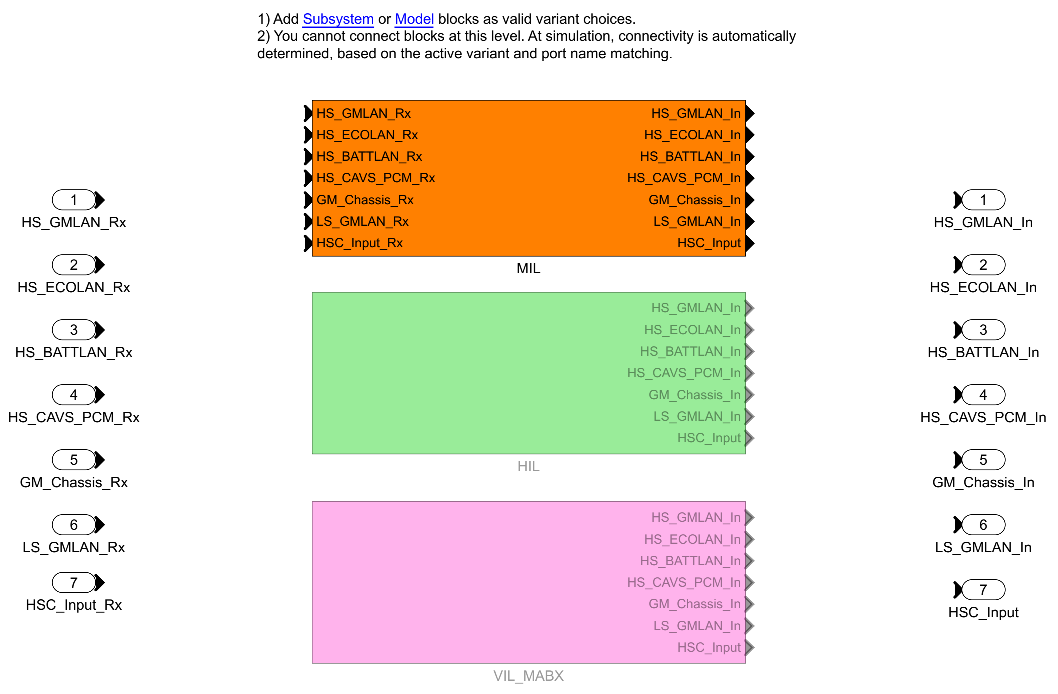


FIGURE 4.13: XIL Input Model Variant

use or reference the calibrations and constants. Currently, a data dictionary has been added for the BAS, Battery, Drivetrain, Engine, Low Voltage System, and Motor Data. Figure 4.15 shows the System Data dictionary within the Vehicle Simulink Model, and its associated referenced dictionaries.

4.3.4 HSC Internal Modularization

As mentioned in Section 3.3, the competition vehicle model is composed of four distinct components: the driver source, the vehicle controllers, the Blazer vehicle plant, and the virtualization block. The Simulink model and virtual CAN I/O have been organized in a modular fashion to create a fast and repeatable transition between XIL Testing, as well as allow for simultaneous software development and deployment among multiple team members. The controller model has been organized into an input conversion layer, an HSC application layer, and an output conversion layer. Inputs coming from the vehicle plant and driver source block have been decomposed into 7 virtual buses within the Controller to HSC Subsystem (Figure 4.16). Six of these outputs represent the six CAN Channels present in the Hybrid Supervisory Controller's serial data interface (see section 4.2.1). The seventh output is reserved for any additional I/O which is not transmitted via CAN. The HSC Model interface has been designed to receive these seven virtual buses, process the signals within them, and transmit them back into the rest of the vehicle model (Figure 4.17).

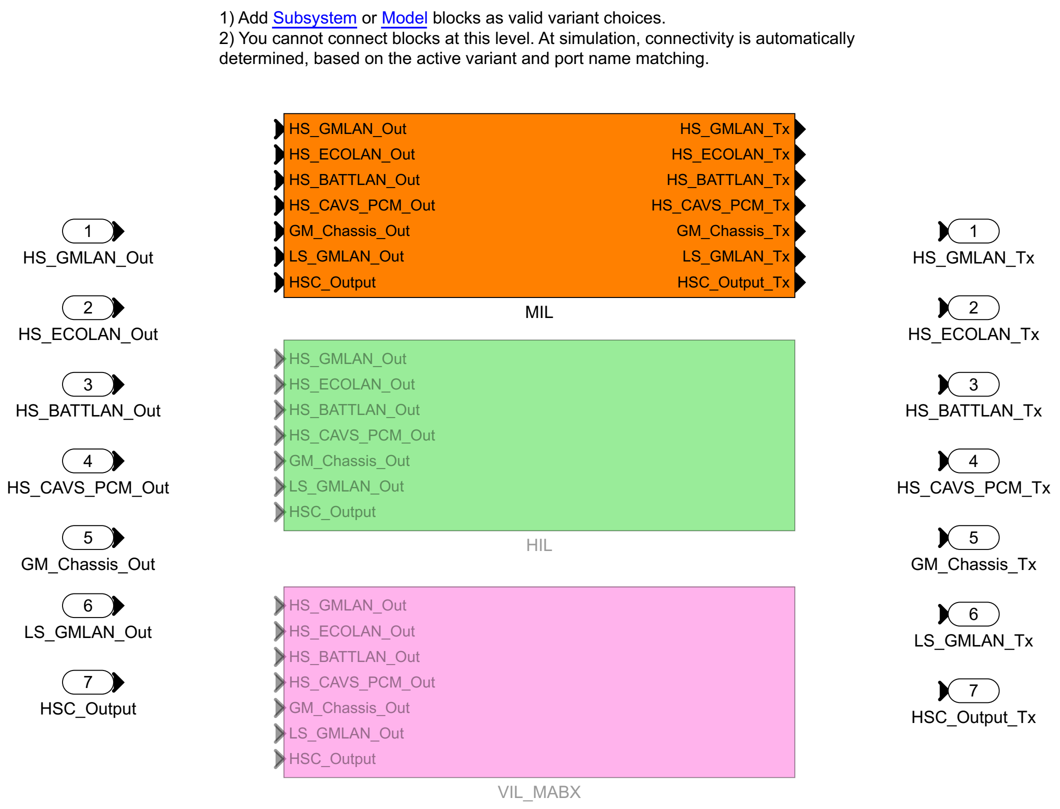
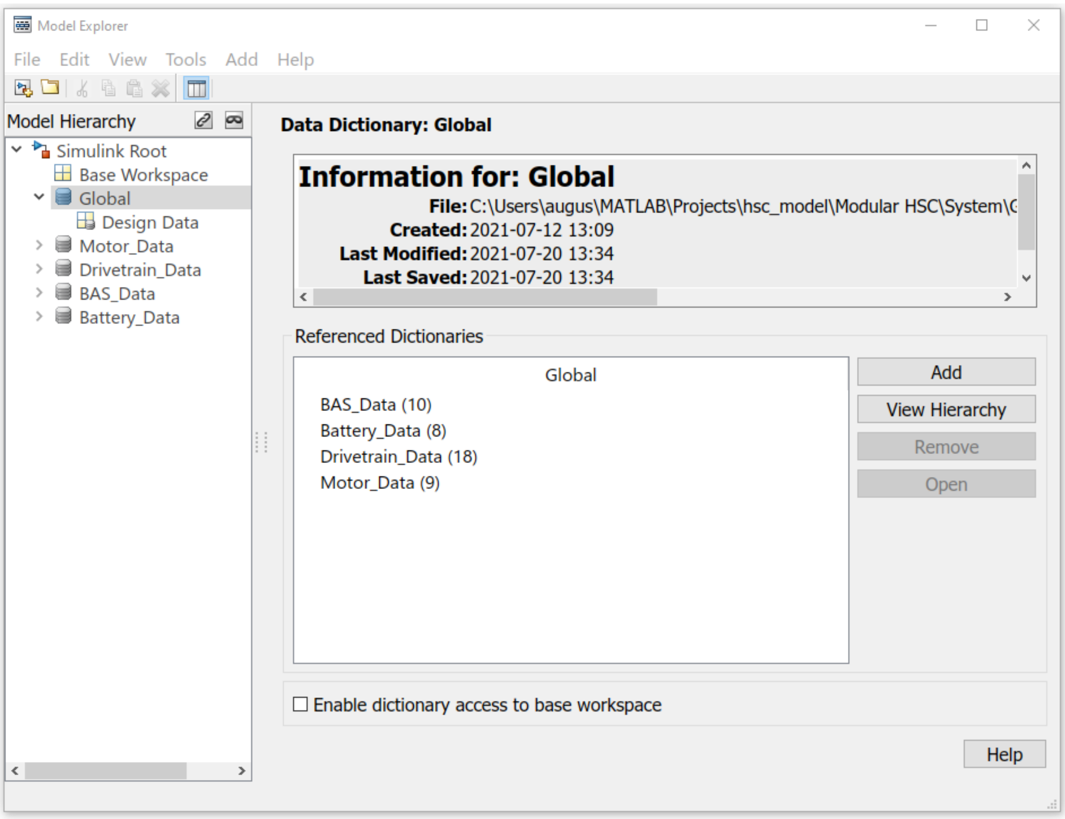

1) Add Subsystem or Model blocks as valid variant choices.
2) You cannot connect blocks at this level. At simulation, connectivity is automatically determined, based on the active variant and port name matching.

FIGURE 4.14: XIL Output Model Variant



Data Dictionary: Global

Information for: Global

File: C:\Users\augus\MATLAB\Projects\hsc_model\Modular HSC\System\G
Created: 2021-07-12 13:09
Last Modified: 2021-07-20 13:34
Last Saved: 2021-07-20 13:34

Referenced Dictionaries

Global
BAS_Data (10)
Battery_Data (8)
Drivetrain_Data (18)
Motor_Data (9)

☐ Enable dictionary access to base workspace

FIGURE 4.15: Model Explorer view of Data Dictionary *Global.sldd*

The internal HSC model has been organized in a modular fashion based on the Functional Supervisory Controller Architecture discussed in Section 4.2.2. A high-level view of the model can be seen in Figure 4.18. The seven bus inputs of the outer HSC interface are redistributed into separate Simulink Bus Objects for each software module within the operation mode and component control layers. This design decision was made for changes to the number of signals required by each module to occur independently and have virtually no effect on the outer interface of any individual feature or component.

All major software modules within the HSC have been organized as model references and all minor modules have been implemented as virtual subsystems. Major modules have been defined as functions or features with high algorithmic complexity, a high number of blocks, and/or functions that will require extensive changes throughout the development process. Model references allow for independent development, testing and verification of these large systems. Minor modules have been defined as functions or features with low algorithmic complexity, low number of blocks required, functionality that does not need to be repeated throughout the model, and/or does not require extensive changes throughout the design process. An example of a major module is the Power Moding Module, and an example of a minor module is the Pedal Interpretation Module, seen in Figures 4.19 and 4.20 respectively. The output of each control module is transmitted to the HSC to Controller subsystem and is condensed back into the six CAN channels and one I/O channel. All outputs from the HSC Model are finally transmitted to the controller to plant subsystem, where they combined with the Soft ECU outputs and condensed into a single Control virtual bus. This single output is passed along to the vehicle plant at the top level of the model where it can be used and distributed as needed.

4.3.5 Interface Implementation via Bus Objects and Signal Tags

All module interfaces have been consolidated into Simulink Bus objects, with each object representing the consolidation of all inputs/outputs from a given module. The Propulsion Control Strategy module can be looked at as an example of this decomposition. As mentioned in Chapter 3, the energy management strategy requires data from several different components and modules, namely the Motor, Engine, and BAS HSC Modules, as well as sensor feedback from the Vehicle Plant, represented by the Bus signal *ECMS_Ctrl_In*. Goto/From tags have been used extensively throughout the application layer to facilitate a more organized visual layout and avoid complex signal routing. The **Tag Visibility** parameter for all tags have been set to local, to avoid sharing of program state. To avoid algebraic loops, initial outputs were created within modules using the Simulink Unit Delay Block. Figures 4.21 - 4.24 demonstrate the data flow involving the PCSM interface. The initial input signal *ECMS_Ctrl_In* coming from the input conversion layer is highlighted in yellow (Figure 4.21). The additional three input signals to the PCSM interface come from the MCM (highlighted in teal), the BASCM (highlighted in green), and the ECMR (highlighted in orange) of the component control layer (Figure 4.22). The single output signal generated by the PCSM (highlighted in red) is routed to each of the powertrain components requiring a torque command, namely the MCM, BASCM, and ECMR (Figure 4.23). Figure

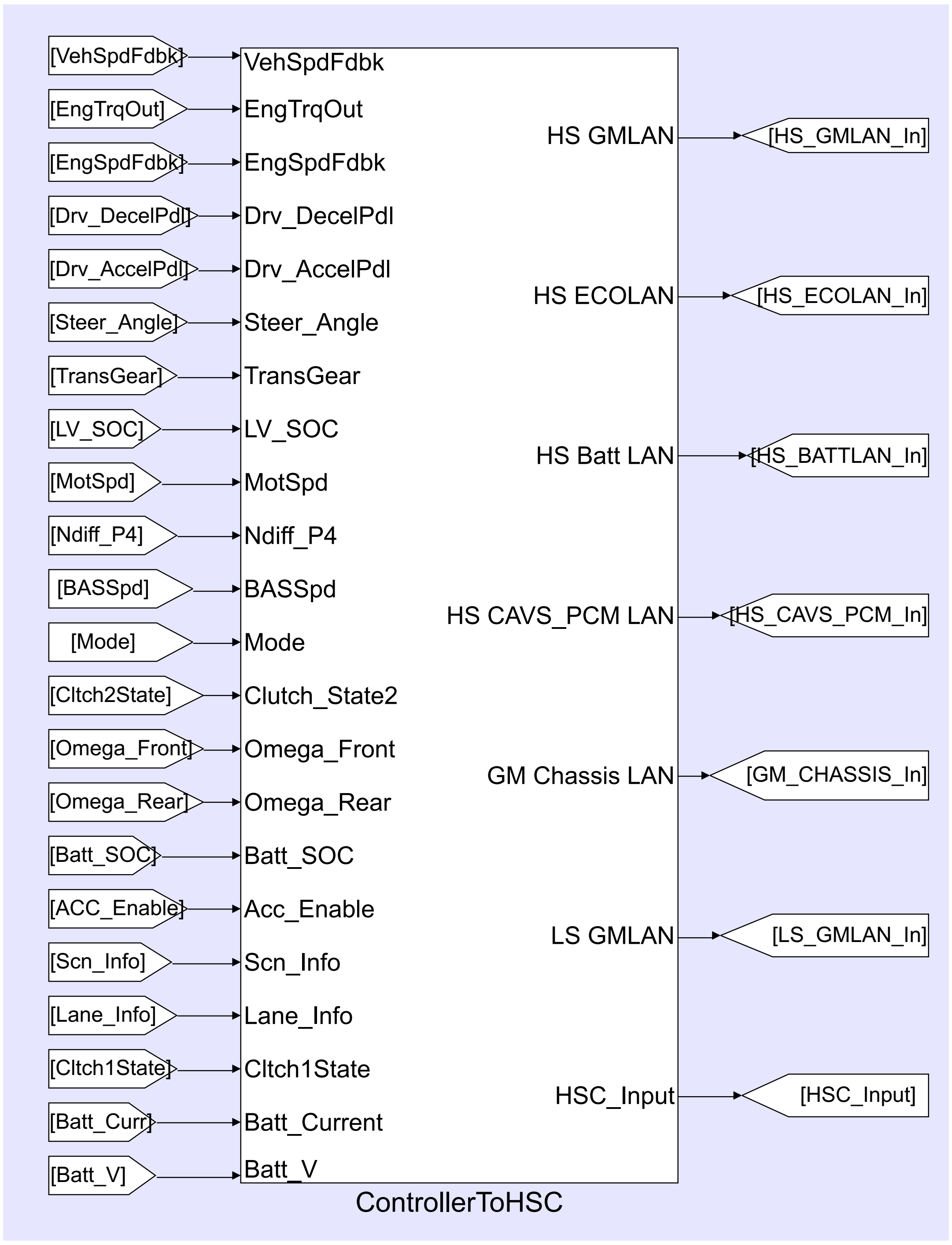


FIGURE 4.16: ControllerToHSC Subsystem in MIL Serial Data Interface

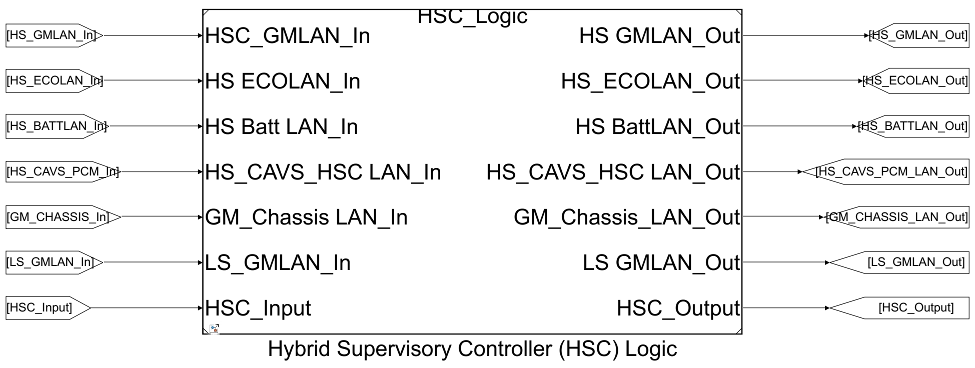


FIGURE 4.17: Top level view of HSC Model Interface

4.24 shows all four of these signals passed to the output conversion layer, which in turn passes them to the top level HSC interface.

4.4 Model and Software Testing

The third step of the vehicle design process is to test the software modules within the target Simulink model, and validate the corresponding requirements set. Testing performed in the MIL environment is discussed in Section 4.4.1, where the Simulink Requirements software package is used to facilitate functional validation. Components which have been successfully validated in the MIL environment are then deployed to target hardware for HIL and/or VIL testing. Section 4.4.2 provides an example of a bench test that was performed in the HIL environment, demonstrating the process for validating requirements.

4.4.1 Requirements Validation via Simulink Requirements

Simulink provides the tools necessary for keeping track of validation through the Simulink Requirements software package. Simulink Requirements supports the creation, testing, and validation of requirements associated with a given Simulink Model [30]. A previously created requirements specification document (RSD) can be uploaded and linked to specific model components and blocks throughout the system hierarchy. Figure 4.25 shows the requirements editor view of the HSC Model. The **rsd** document shown on the left is the requirements set (*slreqx* file) defined within Simulink. This requirement set is directly taken from an external RSD excel document, **rsd_doc_link** which has been imported into Simulink. Once defined, requirements at any abstraction level (system, component, software etc) can be directly linked to the model component or block which implements them. In the example given in Figure 4.25, the **Links** section in the bottom right shows the system requirement II.1 as *Related to:* the corresponding II.1 requirement within the master RSD, as well as *Implemented by:* the Input Data Processor Subsystem within the HSC Model.

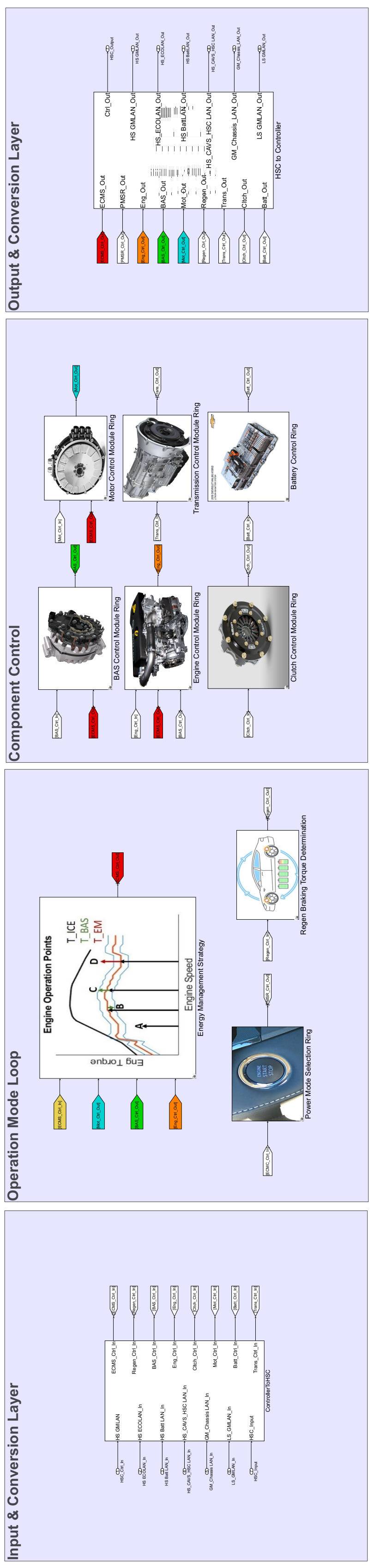

Input & Conversion Layer
Controller
ToHSC
Operation Mode Loop
Engine Operation Points
T_ICE
T_BAS
T_EM
Eng Torque
Engine Speed
Energy Management Strategy
Power Mode Selection Ring
Regen Braking Torque Determination
Component Control
BAS Control Module Ring
Engine Control Module Ring
Clutch Control Module Ring
Motor Control Module Ring
Transmission Control Module Ring
Battery Control Ring
Output & Conversion Layer
HSC to Controller

Figure 4.18: Top Hierarchical view of HSC Simulink Model

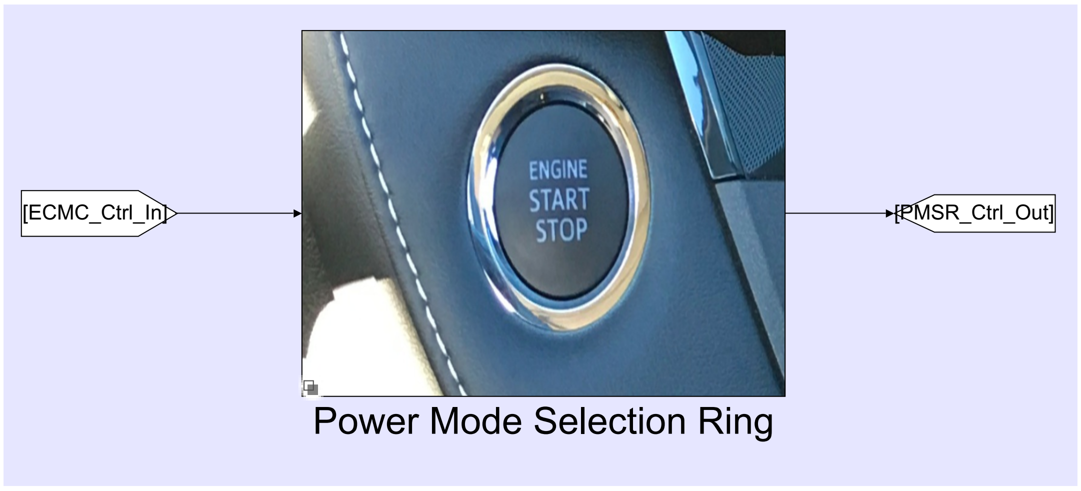


FIGURE 4.19: Example of Major Module implemented as Model Reference

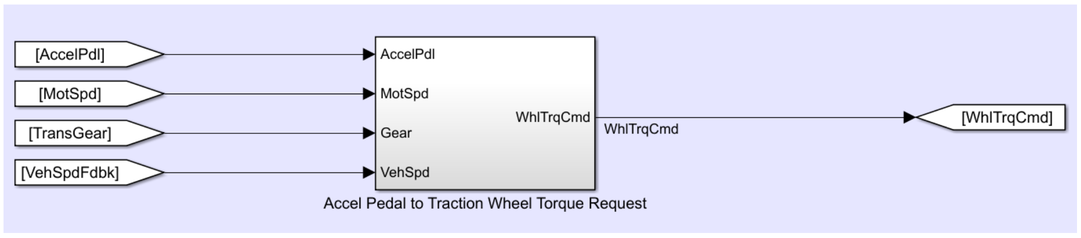


FIGURE 4.20: Example of minor module implemented as a virtual subsystem

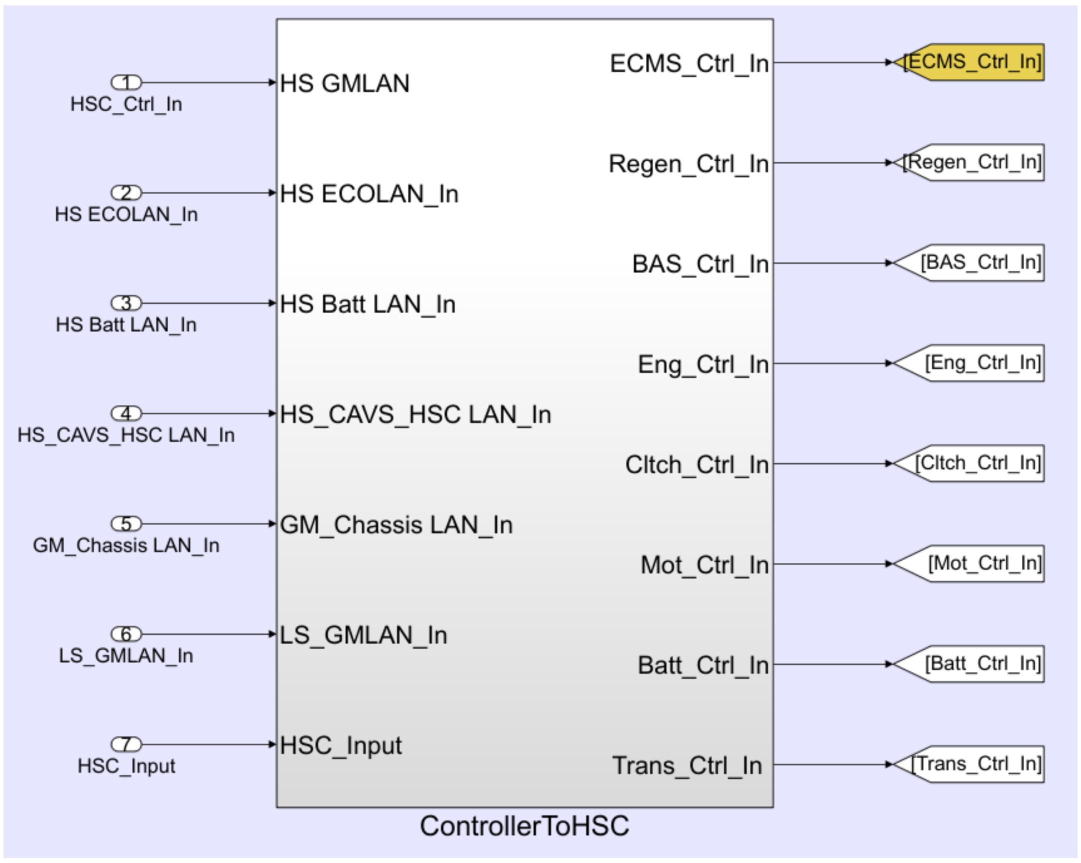


FIGURE 4.21: Data flow originating from Input conversion layer of HSC model

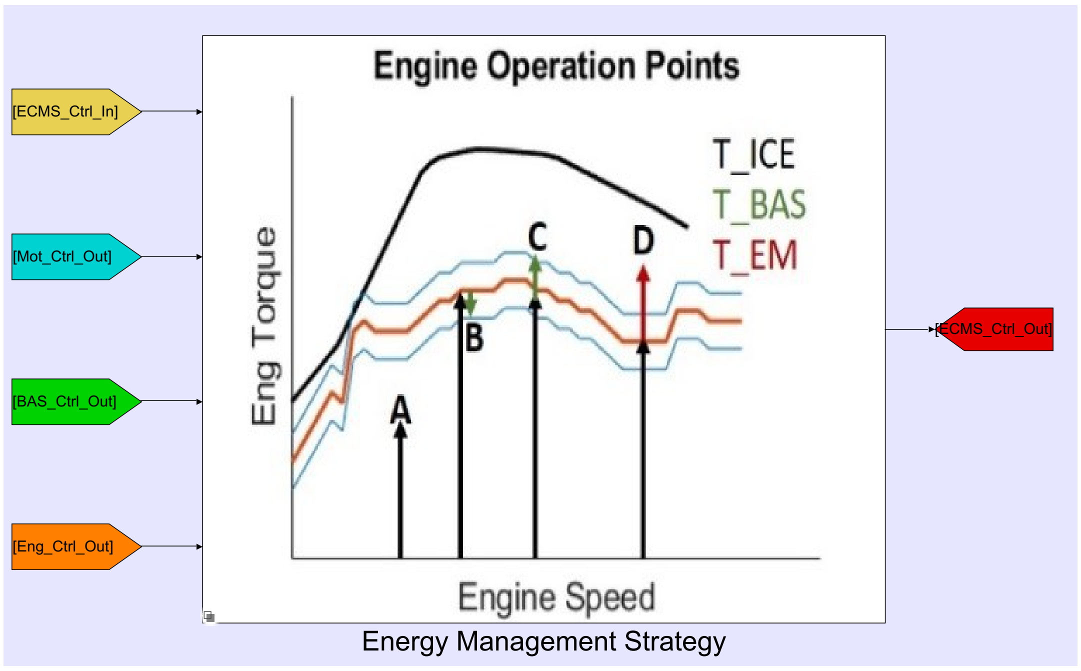


FIGURE 4.22: Data Flow of Propsulsion Control Strategy Module Interface

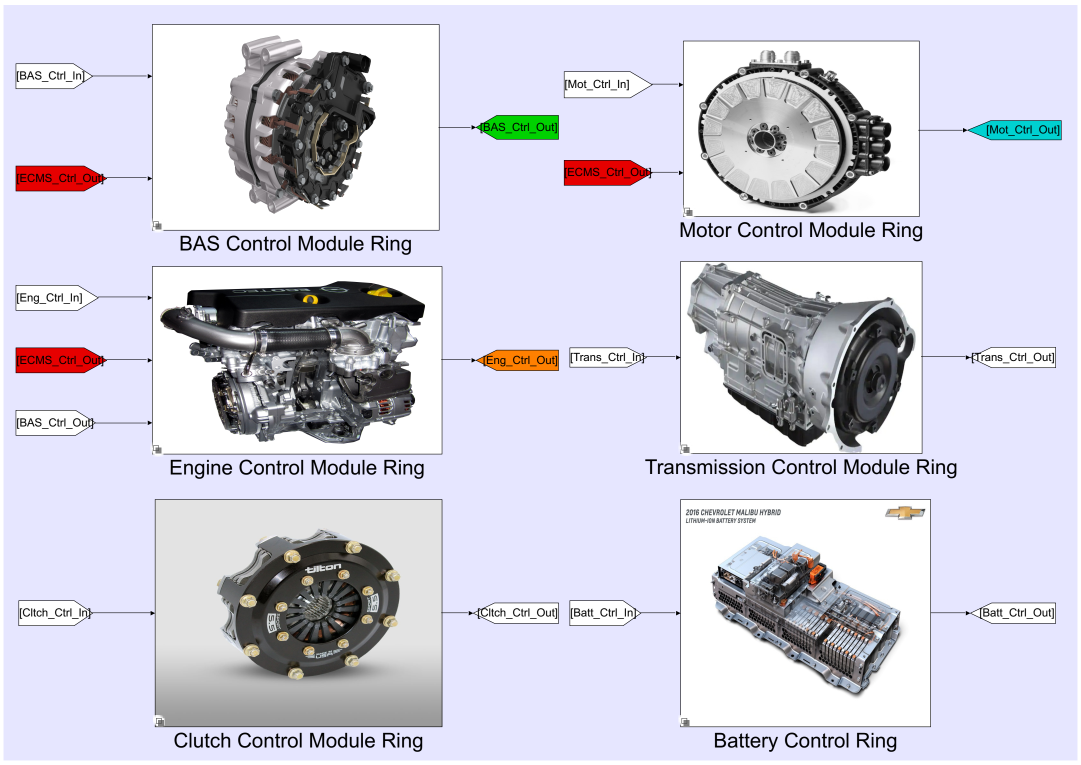


FIGURE 4.23: Data flow within the Component control layer

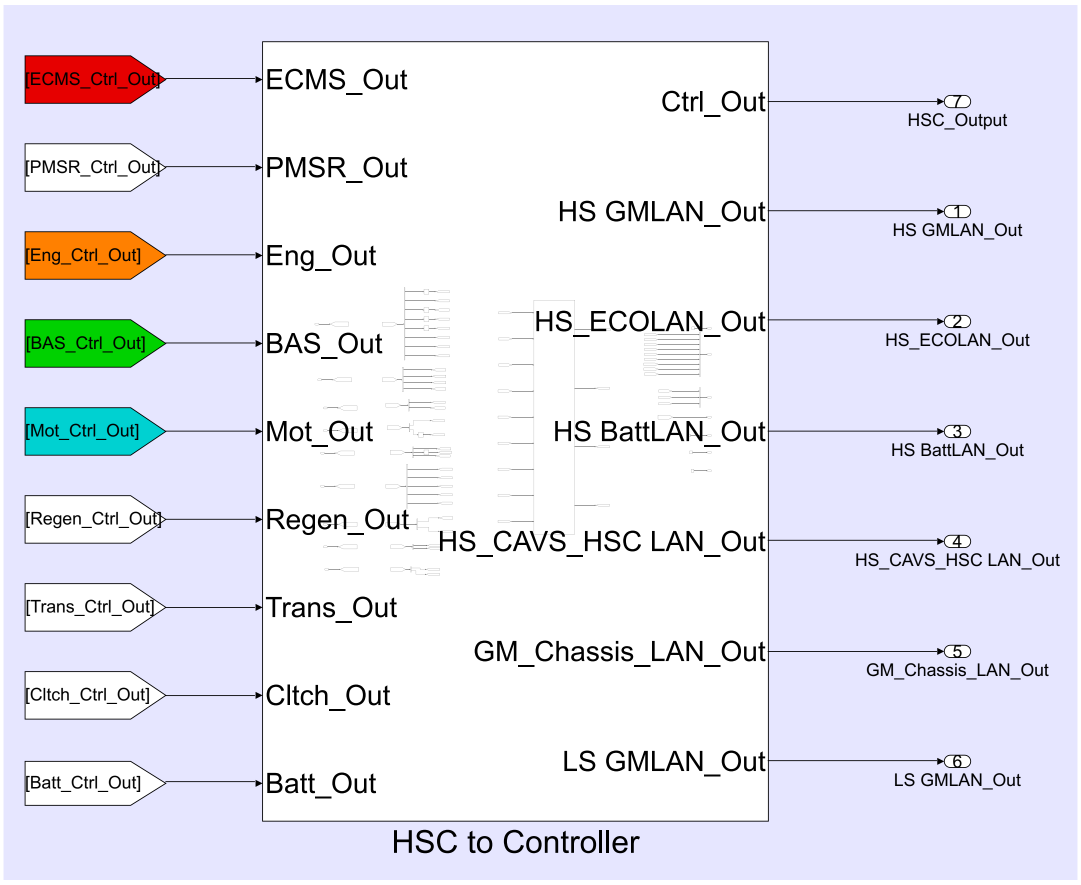
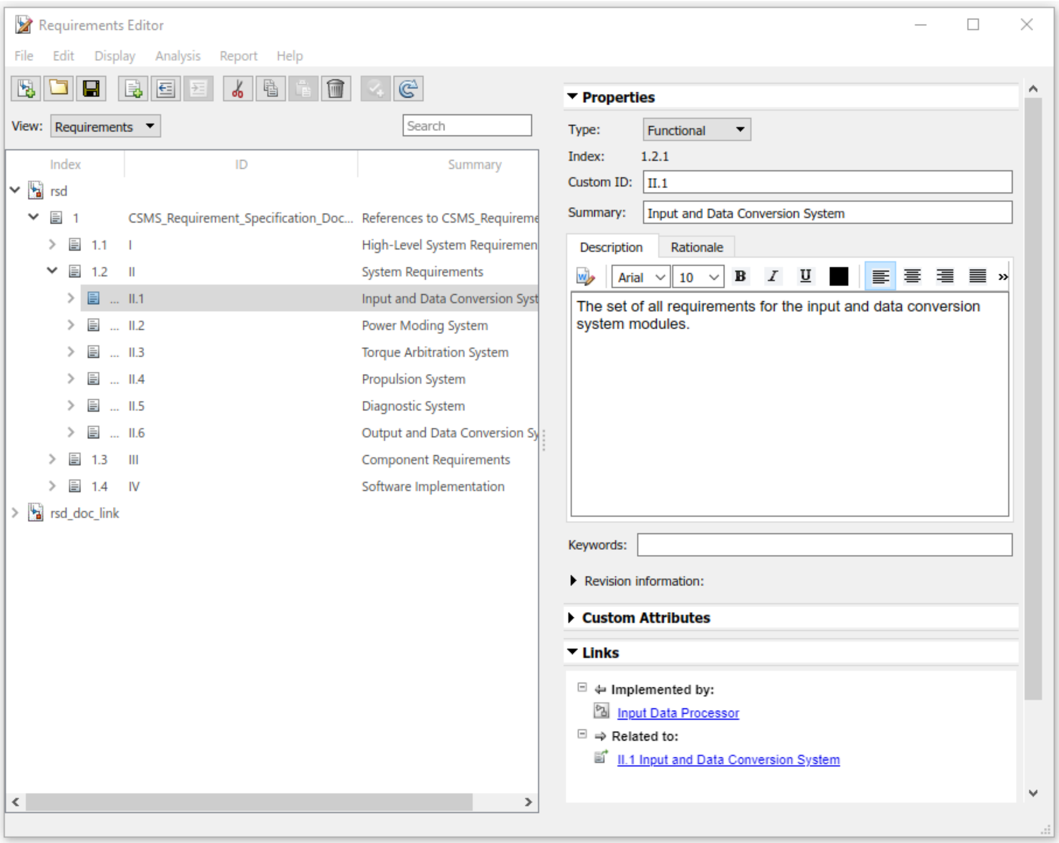


FIGURE 4.24: Data flow within the Output Conversion Layer



Index	ID	Summary
rsd		
1	CSMS_Requirement_Specification_Doc...	References to CSMS_Requireme
1.1	I	High-Level System Requiremen
1.2	II	System Requirements
...	II.1	Input and Data Conversion Syst
...	II.2	Power Moding System
...	II.3	Torque Arbitration System
...	II.4	Propulsion System
...	II.5	Diagnostic System
...	II.6	Output and Data Conversion Sy
1.3	III	Component Requirements
1.4	IV	Software Implementation
rsd_doc_link		

Properties

Type: Functional
Index: 1.2.1
Custom ID: II.1
Summary: Input and Data Conversion System

Description | Rationale

The set of all requirements for the input and data conversion system modules.

Keywords:

Revision information:

Custom Attributes

Links

Implemented by:
Input Data Processor
Related to:
II.1 Input and Data Conversion System

FIGURE 4.25: Requirements Editor view for the HSC Model RSD

Documenting and maintaining an RSD through Simulink Requirements helps facilitate accurate tracking of requirements implementation, testing, and validation.

4.4.2 Accelerator Pedal Processing: Fault Analysis and Test Bench

The accelerator pedal processing module discussed in section 4.1.4 was developed and tested by the author in Year 1 of the competition. To perform a unit test on the module, it was separated from the input conversion layer of the HSC model, and a test bench and harness were created. The developed model used for unit testing can be seen in Figure 4.26, and the signal interface can be seen in Table 4.8. Before the unit test was designed, a fault tree analysis was performed on the accelerator pedal processing system to understand potential system failures and determine a method of risk mitigation. Please note that fault analysis on all modules within the team vehicle is an ongoing and iterative process. As modules are tested in the VIL environment, their fault analysis and corresponding software implementation will change. Figure 4.27 shows the fault tree analysis performed at the time of the unit test discussed below.

Using the fault tree analysis combined with the functional requirement set discussed in Section 4.2.1, as well as the software interface shown in Table 4.8, a test

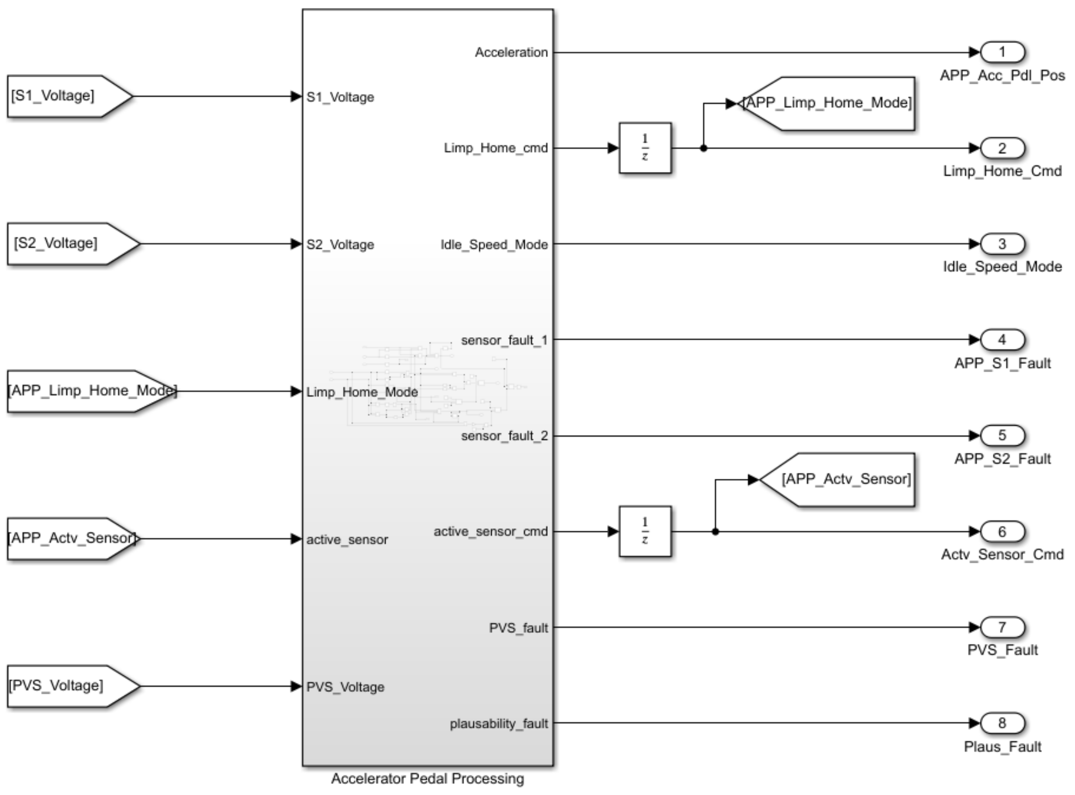


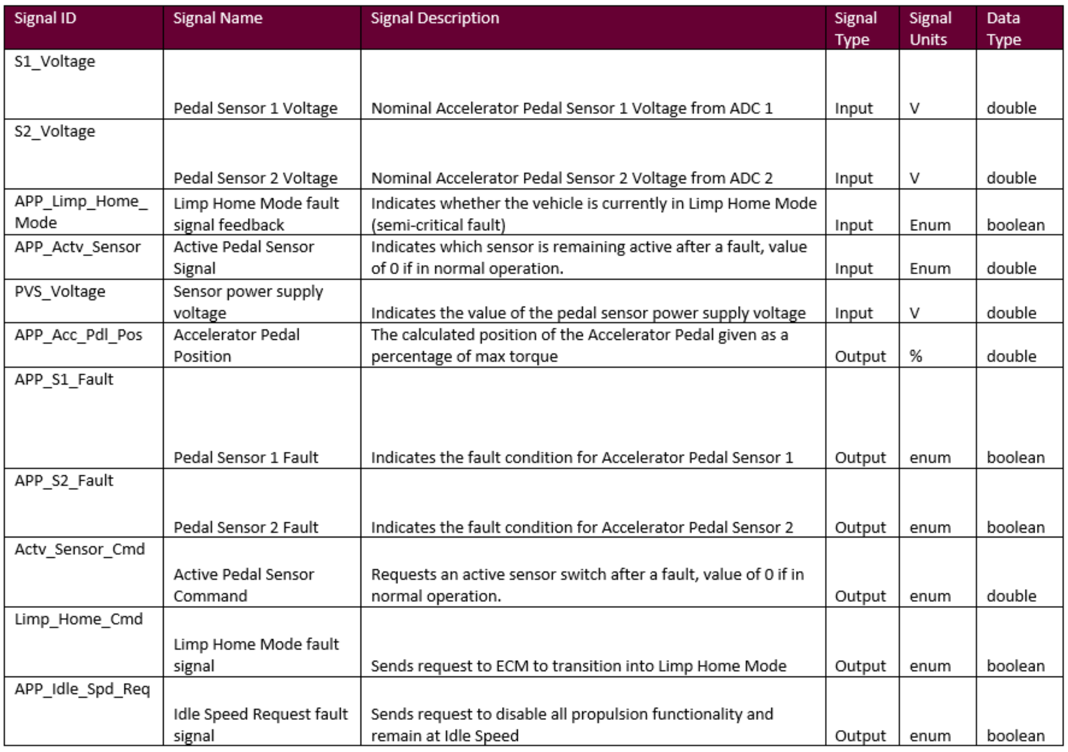
FIGURE 4.26: Accelerator Pedal Processing Test Harness Model

bench was developed. A physical accelerator pedal was wired into the HSC hardware (MABXII). To accurately simulate a difference in pedal sensor voltages, two separate 12V power supplies were connected to PVS1 and PVS2 of the pedal, respectively. Finally, a workstation laptop running ControlDesk [9] and MATLAB Simulink was connected to the HSC hardware via Ethernet and was used to simulate the test and validate the results. Please see Figure 4.28 for a graphical representation of the developed test bench.

4.4.3 Accelerator Pedal Processing: Test Results

The test results described below were performed to validate Requirements *III.4.1.3* and *III.4.1.4* shown in Table 4.1. The expected test results for requirement *III.4.1.3* are shown in Table 4.9. The power supply associated with the input *S1_Voltage* was gradually increased until it surpassed the value defined as *thresh_v*. This led to a successful system reaction of the Sensor 1 Fault signal (*APP_S1_Fault*) becoming active (shown in Figure 4.29). At the point of the Sensor 1 Fault, the module also successfully outputs a Limp Home Mode Command and switches the active sensor to PVS2 (shown in Figure 4.30). Once Limp Home Mode has been activated, the system also restricts acceleration pedal position to 25% of its max torque or less, indicated in Figure 4.31; this completes the verification of requirement *III.4.1.3*.

Table 4.8: Accelerator Pedal Test Harness Interface



Signal ID	Signal Name	Signal Description	Signal Type	Signal Units	Data Type
S1_Voltage	Pedal Sensor 1 Voltage	Nominal Accelerator Pedal Sensor 1 Voltage from ADC 1	Input	V	double
S2_Voltage	Pedal Sensor 2 Voltage	Nominal Accelerator Pedal Sensor 2 Voltage from ADC 2	Input	V	double
APP_Limp_Home_Mode	Limp Home Mode fault signal feedback	Indicates whether the vehicle is currently in Limp Home Mode (semi-critical fault)	Input	Enum	boolean
APP_Actv_Sensor	Active Pedal Sensor Signal	Indicates which sensor is remaining active after a fault, value of 0 if in normal operation.	Input	Enum	double
PVS_Voltage	Sensor power supply voltage	Indicates the value of the pedal sensor power supply voltage	Input	V	double
APP_Acc_Pdl_Pos	Accelerator Pedal Position	The calculated position of the Accelerator Pedal given as a percentage of max torque	Output	%	double
APP_S1_Fault	Pedal Sensor 1 Fault	Indicates the fault condition for Accelerator Pedal Sensor 1	Output	enum	boolean
APP_S2_Fault	Pedal Sensor 2 Fault	Indicates the fault condition for Accelerator Pedal Sensor 2	Output	enum	boolean
Actv_Sensor_Cmd	Active Pedal Sensor Command	Requests an active sensor switch after a fault, value of 0 if in normal operation.	Output	enum	double
Limp_Home_Cmd	Limp Home Mode fault signal	Sends request to ECM to transition into Limp Home Mode	Output	enum	boolean
APP_Idle_Spd_Req	Idle Speed Request fault signal	Sends request to disable all propulsion functionality and remain at Idle Speed	Output	enum	boolean

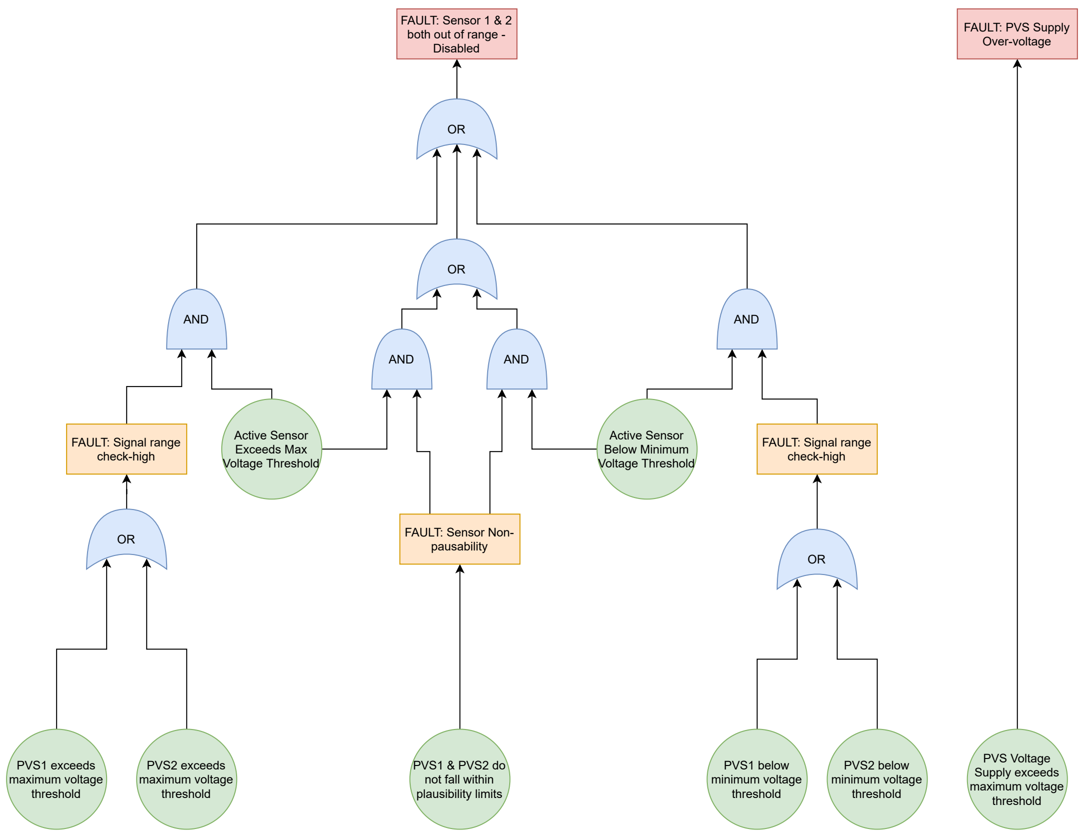

FAULT: Sensor 1 & 2 both out of range - Disabled
FAULT: PVS Supply Over-voltage
OR
OR
AND
AND
AND
AND
FAULT: Signal range check-high
Active Sensor Exceeds Max Voltage Threshold
Active Sensor Below Minimum Voltage Threshold
FAULT: Signal range check-high
OR
FAULT: Sensor Non-pausability
OR
PVS1 exceeds maximum voltage threshold
PVS2 exceeds maximum voltage threshold
PVS1 & PVS2 do not fall within plausibility limits
PVS1 below minimum voltage threshold
PVS2 below minimum voltage threshold
PVS Voltage Supply exceeds maximum voltage threshold

Figure 4.27: Accelerator Pedal Processing Fault Tree Analysis

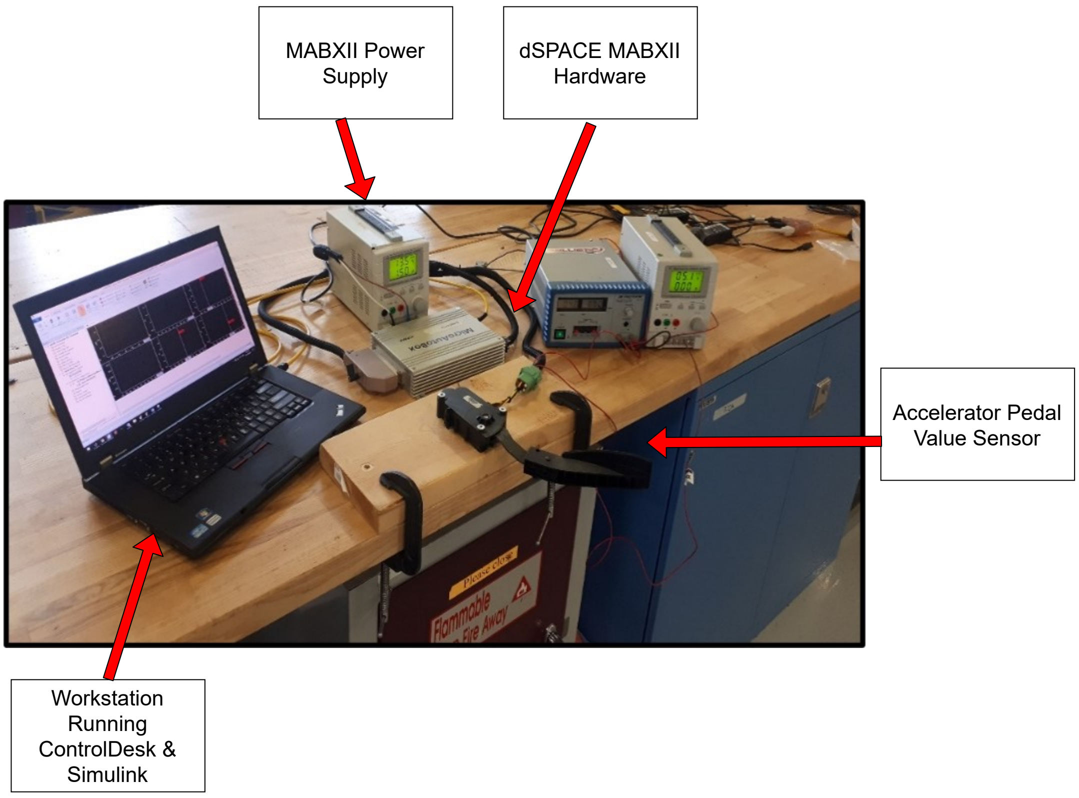

MABXII Power Supply
dSPACE MABXII Hardware
Accelerator Pedal Value Sensor
Workstation Running ControlDesk & Simulink

FIGURE 4.28: Pedal Interpretation Test Bench Setup

TABLE 4.9: Expected Test Results for Requirement *III.4.1.3* validation

Monitoring Fault	Fault Condition	Fault Reaction	Signal Output Changes

Sensor 1 - signal range check high	S1_Voltage > thresh_v	Enter Limp Home Mode, change active sensor to Sensor 2	APP_S1_Fault = 1
			Limp_Home_Cmd = 1
			Actv_Sensor_Cmd = 2
			APP_Acc_Pdl_Pos <=25

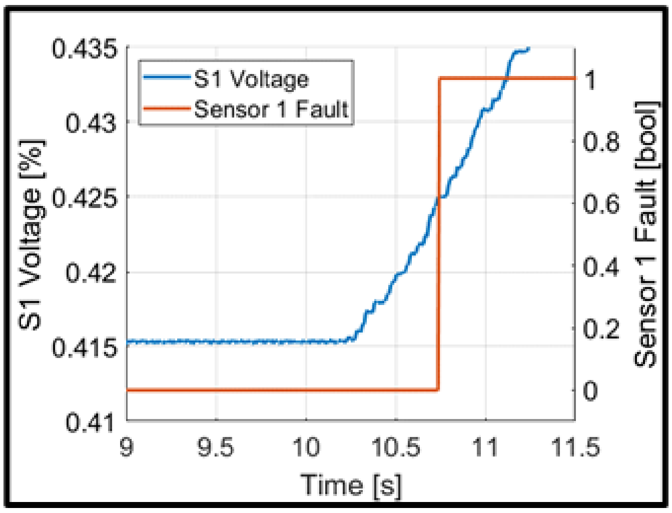


FIGURE 4.29: Sensor Voltage Input, Fault reaction output

The expected test results for requirement *III.4.1.4* are shown in Table 4.10. The power supply associated with the input *S2_Voltage* was gradually increased until it surpassed the value defined as *thresh_v*. This led to a successful system reaction of the Sensor 2 Fault signal (*APP_S2_Fault*) becoming active (shown in Figure 4.32). At the point of the Sensor 2 Fault, the module also successfully outputs a Idle Speed Command and switches the active sensor to 0, indicating both sensors are faulty (shown in Figure 4.33). Once Idle Speed Demand has been activated, the system fully disables propulsion by restricting the acceleration pedal position to 0%, indicated in Figure 4.34; this completes the verification of requirement *III.4.1.4*.

TABLE 4.10: Expected Test Results for Requirement *III.4.1.4* validation

Monitoring Fault	Fault Condition	Fault Reaction	Signal Output Changes

Sensor 12- signal range check high	Limp_Home_Mode == 1
	AND
	S2_Voltage > thresh_v
		Demand Idle Speed, Accelerator Pedal Disabled	APP_S2_Fault = 1
			APP_Idle_Spd_Req = 1
			Actv_Sensor_Cmd = 0
			APP_Acc_Pdl_Pos = 0

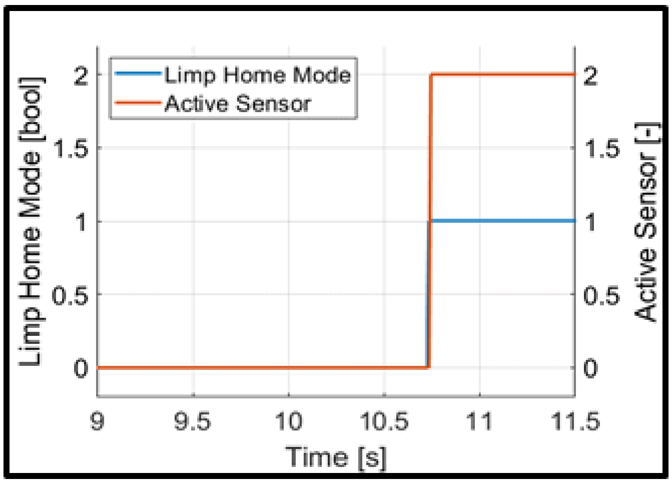


FIGURE 4.30: System output response to Sensor 1 Fault

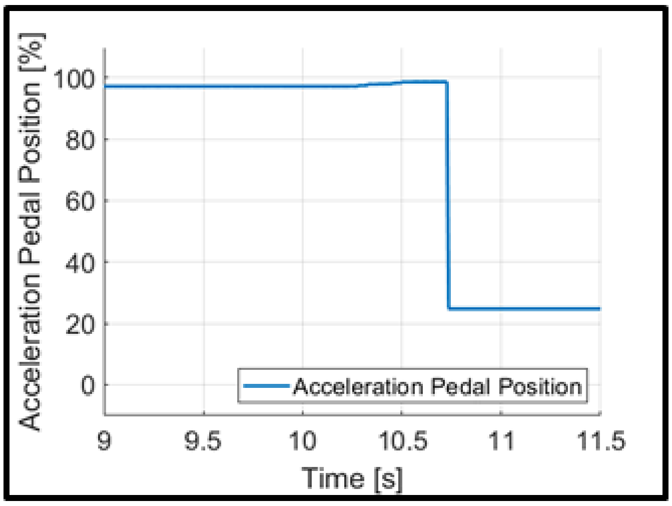


FIGURE 4.31: Verification of Requirement *III.4.1.3*

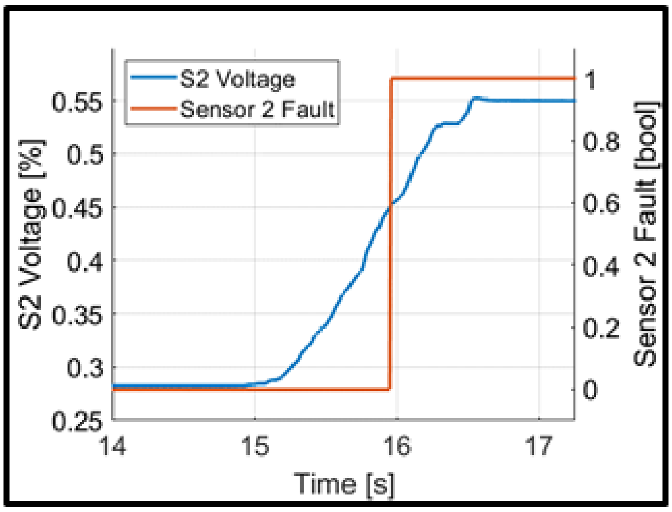


FIGURE 4.32: Sensor Voltage Input, Fault reaction output

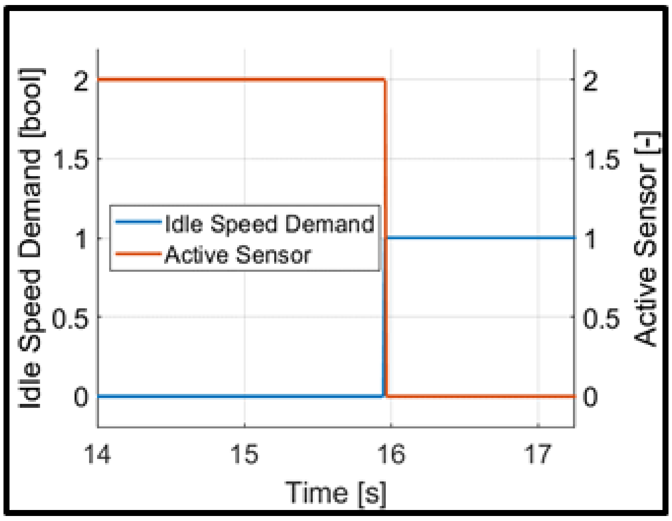


FIGURE 4.33: System output response to Sensor 1 & 2 Fault

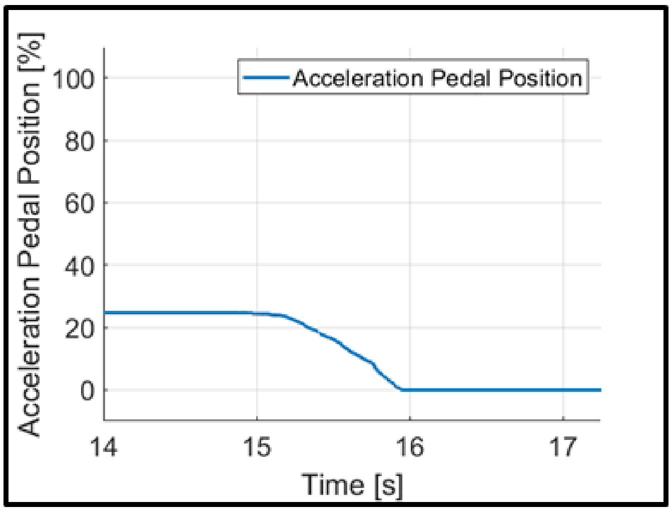


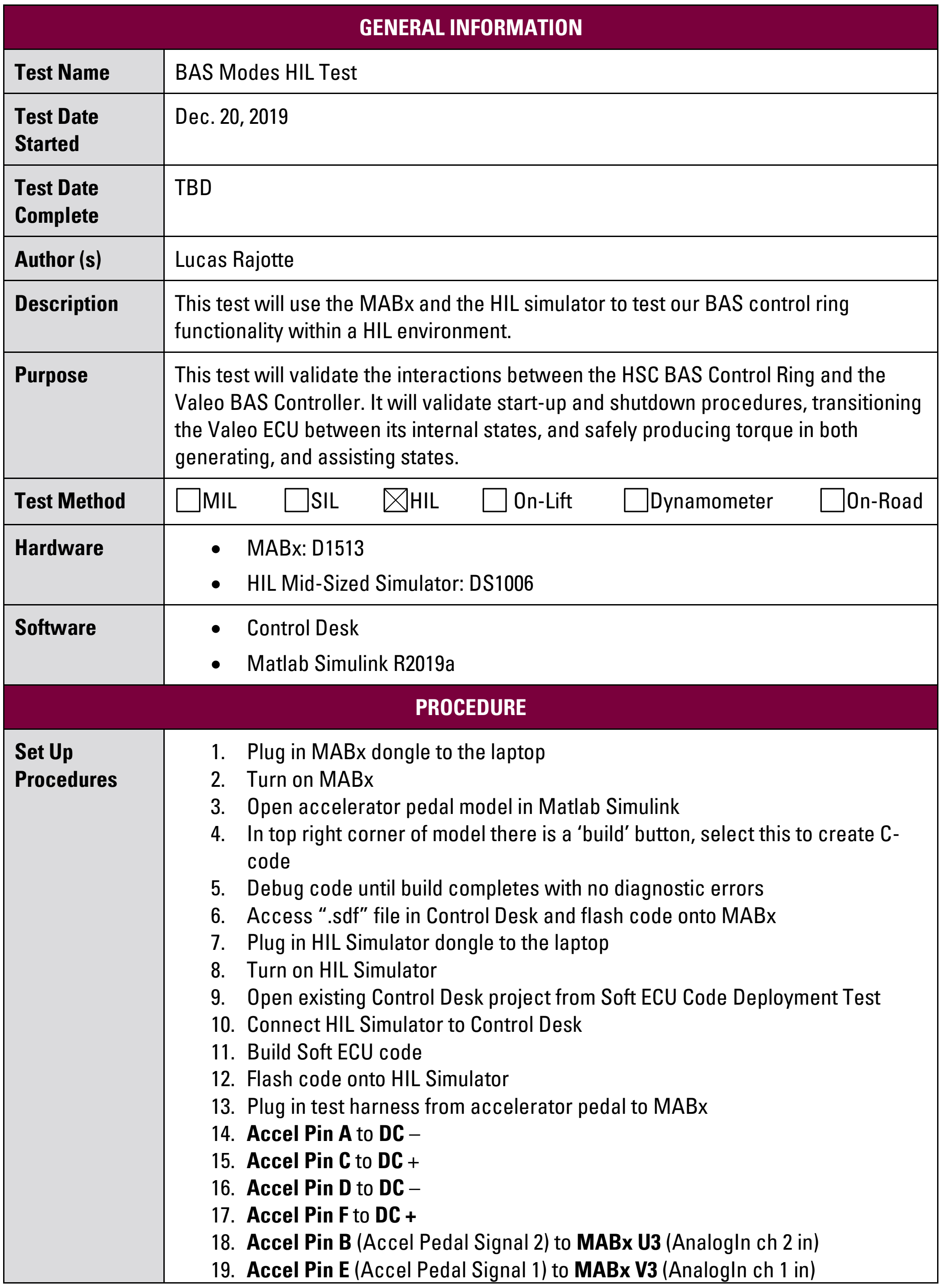
FIGURE 4.34: Verification of Requirement III.4.1.4

4.4.4 Test Plan Documentation

Before a XIL Test Case is performed, a test plan document is created. Figures 4.35 - 4.37 show an example of a Test Plan Document created by the EcoCAR PCM Team, under supervision of the author. This test plan was selected as it was extensive and demonstrates the verification of multiple requirements of a given module. Each test plan document must include the name of the test, the start and end date, the author of the document, and a general description and purpose for the test. A description of all the hardware and software must also be included, as well as a full set-up procedure, and exact instructions on how to perform the test. This ensures that the author of the test does not necessarily need to perform it and allows multiple PCM Team members to gain experience with using the various XIL environments. To update the status of a test case, a section of the test plan has been reserved in which all the major functions and requirements that need validation are included. Each requirement has a general description as well as a checkbox which indicates whether it has been validated through testing or not. As different requirements are complete, the test plan document is updated periodically. Once all requirements have been met, the PCM Lead reviews the test plan document and approves the final version. This document is exported to a pdf and saved with a unique identification number permanently on the team's secure server.

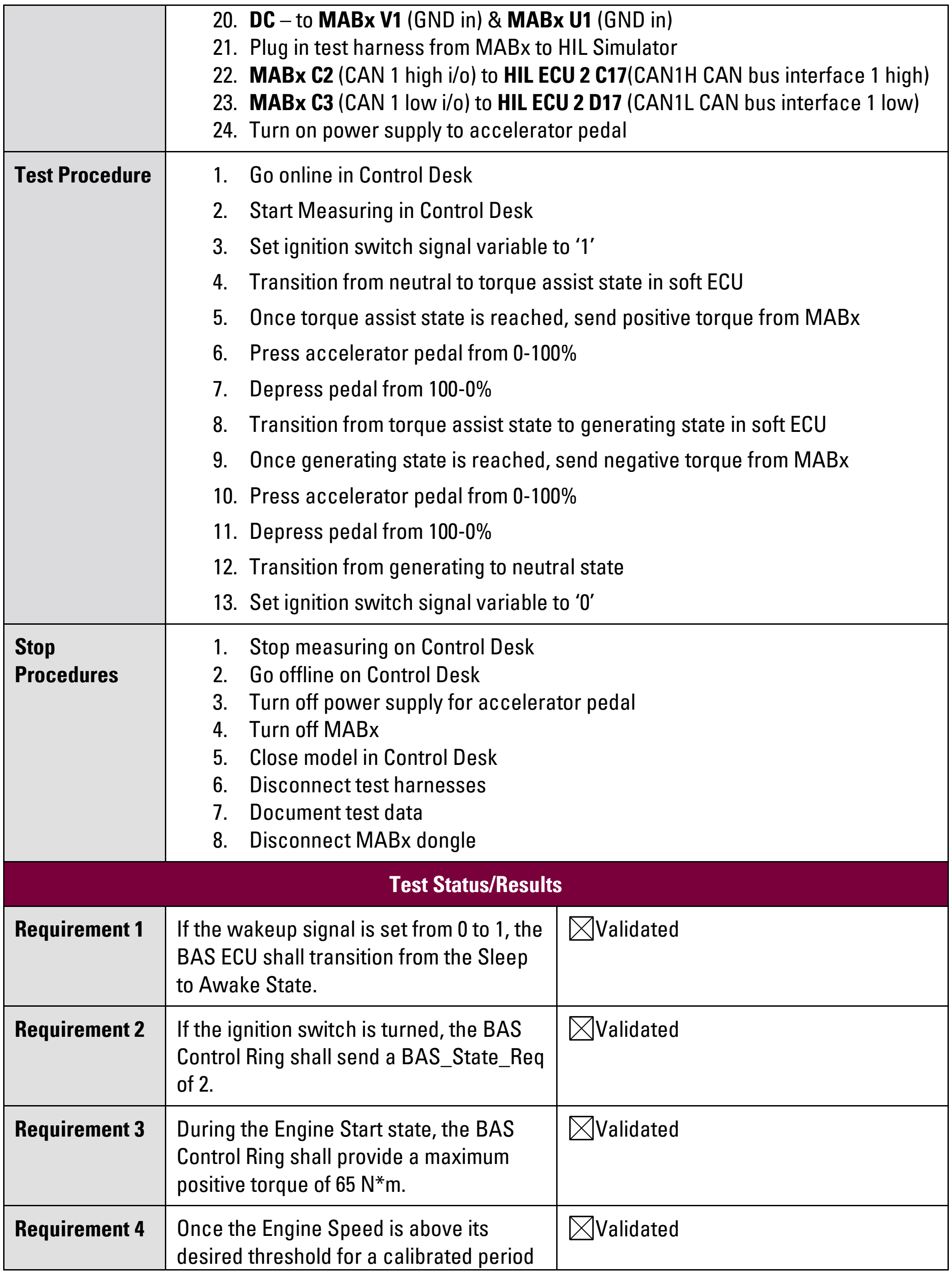
4.5 Summary

Chapter 4 has demonstrated the four major steps of the model development process followed during the design of the EcoCAR Controls System. Requirements for the system are first defined, starting from the VTS and competition rules, and refined until the software implementation level. These requirements are then used to create a serial data network and functional supervisory controller architecture. The controller architecture in combination with the software requirements are used to derive the



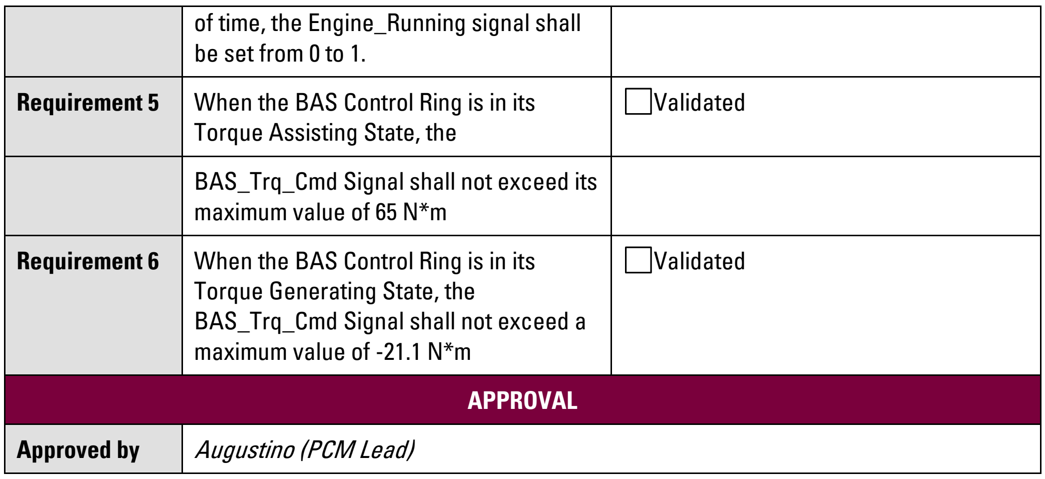
GENERAL INFORMATION	
Test Name	BAS Modes HIL Test
Test Date Started	Dec. 20, 2019
Test Date Complete	TBD
Author (s)	Lucas Rajotte
Description	This test will use the MABx and the HIL simulator to test our BAS control ring functionality within a HIL environment.
Purpose	This test will validate the interactions between the HSC BAS Control Ring and the Valeo BAS Controller. It will validate start-up and shutdown procedures, transitioning the Valeo ECU between its internal states, and safely producing torque in both generating, and assisting states.
Test Method	☐ MIL ☐ SIL [✓ HIL] ☐ On-Lift ☐ Dynamometer ☐ On-Road
Hardware	• MABx: D1513 • HIL Mid-Sized Simulator: DS1006
Software	• Control Desk • Matlab Simulink R2019a
PROCEDURE	
Set Up Procedures	1. Plug in MABx dongle to the laptop 2. Turn on MABx 3. Open accelerator pedal model in Matlab Simulink 4. In top right corner of model there is a 'build' button, select this to create C-code 5. Debug code until build completes with no diagnostic errors 6. Access ".sdf" file in Control Desk and flash code onto MABx 7. Plug in HIL Simulator dongle to the laptop 8. Turn on HIL Simulator 9. Open existing Control Desk project from Soft ECU Code Deployment Test 10. Connect HIL Simulator to Control Desk 11. Build Soft ECU code 12. Flash code onto HIL Simulator 13. Plug in test harness from accelerator pedal to MABx 14. **Accel Pin A** to **DC** – 15. **Accel Pin C** to **DC** + 16. **Accel Pin D** to **DC** – 17. **Accel Pin F** to **DC +** 18. **Accel Pin B** (Accel Pedal Signal 2) to **MABx U3** (AnalogIn ch 2 in) 19. **Accel Pin E** (Accel Pedal Signal 1) to **MABx V3** (AnalogIn ch 1 in)

FIGURE 4.35: HIL Test Plan Part 1



	20. **DC** – to **MABx V1** (GND in) & **MABx U1** (GND in) 21. Plug in test harness from MABx to HIL Simulator 22. **MABx C2** (CAN 1 high i/o) to **HIL ECU 2 C17**(CAN1H CAN bus interface 1 high) 23. **MABx C3** (CAN 1 low i/o) to **HIL ECU 2 D17** (CAN1L CAN bus interface 1 low) 24. Turn on power supply to accelerator pedal	
Test Procedure	1. Go online in Control Desk 2. Start Measuring in Control Desk 3. Set ignition switch signal variable to '1' 4. Transition from neutral to torque assist state in soft ECU 5. Once torque assist state is reached, send positive torque from MABx 6. Press accelerator pedal from 0-100% 7. Depress pedal from 100-0% 8. Transition from torque assist state to generating state in soft ECU 9. Once generating state is reached, send negative torque from MABx 10. Press accelerator pedal from 0-100% 11. Depress pedal from 100-0% 12. Transition from generating to neutral state 13. Set ignition switch signal variable to '0'	
Stop Procedures	1. Stop measuring on Control Desk 2. Go offline on Control Desk 3. Turn off power supply for accelerator pedal 4. Turn off MABx 5. Close model in Control Desk 6. Disconnect test harnesses 7. Document test data 8. Disconnect MABx dongle	
Test Status/Results		
Requirement 1	If the wakeup signal is set from 0 to 1, the BAS ECU shall transition from the Sleep to Awake State.	☒Validated
Requirement 2	If the ignition switch is turned, the BAS Control Ring shall send a BAS_State_Req of 2.	☒Validated
Requirement 3	During the Engine Start state, the BAS Control Ring shall provide a maximum positive torque of 65 N*m.	☒Validated
Requirement 4	Once the Engine Speed is above its desired threshold for a calibrated period	☒Validated

FIGURE 4.36: HIL Test Plan Part 2



	of time, the Engine_Running signal shall be set from 0 to 1.	
Requirement 5	When the BAS Control Ring is in its Torque Assisting State, the	☐Validated
	BAS_Trq_Cmd Signal shall not exceed its maximum value of 65 N*m	
Requirement 6	When the BAS Control Ring is in its Torque Generating State, the BAS_Trq_Cmd Signal shall not exceed a maximum value of -21.1 N*m	☐Validated
APPROVAL		
Approved by	*Augustino (PCM Lead)*	

FIGURE 4.37: HIL Test Plan Part 3

software interface definitions for each module, and the system has been broken down in a modular fashion as suggested within the competition design process as well as the Simulink and MATLAB documentation. Software modules with defined functionality and interfaces are developed and tested with the aid of Simulink Requirements and a Test Plan validation system. Finally, the software modules continue to be periodically updated and refined with the use of a documentation and version control system.

Chapter 5

Application of the Simulink Module Tool

The modular guidelines and functionalities provided by the Simulink Module Tool outlined in Chapter 2 have been applied to the EcoCAR HSC Model. Section 5.1 gives a summary of the guidelines used throughout the model modification process. The initial system decomposition analysis is shown in Section 5.2, where all module secrets and likely system changes are identified and organized into a modular structure. Section 5.3 defines the resulting changes in system decomposition using a comparison to the original structure. The remaining sections (5.4 - 5.8) describe in detail the model changes made to each model component and shows the full extent of model work done to increase the modularity of the HSC. The modification, addition, and/or removal of modules are explained using the secrets they hold as well as the effect that likely changes to the system have.

5.1 Review of Simulink Modular Guidelines

The guidelines for modularization defined by Jaskolka et. al in [36], and summarized in Section 2.5.3, were applied throughout the HSC model discussed in this chapter. The creation of both local and exported Simulink functions was prevalent throughout the HSC system, as well as the use of local functions within Stateflow Chart objects. As a result, Guidelines 1 – 6 were applied extensively throughout the HSC Model, using the Simulink Module Tool to aid in implementation. The HSC system did not have any instances of exported Stateflow functions; the Stateflow charts were used primarily for individual component control logic, not providing functionality for other modules to access. The application of Guideline 7 was therefore not required, but still presents important guidance in the event that a future change would require exported Stateflow functions.

5.2 Overview of Likely Changes and System Secrets

To provide motivation for a new system decomposition, a list of likely model changes was developed, described in detail in Appendix B, Tables A2.1 and A2.2. Ideally, each likely change would only affect one module within the new HSC system, or even one function. To ensure that each major likely change identified was contained within at least one module, the modules affected by the change were added to a second column in Appendix B. For example, changes 1 - 3 in Table A2.1 all involve changes to the Data Interface and can be handled by a newly created Data Interface Module. Changes to the Propulsion Control Strategy, such as the torque splitting method could be handled within the existing Propulsion Control Strategy Module. The remainder of this change list will be referenced throughout Chapter 5, providing motivation for the various changes in decomposition. This list of likely changes is also used in Section 6.2.1 to compare change propagation throughout the HSC system before and after application of the Simulink Module Tool.

5.2.1 System Module Secrets

The complete list of Simulink Modules as well as the secrets they contain have been summarized in Tables 5.1 and 5.2. The secrets each module contains were also classified in terms of the specific changes that were likely to occur throughout the model development cycle. The changes that would be required are classified into three types: hardware, software design (a change in the data structure), and behavioral (changes to functional requirements) [35, page 12], [37]. The list of secrets was created based on knowledge of the existing system as well as analyzing the likely changes described in Appendix B. The PCSM is affected by likely change 16 (change in torque splitting strategy) in Table A2.1, which would propagate to one or more of the corresponding module secrets defined in Table 5.1. The system modules have been organized based on shared groupings of secrets and algorithms, and formed the initial structure needed to design the system decomposition in Section 5.3.

5.3 System Decomposition

Prior to applying modularity guidelines to the HSC model, it is necessary to perform an analysis of the existing model system decomposition. The original system decomposition is described in Section 5.3.1, where the data flow process was the primarily determinant of module organization. The system decomposition for a modular Simulink Model should be designed primarily by grouping algorithms based on both the secrets they hold and the model sample time [35, page 162]. Section 5.3.2 utilizes the system module secrets as well as the guidelines from Section 5.1 to describe the modified system decomposition.

5.3.1 Original Model Decomposition

The system decomposition of the original EcoCAR HSC model is summarized in Figures 5.1 - 5.4 below. For all figures, gray represents model references, orange represents virtual subsystems, green represents Stateflow Charts, and blue represents Simulink Functions. The model decomposition follows directly from the Hybrid Supervisory Controller Diagram and software design discussed in Sections 4.2 and 4.3, respectively. The models and subsystems are primarily grouped together based on their role and position in the data flow process, resulting in a flat and unorganized system hierarchy. The majority of algorithms are located in subsystems within the HSC root model, resulting in likely changes having potentially large propagation throughout the system.

TABLE 5.1: System Module Secrets

Module	Secret	Module Type
Data Interface Module	HSC I/O signal conversions, and data flow algorithm in/out of HSC	Behaviour, Hardware
Input signal subsystem	Input Data interface to HSC Signal conversion	Behaviour, Hardware
Output signal subsystem	HSC signal to Output Data interface	Behaviour, Hardware
Pedal Processing Module	Algorithm for Handling Pedal Inputs and Outputs.	Behaviour, Hardware
Voltage to Pedal Conversion	Algorithm for conversion of a sensor voltage to pedal signal.	Behaviour, Hardware
Pedal to Voltage Conversion	Algorithm for conversion of a pedal signal to sensor voltage	Behaviour, Hardware
CAVs Module	CAVs Control Algorithm, conversions	Behaviour
ACC Pedal Arbitration	Algorithm for arbitration between CAVs signal and drivers pedal command.	Behaviour
Steer Control	Algorithm for processing CAVS lateral commands.	Behaviour
Energy Management Module	Algorithms for vehicle power operation, and energy system control	Behaviour
Propulsion Control Strategy Module	Energy Control Algorithm, hardware limits, conversions	Behaviour
Energy Management Stateflow Logic	Energy Management Control Strategy, calculation of Torque and Power Commands.	Behaviour
Range and Curvature Calculation	Algorithm for calculating control strategy parameters.	Behaviour
Power Torque Conversion	Algorithm for determining component torque requests based on power	Behaviour
Power Moding Module	Power Control Algorithm, conversions	Behaviour
Power Moding Stateflow Logic	Power Moding operations, control algorithms	Behaviour
Regen Power Module	Regen Control Algorithm, hardware limits, conversions	Behaviour
Pressure Regen Calculation	Algorithm for conversion from pressure to regen cmd	Behaviour
Regen Torque Determination	Algorithm for calculating the regenerative torque command for the Motor.	Behaviour
Regen Torque Arbitration	Algorithm for torque arbitration when receiving traction and regenerative torque requests.	Behaviour
Propulsion System Control Module	Propulsion component control logic, Vehicle controls architecture	Behaviour
Motor Control Module	Motor Control Algorithm, hardware limits, conversions	Behaviour
Motor Stateflow logic	Component operations, control algorithms	Behaviour

TABLE 5.2: System Module Secrets, continued

Module	Secret	Module Type
Battery System Manager Module	Battery Control Algorithm	Behaviour
Battery Stateflow logic	Component operations, control algorithms	Behaviour
BAS Control Module	BAS Control Algorithm	Behaviour
BAS Stateflow Logic	Component operations, control algorithms	Behaviour
Clutch Module	Clutch Control Algorithm, conversions	Behaviour
Speed Matching Subsystem	Algorithm for matching clutch speed to motor shaft speed before enabling the clutch.	Behaviour
Clutch Check Subsystem	Algorithm for final torque arbitration based on clutch position status.	Behaviour
Fault Handler Module	Algorithms for Fault Detection and Mitigation	Behaviour
Motor Fault Detection Subsystem	Fault codes, algorithm to decode fault messages	Behaviour
Motor Fault Mitigation Subsystem	Fault conditions, algorithm to determine fault mitigation strategy.	Behaviour
Hardware Limit Control Module	Algorithms for calculating and checking power limits	Behaviour
Wheel Torque Calculation	Algorithm for determining total vehicle wheel torque.	Behaviour
Power Limit Calculation	Algorithm for calculation power limits of various components	Behaviour
Power Limit Check	Algorithms for checking component outputs meet calculation power limits.	Behaviour
Engine Hardware Module	Engine Hardware limits and parameters	Hardware
Engine Axle Torque Calculation	Axle Torque Hardware Limits, Engine Pedal Map	Hardware
Max Engine Wheel Torque Calculation	Wheel Torque Hardware Limits	Hardware
Max Engine Torque Calculation	Engine Torque Hardware Limits, algorithm for calculation.	Hardware
Engine Torque Pedal Output Calculation	Engine Pedal map for conversion from Axle Torque to pedal percentage.	Hardware
Battery Hardware Module	Battery Control Algorithm, hardware limits, conversions	Hardware
Battery Limit Calculation	Hardware limits, algorithm for determining charge/discharge limits.	Hardware
Battery Limit Check	Hardware limits, algorithm for checking limits.	Hardware
Motor Hardware Module	Motor Hardware limits, conversions	Hardware
Motor Torque Limit Check	Hardware Limits, algorithm for checking limits	Hardware
Electrical Power Estimation	Mechanical to Electrical Power Limits	Hardware
Max Motor Torque Calculation	Max Motor Torque Lookup Tables	Hardware
Transmission Hardware Module	Transmission hardware limits, conversions	Hardware

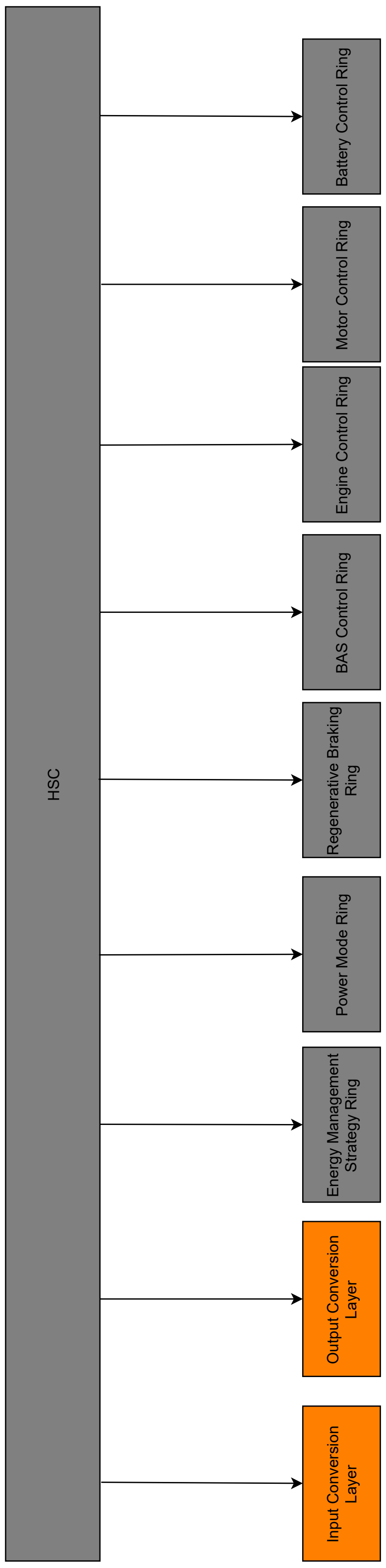


Figure 5.1: HSC root level system decomposition (original model)

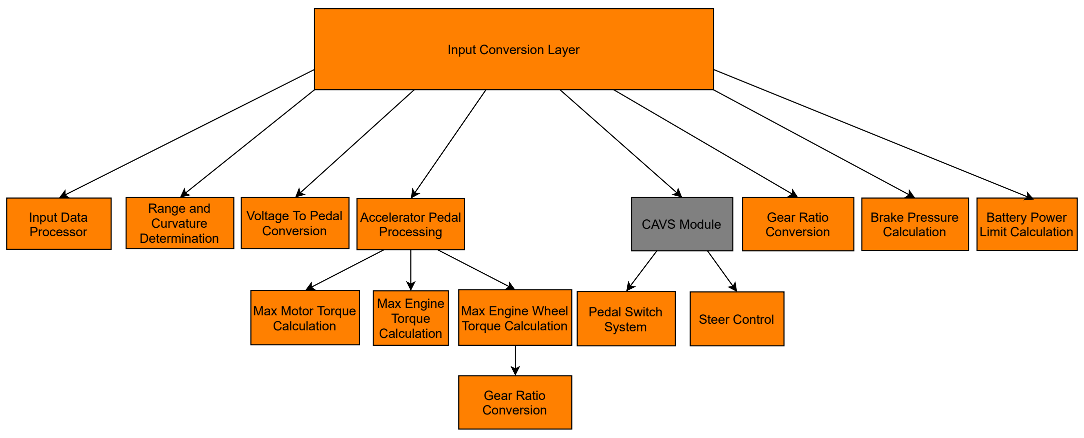

Input Conversion Layer
Input Data Processor
Range and Curvature Determination
Voltage To Pedal Conversion
Accelerator Pedal Processing
CAVS Module
Gear Ratio Conversion
Brake Pressure Calculation
Battery Power Limit Calculation
Max Motor Torque Calculation
Max Engine Torque Calculation
Max Engine Wheel Torque Calculation
Pedal Switch System
Steer Control
Gear Ratio Conversion

FIGURE 5.2: System decomposition of the Input Conversion Layer (original model)

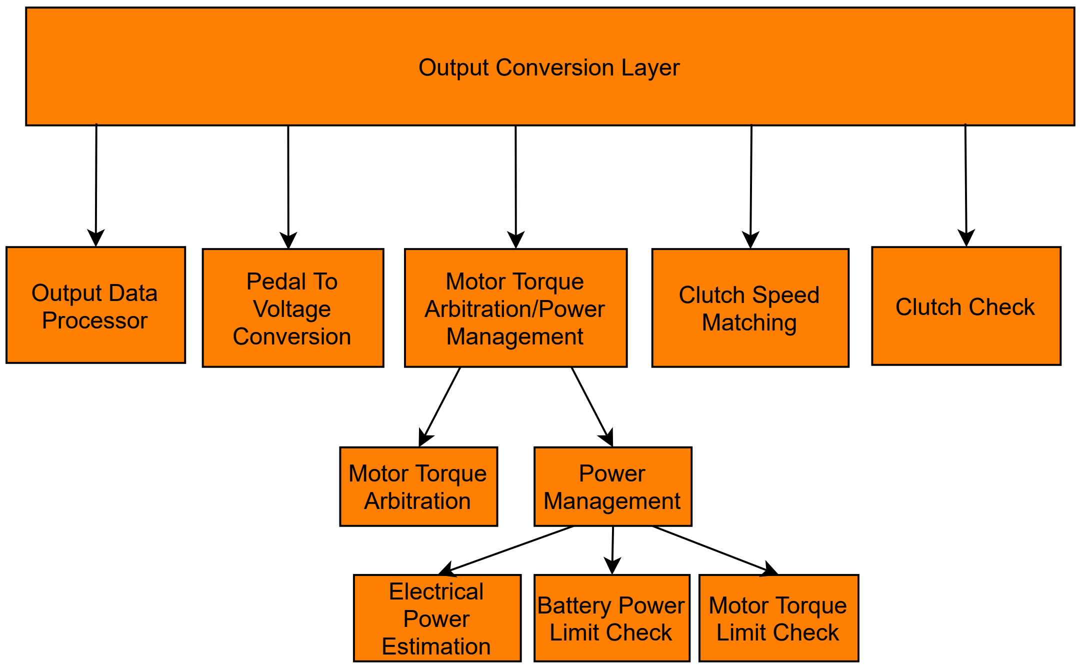


FIGURE 5.3: System decomposition of the Output conversion layer (original model)

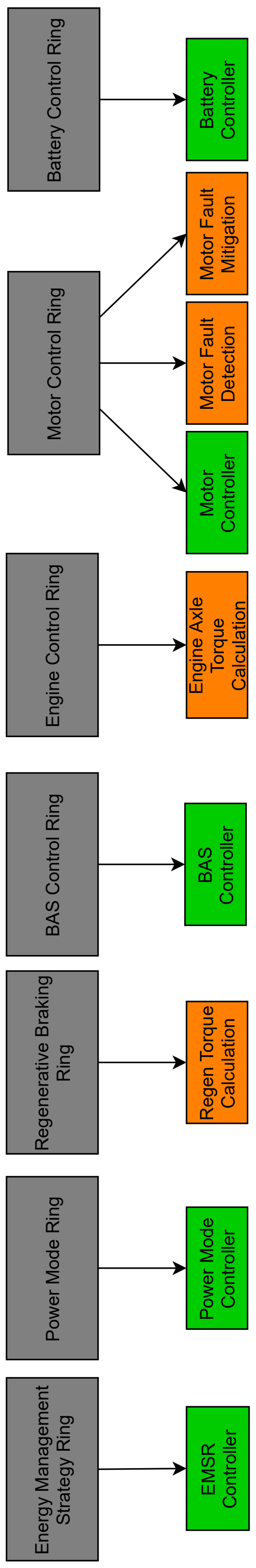


Figure 5.4: System decomposition of Operation Mode and Component Control Loops (original model)

5.3.2 Modular Decomposition

The modified system decomposition was determined based on the algorithms and secrets that each module should hold and the likely changes that may occur, following directly from Tables 5.1 - 5.2 and Appendix B. A graphical representation of the new system decomposition can be seen in Figures 5.5 – 5.10. The guidelines described in Section 5.1 were applied to the original system decomposition, with modules defined based on a grouping of related secrets and algorithms. Most changes occurred in the Input and Output conversion layers, where the various subsystems were organized into both existing modules within the original HSC, as well as newly created ones.

The modularization of the HSC system resulted in the creation of additional modules and corresponding Simulink models, outlined in Table 5.3. Separating hardware secrets from control algorithms motivated the creation of 3 additional hardware modules (BHM, MHM, THM), and 5 additional behaviour modules (DIM, PPM, CCM, FHM, HLCM). Likely changes to the HSC such as modifications to the vehicle architecture and overall energy management system provided the motivation to create 2 final behavioural modules, namely the EMM and PSCM. The full extent of model changes needed to implement the new system decomposition are described in sections 5.4 - 5.8.

5.4 Introduction to Model Changes

The following sections describe the changes that were made to the EcoCAR HSC Model during the modularization process. To draw parity to the data flow process outlined in Section 4.2, Sections 5.5 - 5.8 have been broken into the input conversion layer, operation mode loop, component control loop, and output conversion layer, respectively. For all changes mentioned below, the Simulink Module Tool was utilized, both for creating Simulink Modules as well as verifying and testing modularity guidelines. Section 5.4.1 describes limitations for modularization within the Simulink Environment, while section 5.4.2 discusses modification of the data storage system.

5.4.1 Limitations for Simulink Modularization

A model reference cannot export functions as well as have a fixed sample time; meaning all modules requiring Stateflow charts with specified sample times could not also contain exported Simulink functions. This limitation had a significant impact on the system decomposition, an example being the inability for the Hardware Limit Control Module to be located within the Energy Management Module (see Figure 5.7 for reference). Both the PCSM and PMM contain Stateflow charts which require a specified sample time, thus restricting the ability of the EMM to export the required hardware limit functions. It was deemed a better design decision to encapsulate the exported hardware limit functions in a separate module (HLCM) and keep the Stateflow Chart functions within the EMM. Modules cannot export functions in both the root model and another model reference. For example, if the BSMM contains Simulink functions which are called in the root level, it cannot export functions to any model reference within the HSC. Breaking this guideline results in a MATLAB internal error which prevents simulation and requires an application restart. This

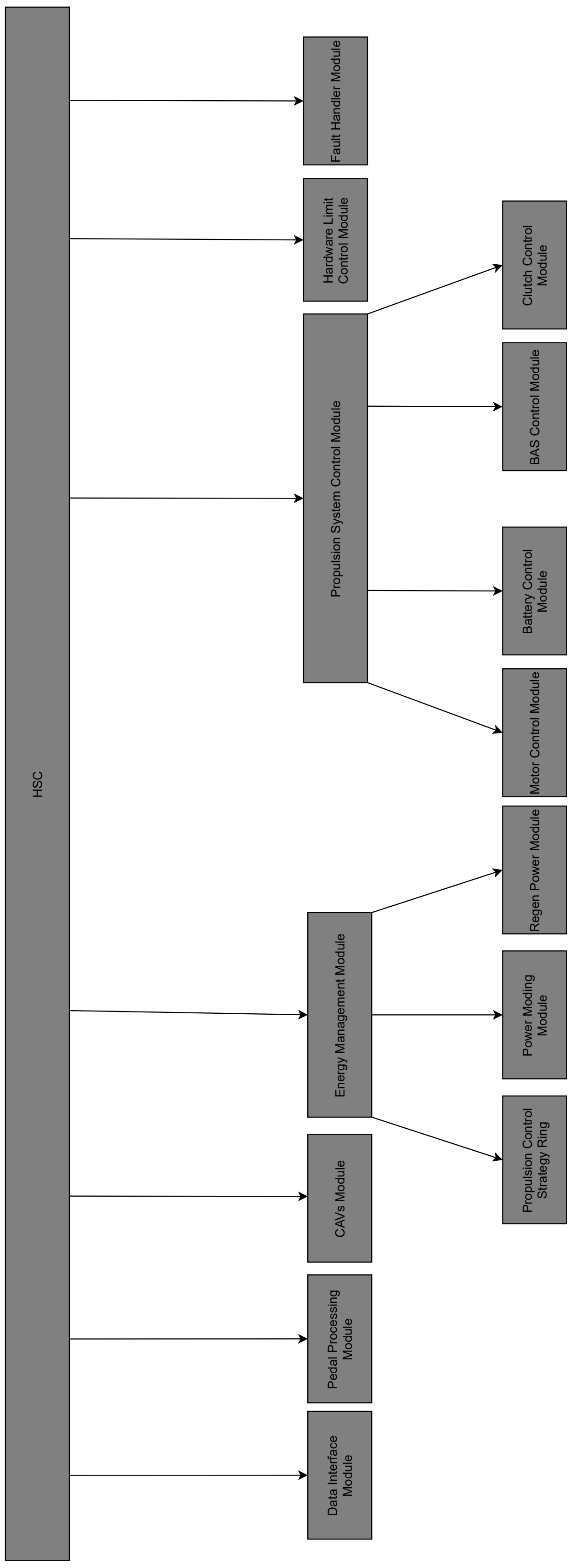


FIGURE 5.5: HSC root level system decomposition (modular model)

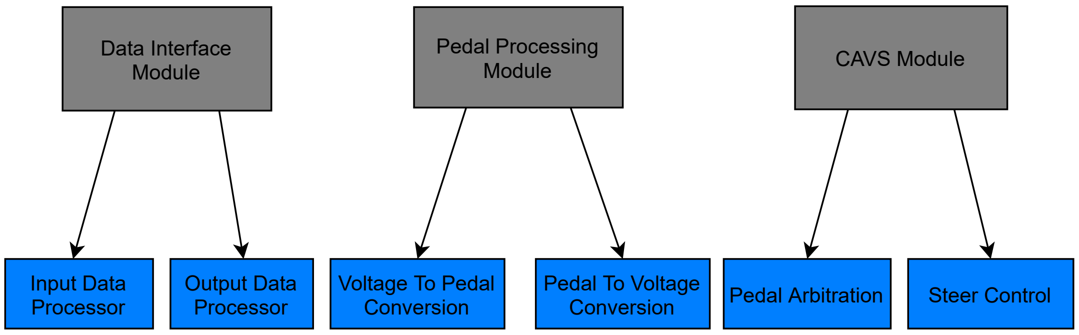


FIGURE 5.6: System decomposition for DIM, PPM, CAVM Models (modular model)

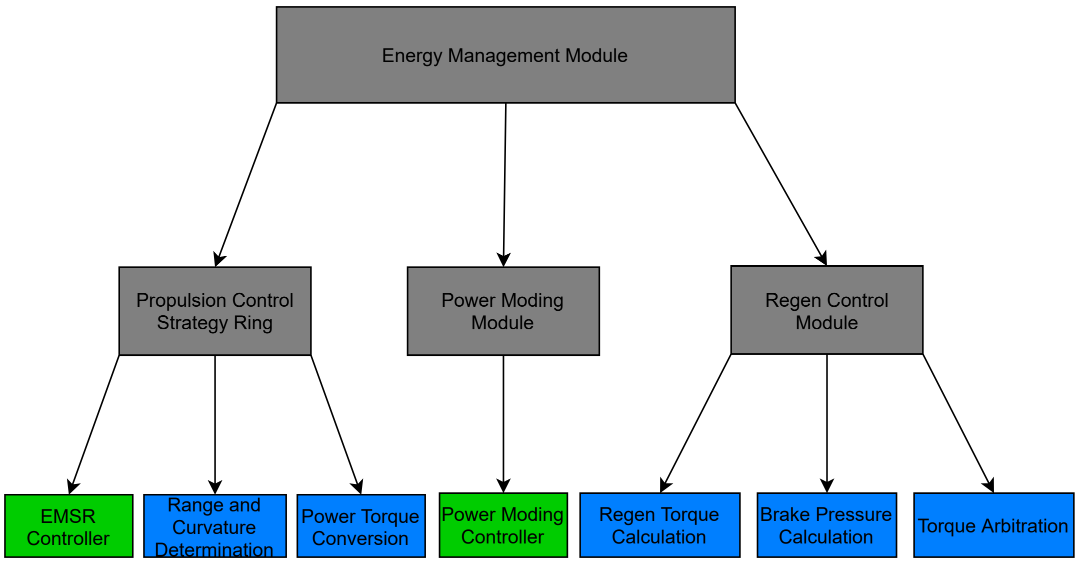


FIGURE 5.7: System Decomposition of Energy Management Module (modular model)

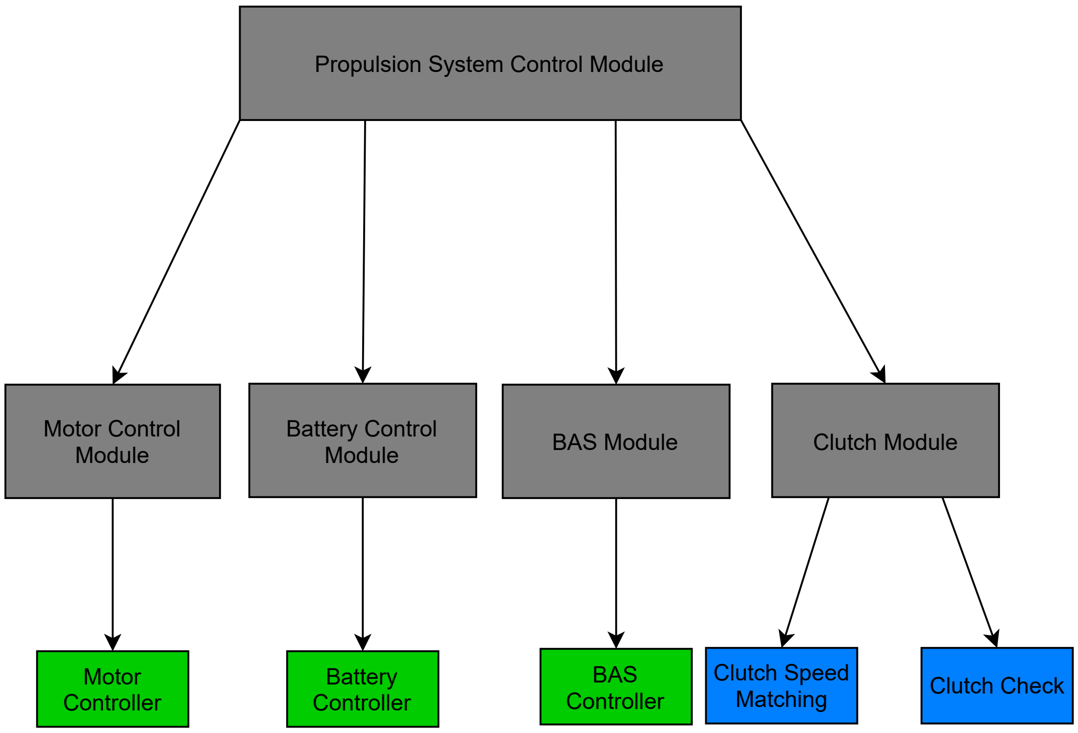


FIGURE 5.8: System decomposition for Propulsion System Control Module (modular model)

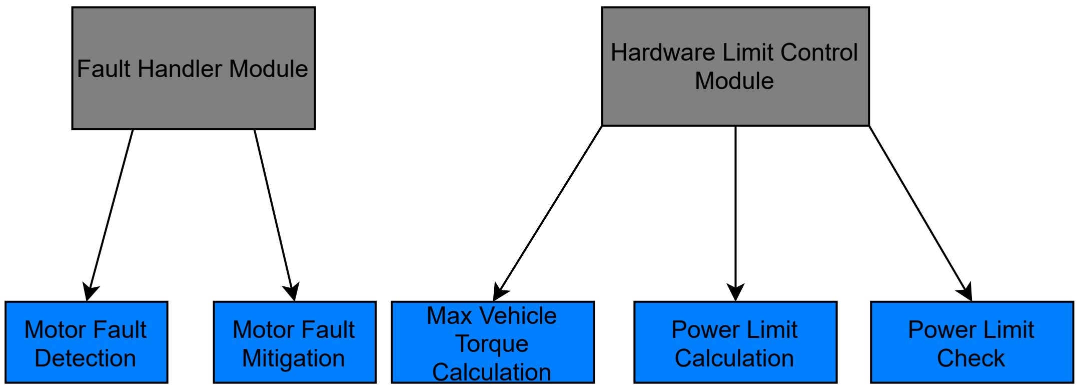


FIGURE 5.9: System decomposition of Fault Handler and Hardware Limit Control Modules (modular model)

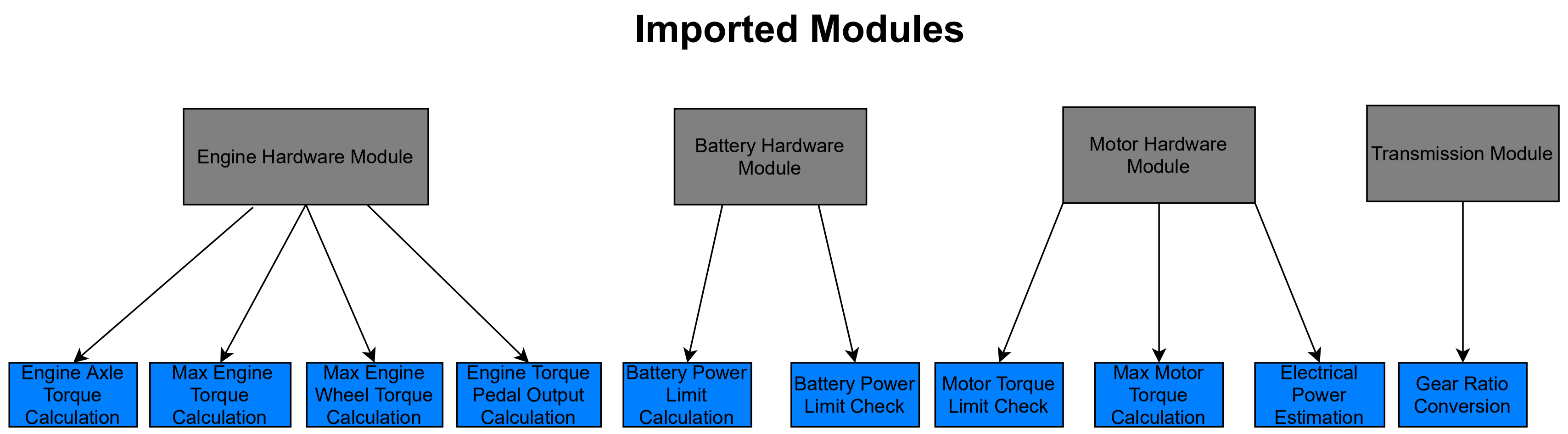

Imported Modules
Engine Hardware Module
Battery Hardware Module
Motor Hardware Module
Transmission Module
Engine Axle Torque Calculation
Max Engine Torque Calculation
Max Engine Wheel Torque Calculation
Engine Torque Pedal Output Calculation
Battery Power Limit Calculation
Battery Power Limit Check
Motor Torque Limit Check
Max Motor Torque Calculation
Electrical Power Estimation
Gear Ratio Conversion

FIGURE 5.10: System decomposition of important hardware Modules (modular model)

Table 5.3: Module structure changes, showing added, modified, and removed models.

Module	Change
Data Interface Module	Added
Pedal Processing Module	Added
CAVs Module	Modified
Energy Management Module	Added
Propulsion Control Strategy Module	Modified
Power Moding Module	Modified
Regen Power Module	Modified
Propulsion System Control Module	Added
Motor Control Module	Modified
Battery System Manager Module	Modified
BAS Control Module	Modified
Clutch Module	Added
Fault Handler Module	Added
Hardware Limit Control Module	Added
Engine Hardware Module	Modified
Battery Hardware Module	Added
Motor Hardware Module	Added
Transmission Hardware Module	Added
Input Conversion Subsystem	Removed
Output Conversion Subsystem	Removed

prevents the development of a module capable of being imported as well as imported other modules.

5.4.2 Data Storage Modifications

Following Guideline 4 from Section 2.5.3, the use of data storage in the base workspace and global data dictionaries was restricted as much as possible. The original Eco-CAR model had all information defined within the the *System.dd* dictionary and its referenced sub dictionaries (discussed in section 4.3.3). This data was reorganized based on which modules required read/write access, and transferred to the respective model workspaces. Data which was required by multiple modules was consolidated to a single global data dictionary, *Global.sldd*. This change in data storage is discussed further in Section 6.3, providing an analysis of the change in interface complexity.

5.5 Input Conversion Layer

The input conversion layer virtual subsystem represents a step in the data flow process and does not group together functions and algorithms sharing common secrets. Each subsystem encapsulated within the Input Conversion Layer was analyzed to determine what secrets were contained within the function and what likely changes may occur. Please refer to Tables 5.1 and 5.2 for a full overview of all functions and the secrets identified, including the resulting module organizational structure.

5.5.1 Data Interface Module

The top-level interface of the controller can be seen in Figures 5.11 and 5.12 below. The HSC root level model interface consists of 7 input and output bus signals, representing the six CAN I/O channels present in the vehicle serial data architecture, and 1 additional input and output port for all signals not utilizing CAN. A new model, the Data Interface Module, has been created to group together the input and output data processor subsystems. This new module contains the blocks necessary to interpret incoming CAN signal data and distribute it as Simulink signals to the rest of the modules. This represents a shared secret which would be affected by changes in the outer CAN interface as well as the changes to the signals that other hardware passes to the HSC (changes 1 - 3 in Table A2.1). Figure 5.13 shows the input data (highlighted in blue), and output data (highlighted in orange) Simulink functions, defined within the Data Interface Module. Function caller blocks for the input and output data processor functions can be seen in Figures 5.14 and 5.15, respectively.

5.5.2 Input Accelerator Pedal Processing

The subsystem shown in Figure 5.16 performed the conversion between two pedal sensor voltages, *PVS1* & *PVS2*, into a single Pedal Command signal (%), *AccelPdl*. The sensor voltages were first converted into individual pedal command percentages, then passed to the pedal sensor arbitrator, highlighted in orange. The resulting Pedal command signal was passed to the 2D Lookup Table, highlighted in yellow, which

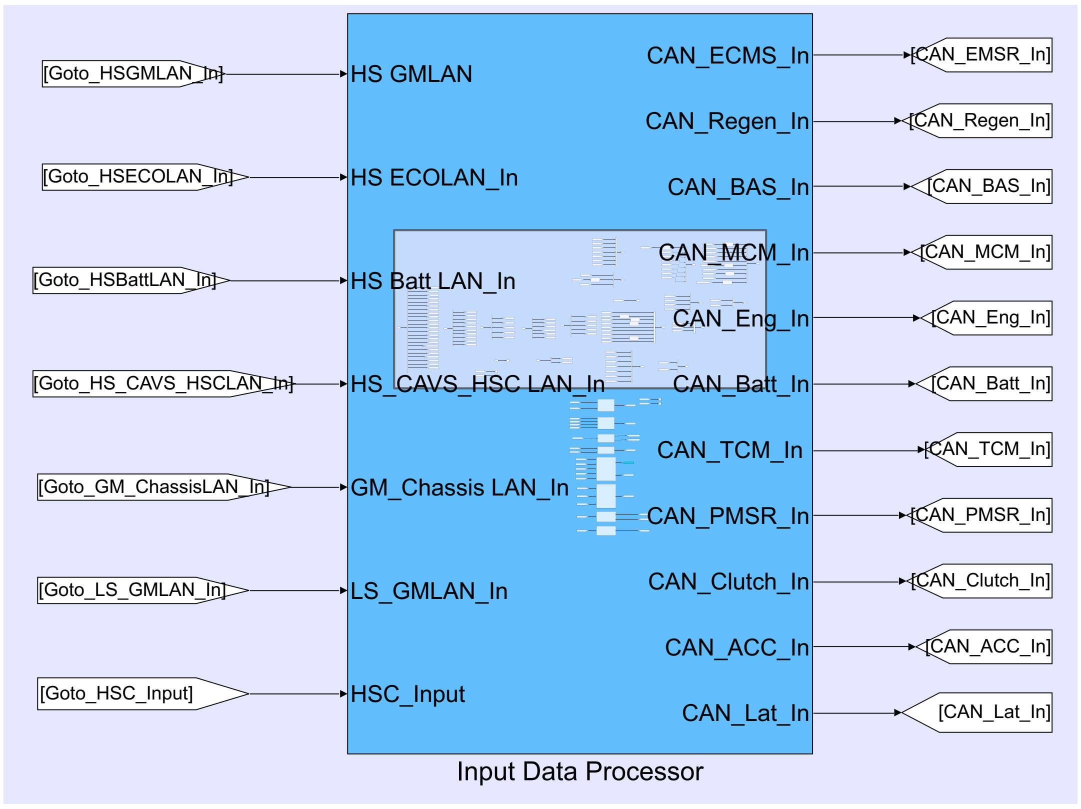


FIGURE 5.11: Input Data Processor subsystem in original model

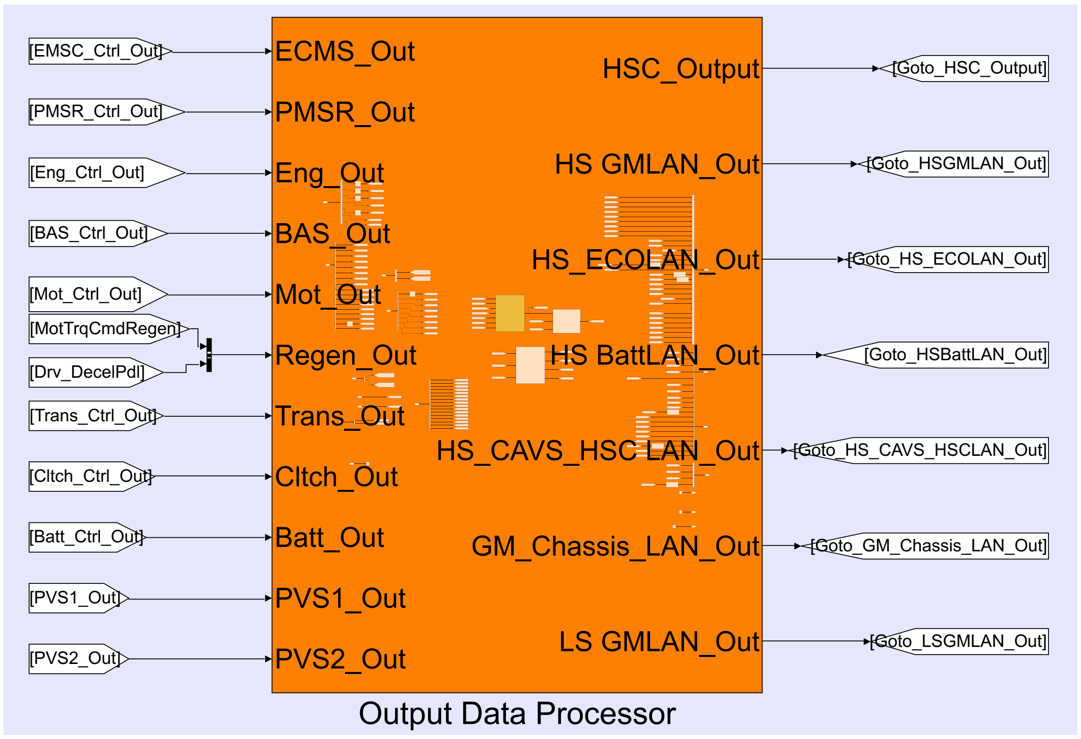


FIGURE 5.12: Output Data Processor Subsystem in original model

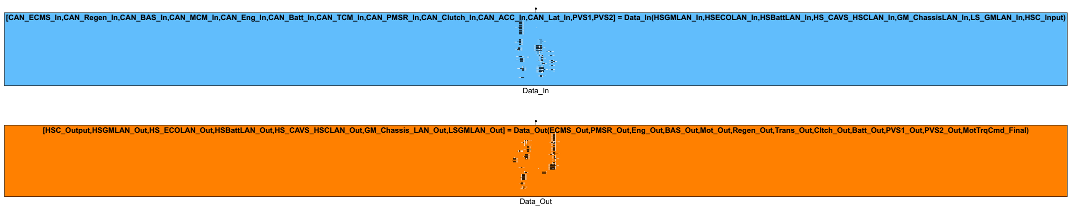

[CAN_ECMS_In,CAN_Regen_In,CAN_BAS_In,CAN_MCM_In,CAN_Eng_In,CAN_Batt_In,CAN_TCM_In,CAN_PMSR_In,CAN_Clutch_In,CAN_ACC_In,CAN_Lat_In,PVS1,PVS2] = Data_In(HSGMLAN_In,HSECOLAN_In,HSBattLAN_In,HS_CAVS_HSCLAN_In,GM_ChassisLAN_In,LS_GMLAN_In,HSC_Input)
Data_In
[HSC_Output,HSGMLAN_Out,HS_ECOLAN_Out,HSBattLAN_Out,HS_CAVS_HSCLAN_Out,GM_Chassis_LAN_Out,LSGMLAN_Out] = Data_Out(ECMS_Out,PMSR_Out,Eng_Out,BAS_Out,Mot_Out,Regen_Out,Trans_Out,Cltch_Out,Batt_Out,PVS1_Out,PVS2_Out,MotTrqCmd_Final)
Data_Out

FIGURE 5.13: Input and Output Simulink functions within Data Interface Module

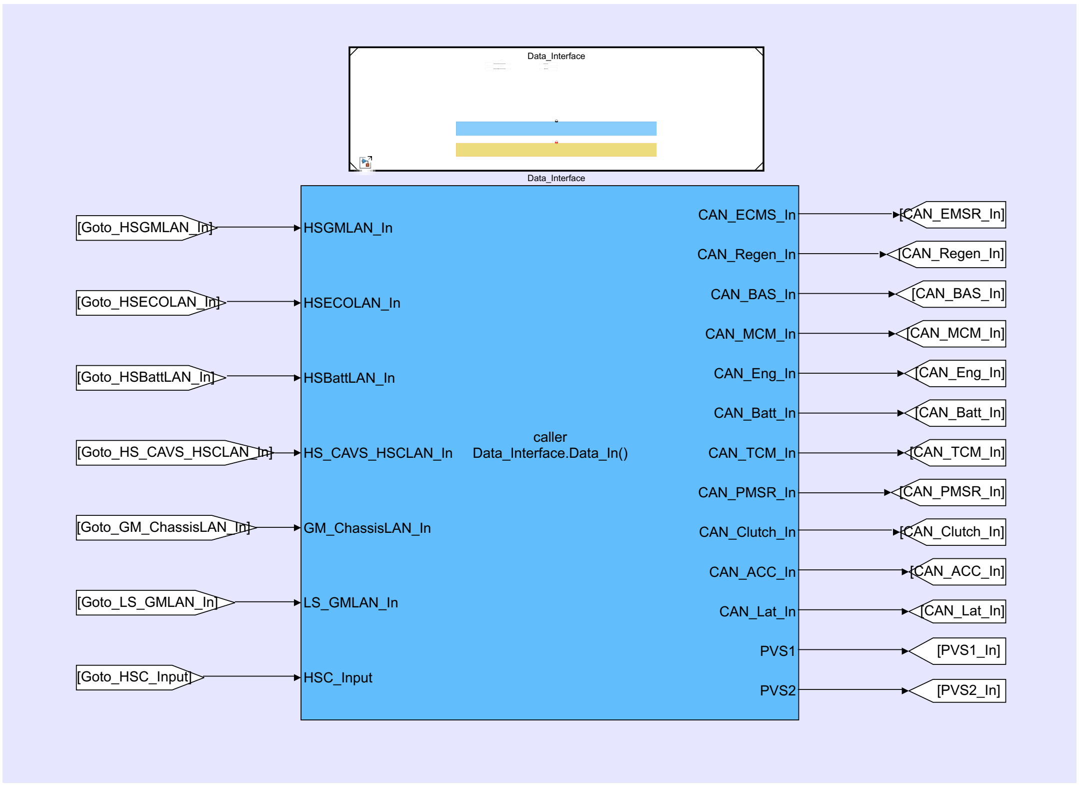


FIGURE 5.14: Data Interface Module as a model reference, with Input function caller block

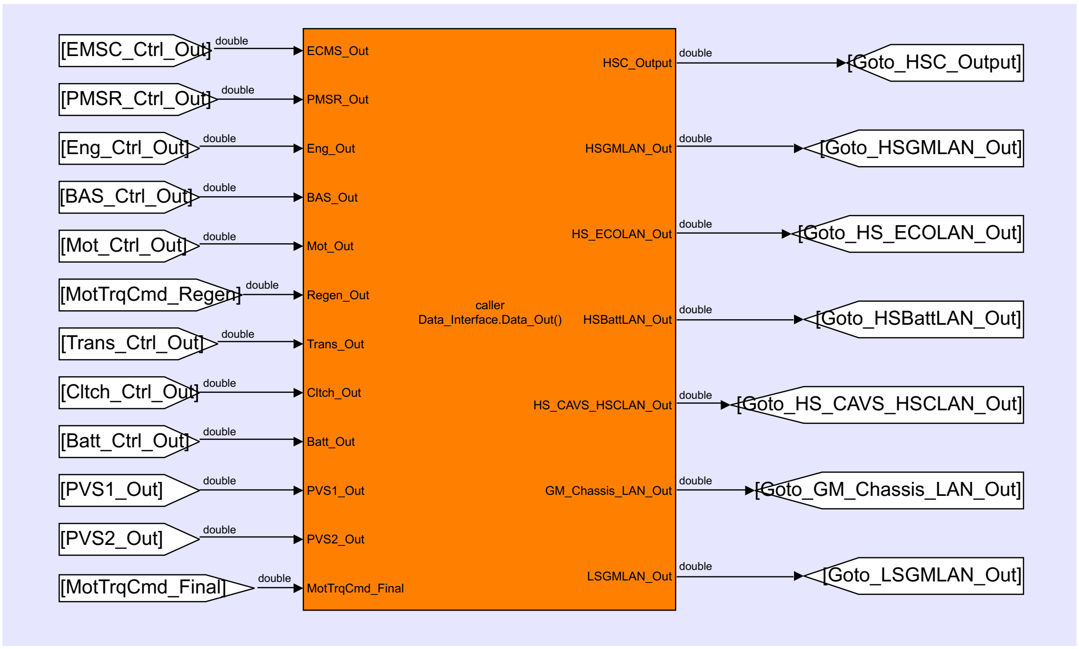


FIGURE 5.15: Output function caller block at HSC model root level

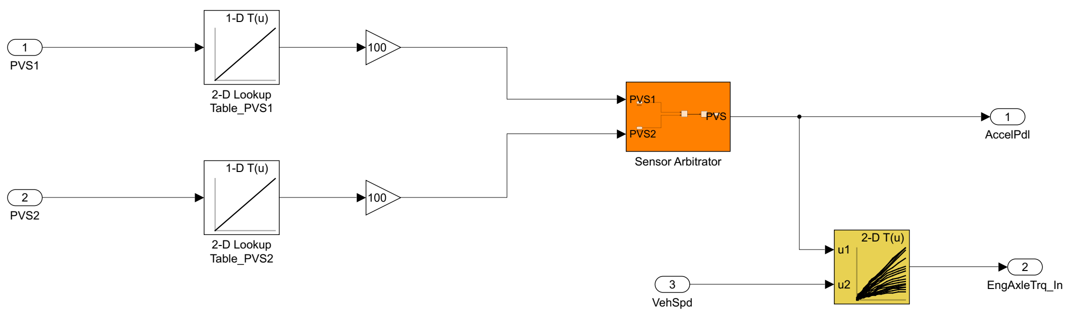


FIGURE 5.16: Internal Algorithm for Input Pedal Processing

contains the Pedal Mapping of the Vehicle Engine Control Module. The resulting output, *EngAxleTrq_In*, is passed to the Propulsion Control Strategy, and used in torque splitting operations.

The existing subsystem was converted into a scoped Simulink function, *Calc_Pedal_In()*, defined within the newly created Pedal Processing Module (PPM) (highlighted blue in Figure 5.19). Several different secrets are contained within *Calc_Pedal_In()*, and must be grouped into different modules. In particular, the Engine Pedal Mapping contains engine hardware related secrets, and was implemented as a scoped Simulink function, *Get_EngAxleTrq()*, within the EHM (Figure 5.17). The corresponding function caller block was defined within *Calc_Pedal_In()*, highlighted in yellow in Figure 5.18. The EHM was imported using a Model Reference block, allowing the set of exported engine hardware functions to be accessible within the PPM. The pedal sensor arbitration subsystem was also moved outside of *Calc_Pedal_In()*, as it contained a separate algorithm which could change independently. A local Simulink function, *Pedal_Arb()* was created, and encapsulated within a virtual subsystem at the root level of the PPM (shown highlighted orange in Figure 5.19. Finally, the corresponding input pedal processing function caller was defined at the HSC level, within the Input Conversion Layer (Figure 5.20). Changes 4, 5 in Figure A2.1 motivated the creation of the PPM, and change 7 motivated the separation of engine hardware functions.

5.5.3 Range and Curvature Determination

This subsystem (shown in Figure 5.21) contains the algorithm for determining range and curvature values integral to the vehicle's propulsion control strategy, and would be affected by likely change 16 in Table A2.1. As a result, this subsystem was moved to the PCSM Model, and converted into a local Simulink function. The scope was made local by encapsulating the Simulink function within a virtual subsystem (Figure 5.22). The scope was then selected via right-clicking on the Simulink function and selecting the *Change Function Scope* property from the Simulink Module Toolbar options, then **To Scoped Local Function**, and finally **In Existing Subsystem** menu options.

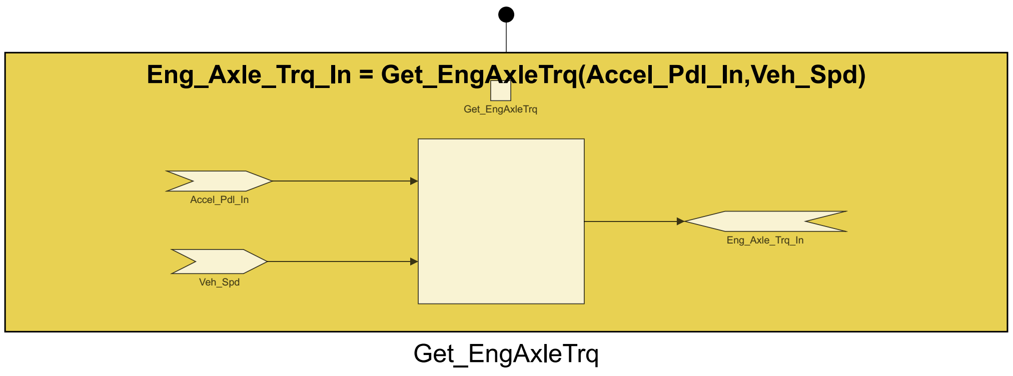


FIGURE 5.17: *Get_EngAxleTrq()* Simulink function defined within the EHM

5.5.4 Max Vehicle Torque Calculation

The vehicle torque conversion subsystem can be seen in Figure 5.23 below (highlighted in yellow), and its internal expansion can be seen in Figure 5.25. One algorithm identified was the calculation of the maximum available motor torque, which holds secrets regarding the motor hardware limits (highlighted in blue). This subsystem was moved into the Motor Hardware Module, converted into a scoped Simulink function (*Calc_Max_MotTrq()*), and exported from the model (Figure 5.26). Another algorithm identified in Figure 5.25 was the calculation of the maximum available engine torque, which held secrets regarding the engine hardware limits (highlighted in orange). This was converted from a subsystem into a scoped Simulink function, *Calc_Max_EngTrq()*, and placed within the Engine Hardware Module (EHM) (Figure 5.27). The third algorithm was the Calculation of the max Engine wheel torque which contained secrets regarding temperature and torque breakpoints within the engine hardware (highlighted in green). The subsystem was converted into a scoped Simulink Function, *Max_EngWhlTrq()*, and placed within the EHM (Figure 5.27). The max engine wheel torque subsystem also contains an algorithm which calculates the transmission gear ratio, shown in Figure 5.24, highlighted in magenta. This functionality was deemed more suited for the Transmission Control Module and was moved out of its previous location in the Engine Hardware Module, defined as a scoped Simulink function (Figure 5.28).

The subsystem from Figure 5.23 was converted into a scoped Simulink function, *Calc_MaxVehTrq()*, and was transferred to the newly created Hardware Limit Control Module (HLCM), shown in Figure 5.29. This separated the vehicle torque limit calculation algorithm (HLCM) from the hardware limit secrets it needed to access (MHM, TCM, EHM). Each of the hardware modules providing information to the HLCM were imported using model reference blocks, allowing access to the corresponding function caller blocks located within the *Calc_MaxVehTrq()* function, shown in (Figure 5.30). Finally, the *Calc_MaxVehTrq()* function caller was placed within the Input conversion layer of the HSC model, as seen in Figure 5.31. The separation of

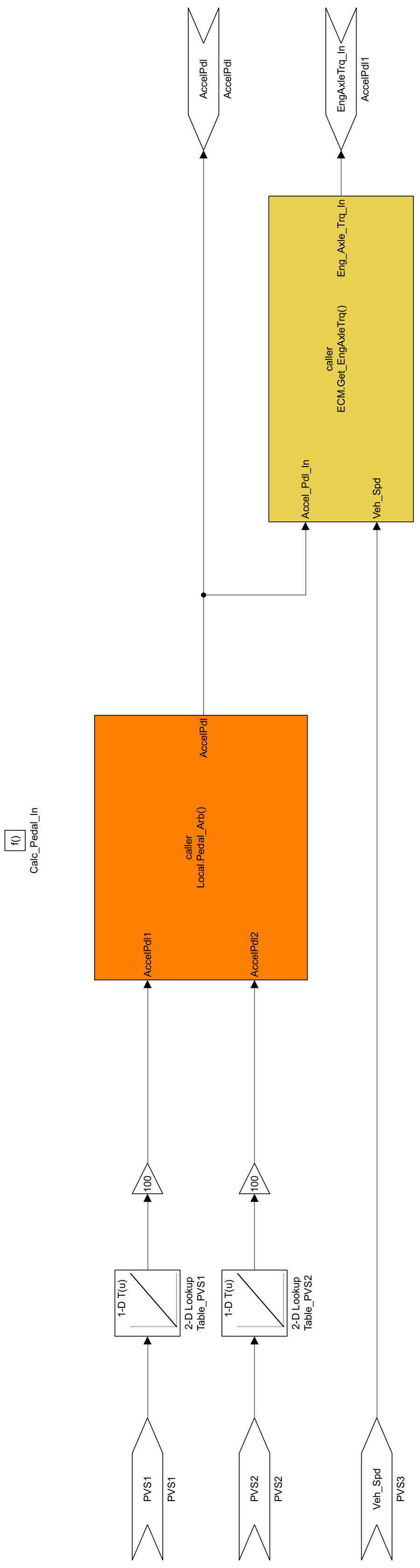


Figure 5.18: Internals of Input Pedal Processing Simulink Function

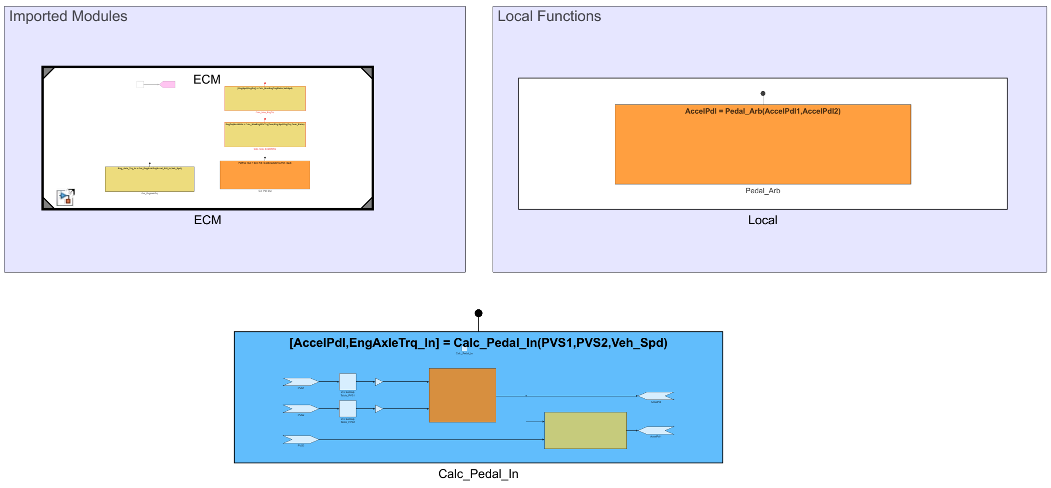

Imported Modules
ECM
ECM
Local Functions
AccelPdl = Pedal_Arb(AccelPdl1,AccelPdl2)
Pedal_Arb
Local
[AccelPdl,EngAxleTrq_In] = Calc_Pedal_In(PVS1,PVS2,Veh_Spd)
Calc_Pedal_In

FIGURE 5.19: Input Pedal Processing Simulink Function defined within the PPM

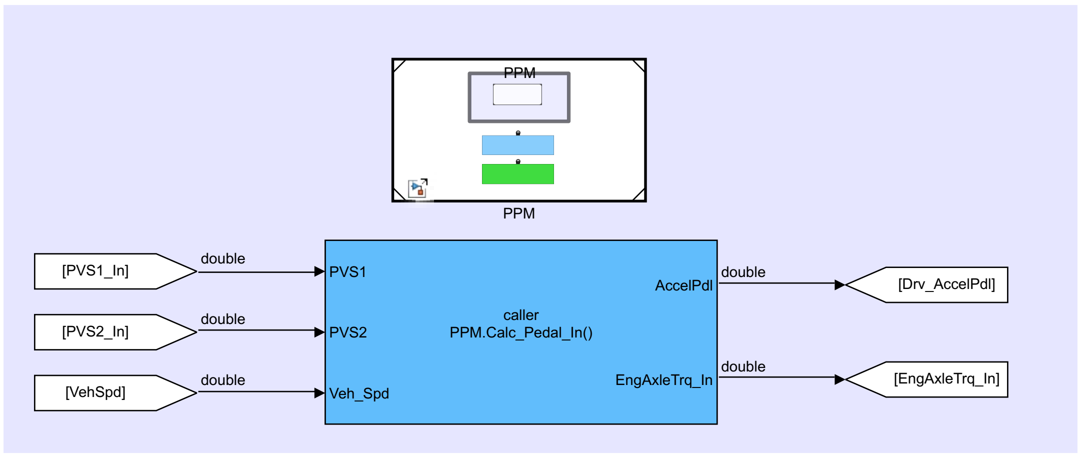


FIGURE 5.20: *Calc_Pedal_In()* function caller block placed at HSC level

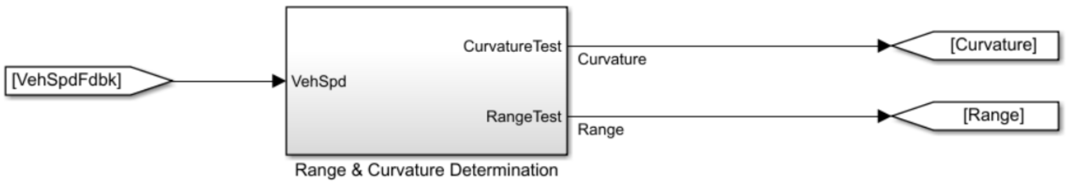


FIGURE 5.21: Range and Curvature Subsystem located in root model

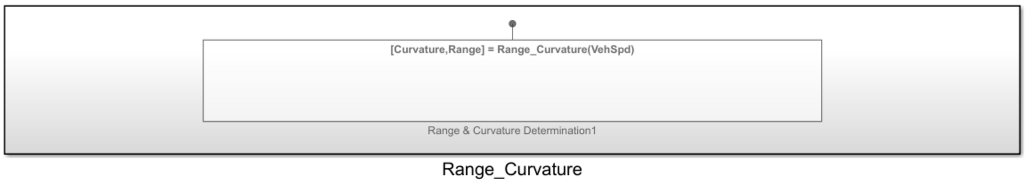

[Curvature,Range] = Range_Curvature(VehSpd)
Range & Curvature Determination1
Range_Curvature

FIGURE 5.22: Subsystem Converted to Local Simulink Function within PCSM Model

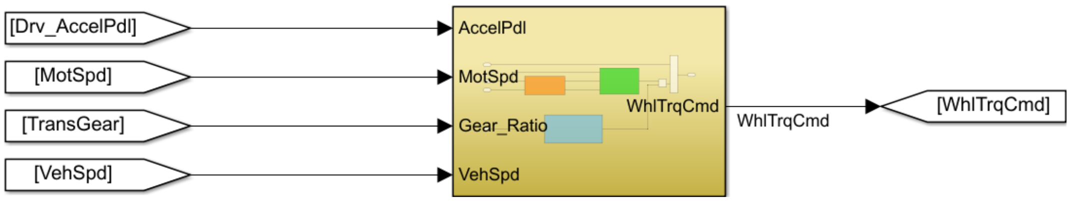


FIGURE 5.23: Original Max Vehicle Torque Conversion implemented as one virtual subsystem.

motor, engine, and transmission hardware secrets was motivated by likely changes 12, 7, and 14, respectively (Table A2.1). The creation of the new HLCM was motivated by likely change 25 in Table A2.2.

5.5.5 CAVs Module

The CAVs module contains two subsystems: the Pedal Switching system and the Steer Controller. Both algorithms are required by other modules within the HSC and were converted into exported scoped Simulink Functions within the CAVS Model (Figure 5.26). Function caller blocks were placed at the root level of the HSC model, shown in Figure 5.27. While the CAVs Module already existed within the original EcoCAR Model, its conversion into exported Simulink functions was motivated by likely changes 23 and 24 in Table A2.2.

5.5.6 Gear Ratio Conversion

The gear to ratio conversion subsystem (seen in Figure 5.34) was moved from the root model to the transmission control model, as it held secrets regarding the transmission hardware. The subsystem was then converted to a scoped Simulink function (Figure 5.28, 5.35) and exported to other modules within the HSC model.

5.5.7 Hardware Limit Calculation

The final algorithm contained within the input conversion subsystem was the Battery Management System (Figure 5.36). This virtual subsystem contains hardware secrets related to the HV Battery Charge and Discharge limits, as well as algorithms to calculate limits based off SOC (Figure 5.37). The battery hardware specific parameters

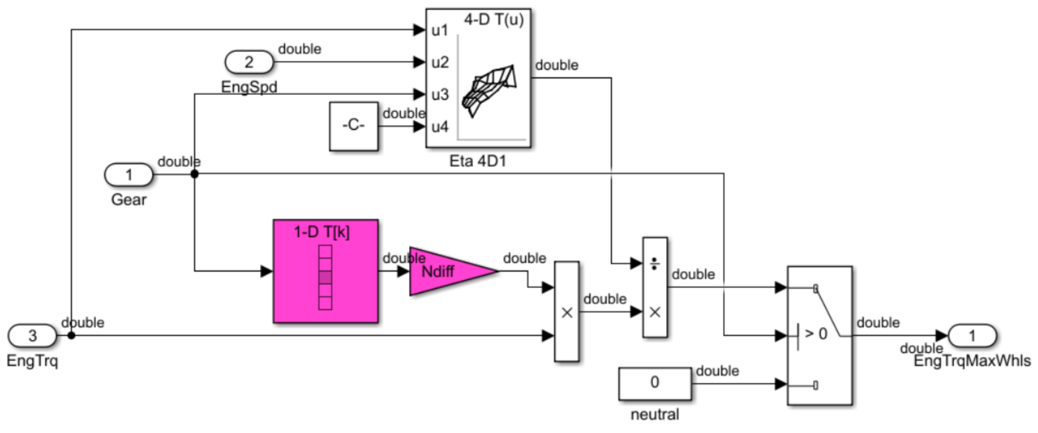


FIGURE 5.24: Original look of Max_EngWhlTrq Function

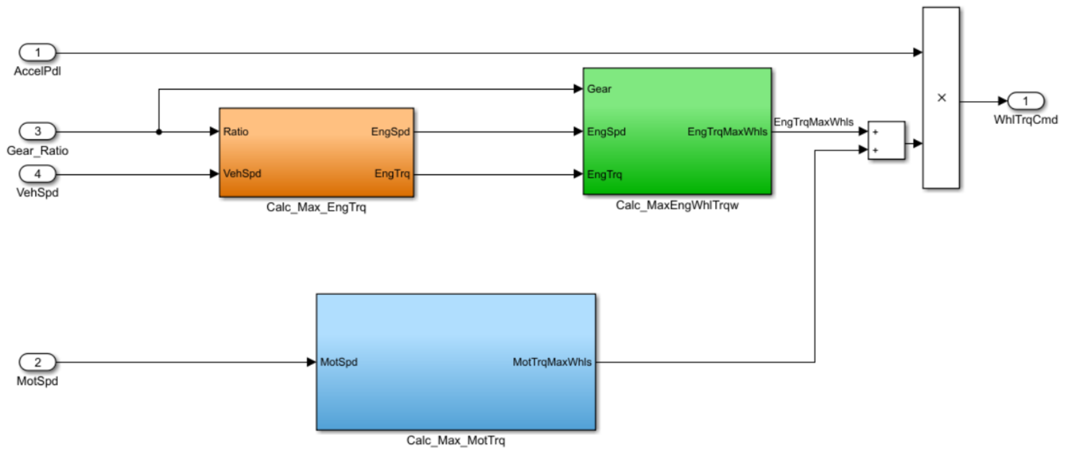


FIGURE 5.25: Algorithms within Max Vehicle Torque Subsystem

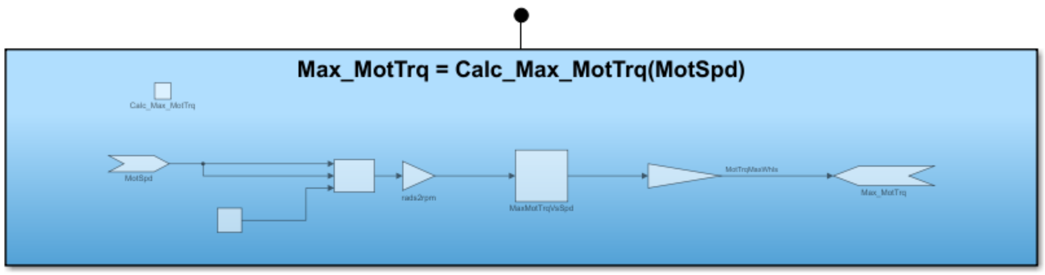


FIGURE 5.26: Max Motor Torque function defined inside MHM Model

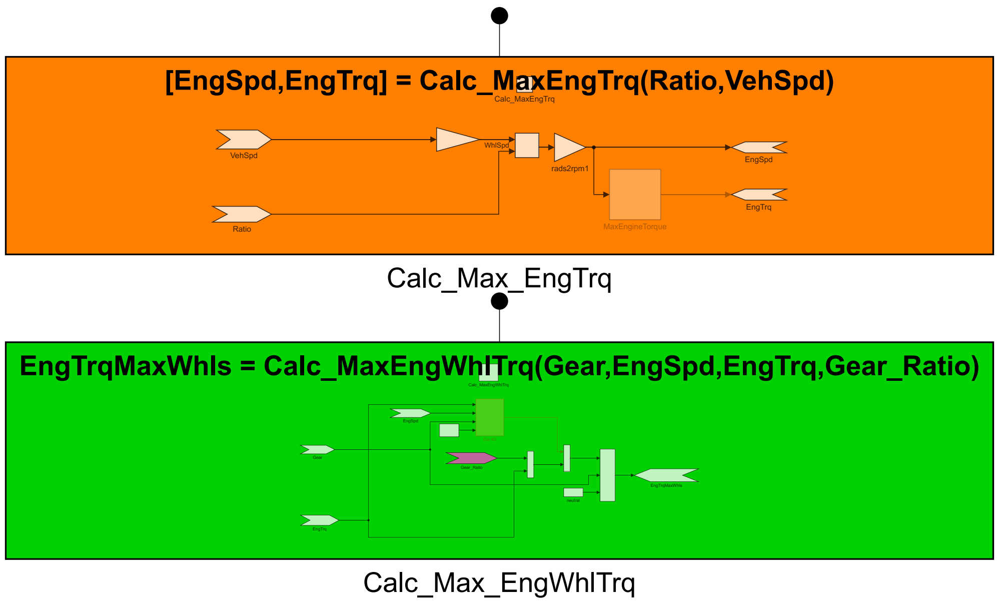

[EngSpd,EngTrq] = Calc_MaxEngTrq(Ratio,VehSpd)
Calc_MaxEngTrq
VehSpd
WhlSpd
rads2rpm1
EngSpd
EngTrq
MaxEngineTorque
Ratio
Calc_Max_EngTrq
EngTrqMaxWhls = Calc_MaxEngWhlTrq(Gear,EngSpd,EngTrq,Gear_Ratio)
Calc_Max_EngWhlTrq

FIGURE 5.27: Engine Hardware functions within EHM Model

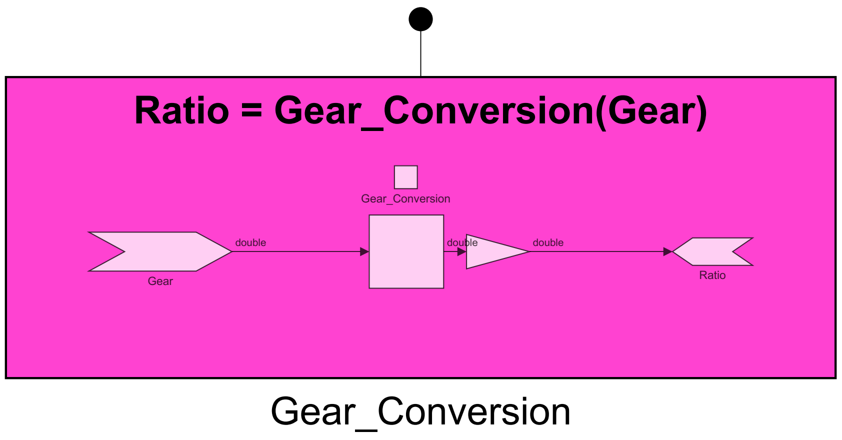


FIGURE 5.28: Gear Ratio Conversion Simulink Function defined within THM

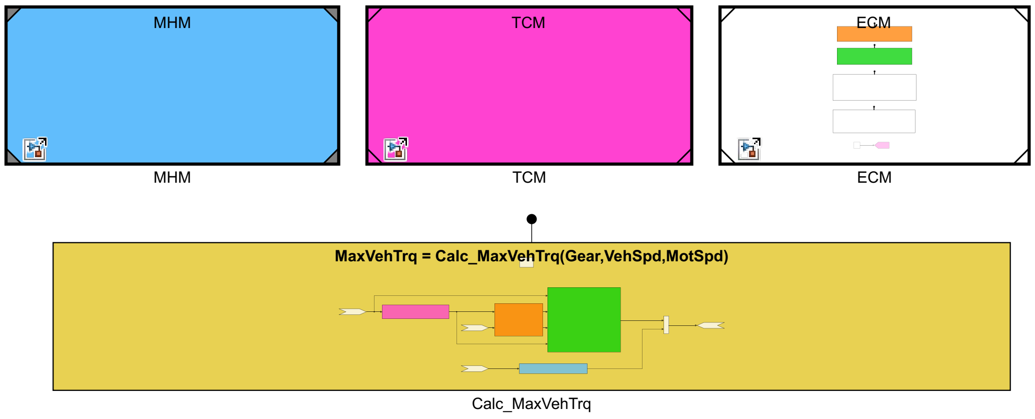


FIGURE 5.29: Calc_MaxVehTrq() Simulink function defined within the HLCM

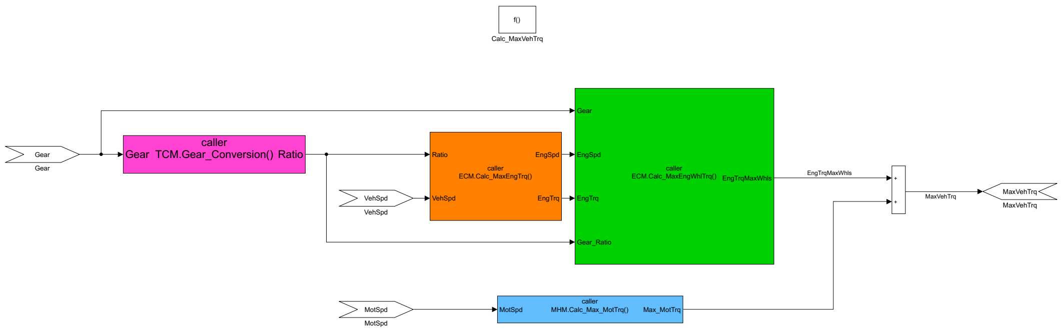


FIGURE 5.30: Calc_MaxVehTrq() Simulink function containing hardware access function callers

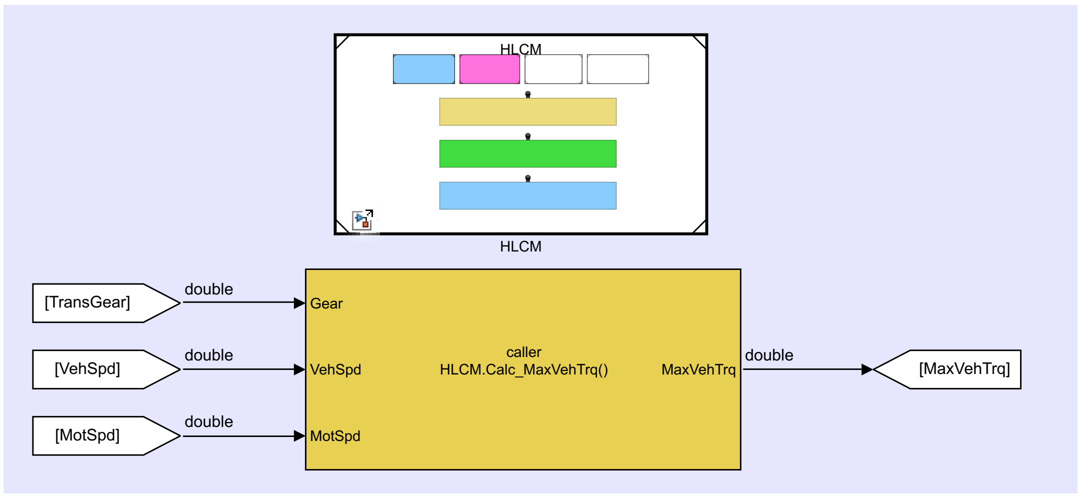


FIGURE 5.31: Calc_MaxVehTrq() function caller block at root level of HSC Model

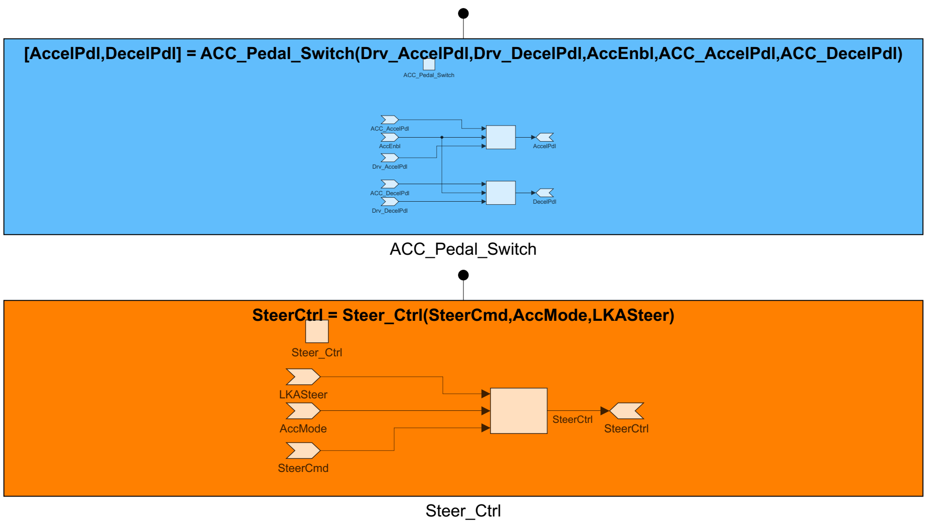


FIGURE 5.32: CAVs Exported Simulink Functions

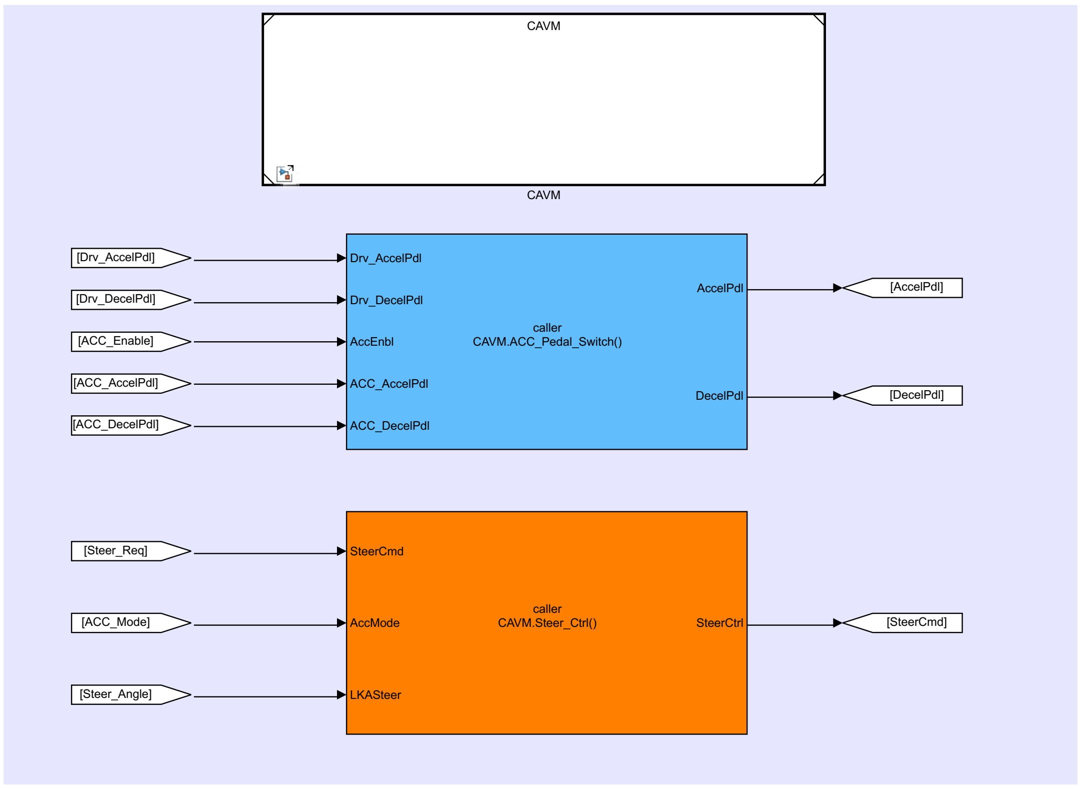


FIGURE 5.33: Top level HSC layer containing CAVs Model Reference and function caller blocks

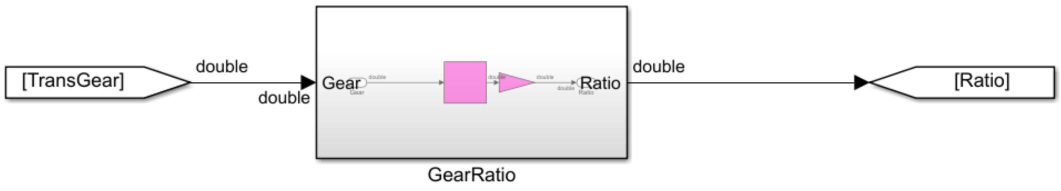


FIGURE 5.34: Gear Ratio Conversion Subsystem in Original Model

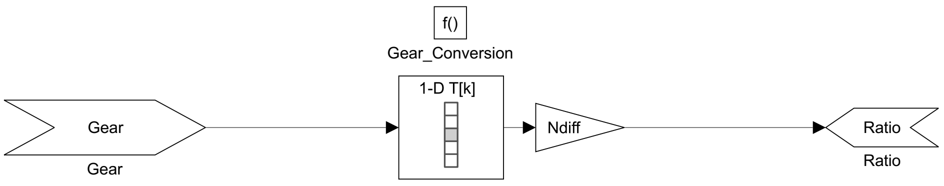


FIGURE 5.35: Internals of Gear Ratio Simulink Function

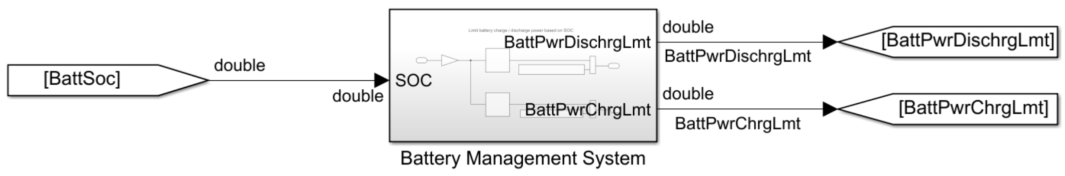


FIGURE 5.36: Original Battery Management System virtual subsystem

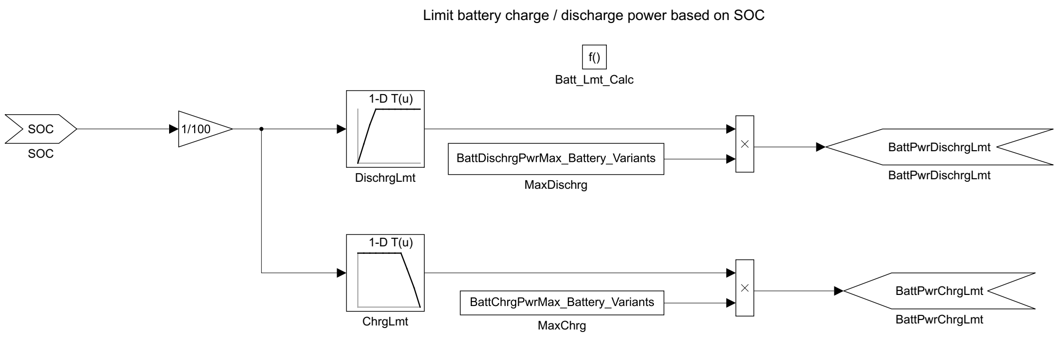


FIGURE 5.37: Internals of Battery Management Simulink Function

and lookup tables were moved to an exported Simulink function, *Batt_Lmt_Calc()*, seen in Figure 5.38 defined within the Battery Hardware Module. The overall hardware limit calculation algorithm remained a separate secret, and was defined as a new Simulink function, *Limit_Calc()*, within the HLCM (shown highlighted blue in Figure 5.39. The limit calculation function caller block was defined within the Input Conversion Layer in the HSC Model, highlighted blue in Figure 5.40. Separating of battery hardware secrets was motivated by likely changes 9 and 10 in Table A2.1.

5.6 Operation Mode Loop

The three major modules within the operation mode loop are the Propulsion Control Strategy Module(PCSM), Power Mode Module (PMM), and the Regenerative Power Module (RPM). These modules all collectively hold the secret of the method of energy management implemented within the HSC model, and would be affected by likely change 15 shown in Table A2.1. As a result, they were all consolidated within the newly created Energy Management Module (EMM), which can be seen at a high level in Figure 5.41.

5.6.1 Propulsion Control Strategy Module

The most significant change within the PCSM was improving the modularization of the PCSR Stateflow Chart shown in Figure 5.42. The original chart contains a

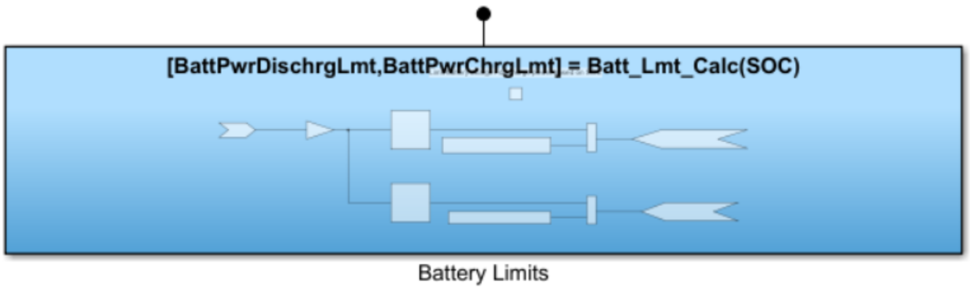


FIGURE 5.38: Battery Management Simulink Function defined within the BHM

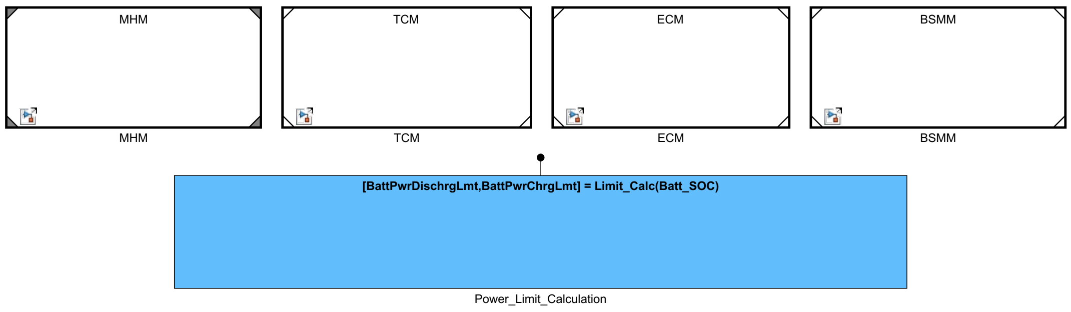


FIGURE 5.39: Battery Management Simulink Function caller at HSC root level

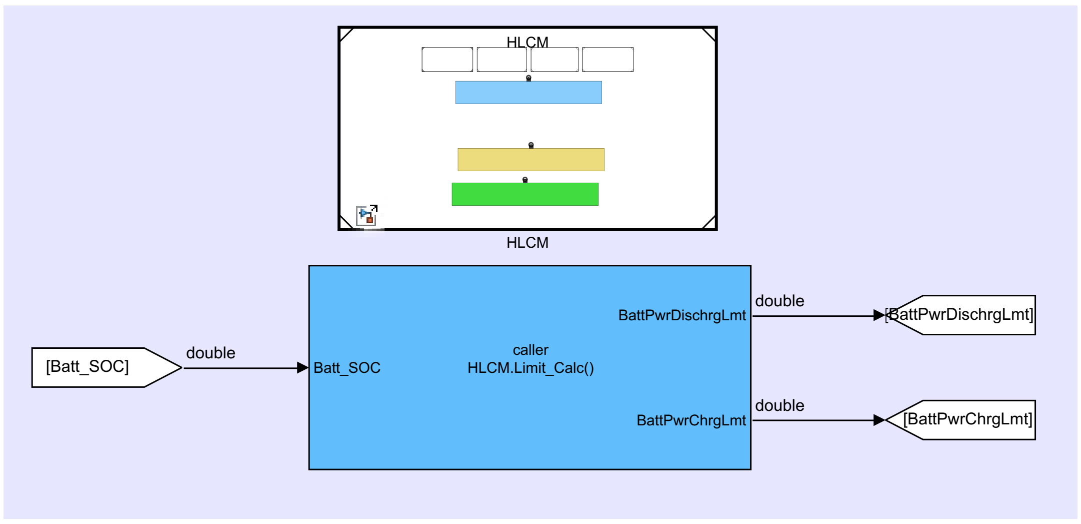


FIGURE 5.40: Battery Management Simulink Function caller at HSC root level

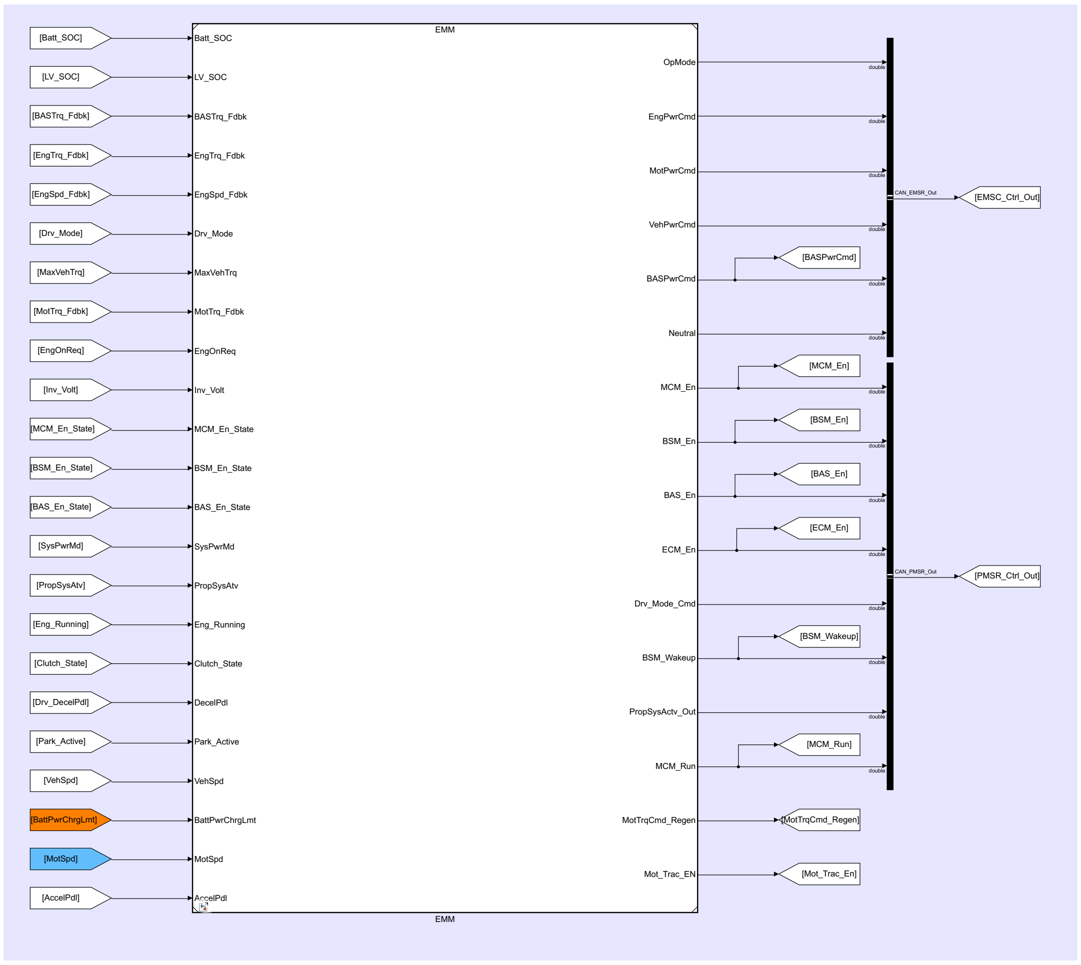


FIGURE 5.41: EMM Model Reference at HSC root level

graphical function comprised of states representing each of the possible vehicle drive modes: *Initial*, *Start/Stop*, *HEV*, *EV*, *ICE*, and *Performance*. Each drive mode state contains a unique Simulink function to calculate the total power needed from each powertrain component. Calls to the Simulink functions are contained within each of the respective Drive Mode states (Figure 5.43). The *HEV* Mode is shown in Figure 5.44 as an example, where both the *PowerCmd()* function and function caller are contained within the State Object.

During the reorganization of the PCSR Stateflow chart, it was determined that the PowerCmd() Simulink functions should be separated from the internal states of the drive mode loop. The drive mode states as well as the conditional statements for transition between them represent one algorithmic secret contained within a single graphical function. The algorithm for calculating power commands within each drive state also represent distinct model secrets and could all potentially require changes independent of one another. In each of the Drive Mode states, the Simulink function was removed, and placed at the root level of the Stateflow chart. All Simulink functions were then encapsulated within a Stateflow Box Object (Figure 5.45). Function calls within the states and transitions were adjusted to properly reference the Stateflow box object, as well as giving each drive mode function a unique name. The graphical function was also encapsulated within a Stateflow box object and remained at the root level of the Stateflow chart (Figure 5.47). An example of a power calculation function being called within the HEV State can be seen in Figure 5.46.

5.6.2 Power Moding Module

The PMM contains the algorithm for coordinating the startup and shutdown of the various hardware components interacting with the HSC. The control module was designed using a Stateflow chart primarily due to the unique timing transitions required between control states (Figure 5.48). The original PMSR Stateflow chart already encapsulates a single algorithm for coordinating component operation and was not further broken down or modularized. Furthermore, it was determined that the original Stateflow implementation was required to maintain controller functionality.

5.6.3 Regen Power Module

All functions containing secrets or algorithms related to regenerative braking have been organized within the RPM. In the original model, the Regen Torque Calculation Subsystem is responsible for determining the regenerative motor torque command, *MotTrqCmdRegen*, that is passed to the motor torque arbitrator subsystem. The subsystem within the Regen Power Model can be seen in Figure 5.49 (highlighted in yellow), and the internal implementation can be seen in Figure 5.50. Several different algorithms are contained within the subsystem, including the secret of calculating the battery charge limit based on SOC, highlighted in orange. This algorithm contains secrets related to the battery hardware limits and is already contained within the Battery Hardware Module. The blocks performing the charge limit calculation were removed from the Regen Power Module, and the *BattChrgLmt* signal was passed as an input into the subsystem (Figure 5.51). Similarly, the algorithm for calculating the

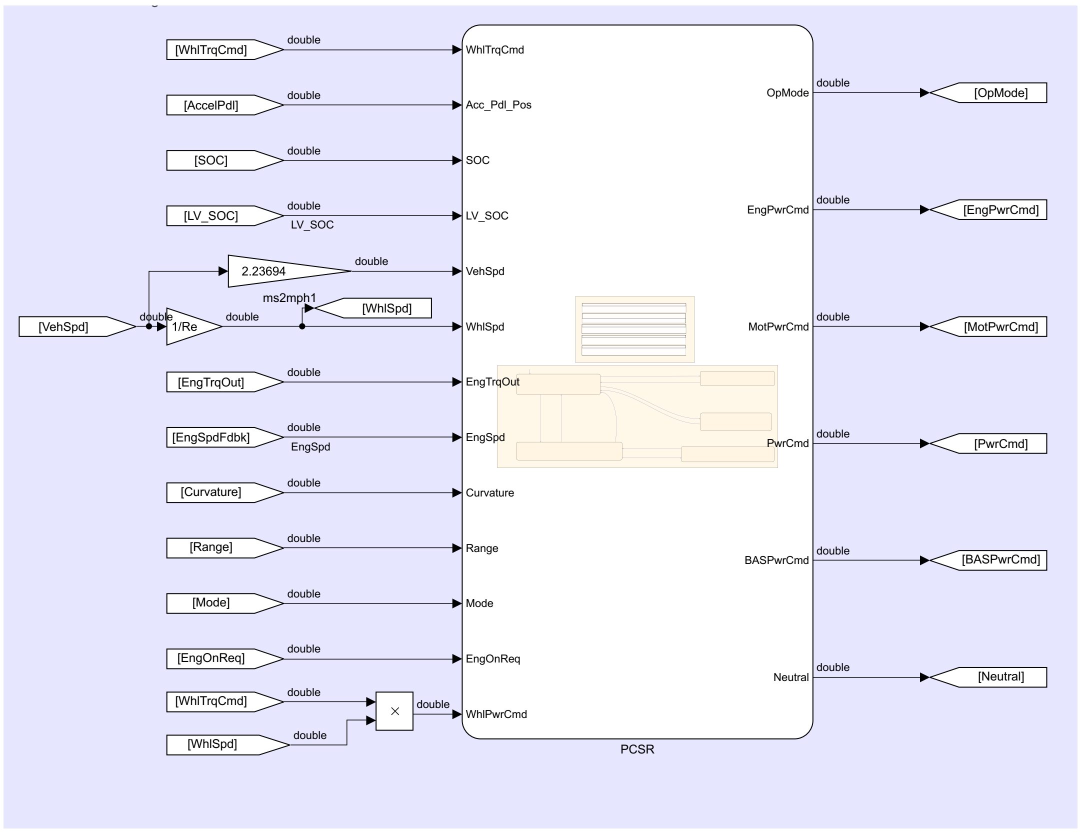


FIGURE 5.42: PCSR Stateflow Chart within the PCSM Model

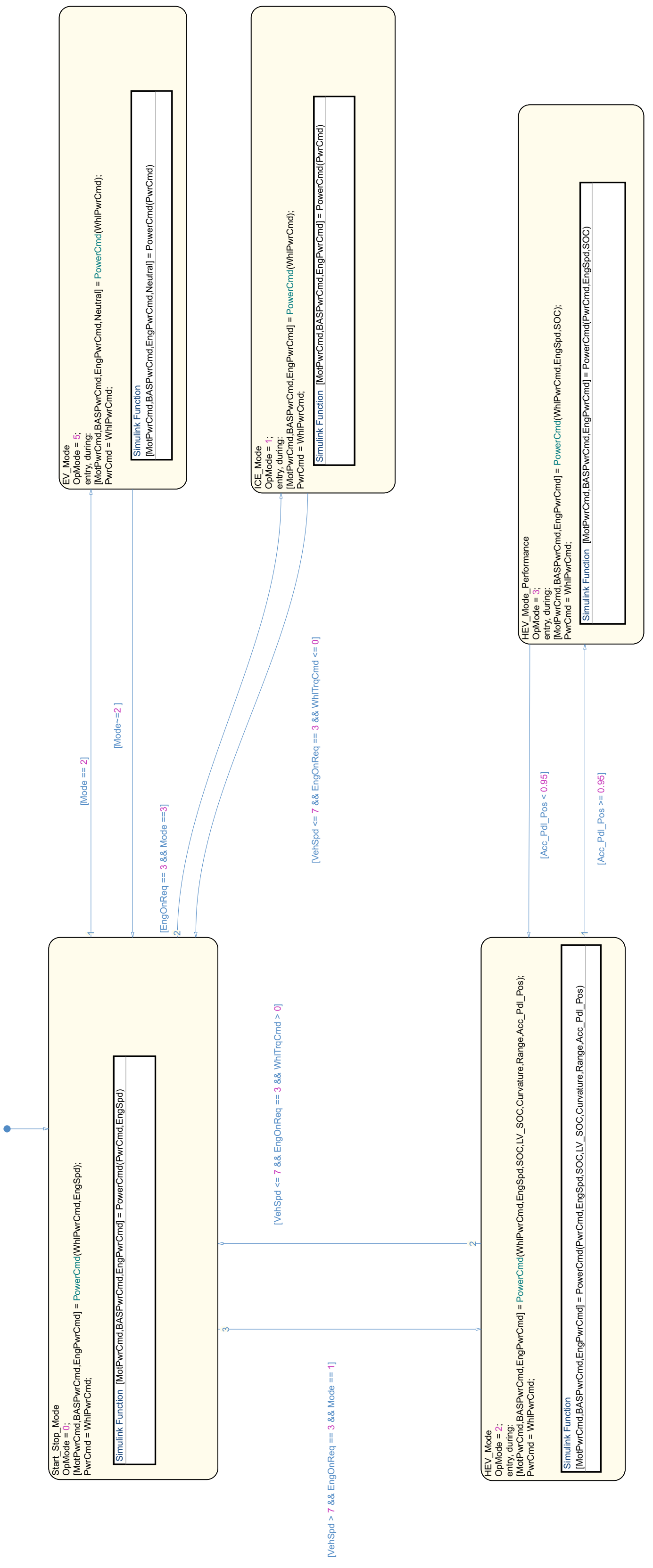
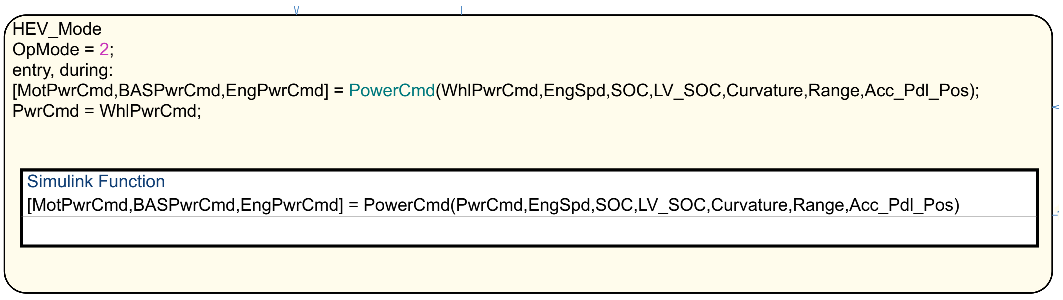
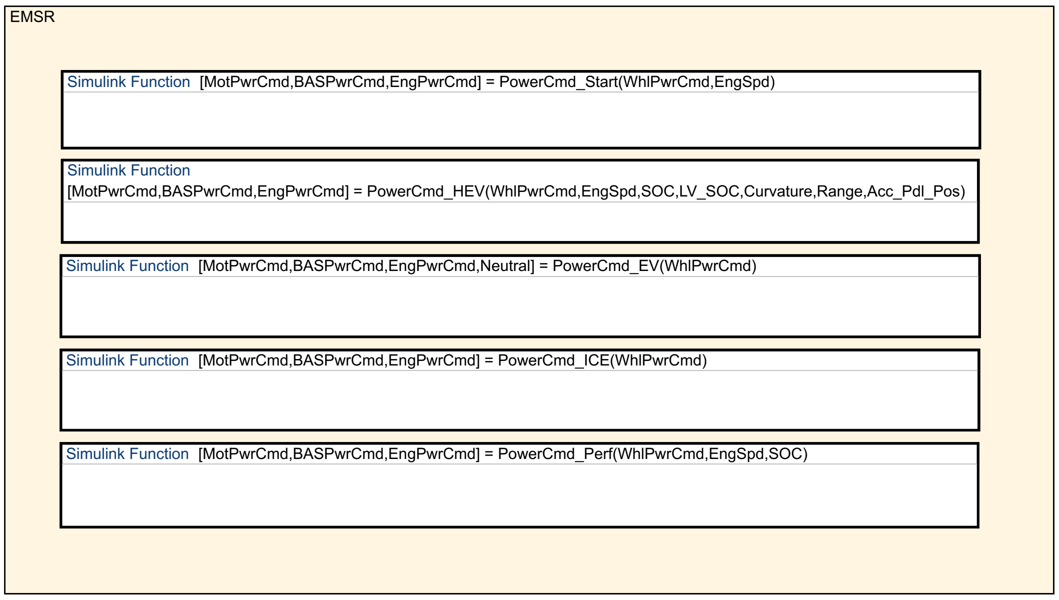

EV_Mode
OpMode = 5;
entry, during:
[MotPwrCmd,BASPwrCmd,EngPwrCmd,Neutral] = PowerCmd(WhlPwrCmd);
PwrCmd = WhlPwrCmd;
Simulink Function
[MotPwrCmd,BASPwrCmd,EngPwrCmd,Neutral] = PowerCmd(PwrCmd)
ICE_Mode
OpMode = 1;
entry, during:
[MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd(WhlPwrCmd);
PwrCmd = WhlPwrCmd;
Simulink Function [MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd(PwrCmd)
HEV_Mode_Performance
OpMode = 3;
entry, during:
[MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd(WhlPwrCmd,EngSpd,SOC);
PwrCmd = WhlPwrCmd;
Simulink Function [MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd(PwrCmd,EngSpd,SOC)
[Mode == 2]
[Mode~=2]
[EngOnReq == 3 && Mode ==3]
[VehSpd <= 7 && EngOnReq == 3 && WhlTrqCmd <= 0]
[Acc_Pdl_Pos < 0.95]
[Acc_Pdl_Pos >= 0.95]
Start_Stop_Mode
OpMode = 4;
[MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd(WhlPwrCmd,EngSpd);
PwrCmd = WhlPwrCmd;
Simulink Function [MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd(PwrCmd,EngSpd)
[VehSpd <= 7 && EngOnReq == 3 && WhlTrqCmd > 0]
[VehSpd > 7 && EngOnReq == 3 && Mode == 1]
HEV_Mode
OpMode = 2;
entry, during:
[MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd(WhlPwrCmd,EngSpd,SOC,LV_SOC,Curvature,Range,Acc_Pdl_Pos);
PwrCmd = WhlPwrCmd;
Simulink Function
[MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd(PwrCmd,EngSpd,SOC,LV_SOC,Curvature,Range,Acc_Pdl_Pos)
1
2
2
3
1

FIGURE 5.43: PCSR Stateflow Chart Internal Structure



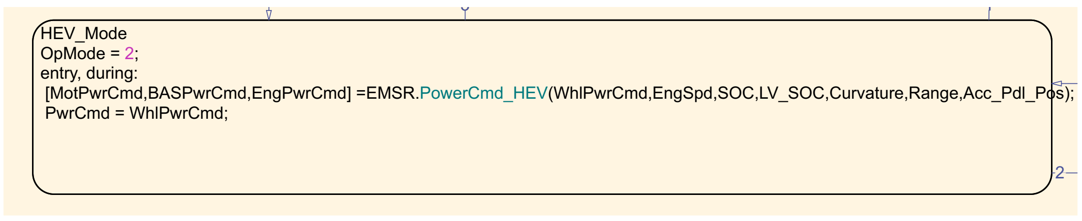
```
HEV_Mode
OpMode = 2;
entry, during:
[MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd(WhlPwrCmd,EngSpd,SOC,LV_SOC,Curvature,Range,Acc_Pdl_Pos);
PwrCmd = WhlPwrCmd;

Simulink Function
[MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd(PwrCmd,EngSpd,SOC,LV_SOC,Curvature,Range,Acc_Pdl_Pos)
```

FIGURE 5.44: State Object implementing the Functionality for the HEV Drive Mode



```
EMSR
Simulink Function  [MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd_Start(WhlPwrCmd,EngSpd)
Simulink Function
[MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd_HEV(WhlPwrCmd,EngSpd,SOC,LV_SOC,Curvature,Range,Acc_Pdl_Pos)
Simulink Function  [MotPwrCmd,BASPwrCmd,EngPwrCmd,Neutral] = PowerCmd_EV(WhlPwrCmd)
Simulink Function  [MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd_ICE(WhlPwrCmd)
Simulink Function  [MotPwrCmd,BASPwrCmd,EngPwrCmd] = PowerCmd_Perf(WhlPwrCmd,EngSpd,SOC)
```

FIGURE 5.45: Stateflow Box containing all local Simulink functions



```
HEV_Mode
OpMode = 2;
entry, during:
 [MotPwrCmd,BASPwrCmd,EngPwrCmd] =EMSR.PowerCmd_HEV(WhlPwrCmd,EngSpd,SOC,LV_SOC,Curvature,Range,Acc_Pdl_Pos);
 PwrCmd = WhlPwrCmd;
```

FIGURE 5.46: : Example of function call within HEV state using Stateflow Box Identifier

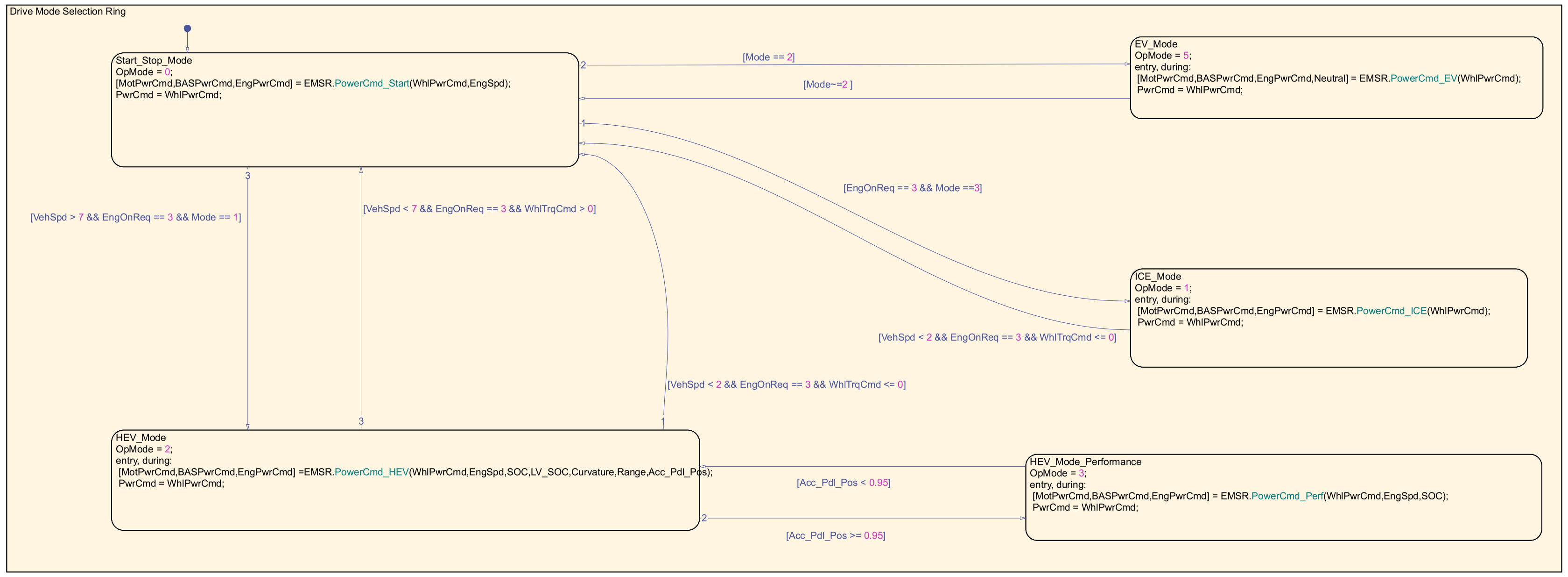

Drive Mode Selection Ring
Start_Stop_Mode
OpMode = 0;
[MotPwrCmd,BASPwrCmd,EngPwrCmd] = EMSR.PowerCmd_Start(WhlPwrCmd,EngSpd);
PwrCmd = WhlPwrCmd;
EV_Mode
OpMode = 5;
entry, during:
[MotPwrCmd,BASPwrCmd,EngPwrCmd,Neutral] = EMSR.PowerCmd_EV(WhlPwrCmd);
PwrCmd = WhlPwrCmd;
ICE_Mode
OpMode = 1;
entry, during:
[MotPwrCmd,BASPwrCmd,EngPwrCmd] = EMSR.PowerCmd_ICE(WhlPwrCmd);
PwrCmd = WhlPwrCmd;
HEV_Mode
OpMode = 2;
entry, during:
[MotPwrCmd,BASPwrCmd,EngPwrCmd] =EMSR.PowerCmd_HEV(WhlPwrCmd,EngSpd,SOC,LV_SOC,Curvature,Range,Acc_Pdl_Pos);
PwrCmd = WhlPwrCmd;
HEV_Mode_Performance
OpMode = 3;
entry, during:
[MotPwrCmd,BASPwrCmd,EngPwrCmd] = EMSR.PowerCmd_Perf(WhlPwrCmd,EngSpd,SOC);
PwrCmd = WhlPwrCmd;
[Mode == 2]
[Mode~=2]
[EngOnReq == 3 && Mode ==3]
[VehSpd < 2 && EngOnReq == 3 && WhlTrqCmd <= 0]
[VehSpd > 7 && EngOnReq == 3 && Mode == 1]
[VehSpd < 7 && EngOnReq == 3 && WhlTrqCmd > 0]
[VehSpd < 2 && EngOnReq == 3 && WhlTrqCmd <= 0]
[Acc_Pdl_Pos < 0.95]
[Acc_Pdl_Pos >= 0.95]
2
1
3
3
1
2

FIGURE 5.47: Stateflow Box containing graphical function

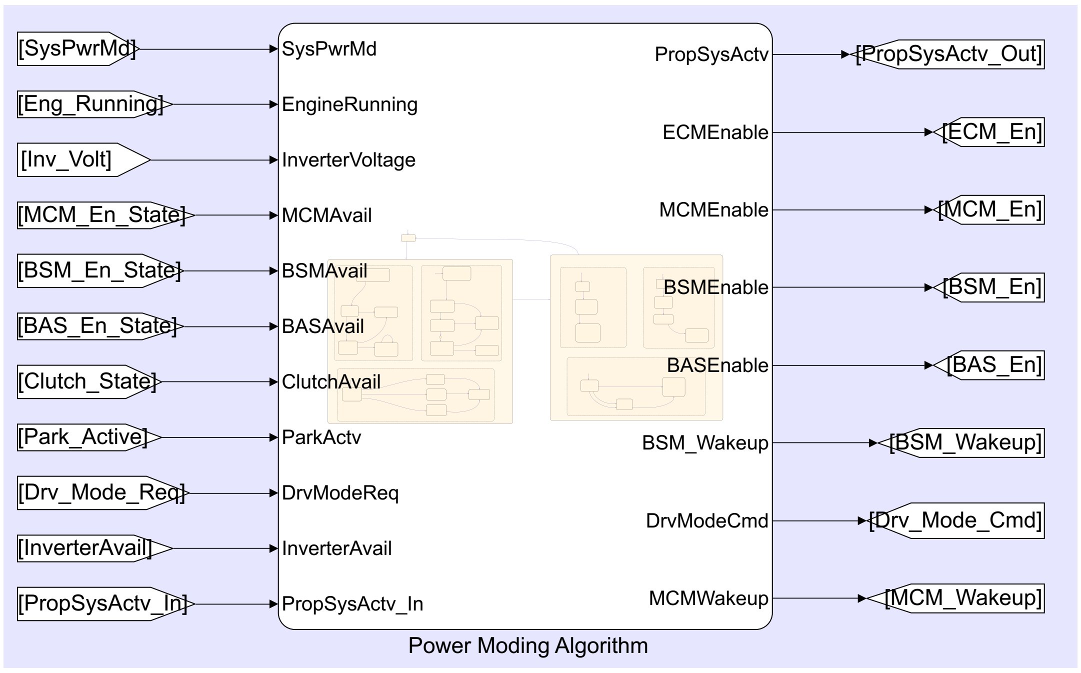


FIGURE 5.48: The PMSR Stateflow chart located within the PMM

Maximum Motor torque (highlighted in blue) contains secrets regarding the motor hardware limits. The algorithm was removed from the RPM, and the max torque signal, *Max_MotTrq* was passed in as an input to the Regen Torque Subsystem (Figure 5.51). The virtual subsystem was converted into a scoped Simulink function and placed within the RPM (Figure 5.52). A corresponding function caller block was placed within the root level of the EMM Model (Figure 5.53).

5.7 Component Control Layer

The modules located within the component control ring in the original model included the BASCM, MCM, BSMM, and EHM. With the exception of the EHM, these modules collectively hold the propulsion controls algorithm secret. Changes to the vehicle architecture would potentially impact all propulsion control components, and as a result, the BASCM, MCM, and BSMM have been grouped within the newly created Propulsion System Control Module (PSCM). These component control modules all contain Stateflow charts which require a shared sample time, providing further motivation for their grouping. A high level view of the PSCM within the HSC root level can be seen in Figure 5.54. The Clutch Control Module (CCM) has also been grouped within the PSCM, and is discussed in detail in Section 5.8.4.

5.7.1 BAS Control Module

The BASCM Model in its current state consists solely of the BAS Control Stateflow Chart, which is responsible for controlling startup, shutdown, and managing torque

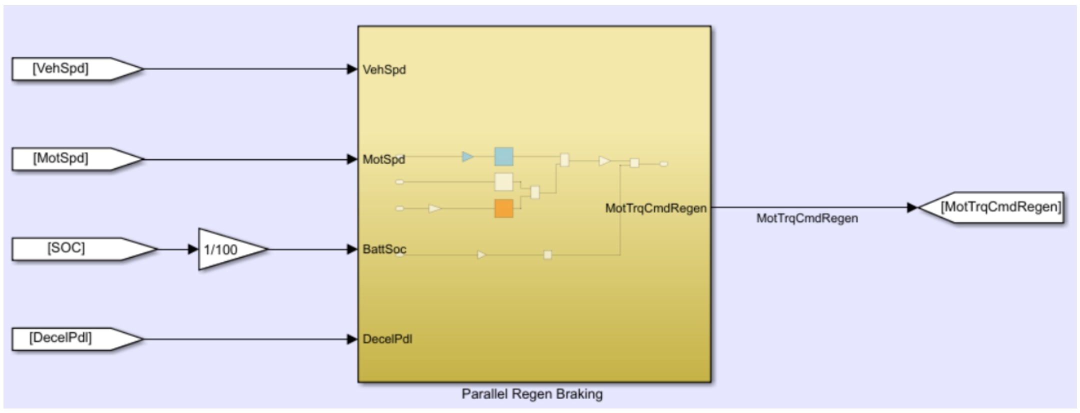


FIGURE 5.49: Original Regen Torque Calculation Subsystem

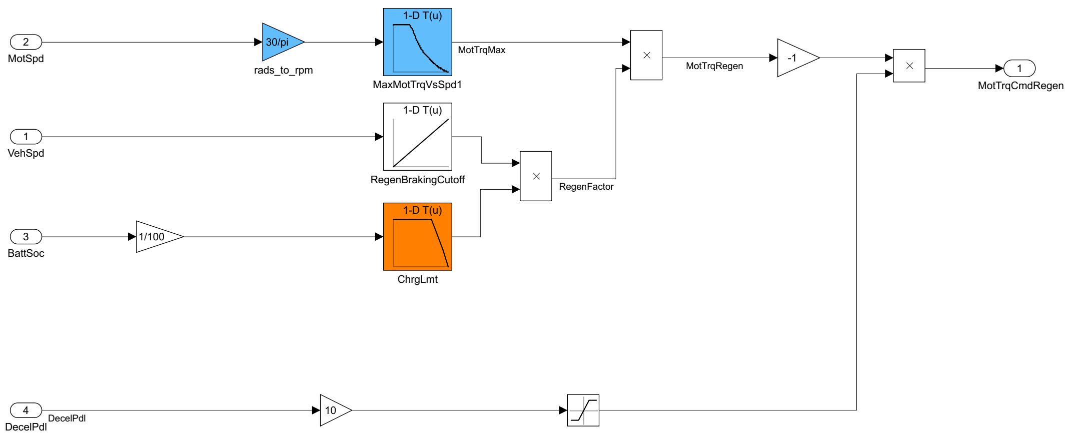


FIGURE 5.50: Original Regen Torque Calculation Subsystem Internals

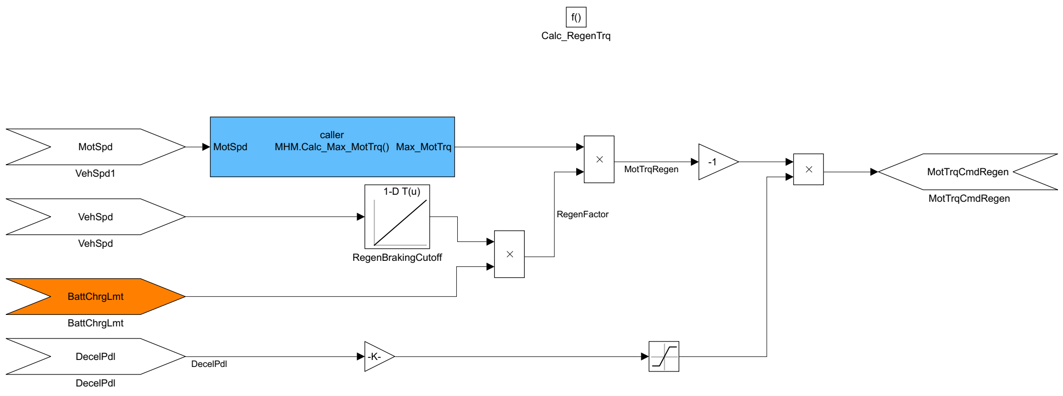


FIGURE 5.51: Modified Regen Torque Calculation Algorithm

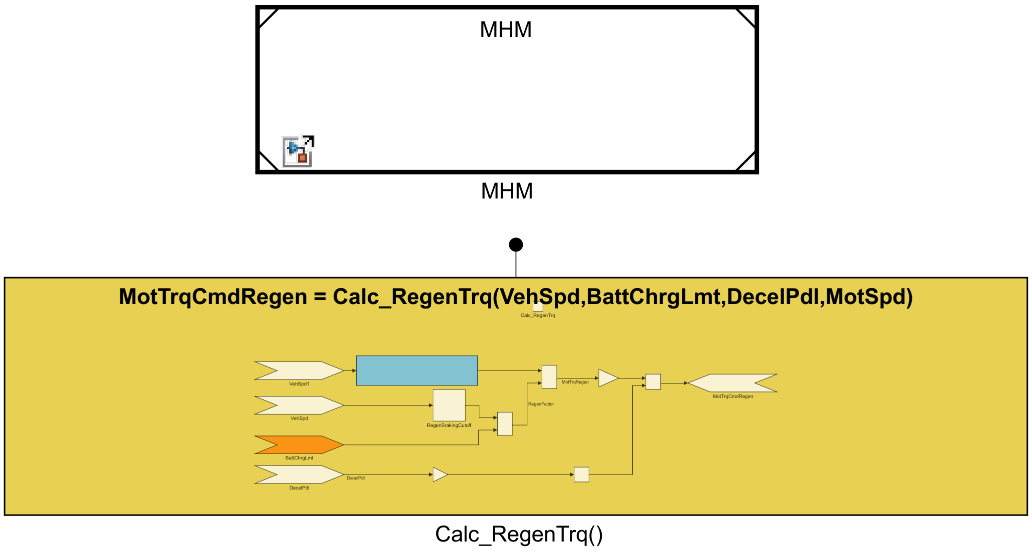

MHM
MHM
MotTrqCmdRegen = Calc_RegenTrq(VehSpd,BattChrgLmt,DecelPdl,MotSpd)
Calc_RegenTrq()

FIGURE 5.52: Scoped Simulink function within Regen Power Module

requests. The stateflow structure is shown in Figure 5.55, but much of the internal functionality has been redacted due to Valeo NDA restrictions. The BASCM functionalities were represented within a graphical function made up of states and transitions and did not require further modularization. The only change that was made, was encapsulating the BAS control function within a Stateflow Box object in order to reduce its scope to be local to the Stateflow Chart. As the vehicle software is further developed, additional functionality such as fault detection and hardware limit checking will need to be added for teh BAS component. These additional modular secrets will then need to be separated from the controller functions via a BAS Hardware Module, and provide necessary data to the BASCM through the use of scoped Simulink Functions. The final Stateflow chart can be seen in Figure 5.56, where the single graphical function is contained within a Stateflow box object; no other subsystems exist within the module, therefore no exported functions are required.

5.7.2 Motor Control Module

The original MCM structure can be seen in Figure 5.57. The model consists of two main sections, the Motor Control Stateflow chart (highlighted blue) and the Fault Detection and Fault Mitigation subsystems (highlighted in orange and green, respectively). To apply the modularization techniques discussed in Section 5.1, subsystems must be converted into Simulink functions, and scoped according to which hierarchical levels they are being used in. The Fault Detection subsystem receives four input CAN signals: *Post_Fault_Low*, *Post_Fault_Hi*, *Run_Fault_Low*, *Run_Fault_Hi*, and determines whether or not a fault has been detected by the Rinehart Motor Controller. The Fault Mitigation subsystem contains the information necessary to determine what action the motor controller must take based on which faults are active.

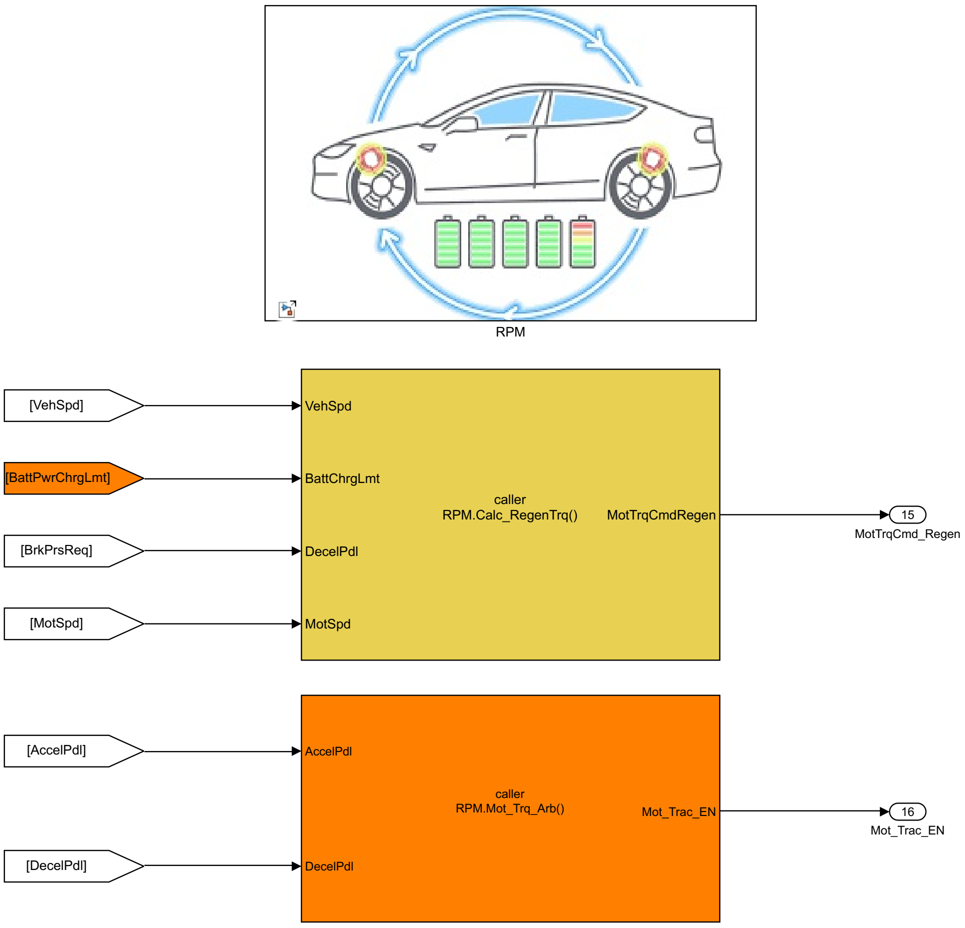


FIGURE 5.53: *Calc_RegenTrq()* function caller defined in EMM

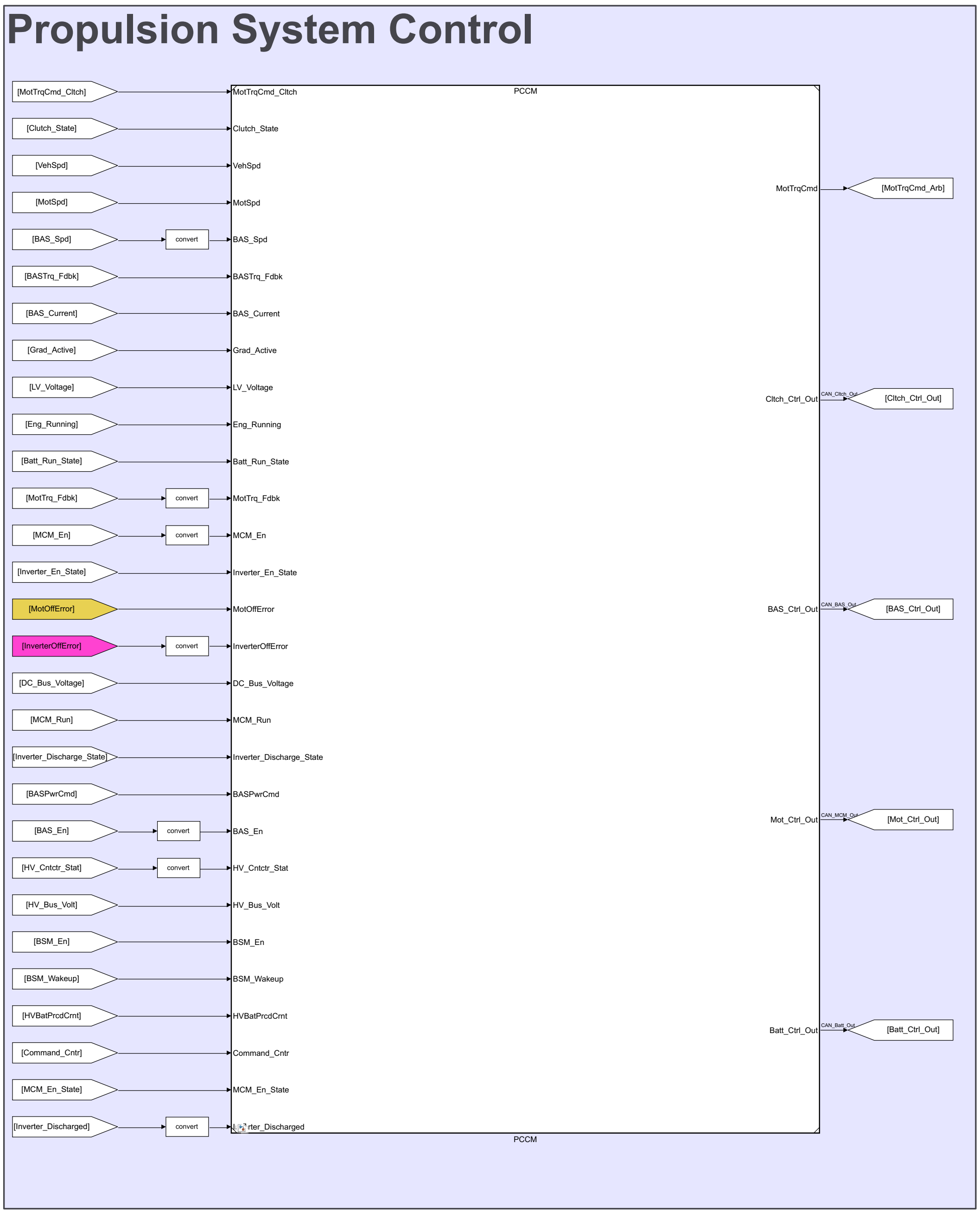


FIGURE 5.54: Propulsion System Control Module Implemented as a model reference within the HSC

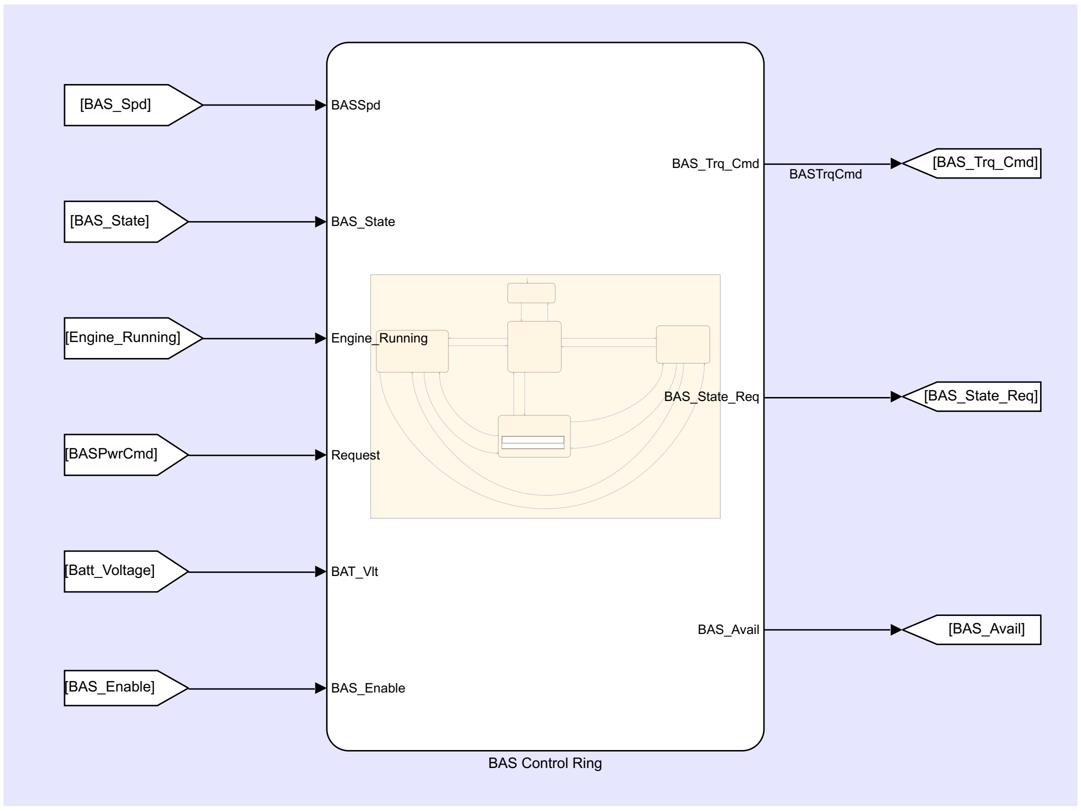


FIGURE 5.55: Top level view of BAS Control Ring defined within BASCM

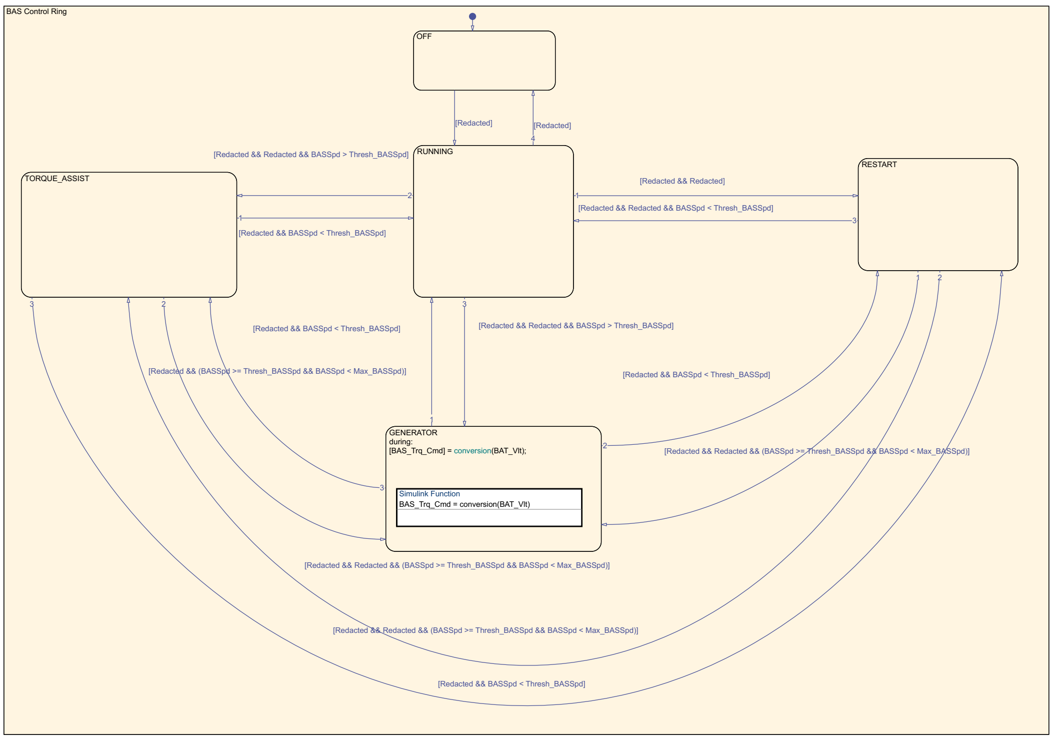

BAS Control Ring
OFF
[Redacted]
[Redacted]
RUNNING
TORQUE_ASSIST
RESTART
[Redacted && Redacted && BASSpd > Thresh_BASSpd]
[Redacted && BASSpd < Thresh_BASSpd]
[Redacted && Redacted]
[Redacted && Redacted && BASSpd < Thresh_BASSpd]
[Redacted && BASSpd < Thresh_BASSpd]
[Redacted && Redacted && BASSpd > Thresh_BASSpd]
[Redacted && (BASSpd >= Thresh_BASSpd && BASSpd < Max_BASSpd)]
[Redacted && BASSpd < Thresh_BASSpd]
GENERATOR
during:
[BAS_Trq_Cmd] = conversion(BAT_Vlt);
Simulink Function
BAS_Trq_Cmd = conversion(BAT_Vlt)
[Redacted && Redacted && (BASSpd >= Thresh_BASSpd && BASSpd < Max_BASSpd)]
[Redacted && Redacted && (BASSpd >= Thresh_BASSpd && BASSpd < Max_BASSpd)]
[Redacted && Redacted && (BASSpd >= Thresh_BASSpd && BASSpd < Max_BASSpd)]
[Redacted && BASSpd < Thresh_BASSpd]

FIGURE 5.56: BAS Control logic locally encapsulated within a Box Object

Both algorithms were converted to local Simulink functions using the Simulink Module Tool and defined within the newly created Fault Handling Module (FHM). Rather then being defined directly within the same module as the motor controller, it was deemed more appropriate to group together all fault handling algorithms in a separate module, and pass in relevant fault information as inputs. This was also motivated by handling likely changes 26 and 27 in Table A2.2 The Fault detection and mitigation Simulink function callers are located within the FHM function, *MotFault_Handler()*, seen highlighted orange and green, respectively in Figure 5.58. The *MotFault_Handler()* function is defined at the root level of the FHM, accessing the local *Fault_Detection()* and *Fault_Mitigation()* functions defined within the virtual subsystem seen in Figure 5.59. The resulting output signals (highlighted in yellow and magenta) from the fault handler function call shown in Figure 5.60 are passed as inputs to the Motor Control Module (Figure 5.61).

5.7.3 Battery System Manager Module

The BSMM contains a Stateflow Chart which hides the secret of battery control logic. The original Stateflow chart consists of a graphical function in the form of states and transitions, as well as a Simulink Function, *set_Voltages()*, defined at the root level of the Stateflow Chart. The Stateflow Chart is shown in Figure 5.62, and the *set_Voltages()* Simulink function is shown in Figure 5.63. Confidential Battery signals and control logic have been removed from the image due to NDA restrictions. In order to improve the modularization of this model, a Stateflow Box object was created to encapsulate both the graphical control logic function, as well as the Simulink function *set_Voltages()*. The scope of the Simulink function remains local, as it is not required at higher hierarchical levels of the model and does not need to be exported from the Stateflow Chart. The improved battery control Stateflow Chart can be seen in Figures 5.64 and 5.65, with each function separated into distinct Stateflow Box objects.

5.7.4 Engine Hardware Module

Much of the fault detection and algorithmic complexity of the GM provided Engine has been hidden from the HSC software and is located within the GM factory ECM hardware. While the internal engine controls are not present within the HSC, there does exist a need for the EHM, which is composed of functions containing the various engine hardware parameters needed by other modules within the HSC. The EHM was implemented without the use of Stateflow Chart Objects and is entirely composed of engine hardware secrets implemented as scoped Simulink functions. As a result, it did not share a component control secret with the other modules within the component control layer and did not get grouped into the PCSM.

The internal structure of the EHM can be seen in Figure 5.66, containing the Input Pedal Mapping function, *Get_EngAxleTrq()*, highlighted in blue. The Output Pedal Mapping function, *Get_Pdl_Out()*, is also contained within the EHM, highlighted in yellow. The EHM also contains two engine related Simulink functions, *Calc_WhlTrq()* and *Calc_MaxEngTrq()*, discussed in sections 5.5.2 and 5.5.4, respectively. The EHM is imported throughout the HSC system hierarchy using model

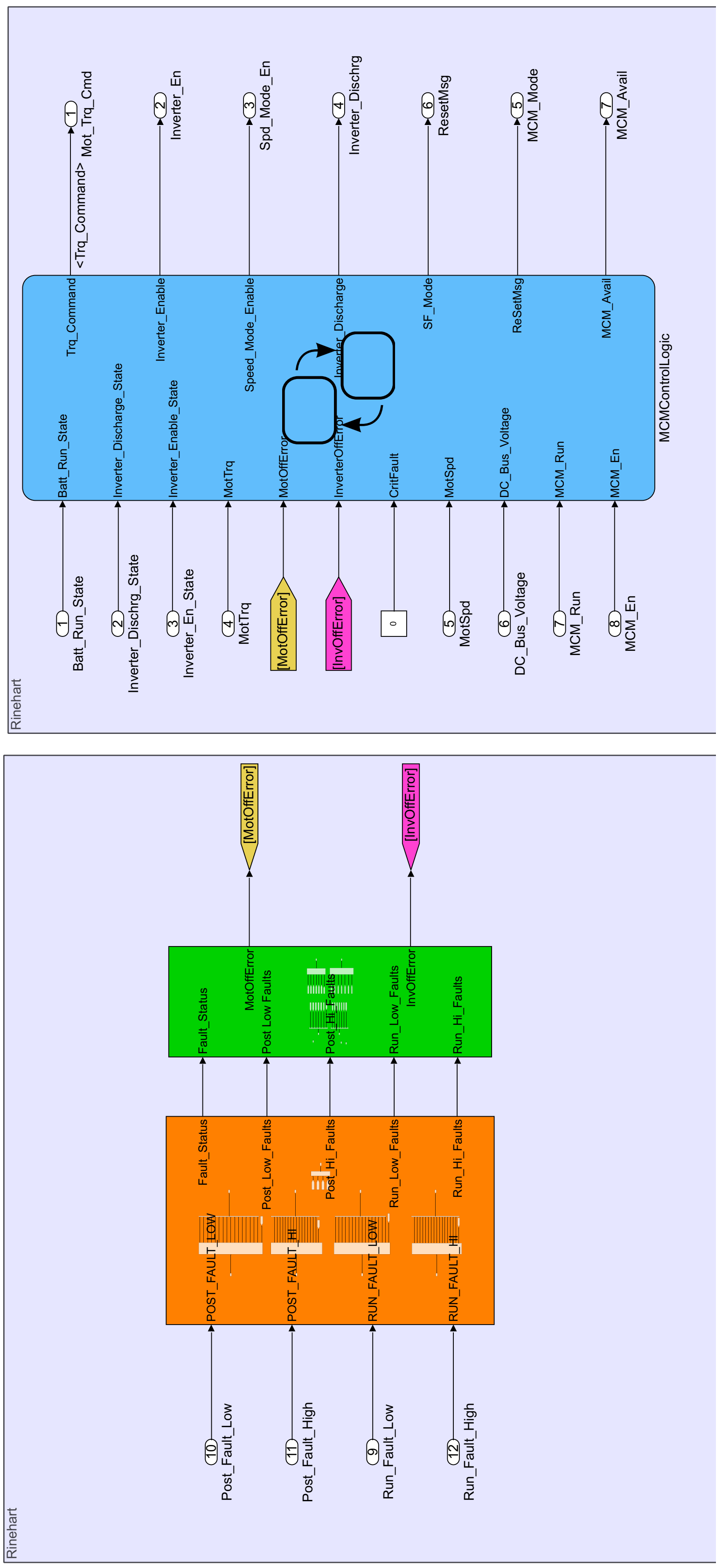


Figure 5.57: Original Motor Control Model Layout

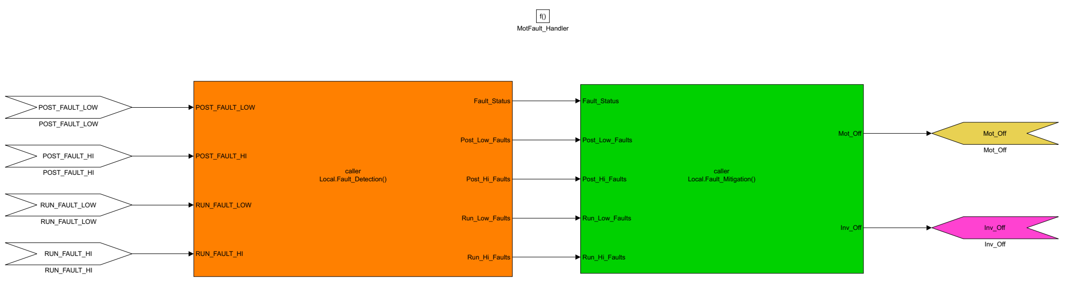


FIGURE 5.58: Fault Detection function caller blocks and data flow to MHM

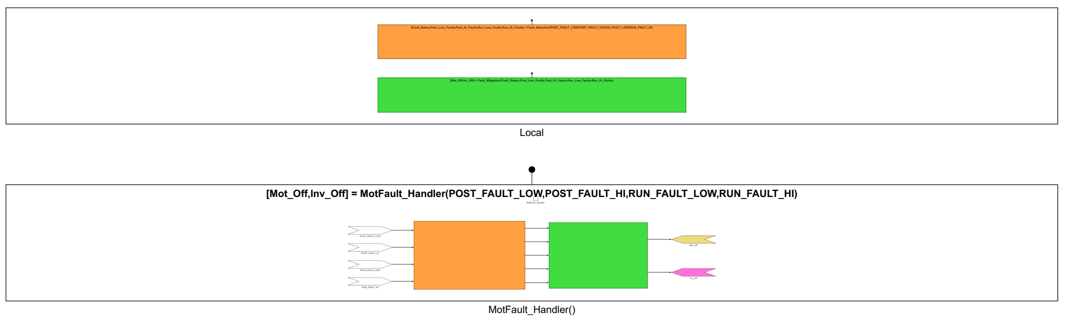

[Fault_Status,Post_Low_Faults,Post_Hi_Faults,Run_Low_Faults,Run_Hi_Faults] = Fault_Detection(POST_FAULT_LOW,POST_FAULT_HI,RUN_FAULT_LOW,RUN_FAULT_HI)
[Mot_Off,Inv_Off] = Fault_Mitigation(Fault_Status,Post_Low_Faults,Post_Hi_Faults,Run_Low_Faults,Run_Hi_Faults)
Local
[Mot_Off,Inv_Off] = MotFault_Handler(POST_FAULT_LOW,POST_FAULT_HI,RUN_FAULT_LOW,RUN_FAULT_HI)
MotFault_Handler()

FIGURE 5.59: Fault Handler Simulink functions defined within the FHM

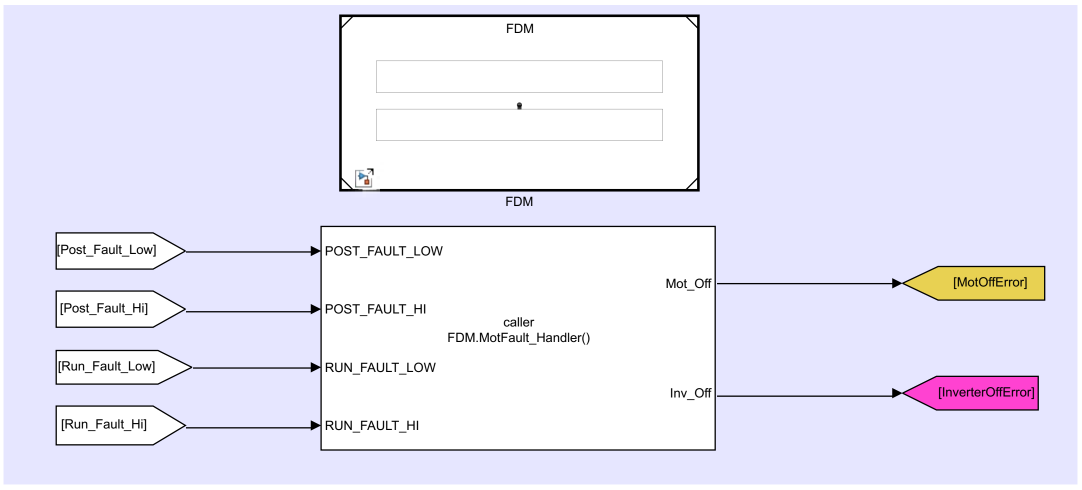


FIGURE 5.60: *MotFault_Handler()* function caller defined within the HSC Input Conversion Layer

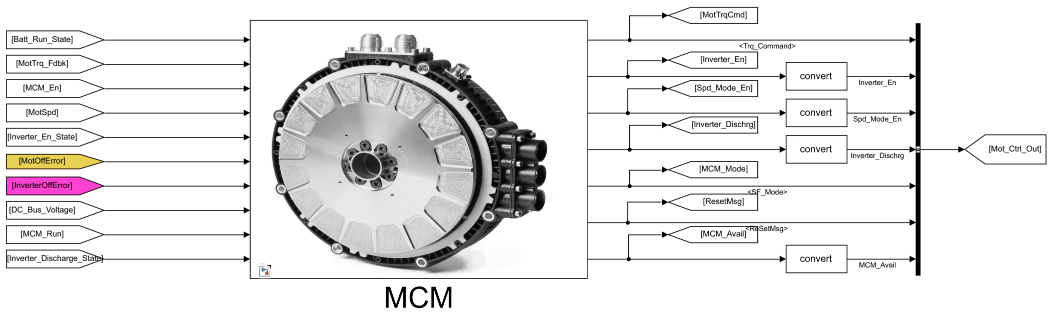


FIGURE 5.61: Fault Inputs passed to MCM within the PSCM

reference blocks. Function caller blocks are defined within other HSC system modules if any engine hardware related information is required. While all engine related functions have been grouped within the same model, their function calls appear in very different areas throughout the data flow process; this is a key benefit of modularization with the use of scoped Simulink Functions.

5.8 Output Conversion Layer

The Output Conversion Layer subsystem was decomposed, with each individual component analyzed to determine what secrets it held. Similar to the Input Conversion Subsystem described in Section 5.5, the original decomposition was not based on shared algorithm secrets but rather the components role in the data flow process. This decomposition was improved significantly by converting subsystems to Simulink functions, and redefining them within the HSC hierarchy according to the secrets they hold, and the likely changes that would affect them.

5.8.1 Motor Torque Arbitration and Power Management

The subsystem shown in Figures 5.67 and 5.68 contains two algorithms: torque arbitration between traction and regenerative modes (highlighted in blue), as well as a power management algorithm for the motor output (highlighted in orange). These algorithms are distinct functions from one another and therefore should not be implemented within the same subsystem. In preparation for the application of the Simulink Module Tool, the torque arbitration algorithm was removed from its current subsystem and placed higher in the model hierarchy, within the Output Conversion Layer. The Power Management algorithm was already implemented within a subsystem, which was brought to the output conversion layer of the model hierarchy (Figure 5.69).

5.8.2 Power Management Subsystem Decomposition

Prior to decomposing the individual algorithms within the power management subsystem, it was useful to first consider the overall function that was held secret; the process for checking the hardware limits of the electrical powertrain and validating

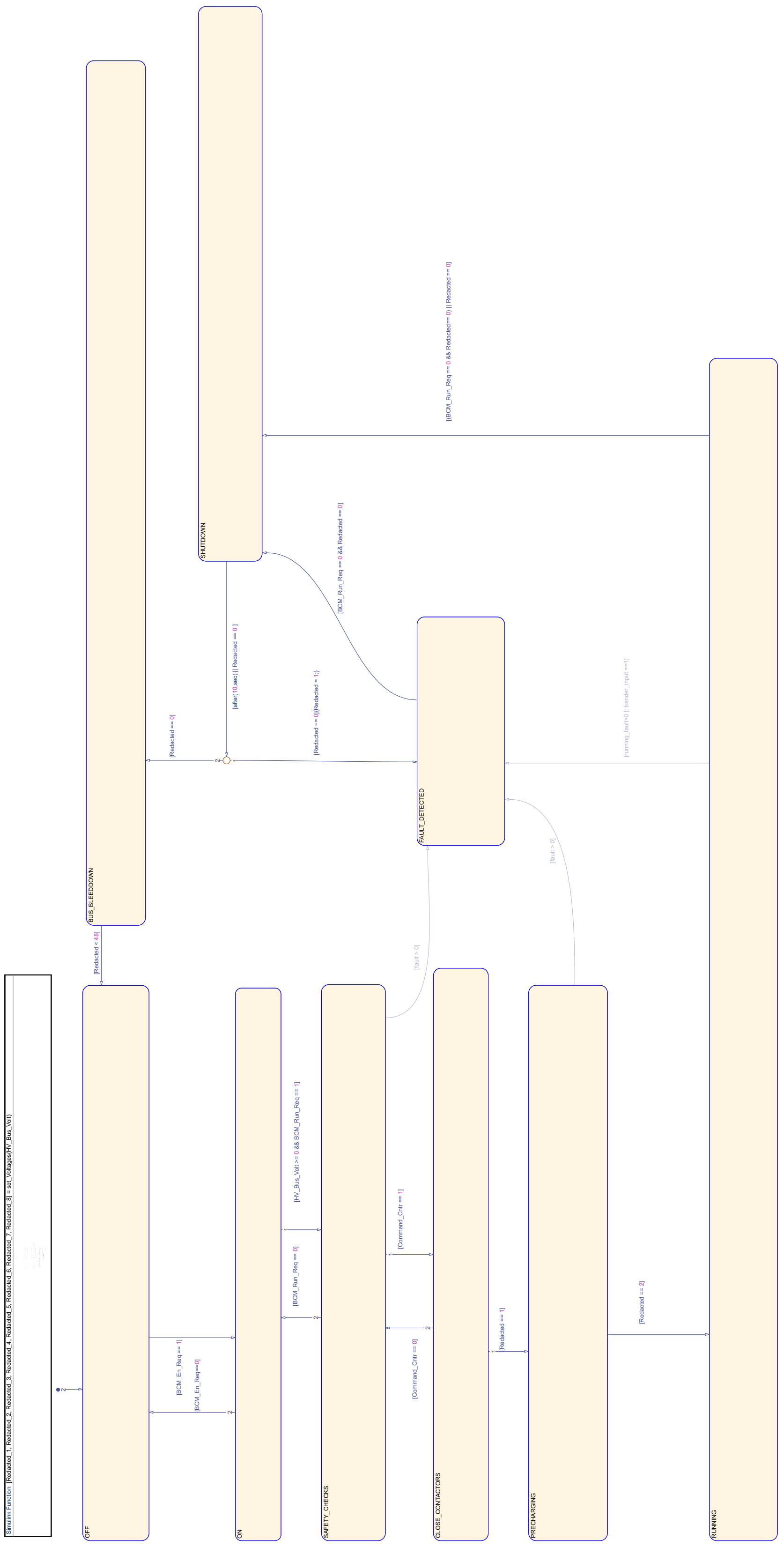

Simulink Function [Redacted_1, Redacted_2, Redacted_3, Redacted_4, Redacted_5, Redacted_6, Redacted_7, Redacted_8] = set_Voltages(HV_Bus_Volt)
OFF
ON
SAFETY_CHECKS
CLOSE_CONTACTORS
PRECHARGING
RUNNING
BUS_BLEEDDOWN
SHUTDOWN
FAULT_DETECTED
[BCM_En_Req == 1]
[BCM_En_Req==0]
[BCM_Run_Req == 0]
[HV_Bus_Volt >= 0 && BCM_Run_Req == 1]
[Command_Cntr == 1]
[Command_Cntr == 0]
[Redacted == 1]
[Redacted == 2]
[Redacted < 48]
[Redacted == 0]
[after(10,sec) || Redacted == 0]
[Redacted == 0||Redacted = 1;]
[BCM_Run_Req == 0 && Redacted == 0]
[BCM_Run_Req == 0 && Redacted== 0) || Redacted == 0]
[fault = 0]

FIGURE 5.62: set_Voltage() Simulink function defined at root level of Stateflow Chart

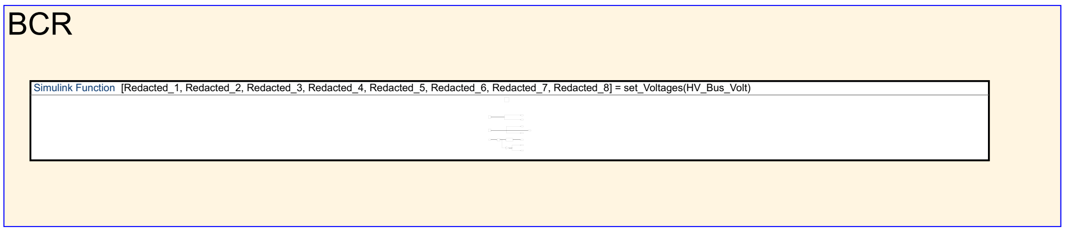

BCR
Simulink Function [Redacted_1, Redacted_2, Redacted_3, Redacted_4, Redacted_5, Redacted_6, Redacted_7, Redacted_8] = set_Voltages(HV_Bus_Volt)

FIGURE 5.63: set_Voltages() Simulink function within modified Stateflow Chart

outgoing control signals. This functionality fell within the behavior contained in the Hardware Limit Control Module, and would be affected by likely change 25 in Table A2.2. As a result the subsystem was transferred to the HLCM Model and converted to a scoped Simulink function, *Limit_Check()*. Prior to Simulink function conversion, the subsystem contained several different algorithms, one of which was the subsystem that converted Motor Torque to an Electrical Power Estimate, shown highlighted in blue in Figure 5.70. The internal algorithm (shown in Figure 5.71) contains secrets regarding the motor hardware and efficiency values and was moved accordingly into the MHM Model after being converted into a scoped Simulink Function (Figure 5.72).

The second algorithm located in the Power management subsystem (highlighted orange in Figure 5.70) checked whether the requested electrical power fell within acceptable battery power limits and adjusted the command accordingly. The subsystem contained secrets regarding the Battery Hardware and was converted into a scoped Simulink function within the Battery Hardware Model, seen highlighted in orange in Figure 5.73. The final algorithm contained within the power management subsystem is the Motor Torque Limit Check (highlighted green in Figure 5.70), which contains secrets regarding the motor hardware components. The Motor Torque Limit Check subsystem was transferred to the MHM Model, and then converted into a scoped Simulink function, highlighted in green in Figure 5.72. The internal structure of the *Limit_Check()* function can be seen in 5.74, with the function caller blocks for the three hardware algorithms highlighted in blue, orange, and green. References to the relevant hardware modules are defined at the root level of the HLCM, at the same level as the *Limit_Check()* function (shown highlighted green in Figure 5.75).

5.8.3 Motor Torque Arbitration Decomposition

The original subsystem contains the algorithm for determining whether regenerative torque command signals are present and should be honored, before sending the final arbitrated signal to the motor hardware. This algorithm groups well within the Regen Power Module as it is affected by likely change 18 in Table A2.2. The subsystem was transferred to the RPM model and implemented as a scoped Simulink Function, shown in Figure 5.76. A corresponding function caller block was placed within the EMM (highlighted orange Figure 5.53). The newly added output signal, *Mot_Trac_En*, is passed from the EMM to the Output Conversion layer, where it is used within the *Limit_Check()* function to determine final output motor torque.

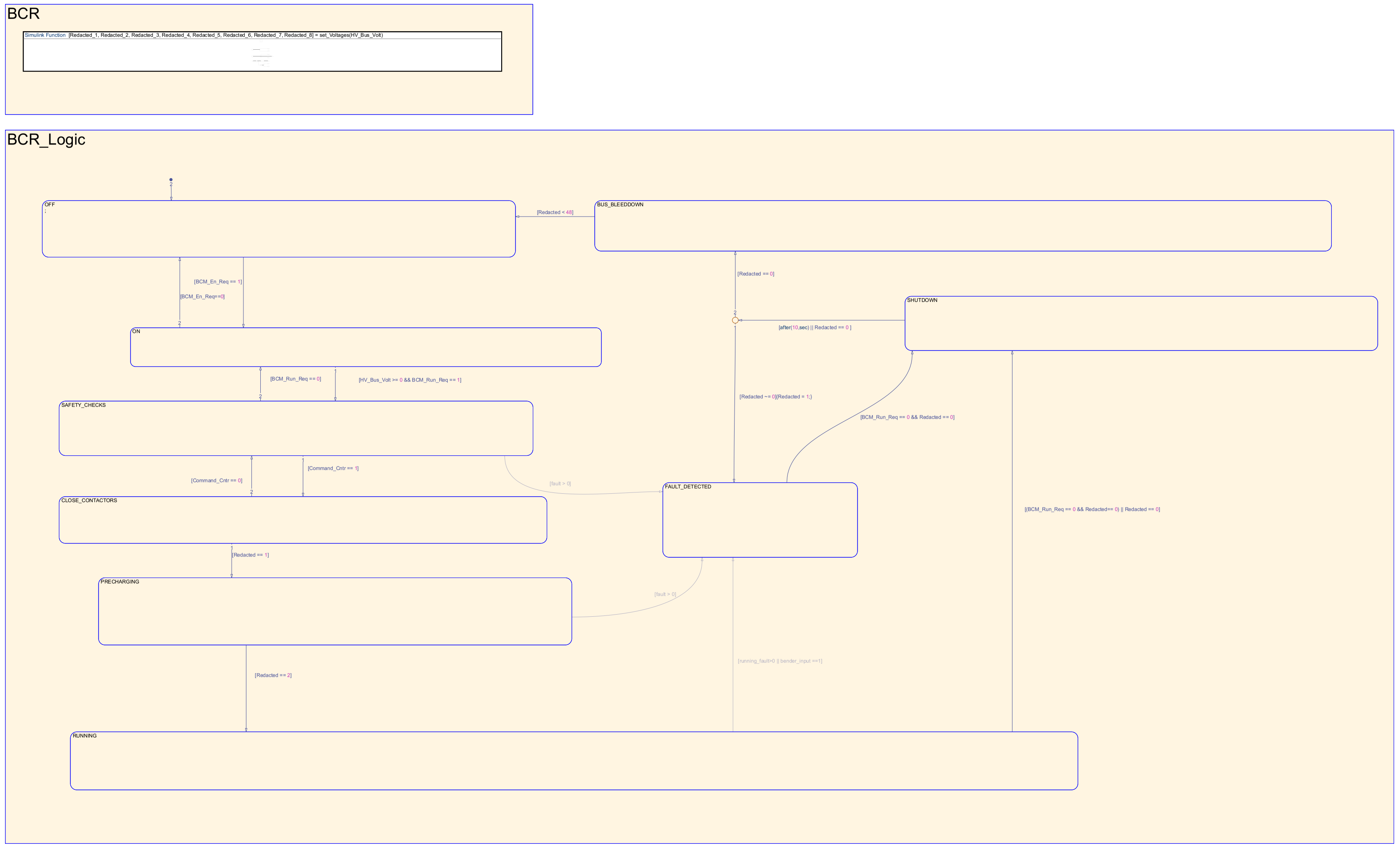

BCR
Simulink Function [Redacted_1, Redacted_2, Redacted_3, Redacted_4, Redacted_5, Redacted_6, Redacted_7, Redacted_8] = set_Voltages(HV_Bus_Volt)
BCR_Logic
OFF
BUS_BLEEDDOWN
[Redacted < 48]
[BCM_En_Req == 1]
[BCM_En_Req==0]
ON
[BCM_Run_Req == 0]
[HV_Bus_Volt >= 0 && BCM_Run_Req == 1]
SAFETY_CHECKS
[Command_Cntr == 1]
[Command_Cntr == 0]
CLOSE_CONTACTORS
[Redacted == 1]
PRECHARGING
[Redacted == 2]
RUNNING
[Redacted == 0]
SHUTDOWN
[after(10,sec) || Redacted == 0]
[Redacted ~= 0]{Redacted = 1;}
[BCM_Run_Req == 0 && Redacted == 0]
FAULT_DETECTED
[fault > 0]
[fault > 0]
[running_fault>0 || bender_input ==1]
[(BCM_Run_Req == 0 && Redacted== 0) || Redacted == 0]

Figure 5.64: Battery Control Graphical function within modified Stateflow Chart

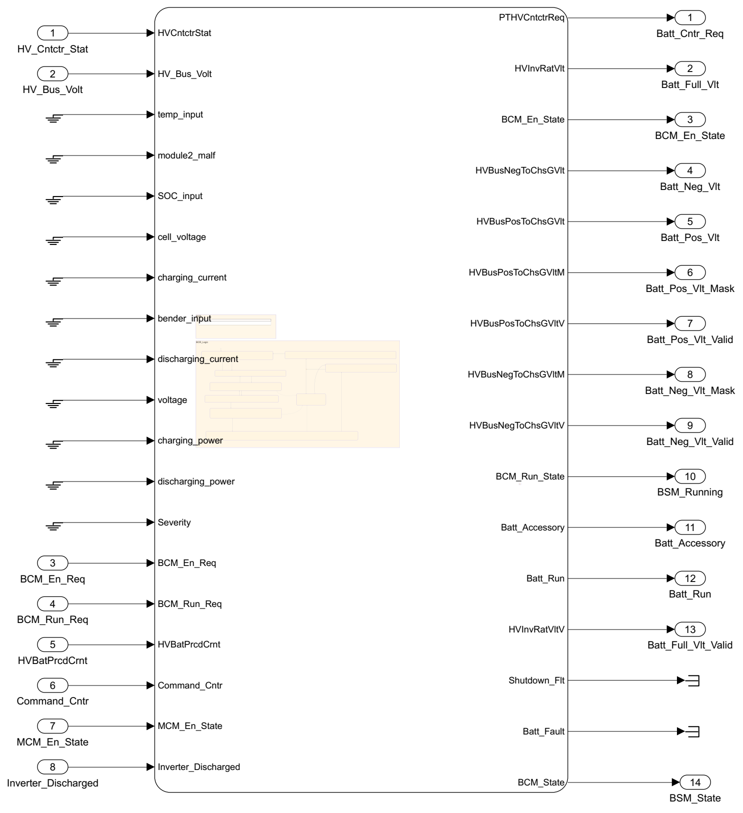


Figure 5.65: Modularized Battery Control Module

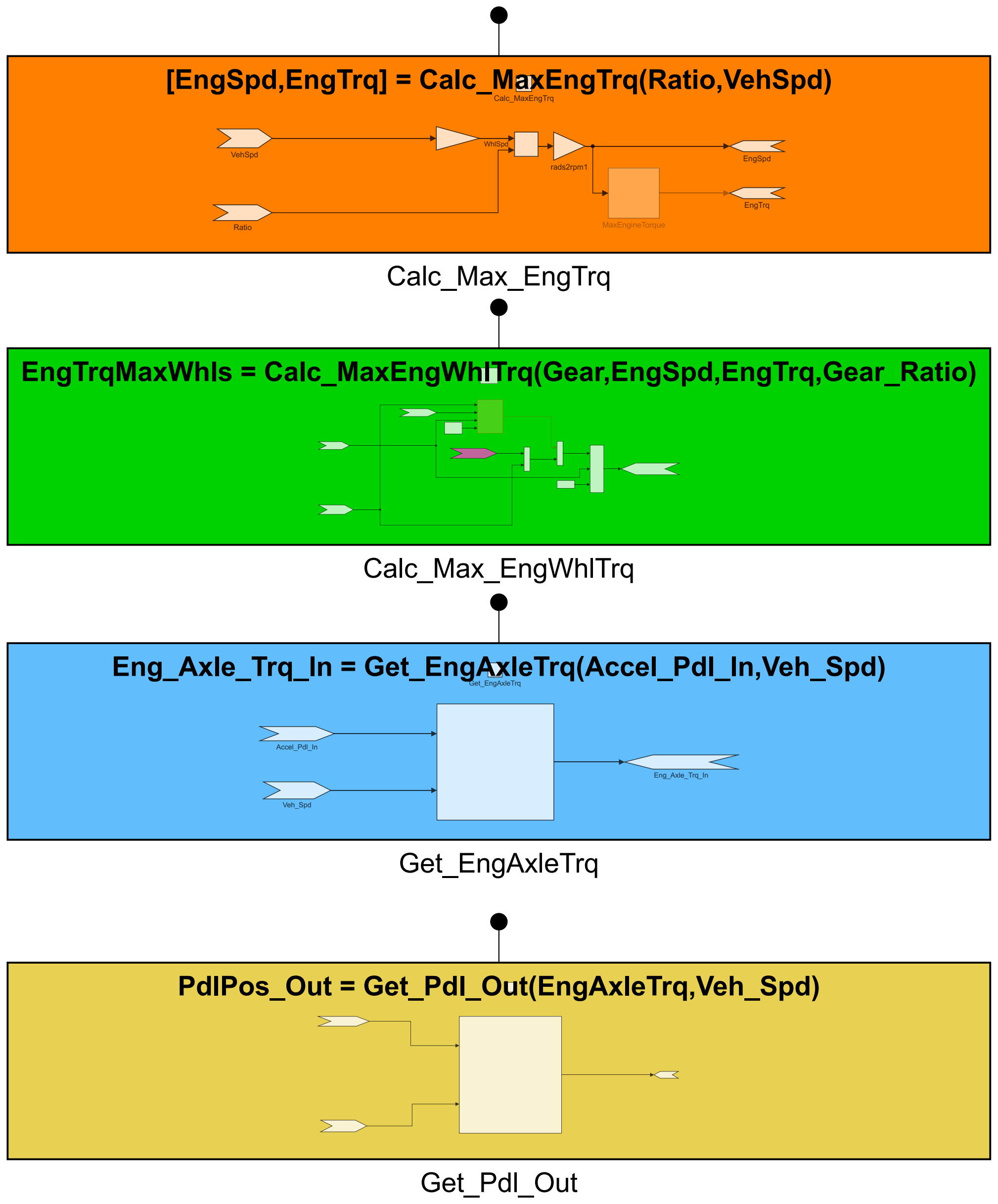

[EngSpd,EngTrq] = Calc_MaxEngTrq(Ratio,VehSpd)
Calc_MaxEngTrq
VehSpd
WhlSpd
rads2rpm1
EngSpd
EngTrq
Ratio
MaxEngineTorque
Calc_Max_EngTrq
EngTrqMaxWhls = Calc_MaxEngWhlTrq(Gear,EngSpd,EngTrq,Gear_Ratio)
Calc_Max_EngWhlTrq
Eng_Axle_Trq_In = Get_EngAxleTrq(Accel_Pdl_In,Veh_Spd)
Get_EngAxleTrq
Accel_Pdl_In
Veh_Spd
Eng_Axle_Trq_In
Get_EngAxleTrq
PdlPos_Out = Get_Pdl_Out(EngAxleTrq,Veh_Spd)
Get_Pdl_Out

FIGURE 5.66: Function callers Calc_Eng_Axle_Trq() and Calc_Pedal_Out(), located in output layer

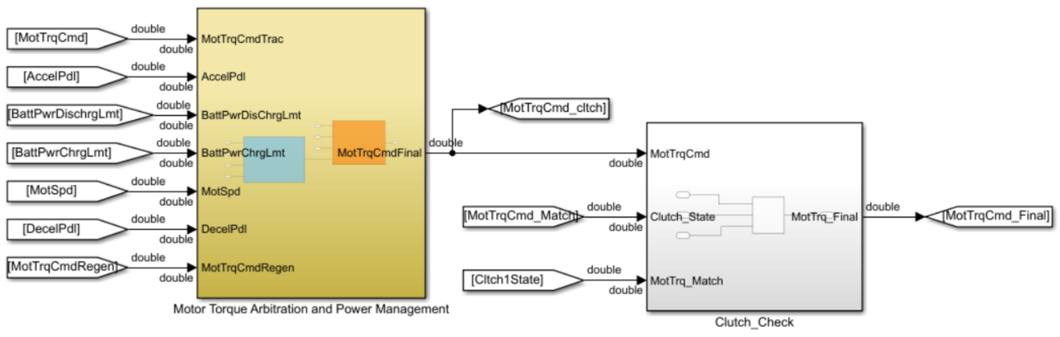


FIGURE 5.67: Original subsystem layout within output conversion layer

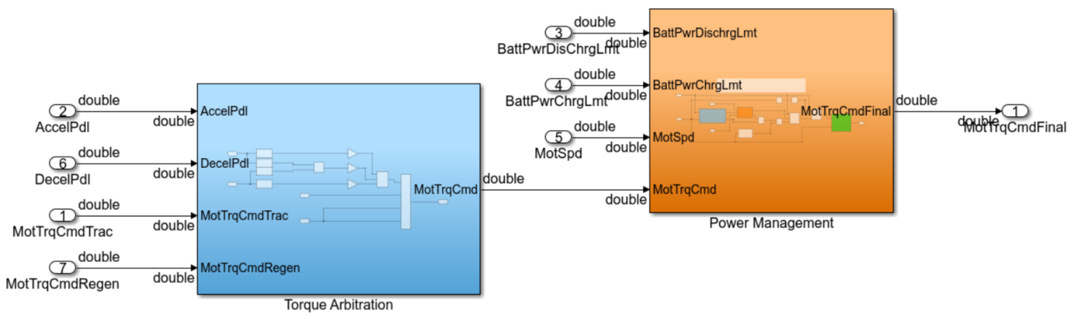


FIGURE 5.68: Internals of original subsystem containing two distinct algorithms

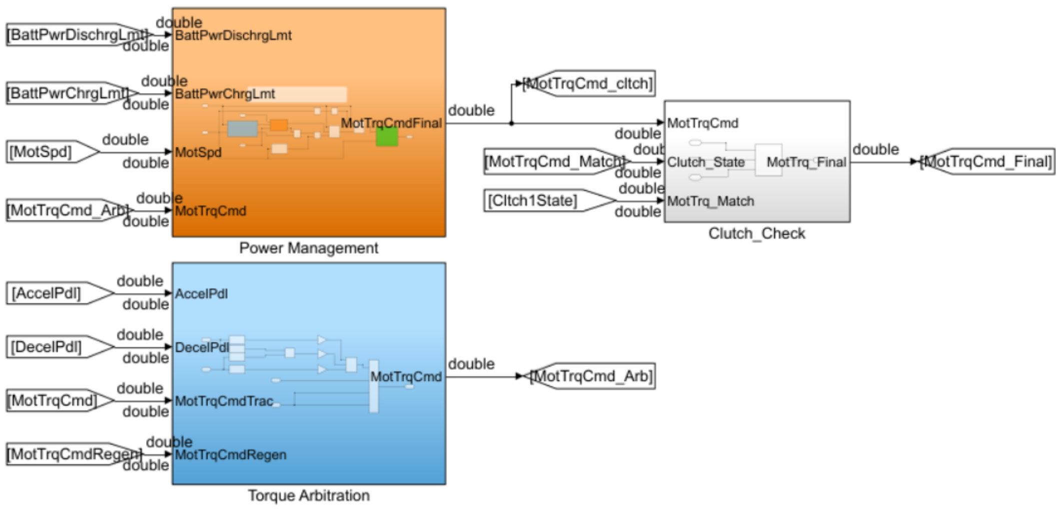


FIGURE 5.69: Output conversion layer view of the separated virtual subsystems

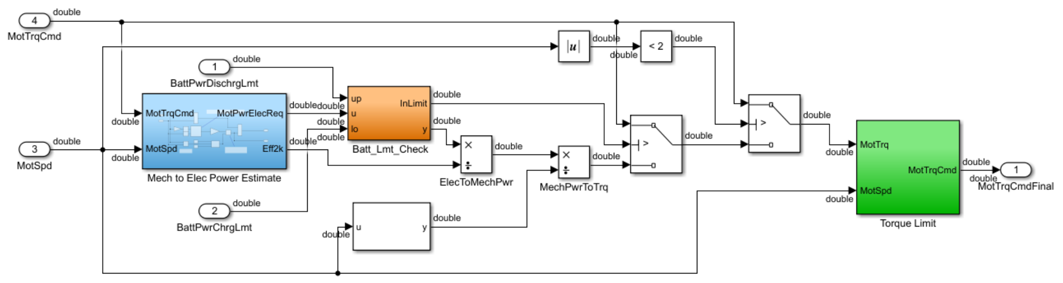
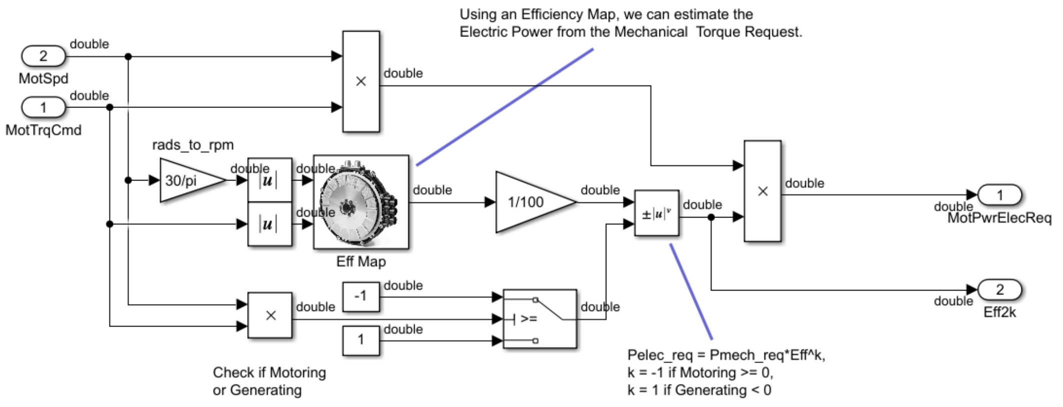


FIGURE 5.70: Power Management Algorithm in original virtual subsystem



Using an Efficiency Map, we can estimate the Electric Power from the Mechanical Torque Request.
2
MotSpd
1
MotTrqCmd
rads_to_rpm
30/pi
|u|
Eff Map
1/100
±|u|^v
-1
1
>=
Check if Motoring or Generating
Pelec_req = Pmech_req*Eff^k,
k = -1 if Motoring >= 0,
k = 1 if Generating < 0
1
MotPwrElecReq
2
Eff2k
double

FIGURE 5.71: Motor Torque to Electrical Algorithm in original virtual subsystem

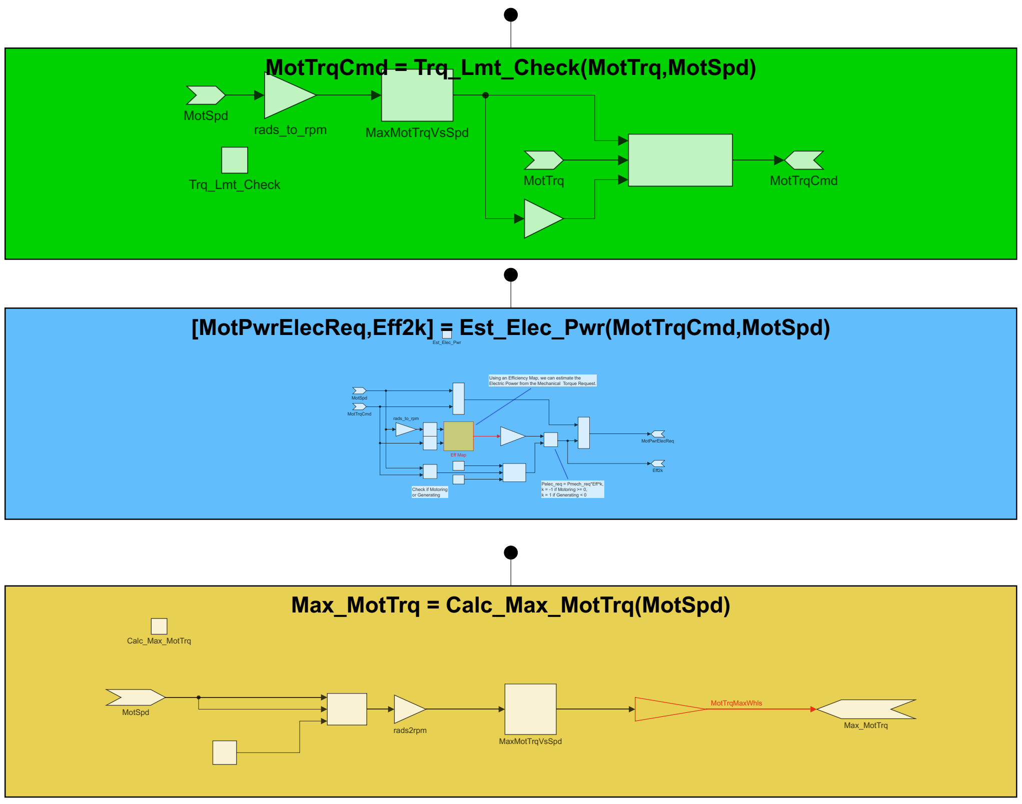

MotTrqCmd = Trq_Lmt_Check(MotTrq,MotSpd)
MotSpd
rads_to_rpm
MaxMotTrqVsSpd
Trq_Lmt_Check
MotTrq
MotTrqCmd
[MotPwrElecReq,Eff2k] = Est_Elec_Pwr(MotTrqCmd,MotSpd)
Max_MotTrq = Calc_Max_MotTrq(MotSpd)
Calc_Max_MotTrq
MotSpd
rads2rpm
MaxMotTrqVsSpd
MotTrqMaxWhls
Max_MotTrq

FIGURE 5.72: Exported Simulink functions located in the MHM Model

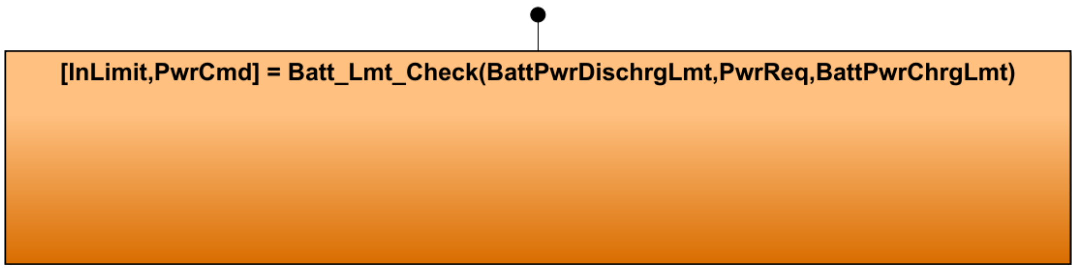

[InLimit,PwrCmd] = Batt_Lmt_Check(BattPwrDischrgLmt,PwrReq,BattPwrChrgLmt)

FIGURE 5.73: *Batt_Lmt_Check()* Simulink function defined in the BHM Model

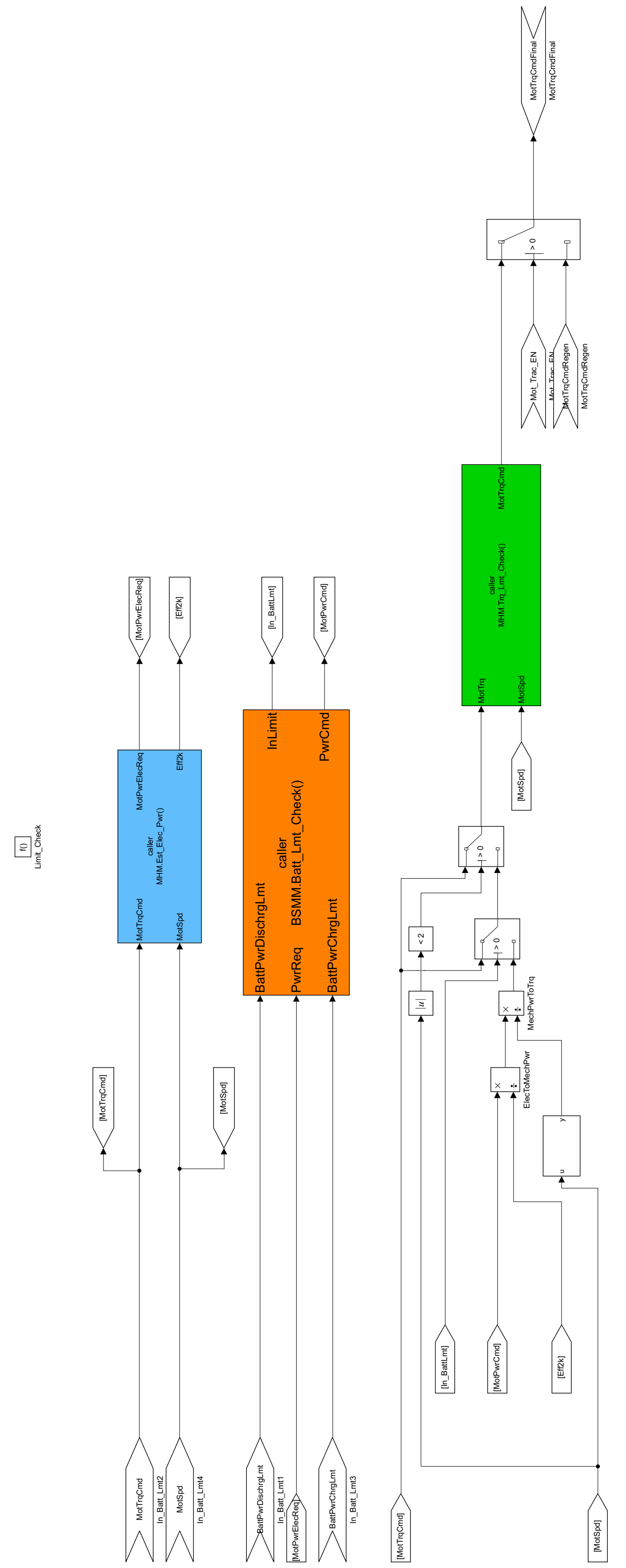


FIGURE 5.74: *Limit_Check()* Internal Algorithm with hardware function caller blocks

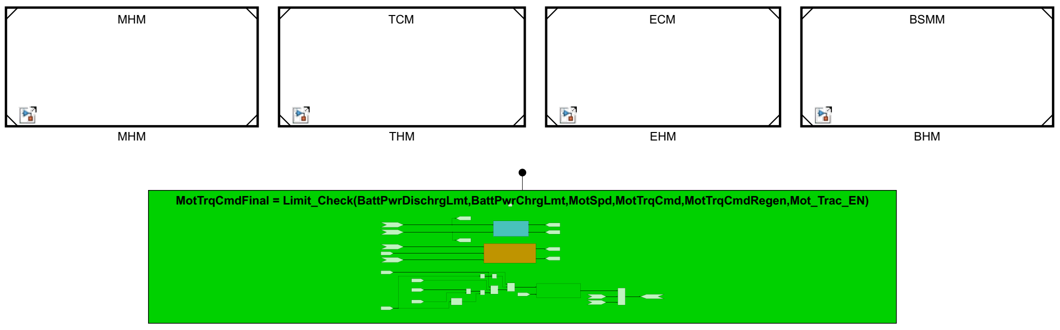

MHM
MHM
TCM
THM
ECM
EHM
BSMM
BHM
MotTrqCmdFinal = Limit_Check(BattPwrDischrgLmt,BattPwrChrgLmt,MotSpd,MotTrqCmd,MotTrqCmdRegen,Mot_Trac_EN)

FIGURE 5.75: *Limit_Check()* Simulink function defined within the HLCM

This restructuring allowed for the Regen torque arbitration algorithm to be grouped within a more appropriate module, while maintaining its execution point within the Output Conversion Layer.

5.8.4 Clutch Control Module

The original CCM contained two algorithms defined within the Output Conversion Layer; the Clutch Speed Matching and Clutch Check Subsystems. The speed matching subsystem contains the algorithm for matching the speed of the motor to the speed of the driveshaft to prepare for a clutch engagement. This function would typically be called if the driver wanted to provide torque requests, both traction and regenerative to the electric motor (Figure 5.77). It was determined that this subsystem was more accurately categorized as a component control module, and as a result was moved inside the PSCM (Section 5.7).

The virtual subsystem containing the algorithm was then converted to a scoped Simulink function using the Simulink Module Tool, highlighted blue in Figure 5.79. The Clutch check subsystem was also implemented as a scoped Simulink function and was placed within the CCM Model, highlighted in orange in Figure 5.79. The absolute time used within the Simulink PID controller block in Figure 5.78 cannot be located within a Simulink function. In order to maintain the algorithm functionality while keeping the contents within a Simulink function, the PID controller was moved outside of the Simulink function, and placed as an input to the function caller blocks within the PSCM (highlighted magenta in Figure 5.80).

5.8.5 Output Accelerator Pedal Processing

The internal algorithm of the output pedal processing subsystem is shown in Figure 5.81, with the conversion of the Engine Torque Command to a output pedal command highlighted in yellow. This lookup table contains engine hardware secrets, and was redefined as a Simulink function, *Get_Pdl_Out()*, within the EHM Model (highlighted yellow in Figure 5.66. The remaining content in Figure 5.81 contains the output pedal processing algorithm secret, and was converted into a new function,

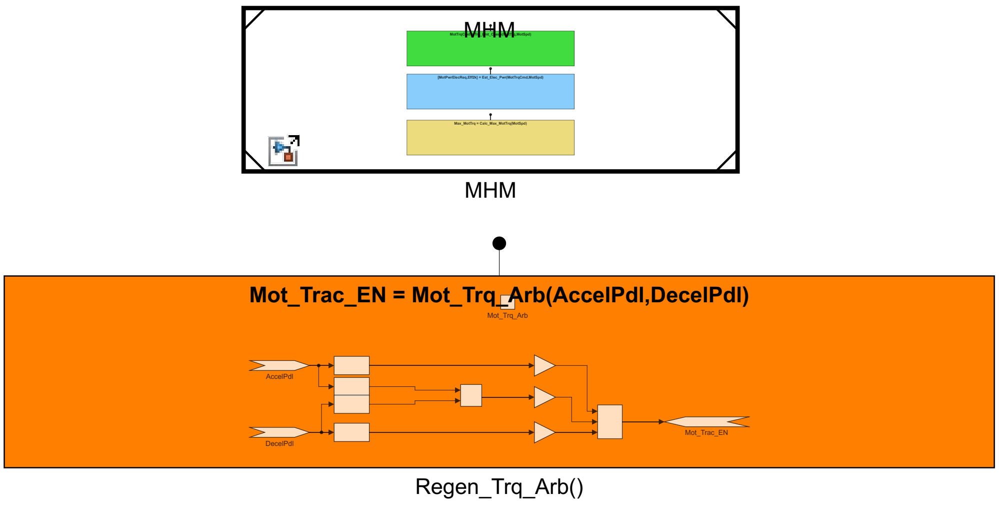


FIGURE 5.76: Regen Torque Arbitration Simulink Function defined within the RPM

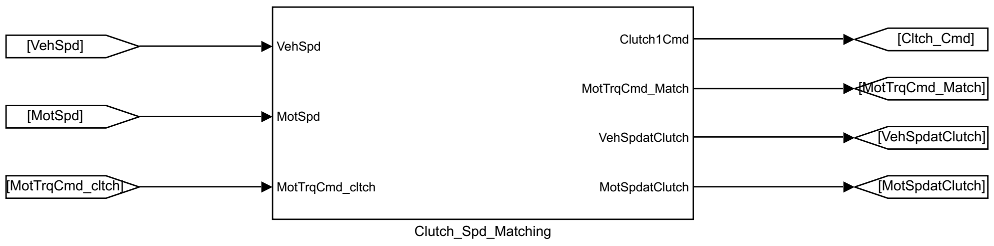


FIGURE 5.77: Clutch Speed Matching Subsystem in original model

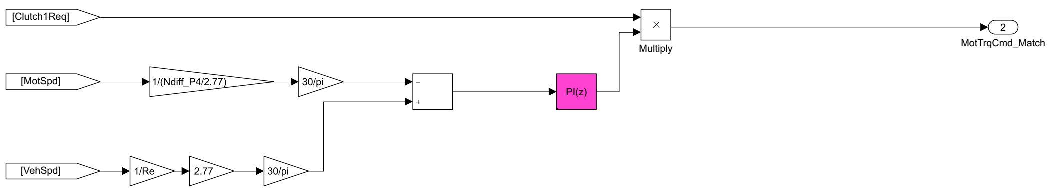


FIGURE 5.78: PID controller block located within Clutch Speed Matching Subsystem

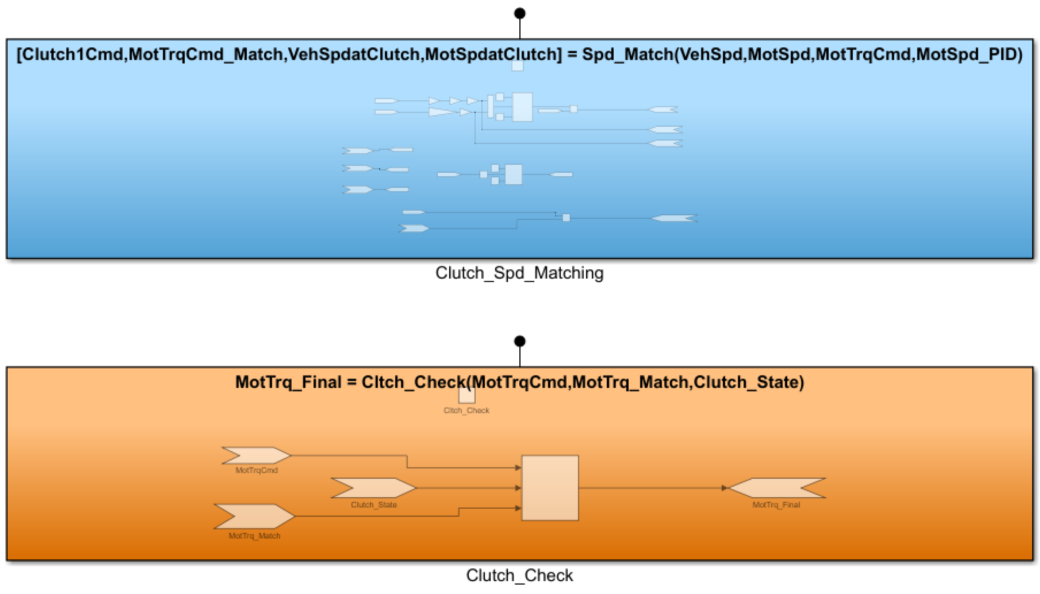

[Clutch1Cmd,MotTrqCmd_Match,VehSpdatClutch,MotSpdatClutch] = Spd_Match(VehSpd,MotSpd,MotTrqCmd,MotSpd_PID)
Clutch_Spd_Matching
MotTrq_Final = Cltch_Check(MotTrqCmd,MotTrq_Match,Clutch_State)
Cltch_Check
MotTrqCmd
Clutch_State
MotTrq_Match
MotTrq_Final
Clutch_Check

FIGURE 5.79: Clutch Simulink functions located in CCM Model

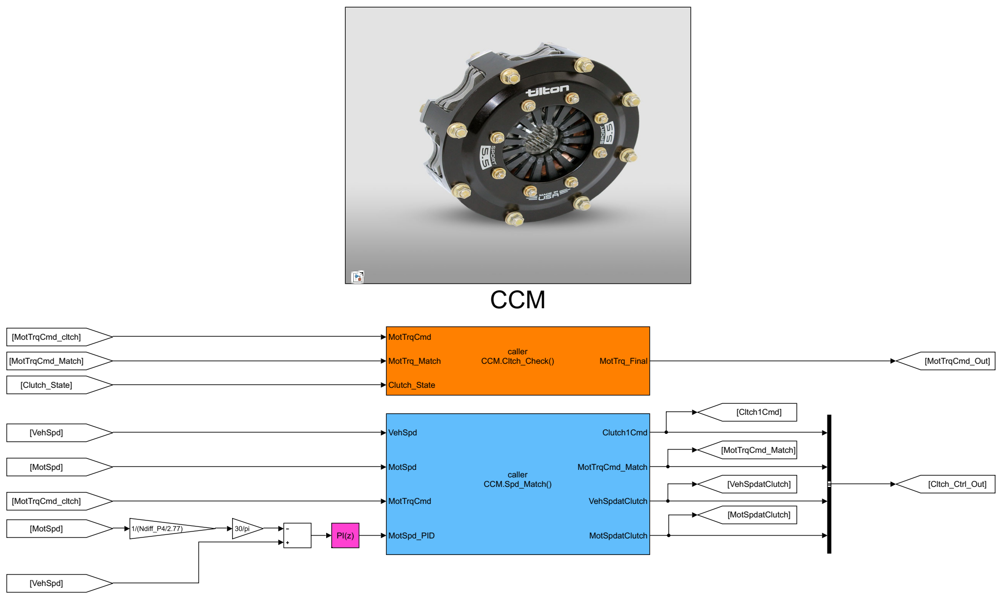


FIGURE 5.80: Clutch function caller blocks and PID block defined within PSCM

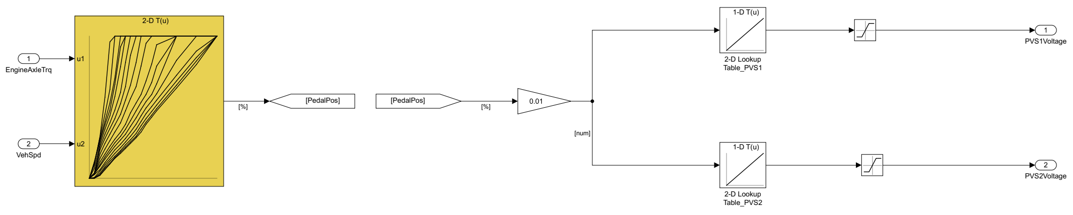


FIGURE 5.81: Input Pedal Processing Subsystem containing engine hardware secrets

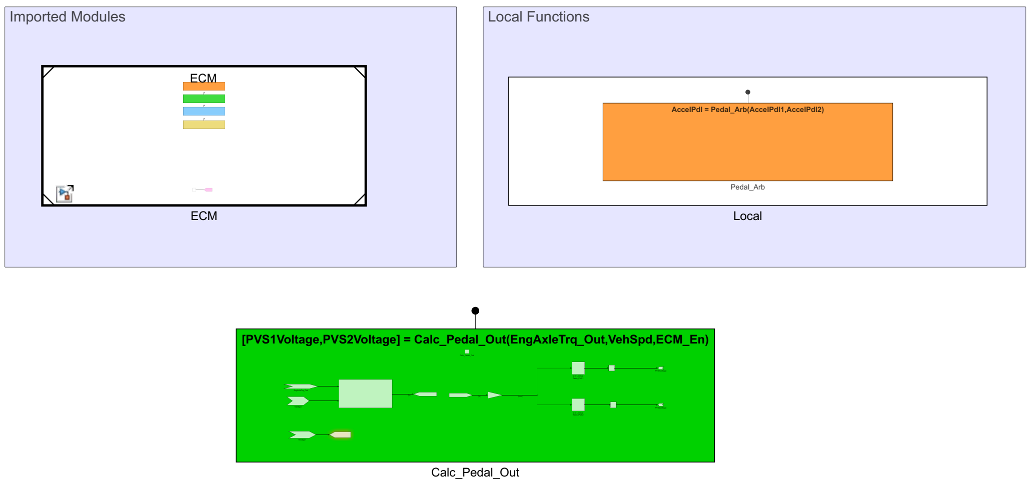

Imported Modules
ECM
ECM
Local Functions
AccelPdl = Pedal_Arb(AccelPdl1,AccelPdl2)
Pedal_Arb
Local
[PVS1Voltage,PVS2Voltage] = Calc_Pedal_Out(EngAxleTrq_Out,VehSpd,ECM_En)
Calc_Pedal_Out

FIGURE 5.82: *Calc_Pedal_Out()* Simulink function defined within the PPM

Calc_Pedal_Out(), defined within the Pedal Processing Module (PPM) (highlighted green in Figure 5.82. The corresponding function caller block was defined within *Calc_Pedal_Out()*, highlighted yellow in Figure 5.83. Finally, the corresponding output pedal processing function caller was defined at the HSC level, within the Output Conversion Layer (Figure 5.84). Similar to the input pedal processing, separation of engine hardware secrets was motivated by changes 7 and 8 in Table A2.1.

5.9 Summary

This Chapter has described the application of the Simulink Module Tool, as well as the redesign of the system decomposition and internal components. The motivation for changing the system decomposition came from a list of likely changes, described in Tables A2.1 and A2.2, as well as identifying all the system secrets contained within the existing HSC. The resulting change in system decomposition was discussed, consolidating modules based on shared secrets rather than role in the data flow process.

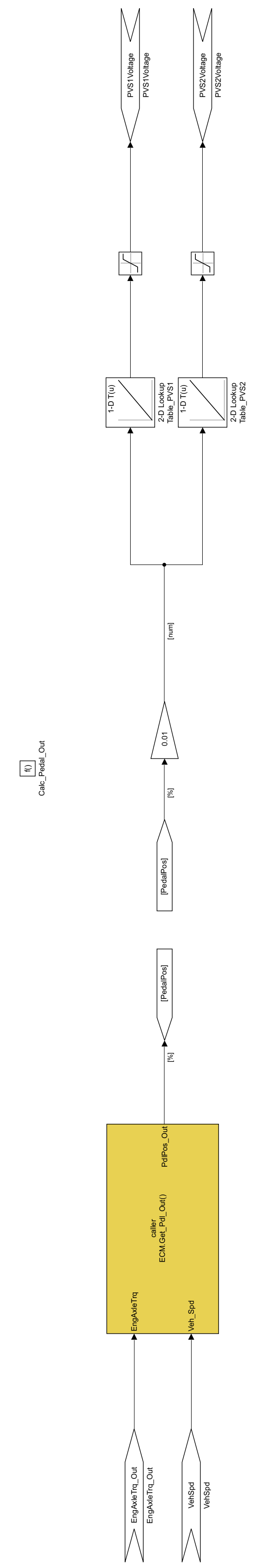


FIGURE 5.83: *Get_Pdl_Out()* engine hardware function caller defined within output pedal processing function

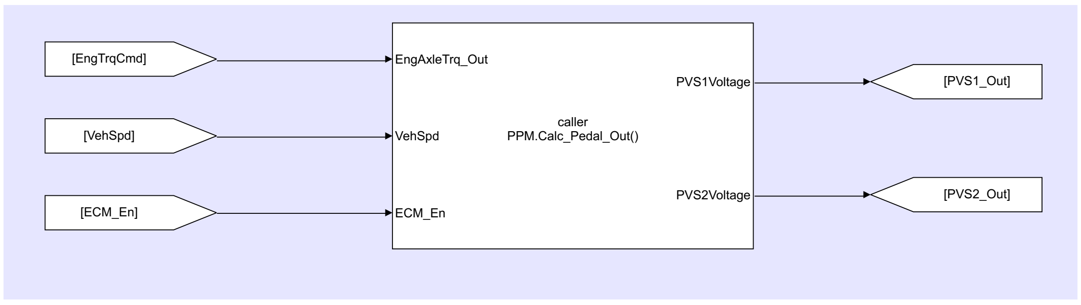


FIGURE 5.84: Output Pedal Processing Function caller defined within HSC Output Conversion Layer

The remainder of the Chapter reviewed in detail the model changes made throughout the HSC, including any additional modules that were created.

Chapter 6

Model Comparison

In this Chapter, a comparison is performed between the two models described in Chapters 4 and 5, respectively. The HSC model is evaluated before and after application of the Simulink Module Tool to determine the degree to which modularity was improved. A similar evaluation approach to that used in [35] and [36] was followed, with modularity being compared according to well known software metrics. Evaluation results will potentially determine faults within the modularization system followed, as well as indicate the potential impact that application of Simulink modular principles could have as the HSC Model development progresses. The remainder of this Chapter is as follows: Section 6.1 outlines the process used to prove functional equivalence between the original and modified models, prior to comparison. Section 6.2 describes the changes in information hiding, both in the system changeability as well as the hidden internal implementation. A comparison of the interface complexities is then discussed in Section 6.3, with a specific focus put on changes in Data Dictionaries, Model references, and exported Simulink functions. Section 6.4 evaluates changes in coupling and cohesion between the two models, and determines whether modularization led to decreased coupling and increased cohesion. The changes in cyclomatic complexity and testability are discussed in Sections 6.5 and 6.6, respectively. Finally, the results of the model comparison are summarized in Section 6.7.

Several software metrics are evaluated throughout this Chapter. In some cases, MATLAB provided tools which aided in the facilitation of this analysis, and in other cases a manual method was required. When determining system changeability, the primary analysis was performed by manually analyzing the two decompositions and Simulink modules to determine the level of propagation for a likely change. Similarly for the coupling and cohesion, a manual analysis was required involving looking at each module within the system and determining both the interdependence it had with other modules, as well as within its internal components. Cyclomatic complexity was analyzed using existing MATLAB tools, specifically the Simulink Model Advisor, which provides a detailed report of model block counts, as well as complexity. Lastly, the level of testability within each system was determined using the Simulink Design Verifier. The test generation mode of the SDV was leveraged to determine the number of objective and their level of satisfaction. The testing coverage was also determined using the SDV, providing a detailed breakdown of the condition, decision, and execution objective coverage.

6.1 Model Equivalence Verification

Using the Simulink Design Verifier to prove equivalence between the original EcoCAR and Modularized Models requires all model conditions to be met according to the SDV Documentation [29]. Certain blocks used throughout both models are not supported within the SDV Environment and were replaced with compatible alternatives [29, page 93]. While this replacement process changed the original functionality of the control system, they were necessary to provide an accurate analysis of the two models, and for the purposes of verification analysis are equivalent to the original versions. Copies of both the original EcoCAR and modular models were created for the purpose of performing the equivalence analysis while maintaining the functional integrity of the original control systems. Design Verifier test models were created for the model root and associated model references, an example of which can be seen in Figure 6.1. Functional equivalence was proven using logical equivalence operators to compare the outputs from the original EcoCAR and Modularized Models.

6.2 Information Hiding

The extent to which the two Simulink Models implement information hiding is analyzed within this section. Section 6.2.1 discusses likely model changes, and the resulting impact they have on the software system. Section 6.2.2 discusses hidden internal implementation within the models and uses a test probe model to analyze and compare them.

6.2.1 System Changeability

System Changeability evaluates the impact that a software change can have on its larger system and other modules. In a modular system with strong information hiding, changes within a given module should have little to no impact on the rest of the modules within the software system [40, page 3]. A system with poor information hiding would likely experience changes that propagate throughout several modules, resulting in increased development time and complexity. The likely changes from Tables A2.1 - A2.2 have been analyzed to determine the change propagation before and after modular decomposition. An example of changing motor hardware limits is presented in Section 6.2.1.1, and shows the process of analyzing system changeability in detail. The remainder of the analysis results can be seen in Appendix C, Tables A3.1 - A3.3, outlining the change in both modules as well as components. The term component refers to any Simulink Subsystem, function, or data dictionary that could potentially require modifications as a result of a particular software change. Changes to engine, motor, battery, and transmission hardware secrets result in reduced propagation throughout the model hierarchy after the creation of separate hardware modules. Another improvement comes from the creation of the Pedal Processing Module which prevents a change in engine pedal maps from propagating to the pedal behavioral algorithms, instead handling all engine hardware secrets within a singular Engine Hardware Module. The creation of the Data Interface Module restricts propagation of changes to the outer interface or network protocol from affecting the root level of the model, where previously both the input and output conversion

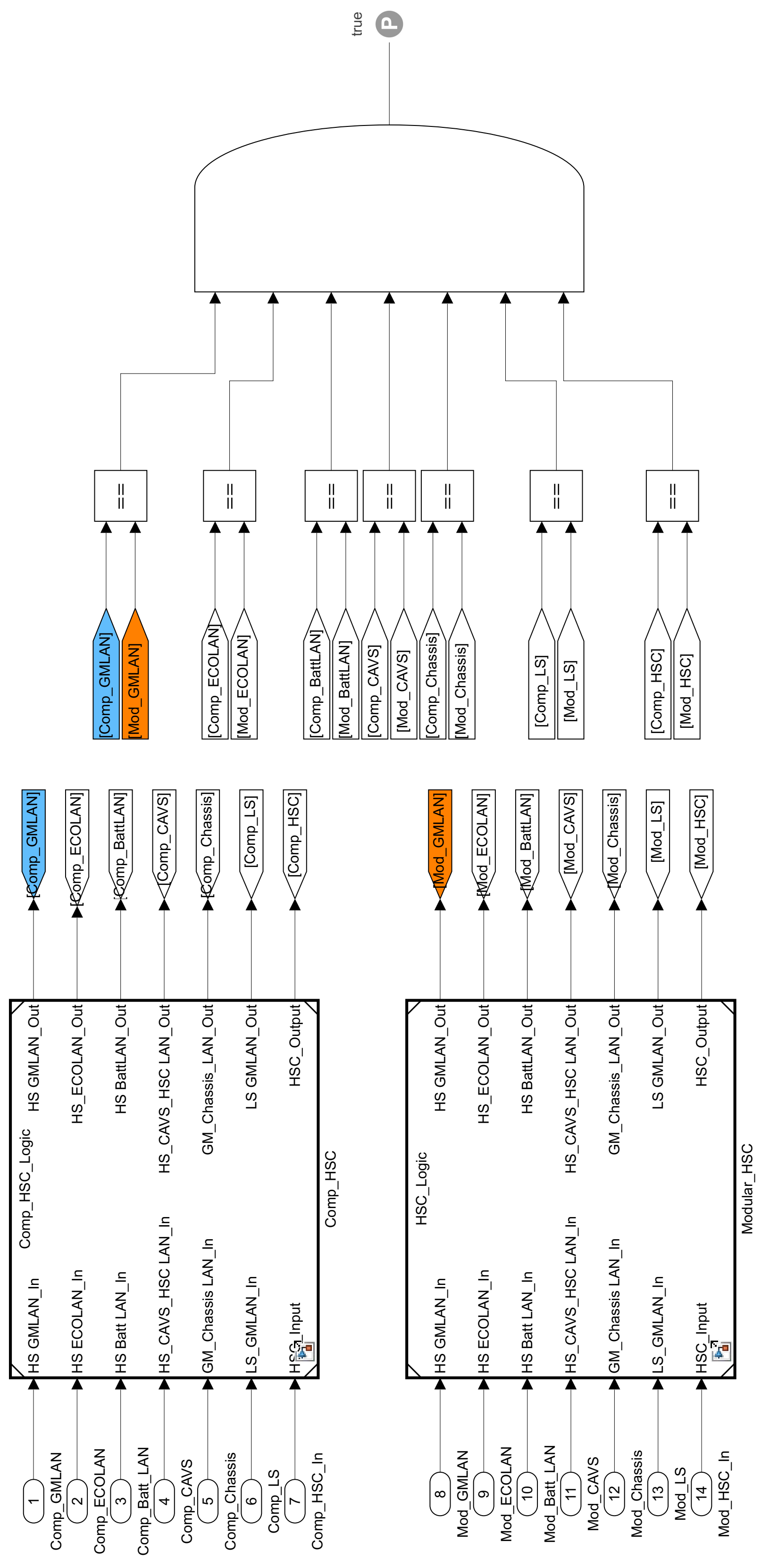


FIGURE 6.1: Example of Design Verification Model used to prove functional equivalence

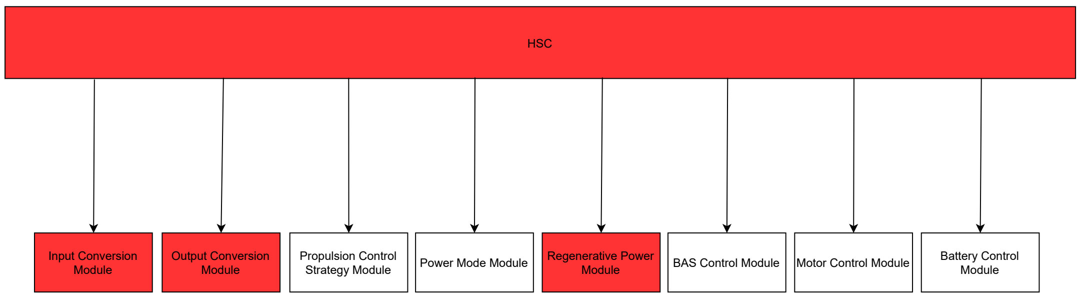


FIGURE 6.2: Root level components requiring changes within EcoCAR Model

layers would require modifications. Overall, system changeability was improved substantially, with the majority of benefits coming from the separation of component hardware and behavioral secrets.

6.2.1.1 Changing Motor Hardware Limits: Old Decomposition

To perform a system changeability analysis on the Simulink models discussed in Chapters 4 and 5, it is first necessary to identify a likely change to the system. Throughout the EcoCAR Competition, new information and data regarding the vehicle hardware components is frequently updated. As a result, a likely change to the HSC system would be the modification of the Motor hardware limits and parameters (likely change 12 in A2.1). If more accurate limit data was obtained, either through bench testing or information received from the supplier, it is a reasonable expectation that the Simulink parameters and data associated with these motor hardware limits will need to be modified in order for the system to continue to operate correctly. The propagation of changes required throughout the original EcoCAR model are shown in Figures 6.2 – 6.4. Components requiring changes are highlighted in red, and untouched components are white. Overall, four root level components required changes: the Input and Output conversion modules (virtual subsystems), the Regenerative Power Module (model) as well as the HSC root model itself. Within the input conversion layer (Figure 6.3) two virtual subsystems required modification, with the *Max Vehicle Torque Calculation* Algorithm being affected by the *Max Motor Torque Calculation*. The *Regen Torque Calculation* subsystem within the RPM also contains motor hardware limits and would be affected by the change (Figure 6.4). Lastly the output conversion layer also contains several subsystems containing motor hardware limits, including the *Power Management Algorithm*, *Electrical Power Estimation*, and *Motor Torque Limit Check*. The summary of change propagation in the EcoCAR Model can be seen in Table 6.1 in the *Original* Columns. In total, the propagated changes affected 3 modules: the input conversion, output conversion, and Regen Power Modules. In addition, eight separate components were affected, the specifics of which can be seen in Table 6.2. The original system changeability included two models, nine virtual subsystems, and four components located at the root HSC level.

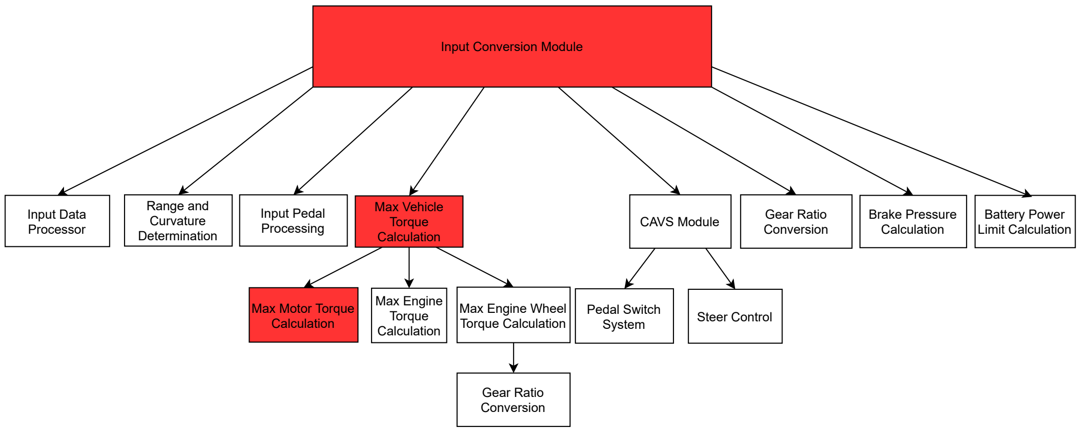

Input Conversion Module
Input Data Processor
Range and Curvature Determination
Input Pedal Processing
Max Vehicle Torque Calculation
CAVS Module
Gear Ratio Conversion
Brake Pressure Calculation
Battery Power Limit Calculation
Max Motor Torque Calculation
Max Engine Torque Calculation
Max Engine Wheel Torque Calculation
Pedal Switch System
Steer Control
Gear Ratio Conversion

FIGURE 6.3: Changes required within the Input Conversion Module

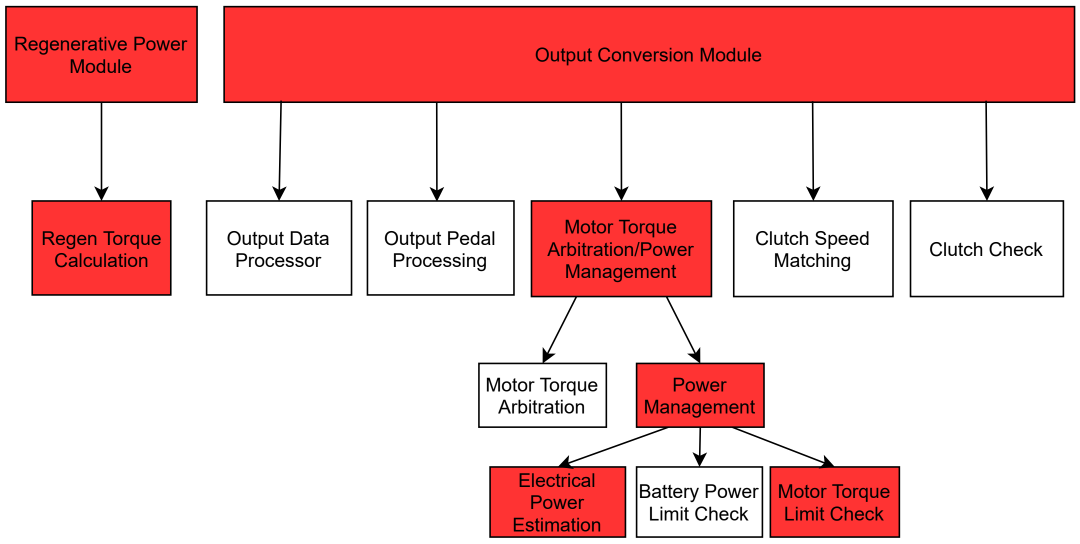


FIGURE 6.4: Changes in RPM and Output Conversion Module

Table 6.1: Motor Hardware System Changeability

Change ID	Likely Change	Modules Affected (Original)	Components Affected (Original)	Modules Affected (New)	Components Affected (New)	Module Propagation (Difference)	Component Propagation (Difference)
12	Motor Hardware parameters (more accurate limit data obtained)	3	8	1	4	-2	-4

Table 6.2: Change Propagation before and after Modularization

Component	Before	After	Difference
Models	2	1	-1
Virtual Subsystems	9	0	-9
Simulink Function	0	3	+3
Root Level Components	4	0	-4

6.2.1.2 Changing Motor Hardware Limits: New Decomposition

The propagation of changes required in the modular decomposition can be seen in Figures 6.5 and 6.6. Within the new model, all motor hardware secrets have been moved from their role within the data flow process and are contained within the Motor Hardware Module. Any changes to hardware limits or parameter values only require changes within a single Simulink Module. Additionally, each of the three hardware algorithms contained within the MHM (shown in Figure 6.6) are contained within a Simulink Function. Changes required to just the Max Motor Torque, for example, would only propagate to the *Calc_Max_MotTrq()* Simulink Function, leaving the remaining hardware algorithms within the MHM unaffected.

The summary of changes required in the modularized model can be seen in Table 6.1, in the *After* Column. The number of modules was reduced to just one (Motor Hardware Module), and the number of components was reduced by 4. The details of the change propagation in the modularized model is shown in Table 6.2. The number of models requiring changes was reduced by one, as the Regenerative Power Module no longer contains any motor related secrets. Three of the virtual subsystems from the original model were converted into Simulink Functions (Figure 6.6) and require changes. The remaining subsystems are not present in the modular decomposition, as the motor hardware secrets are only accessed with the use of Simulink function caller blocks, reducing the amount of repeated code. Finally, the number of root level components affected by the change was reduced from 3 to 0, with the removal of hardware secrets from the input and output conversion layers, as well as the transfer of hardware related parameters from a global data dictionary to a local MHM model workspace. Overall, the modularization of the Simulink Model clearly improved the system changeability and reduced the number of changes that propagate throughout the HSC.

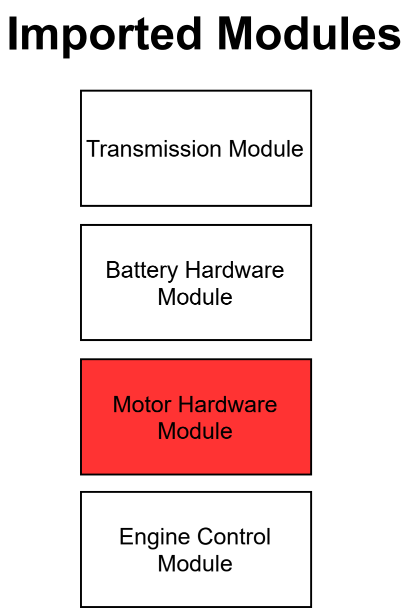


FIGURE 6.5: Changes required within the imported hardware modules

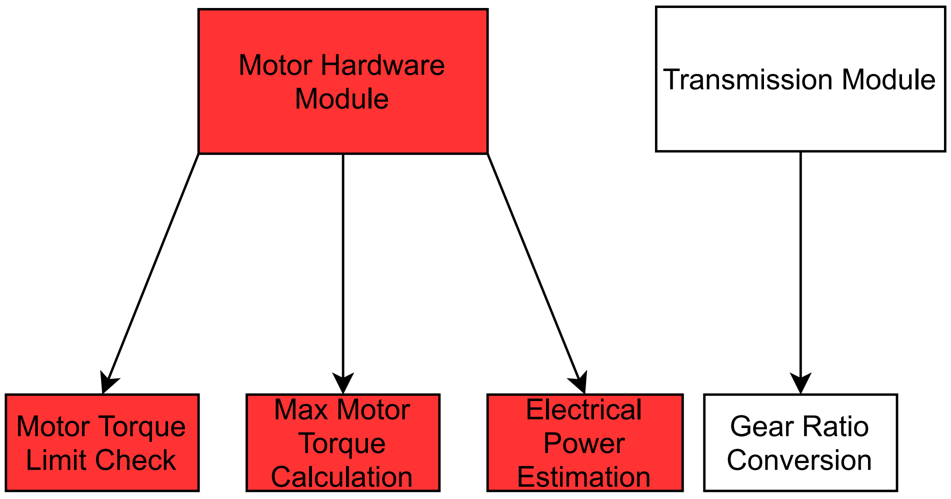


FIGURE 6.6: All changes required are located within the Motor Hardware Module

6.2.2 Hidden Internal Implementation

An approach based on the construct comparison outlined in Section 2.4 was adopted to determine the degree to which each model hides internal implementation. A test model was created for both the EcoCAR and Modular HSC Models, and probes were inserted to record test results. A total of three tests were performed and are outlined below. The first and second tests were performed to determine whether data can be implicitly passed as inputs or outputs to the *Max Vehicle Torque Calculation* Subsystem. The third test analyzed the degree to which components within each model exposed their internal parameters. The expected results are that implicit input and output data will be successfully passed to the virtual subsystem located within the EcoCAR Model but will not be successfully passed to the Simulink Function located within the Modularized Model. In addition, the use of global data dictionaries within the EcoCAR Model will likely expose more data then with the use of model workspaces within the Modularized Model.

6.2.2.1 Goto/From as Implicit Input

The first test performed involved the implicit passing of input data using global Goto/From tags. A constant block was inserted into the root level of the EcoCAR Model, and used as a test input to a Global Goto Tag *Test_In*, highlighted in magenta in Figure 6.7. A corresponding Global From Tag, *Test_In*, was placed within the internal *Max Vehicle Torque Calculation* virtual subsystem. The Global From Tag was then connected to a Simulink display block, highlighted in magenta in Figure 6.8. Simulation of the test model was successful, with the output display block showing the value sent by the constant block test input. This confirms that data can be passed implicitly within the EcoCAR Model using a global Goto/From Tag combination.

Identical constant test inputs and Global Goto/From tags were placed in the Modularized Model, and within the *Calc_MaxEngTrq()* Simulink function. The test input connected to the Global Goto Tag, and the display output connected to the Global From Tag are highlighted in magenta in Figures 6.9 and 6.10, respectively. Simulation of the test model resulted in an error, shown in Figure 6.11. The Goto/From Tags crossed non-virtual subsystem boundaries, which is not permitted using non-state output ports. This test is sufficient to show that any instance of Simulink Functions will restrict implicit data flow using Global Goto/From Tags.

6.2.2.2 Goto/From as Implicit Output

The second test performed involved the implicit passing of output data using global Goto/From tags. The constant test input and Global Goto Tag were placed within the Max Engine Torque Calculation virtual subsystem, highlighted in magenta in Figure 6.12. The Global From Tag and display output were placed at the root level of the EcoCAR Model, highlighted in magenta in Figure 6.13. The EcoCAR test model was run successfully, and the test output displayed the correct input value. This test confirmed that data can be passed implicitly as output from within internal subsystems.

The test was also performed on the modularized model, with data being passed implicitly from within the *Calc_MaxEngTrq()* Simulink Function. The test input

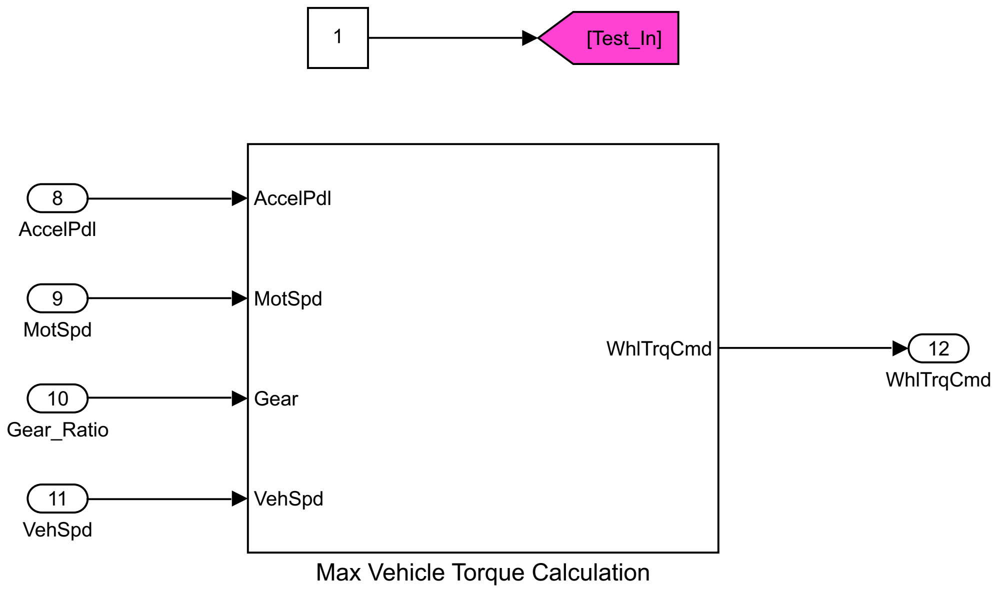


FIGURE 6.7: Implicit Input passed to Max Vehicle Torque Calculation within EcoCAR Model

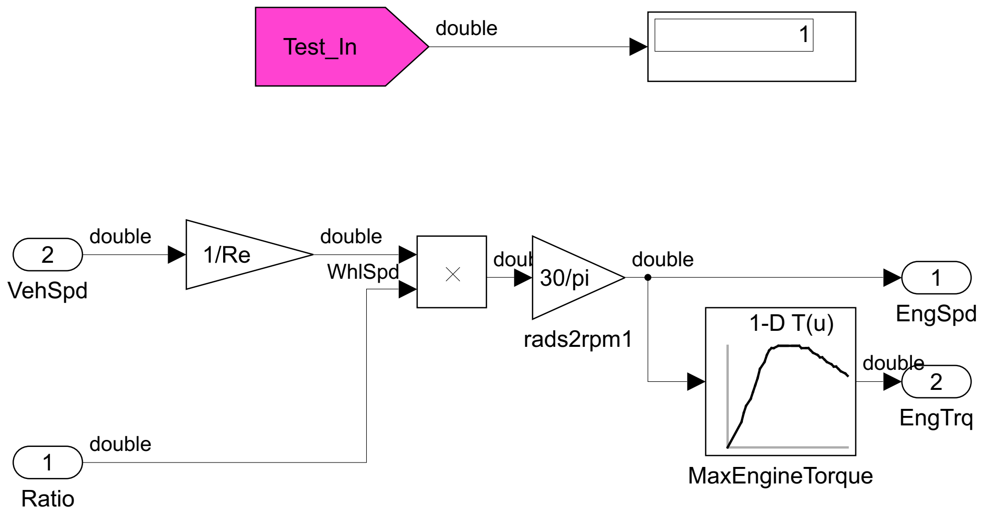


FIGURE 6.8: Implicit Input received within Max Engine Torque Calculation subsystem (Original EcoCAR Model)

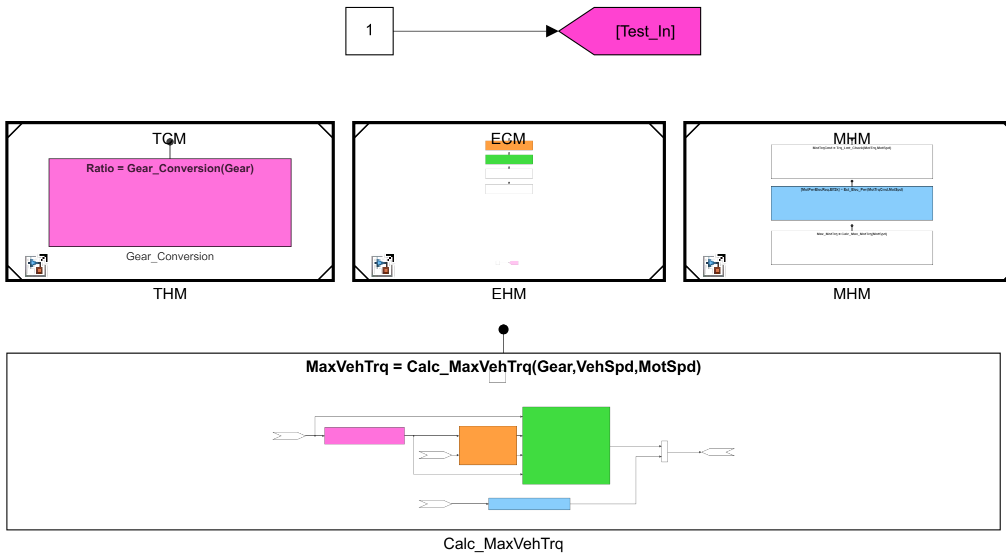

1
[Test_In]
TCM
Ratio = Gear_Conversion(Gear)
Gear_Conversion
THM
ECM
EHM
MHM
MHM
MaxVehTrq = Calc_MaxVehTrq(Gear,VehSpd,MotSpd)
Calc_MaxVehTrq

FIGURE 6.9: Implicit Input passed to Simulink Function within Modularized Model

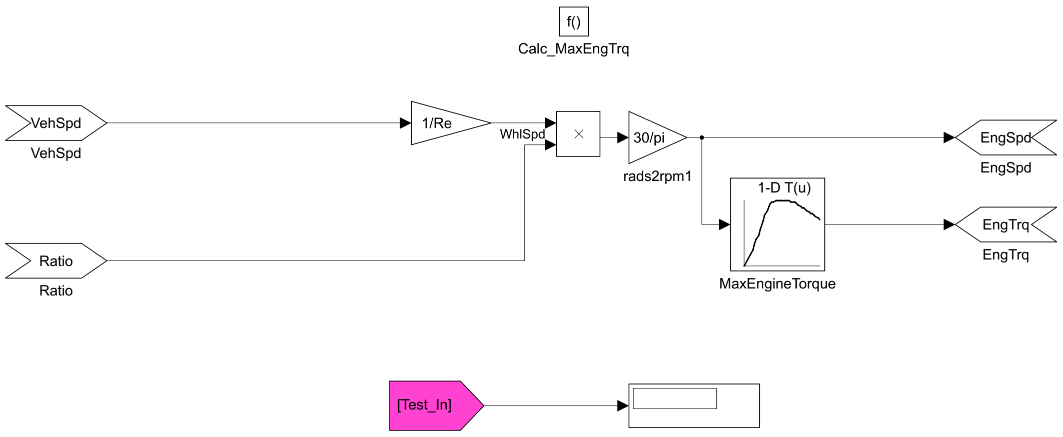
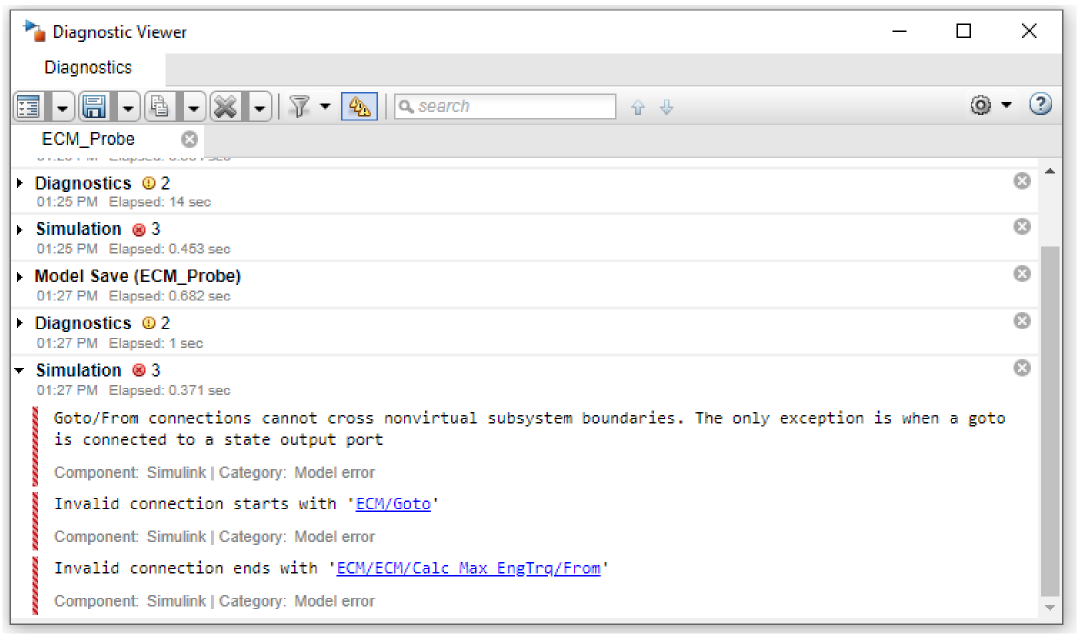


FIGURE 6.10: Implicit Input does not pass-through Simulink Function boundaries



```
Diagnostics  2
01:25 PM  Elapsed: 14 sec
Simulation  3
01:25 PM  Elapsed: 0.453 sec
Model Save (ECM_Probe)
01:27 PM  Elapsed: 0.682 sec
Diagnostics  2
01:27 PM  Elapsed: 1 sec
Simulation  3
01:27 PM  Elapsed: 0.371 sec
Goto/From connections cannot cross nonvirtual subsystem boundaries. The only exception is when a goto
is connected to a state output port
Component: Simulink | Category: Model error
Invalid connection starts with 'ECM/Goto'
Component: Simulink | Category: Model error
Invalid connection ends with 'ECM/ECM/Calc_Max_EngTrq/From'
Component: Simulink | Category: Model error
```

FIGURE 6.11: Error received when attempted to pass implicit input to a Simulink Function

connected to the Global Goto Tag, and the display output connected to the Global From Tag are highlighted in magenta in Figures 6.13 and 6.14, respectively. Simulation of the output test model also resulted in an error, shown in Figure 6.15. For both input and output data flow, Simulink Functions successfully restrict the implicit passing of data using Global Tags. It is clear from these test results that the conversion from virtual subsystems to Simulink functions resulted in an increased restriction to implicit data flow.

6.2.2.3 Accessing Internal Data

The third and final test that was performed involved attempting to access internal data parameters within HSC system components. Within the EcoCAR Model, the calculation of maximum engine torque is in the Max Vehicle Torque Calculation virtual subsystem. The Modularized Model converted this virtual subsystem into a Simulink function, *Calc_MaxEngTrq()*, which is located within the EHM Model. In both cases, the max engine torque is determined using a 1-D Lookup table Simulink Block, highlighted yellow in Figures 6.17 and 6.18, respectively. This Lookup Table is dependent upon the parameter *f_tbrake_n_bpt* to output the correct engine torque value. A test procedure was developed to determine whether the *f_tbrake_n_bpt* parameter could be accessed at the root level of both the EcoCAR and Modularized Models. If access is permitted, then the level of information hiding is reduced significantly. Data used within a given module or component should have as local of a scope as possible to restrict root level model change propagation. Exposing a parameter used locally within a module also increases the possibility of unintended modification.

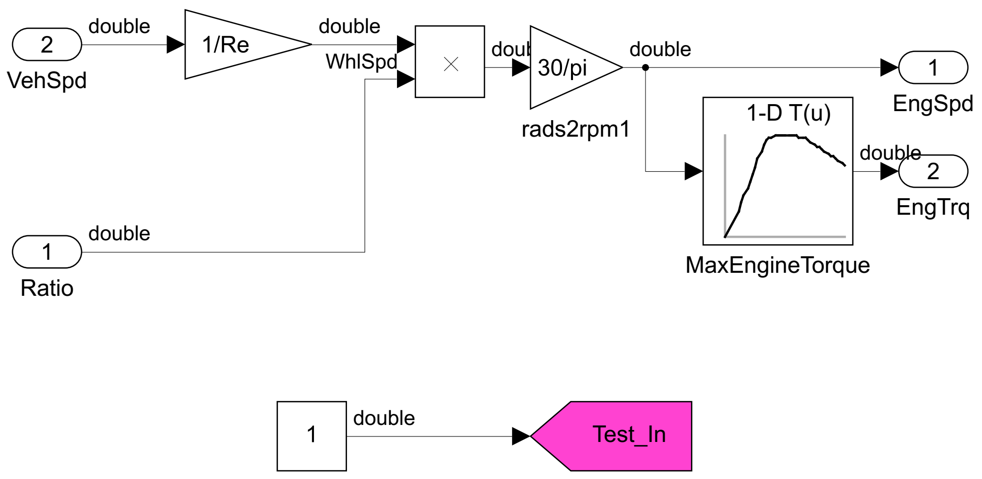


FIGURE 6.12: Implicit Output passed from Max Engine Torque Calculation Subsystem within EcoCAR Model

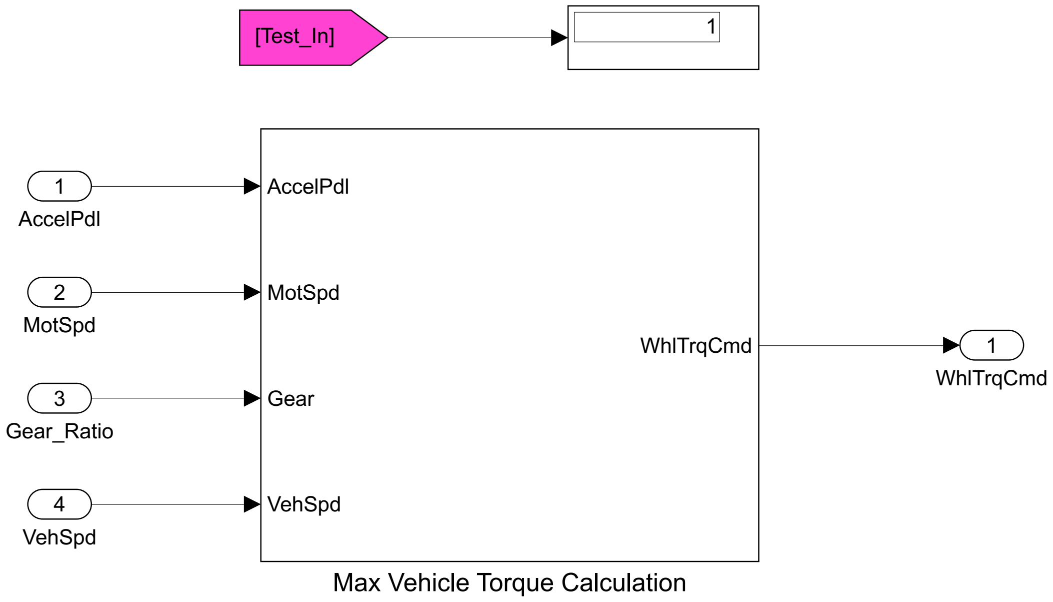


FIGURE 6.13: Implicit output received from virtual subsystem (Original EcoCAR Model)

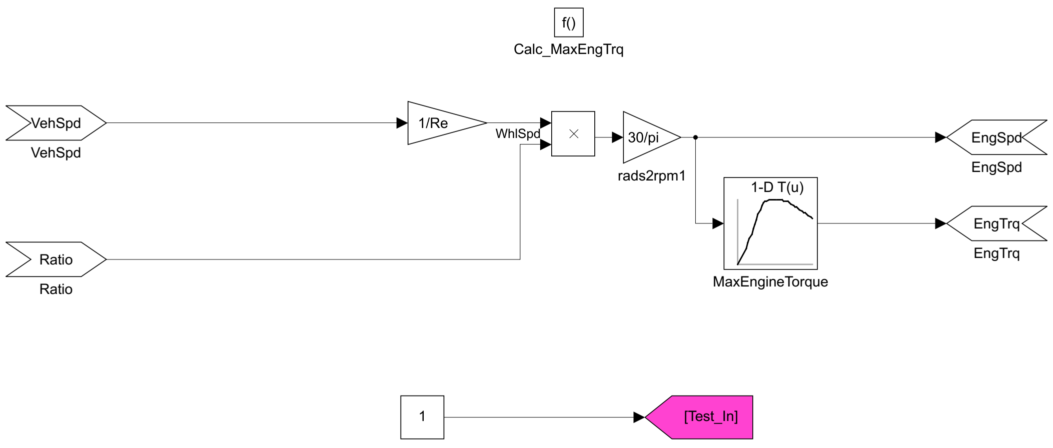


Figure 6.14: Implicit Output passed from Simulink Function within Modularized Model

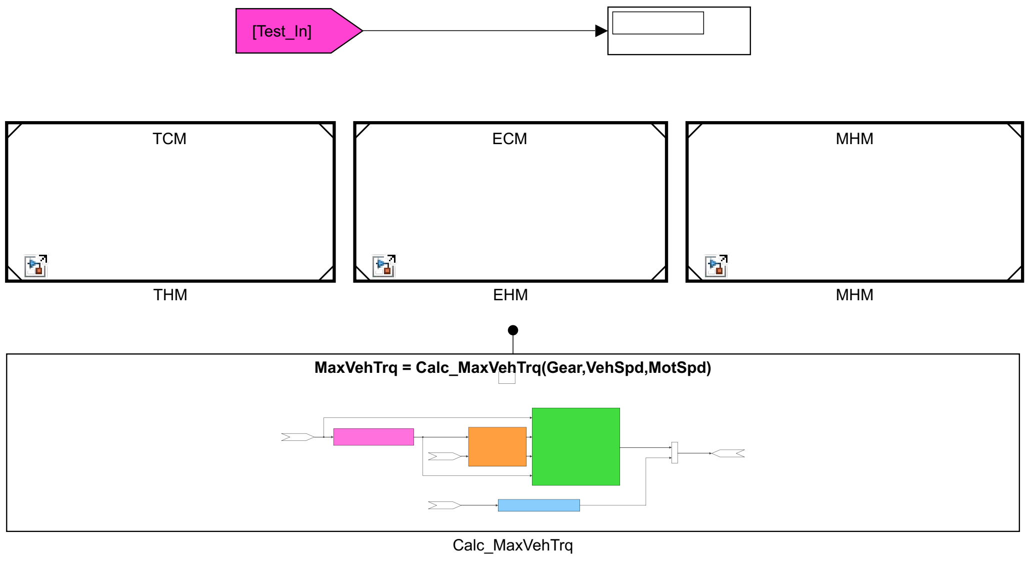
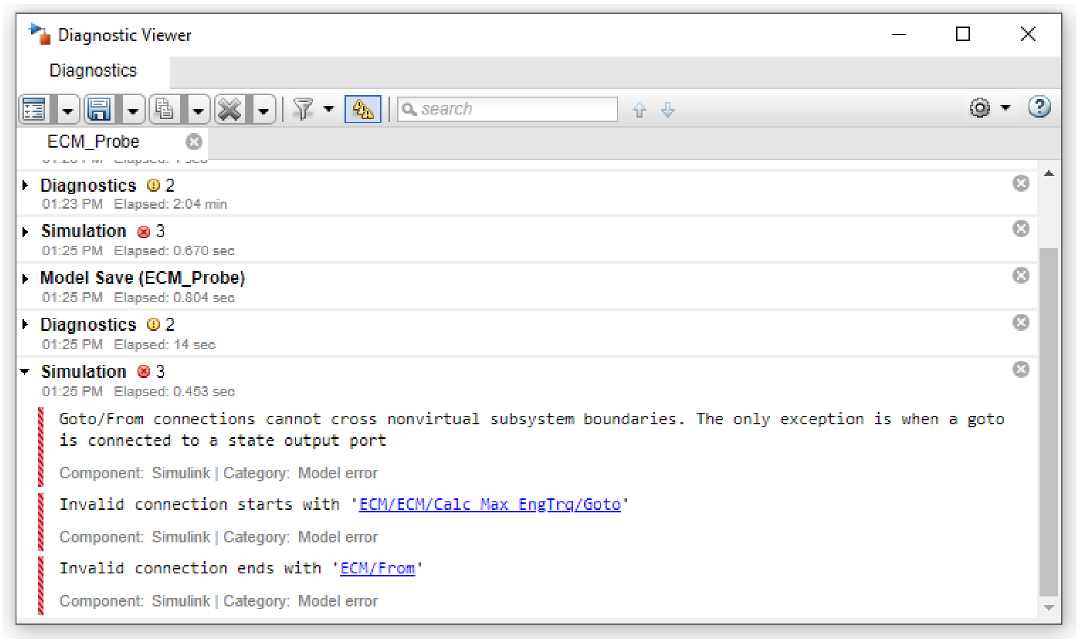


Figure 6.15: Implicit Output does not pass-through Simulink Function boundaries



```
Diagnostics ⓘ 2
01:23 PM  Elapsed: 2:04 min
Simulation ⊗ 3
01:25 PM  Elapsed: 0.670 sec
Model Save (ECM_Probe)
01:25 PM  Elapsed: 0.804 sec
Diagnostics ⓘ 2
01:25 PM  Elapsed: 14 sec
Simulation ⊗ 3
01:25 PM  Elapsed: 0.453 sec
Goto/From connections cannot cross nonvirtual subsystem boundaries. The only exception is when a goto
is connected to a state output port
Component: Simulink | Category: Model error
Invalid connection starts with 'ECM/ECM/Calc Max EngTrq/Goto'
Component: Simulink | Category: Model error
Invalid connection ends with 'ECM/From'
Component: Simulink | Category: Model error
```

FIGURE 6.16: Error received when attempted to pass implicit output from a Simulink Function

This problem would be particularly prevalent in large scale models where multiple developers are accessing the root level parameters simultaneously. Accidental changes to parameters with a global scope could result in a multiple functions or modules producing inaccurate results.

In both models a constant block was created as a test input at the root level, and connected to the *MaxEngineTorque* Lookup Table block, highlighted yellow in Figures 6.19 and 6.20, respectively. The output engine torque value of the Lookup Table was connected to a display block and used as an indication of whether the *f_tbrake_n_bpt* was successfully accessed. Simulation of the EcoCAR Model was successful and resulted in an output engine torque value of 0.1019. The *f_tbrake_n_bpt* parameter is defined within a Simulink Data Dictionary linked to the root model. As a result, the scope of the parameter is global, and can be referenced or modified in any model which uses the data dictionary. In the case of the Modularized Model, Simulation resulted in an error, shown in Figure 6.21. The *f_tbrake_n_bpt* parameter was not recognized at the root level of the model, and therefore the Lookup Table could not produce an engine torque output. The parameter was defined within the EHM model workspace, and as a result its scope is restricted. An attempt to access or modify the *f_tbrake_n_bpt* parameter from anywhere outside of the EHM model will result in an error, preventing unwanted changes and conflicts. By following modularization guidelines outlined in Section 2.5.3, local data within each module has been restricted from external access through definition in the model workspace. All parameters previously defined within Data Dictionary Objects were either redefined within a module model workspace, or added to the single global data dictionary, *Global.sldd*. These model changes resulted in increased restriction to internal data, and as a result, improved

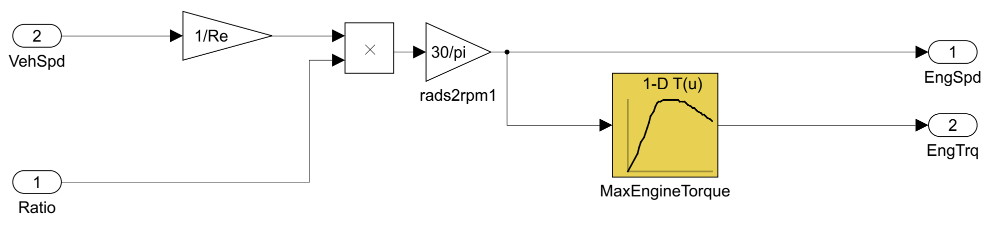


FIGURE 6.17: Engine Lookup Table containing Parameter defined in Data Dictionary (Original EcoCAR Model)

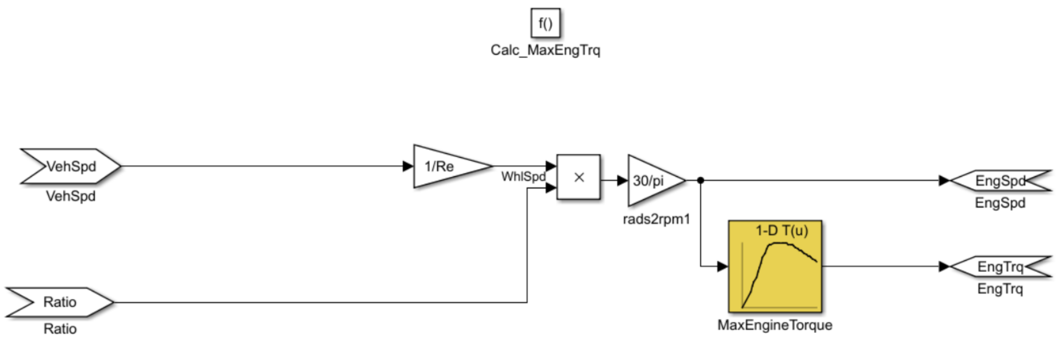


FIGURE 6.18: Engine Lookup Table containing Parameter defined in model workspace of EHM (Modularized Model)

information hiding.

6.3 Interface Complexity

The interface and dependency information for both the EcoCAR and Modularized versions of the HSC Model can be seen in Figure 6.22. The seven inputs and outputs represent the six CAN channels communicating with the HSC as well as an addition physical signal output. Following **Guideline 4 (use of base workspace)**, the number of data dictionary dependencies reduced from 5 to 1. The data dictionary objects linked within the MCM, BSMM, PCSM, and BASCM were removed, and all the containing signals were defined within each module's respective model workspace. Any signal values which were required in multiple Simulink modules were defined within a single data dictionary, *Global.sldd*. This reduced the interface complexity of all modules that previously required a separate data dictionary, an example of the motor modules can be seen in Figure 6.23. The creation of the MHM reduced the Inputs of the MCM by 2, and created 3 additional exports via scoped Simulink functions. Overall, the use of the Simulink Module Structure resulted in reduced data storage dependencies but increased the number of model references. This increase in model references is expected, due to the nature of the modularization process. The

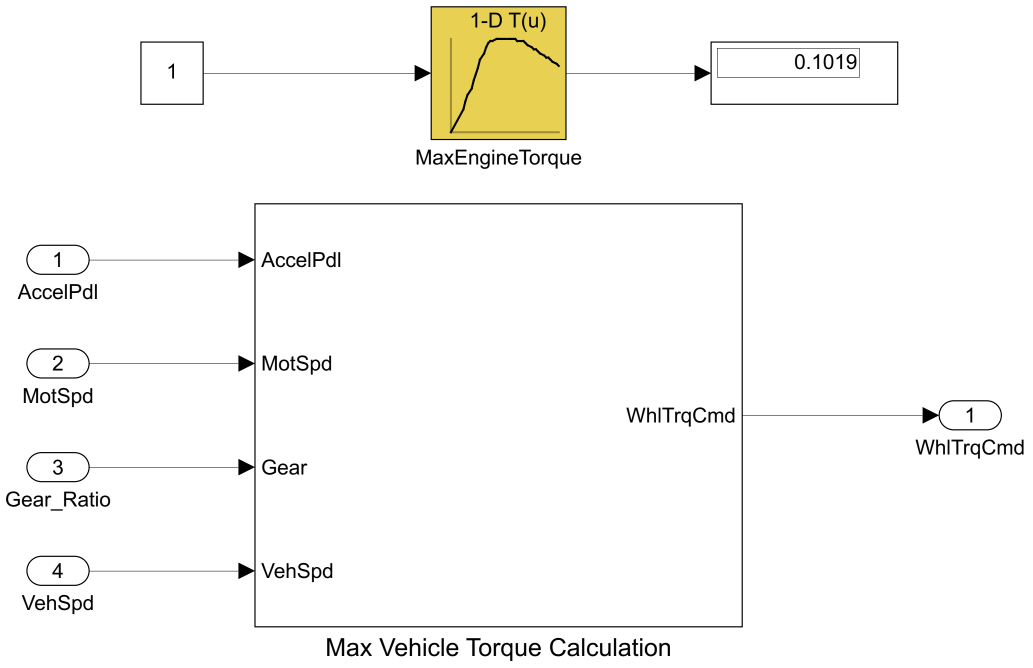


FIGURE 6.19: Internal Data accessible outside of virtual subsystem within EcoCAR Model

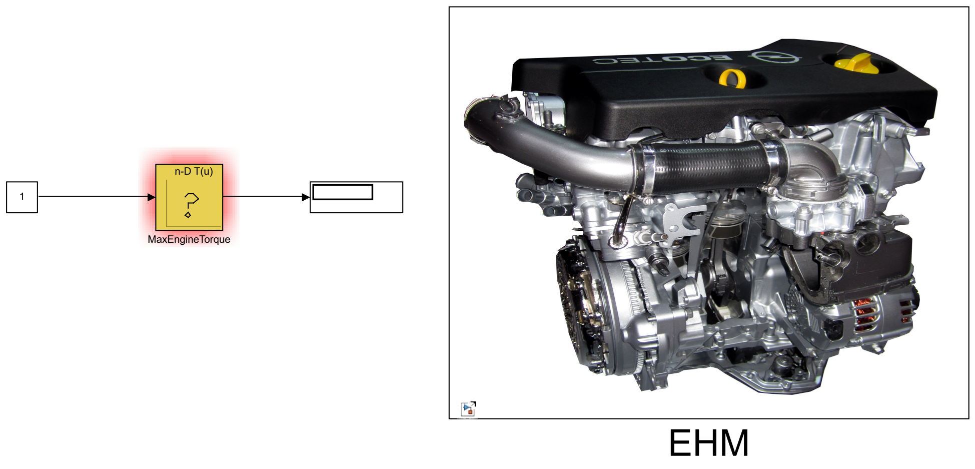
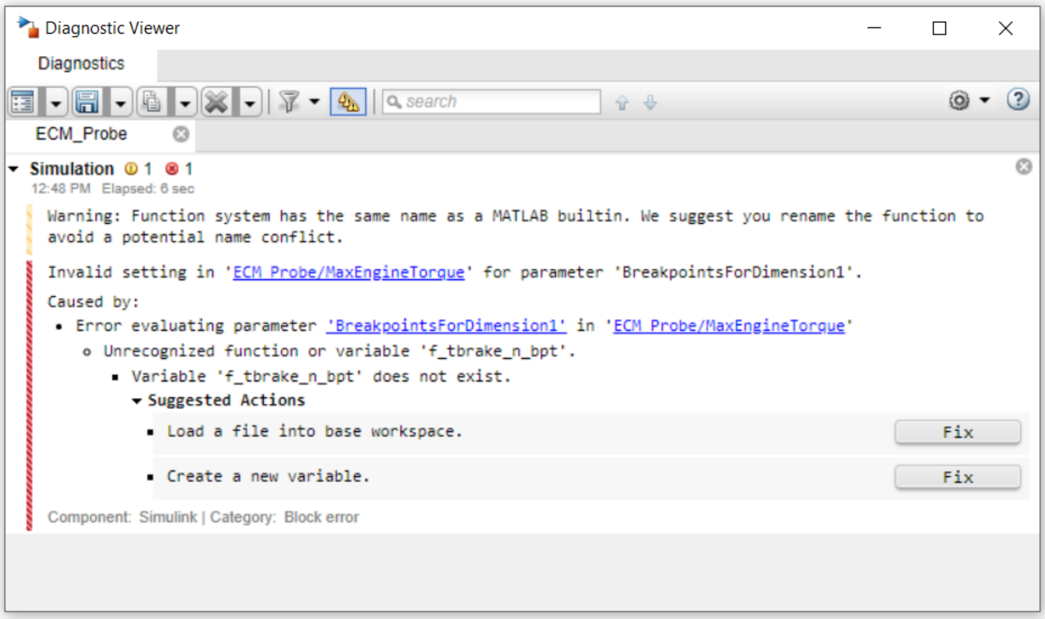


FIGURE 6.20: Internal Data inaccessible outside of EHM within Modularized Model



```
Warning: Function system has the same name as a MATLAB builtin. We suggest you rename the function to
avoid a potential name conflict.

Invalid setting in 'ECM_Probe/MaxEngineTorque' for parameter 'BreakpointsForDimension1'.
Caused by:
  • Error evaluating parameter 'BreakpointsForDimension1' in 'ECM_Probe/MaxEngineTorque'
    o Unrecognized function or variable 'f_tbrake_n_bpt'.
      ▪ Variable 'f_tbrake_n_bpt' does not exist.
        ▾ Suggested Actions
          ▪ Load a file into base workspace.                                    Fix
          ▪ Create a new variable.                                              Fix
Component: Simulink | Category: Block error
```

Figure 6.21: Error received when referencing parameter defined within EHM Model Workspace

original EcoCAR system had much of its functionality defined within subsystems placed within the root Comp_HSC.slx Model, thus resulting in a small number of model references used. The benefits provided by separating areas of concerns drove the decision to create additional modules, and as a result increased the number of model reference dependencies. Furthermore, despite the addition of 12 additional modules to the system, the number of root level model dependencies only increased by 1 (Figure 6.22). This increase in model dependencies was deemed acceptable and still provides value in increasing Simulink model modularization.

6.4 Coupling and Cohesion

Another critical metric for evaluating a modular system is the coupling and cohesion contained within it. A well modularized system typically results in a low level of coupling between modules, and a high cohesion of functions within each module. This section evaluates whether changes made to the EcoCAR Model resulted in increased cohesion among modules and decreased coupling between them. Section 6.4.1 describes the changes to coupling before and after application of the Simulink Module Tool and discusses the interdependence between modules. Section 6.4.2 looks at the overall change in cohesion and resulting interdependence of components within a given Simulink module.

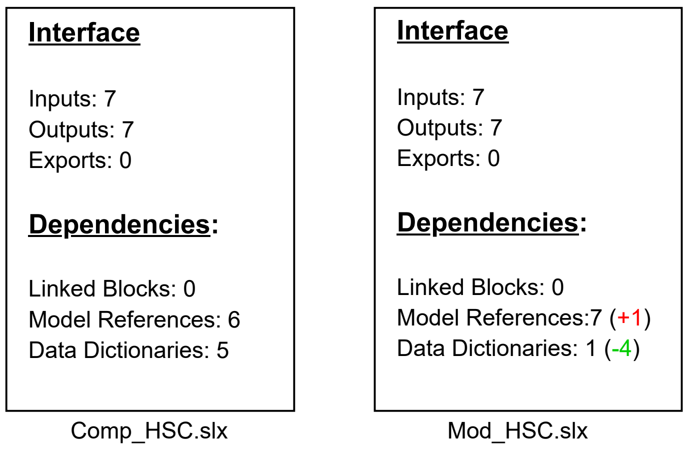


FIGURE 6.22: Interface and Dependencies Before and After Modularization

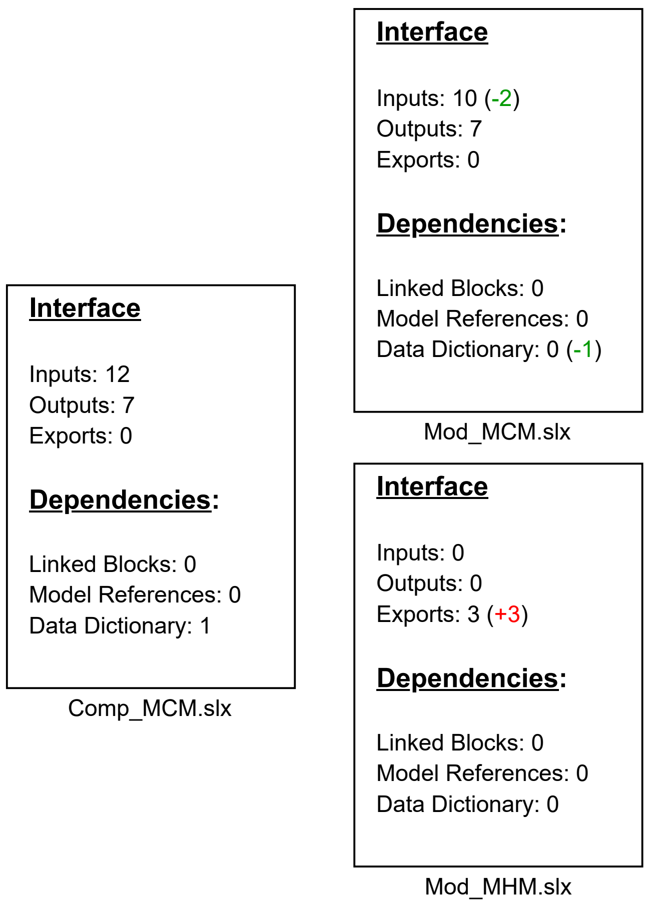


FIGURE 6.23: Motor Interface Before and After Modularization

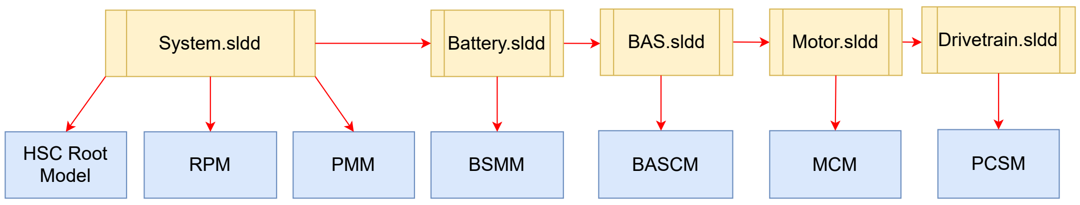


Figure 6.24: Model References with high data coupling to Data Dictionaries

6.4.1 Coupling

The data coupling was reduced significantly through the use of model workspaces as opposed to global data dictionaries. In the original EcoCAR Model, the Simulink modules used several Simulink Data Dictionaries as their primary data source, shown in Figure 6.24. The data dictionaries were all placed within a reference hierarchy and as a result were highly coupled with one another. Changes made to parameters in one data dictionary could potentially affect multiple Simulink modules, leading to increased system complexity and instability. The modularized model removed 4 of the 5 data dictionaries by using model workspaces as the primary data source for the Simulink modules, seen in Figure 6.25. Despite the increase in total Simulink modules from 6 to 18, data coupling between them was reduced, with 15 out of the total 19 models using the model workspace for data storage. Data utilized in multiple Simulink modules was consolidated to a single global Data Dictionary, *Global.sldd*.

6.4.1.1 Example of Functional Coupling Reduction

The Voltage to Pedal Conversion and Pedal to Voltage Conversion subsystems were highly coupled within the EcoCAR Model (highlighted purple in Figures 6.29 and 6.30, respectively). Both subsystems contained secrets regarding the engine pedal maps and pedal voltage limits to interface with the accelerator pedal and engine hardware. Despite their interdependence, the Voltage to Pedal Conversion subsystem was placed in a separate component (input conversion layer) from the Pedal to Voltage subsystem (output conversion layer). In the Modularized Model, both pedal functions *Calc_Pedal_In()* and *Calc_Pedal_Out()* are defined within the PPM (Figure 6.26), reducing the coupling in the root model. In addition, the engine hardware secrets have been removed and are accessed using EHM function caller blocks.

6.4.2 Cohesion

The input conversion layer contained within the original EcoCAR Model can be seen in Figures 6.28 and 6.29, and the output conversion layer is shown in Figure 6.30. The virtual subsystems have been color coded based on the functional secrets they hold (details shown in Figure 6.27). Subsystems with the same color contain algorithms which share secrets with one another, and subsystems with different colors are functionally independent. In total, there were nine different groupings of functionality within the input layer, and five different groupings within the output layer. In

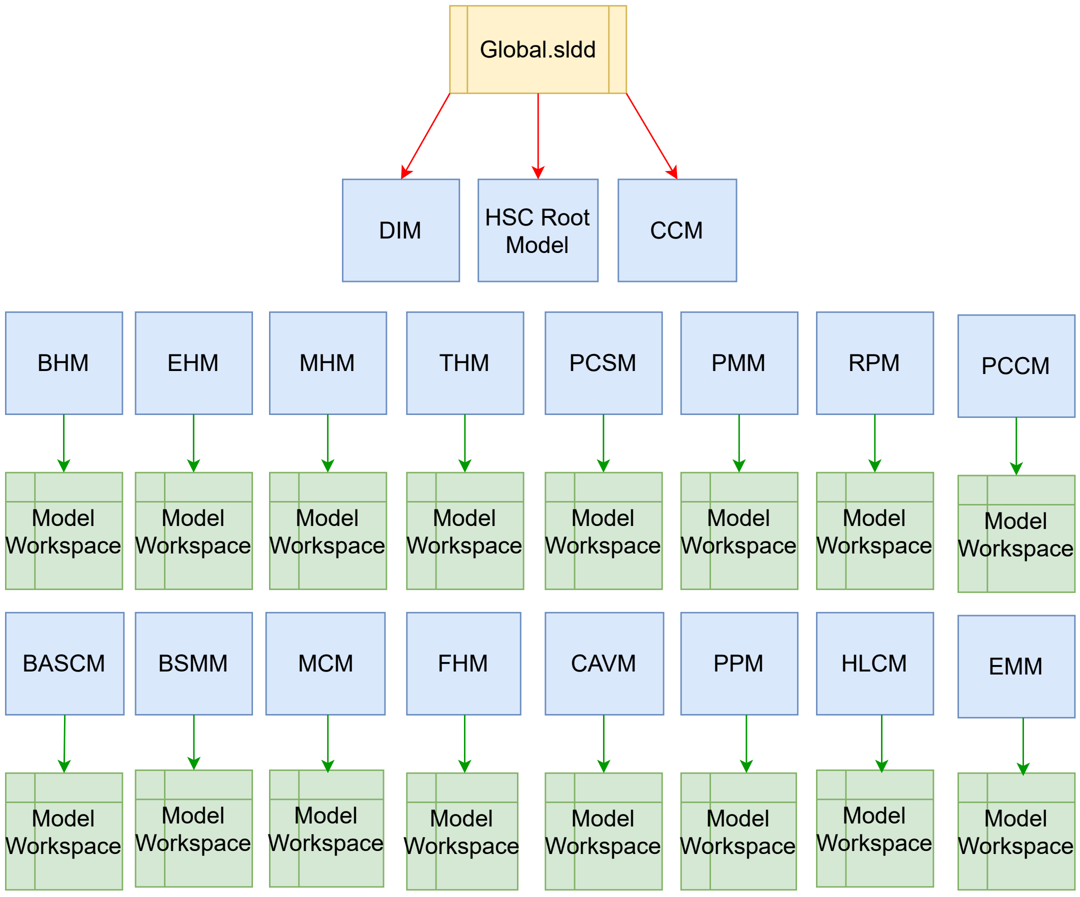


FIGURE 6.25: Reduced data coupling after Module Data defined in Model workspaces

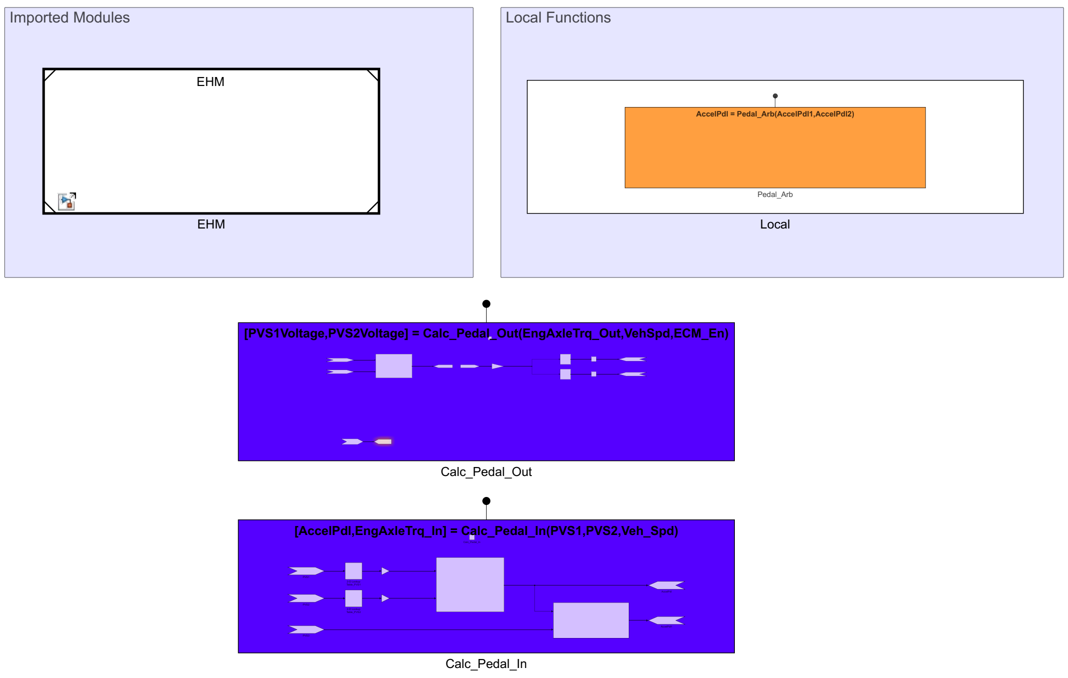

Imported Modules
EHM
EHM
Local Functions
AccelPdl = Pedal_Arb(AccelPdl1,AccelPdl2)
Pedal_Arb
Local
[PVS1Voltage,PVS2Voltage] = Calc_Pedal_Out(EngAxleTrq_Out,VehSpd,ECM_En)
Calc_Pedal_Out
[AccelPdl,EngAxleTrq_In] = Calc_Pedal_In(PVS1,PVS2,Veh_Spd)
Calc_Pedal_In

FIGURE 6.26: Interdependent Functions placed within singular Module to reduce coupling

both cases, the large variation of functionality contained within the same components results in low cohesion within the HSC root model. The input and output conversion layers after application of the Simulink Module Tool can be seen in Figures 6.31 and 6.34, respectively. The virtual subsystems were decomposed and converted into Simulink functions, with function caller blocks placed at the model root level. In the case of the Range and Curvature Calculation, it was determined that the function caller block could be placed outside of the root model, and locally scoped within the PCSM Model (shown in Figure 6.32). The Brake Conversion Algorithm was also converted to a Simulink function, and redefined within the Regenerative Power Module (shown in Figure 6.33). All root level functions were grouped based on functionality and defined within a corresponding Simulink module. Examples demonstrating the increased module cohesion can be seen in Figures 6.35 and 6.36, where model references are color coded based on the secrets they hold. Comparing the functionality groupings to those shown in Figures 6.28-6.30, the cohesion was greatly increased at the root level of the model, as well as within each Model Reference.

6.5 Cyclomatic Complexity

The Simulink Model Advisor was leveraged to evaluate the EcoCAR and Modular Models block composition and resulting cyclomatic complexity. The Model Advisor was run with the *Model Metrics* check enabled (shown in Figure 6.37). More specifically, the number of Simulink Blocks, Subsystems, Library Links, and Stateflow Chart objects were evaluated. In addition, the cyclomatic complexity check was evaluated

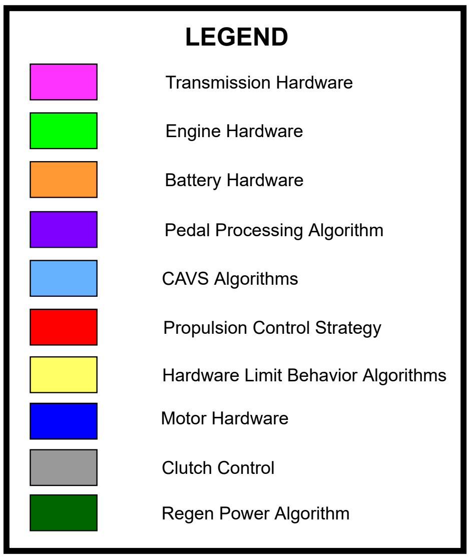


FIGURE 6.27: Legend for Figures 6.28 - 6.33

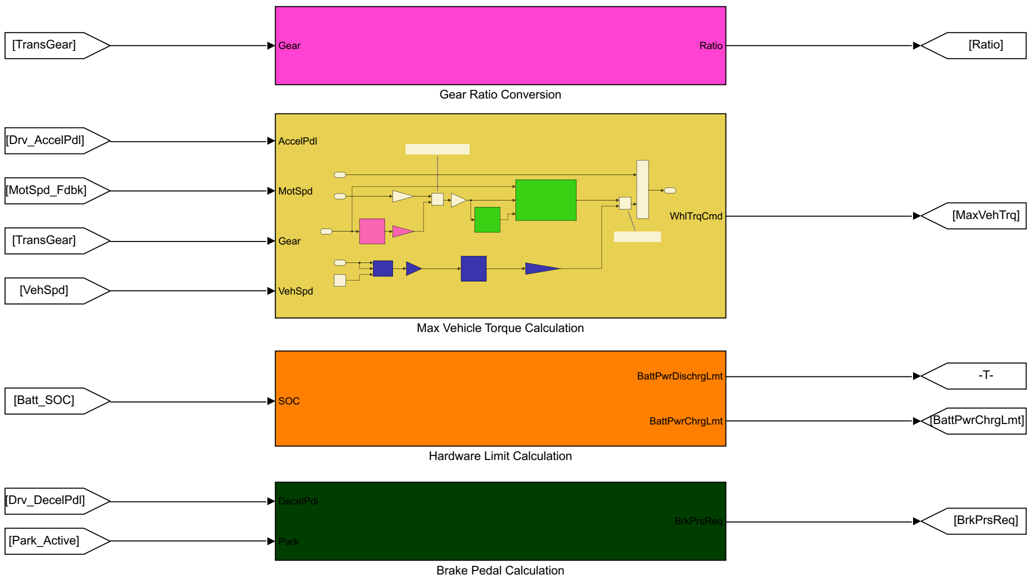


FIGURE 6.28: Low Cohesion among virtual subsystems in Input Conversion Layer (Original EcoCAR Model)

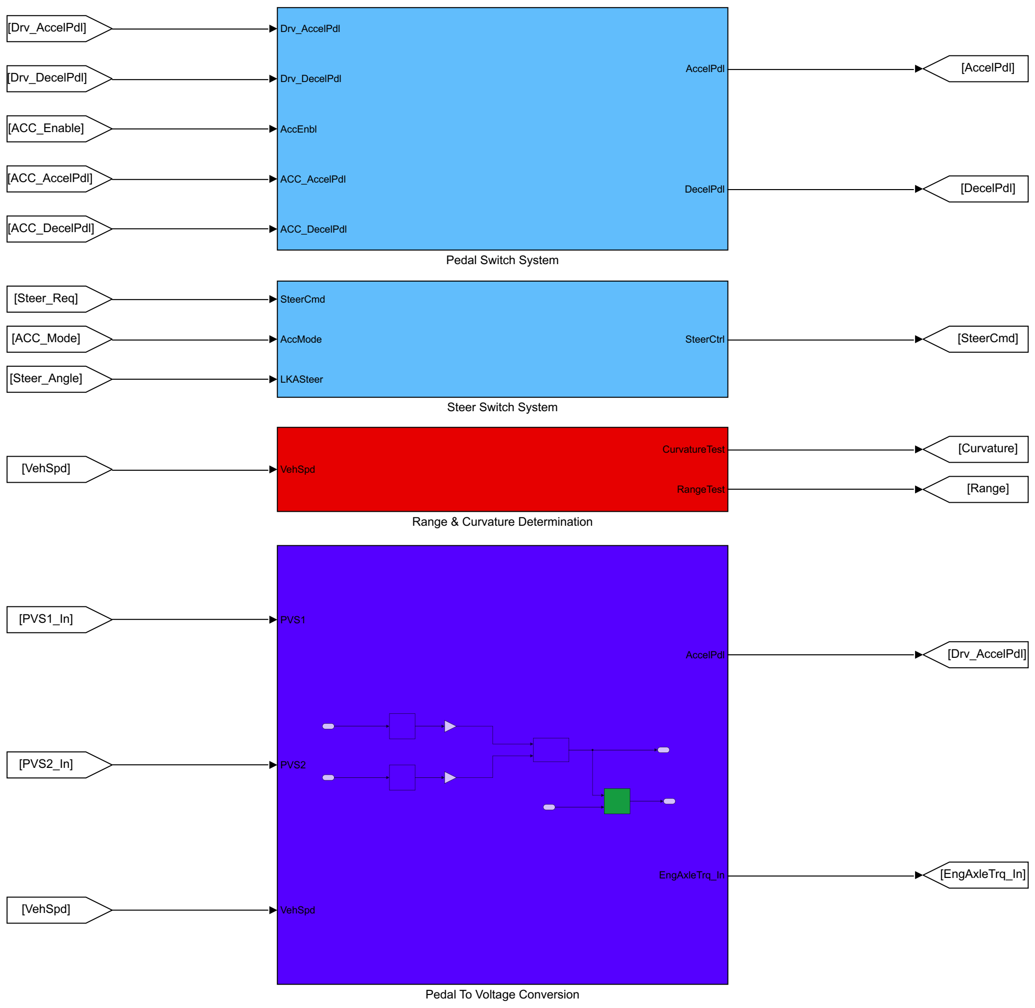


FIGURE 6.29: Low Cohesion among virtual subsystems in Input Conversion Layer, continued (Original EcoCAR Model)

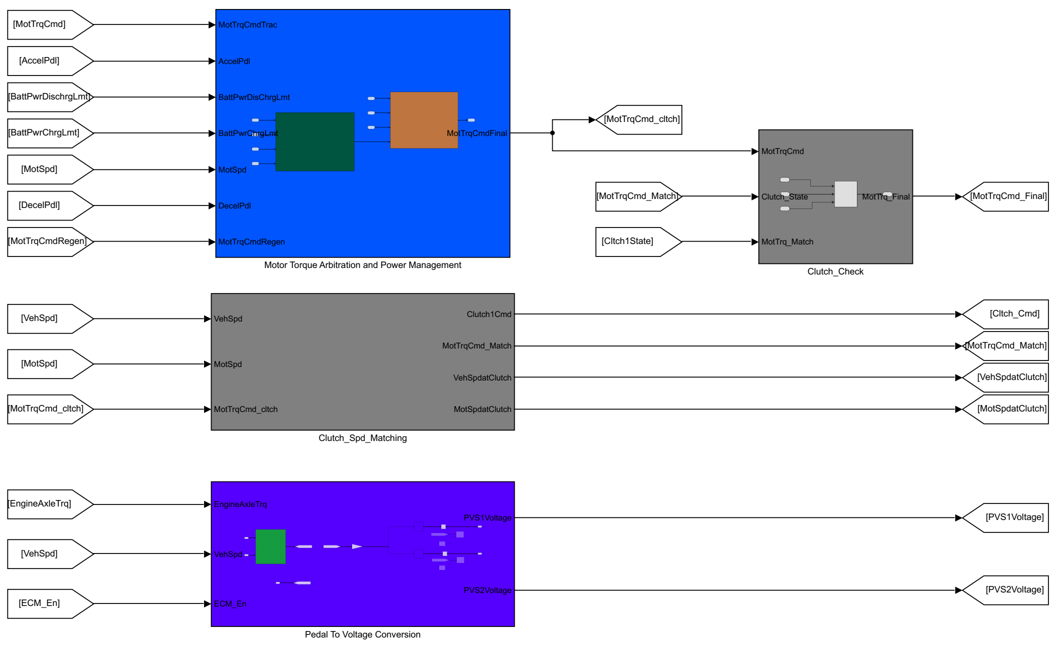


Figure 6.30: Low Cohesion among virtual subsystems in Output Conversion Layer (Original EcoCAR Model)

to determine the impact that modularization had on overall program intricacy. The results of Model Advisor checks ran on the EcoCAR and Modular Models are shown in Table 6.3 and 6.4, respectively. The count and complexity metrics for the root HSC Model as well all system models have been included.

In total, the original EcoCAR system contained 6 model references (7 total models), 1612 blocks, 105 subsystems, 164 Stateflow objects, with an overall cyclomatic complexity of 300. After model restructuring and modularization, the system had 20 model references (19 total models), 2013 blocks, 115 subsystems, 162 Stateflow objects, with an overall cyclomatic complexity of 338. The comparison of these metric totals can be seen in Table 6.5. The Modular Model resulted in a 70% increase in model references, which was expected due to the addition of new Simulink Modules to the system decomposition. Various algorithms originally implemented at the root level of the HSC were grouped based on functionality and moved into newly created Simulink Models, thus increasing the total count. The number of blocks and subsystems also increased by 19.9% and 8.7%, respectively. The increase in blocks can be attributed to the conversion of all virtual subsystems within the model to Simulink functions. This conversion introduces addition Simulink function caller blocks, function input and output port blocks, as well as GoTo/From Tags for organized data flow. Furthermore, certain subsystems in the original model contained multiple algorithms which limited the number of subsystems used. During modularization, these algorithms were separated and placed within their own Simulink function and executed with a corresponding Simulink function caller block. The number of Stateflow objects stayed relatively the same, but the slight decrease can be attributed to the addition

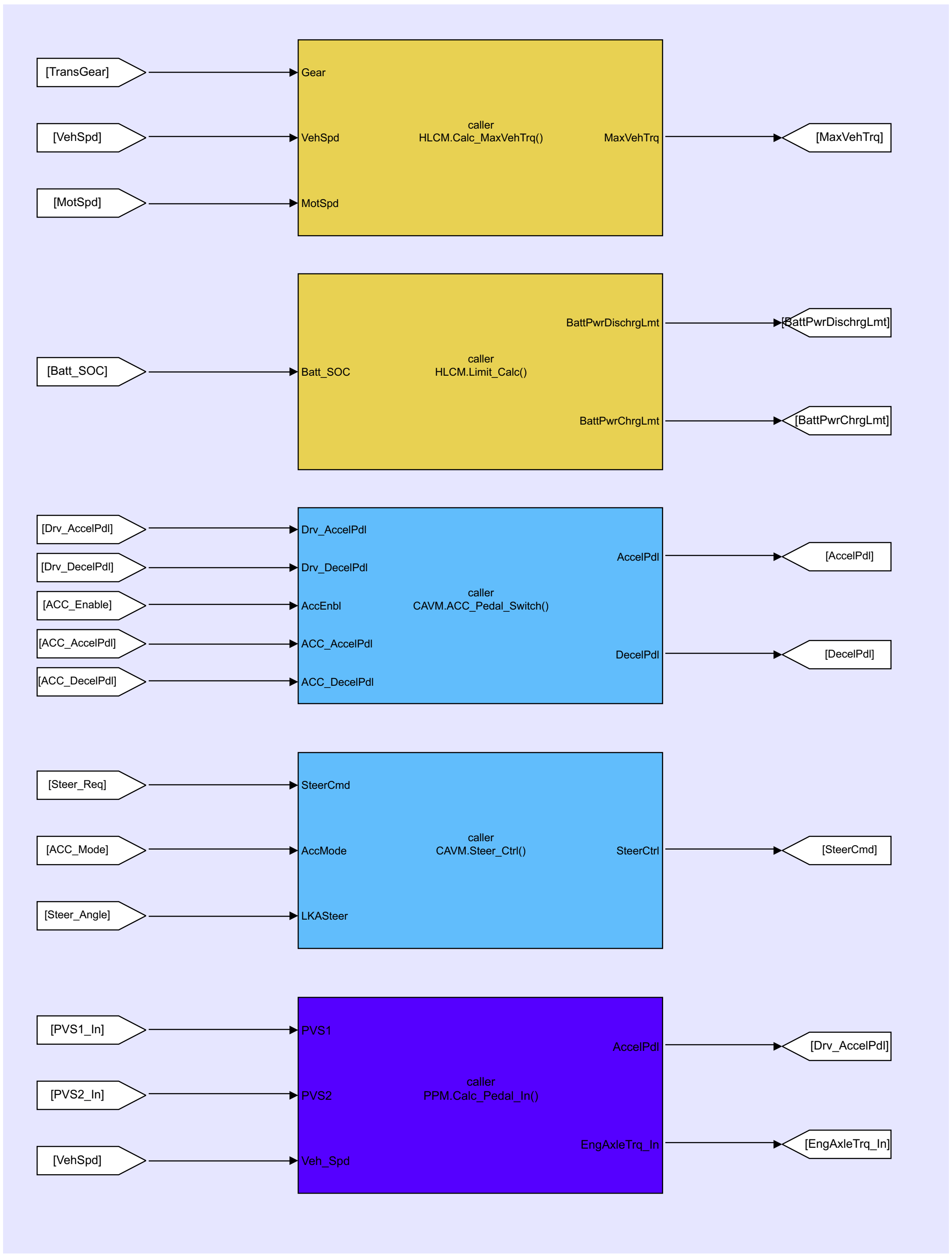


FIGURE 6.31: Input Subsystems replaced by Simulink function calls at HSC Root level (Modularized Model)

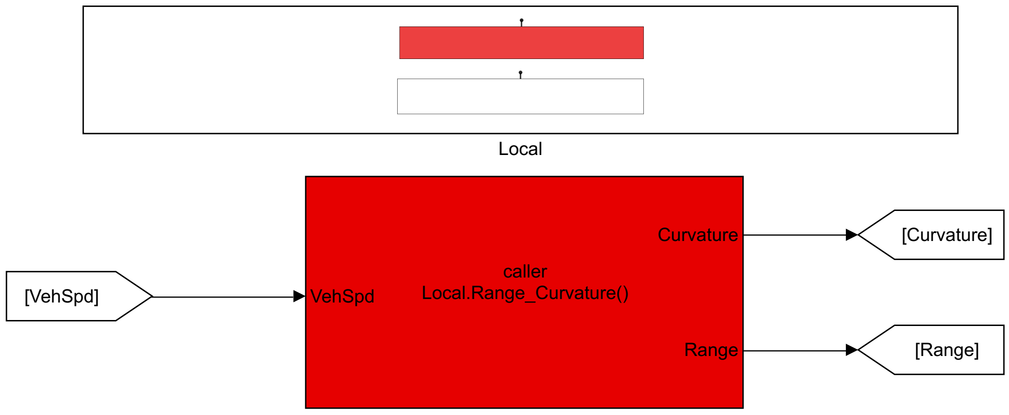
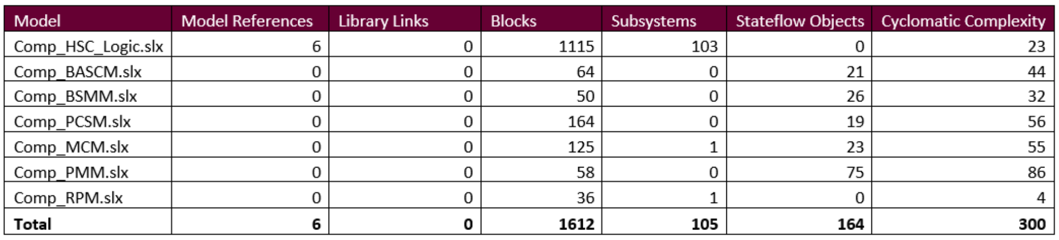


Figure 6.32: Range Curvature function moved to PCSM, improving cohesion

Table 6.3: Model Advisor Results for Original EcoCAR Model



Model	Model References	Library Links	Blocks	Subsystems	Stateflow Objects	Cyclomatic Complexity
Comp_HSC_Logic.slx	6	0	1115	103	0	23
Comp_BASCM.slx	0	0	64	0	21	44
Comp_BSMM.slx	0	0	50	0	26	32
Comp_PCSM.slx	0	0	164	0	19	56
Comp_MCM.slx	0	0	125	1	23	55
Comp_PMM.slx	0	0	58	0	75	86
Comp_RPM.slx	0	0	36	1	0	4
Total	**6**	**0**	**1612**	**105**	**164**	**300**

of Simulink Box objects to locally scope functions within each module's Stateflow Chart, reducing certain instances of repeated functionality. Finally, the change in modular structure resulted in a 11.2% increase in cyclomatic complexity. The additional 38 complexity added to the Modular Model is a result of the added Simulink functions, as well as the addition of new models needed to group algorithmic secrets more accurately. The increase in cyclomatic complexity was relatively small and was deemed necessary to greatly improve the information hiding and modular structure.

6.6 Testability

The SDV Tool was applied to both the original EcoCAR and Modularized Models, and configured to simulate in *Test Generation Mode*. Test objectives within each model system were identified, and test cases were automatically generated to maximize structural coverage. The test generation analysis results before and after modularization can be seen in Table 6.6. The total number of objectives increased by 4.6% from 581 to 608, which can be attributed to newly added Simulink Functions

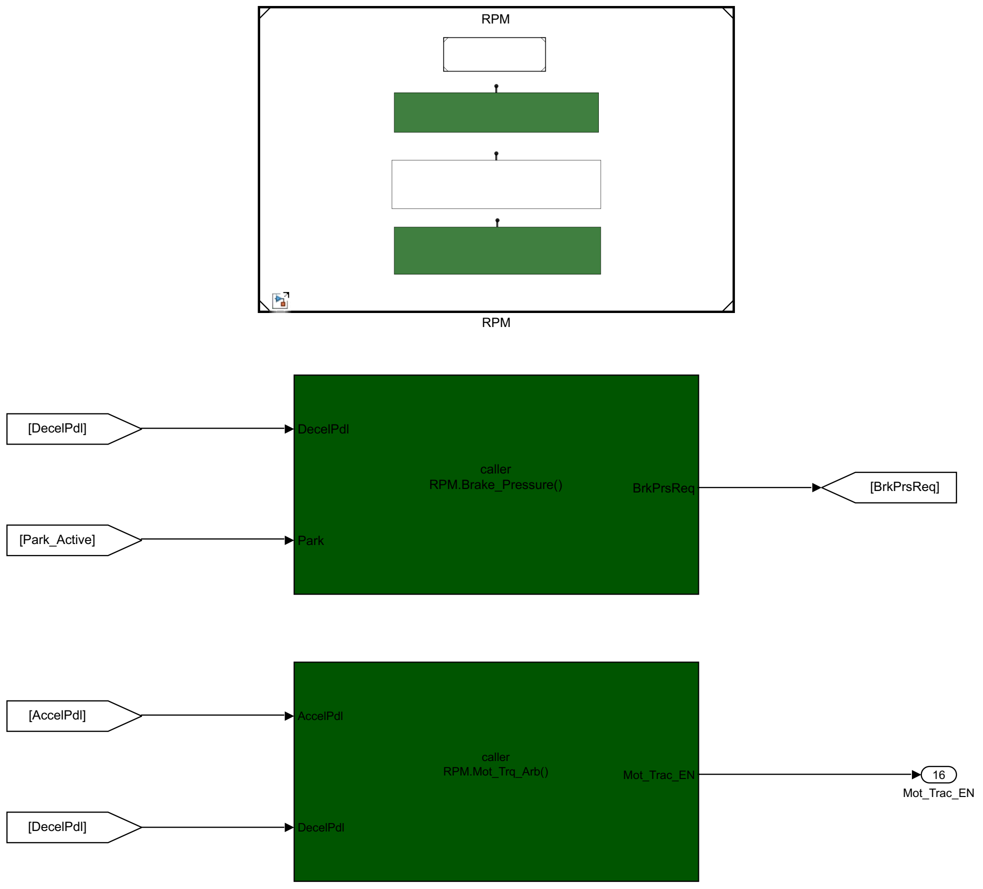


FIGURE 6.33: Brake Calculation function moved to RPM, improving cohesion

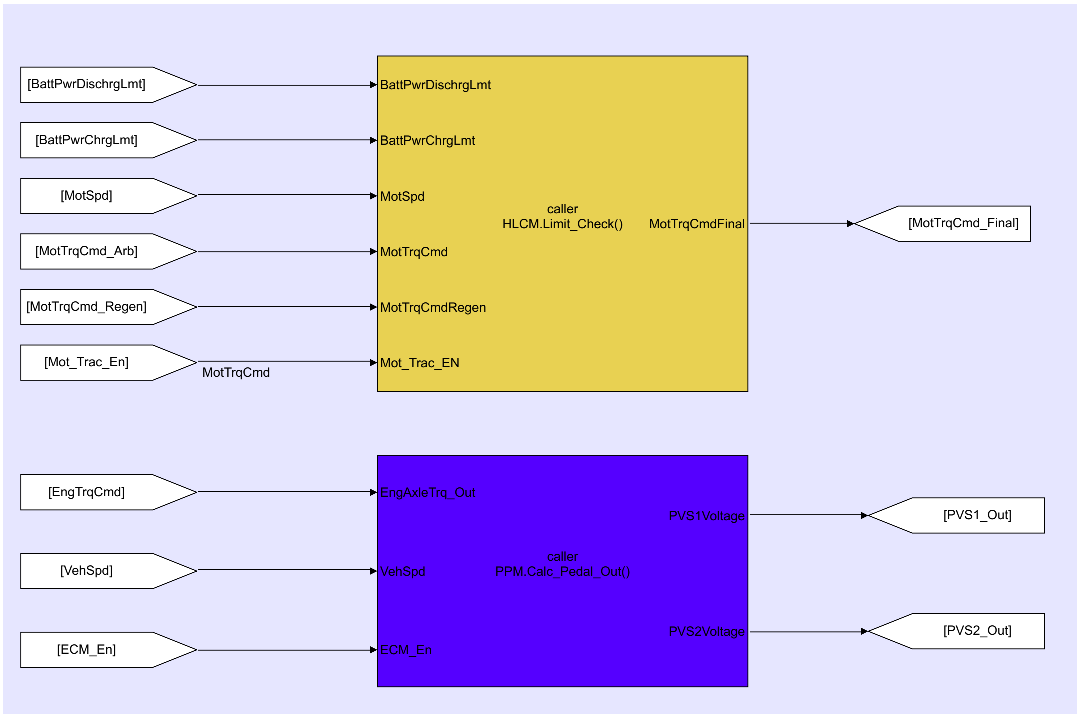


FIGURE 6.34: Output Subsystems replaced by Simulink function calls at HSC Root level

TABLE 6.4: Model Advisor Results for Modularized Model

Model	Model References	Library Links	Blocks	Subsystems	Stateflow Objects	Cyclomatic Complexity
Mod_HSC_Logic.slx	7	0	283	0	0	0
Mod_CAVM.slx	0	0	18	2	0	6
Mod_DIM.slx	0	0	311	2	0	3
Mod_HLCM.slx	4	0	65	3	0	12
Mod_PPM.slx	1	0	38	4	0	11
Mod_FHM.slx	0	0	84	4	0	28
Mod_EMM.slx	3	0	128	0	0	0
Mod_PCSM.slx	0	0	196	3	22	65
Mod_PMM.slx	0	0	57	0	75	86
Mod_RPM.slx	1	0	39	3	0	7
Mod_PCCM.slx	4	0	483	82	0	0
Mod_BSMM.slx	0	0	51	0	28	32
Mod_BASCM.slx	0	0	64	0	22	44
Mod_CCM.slx	0	0	41	2	0	6
Mod_MCM.slx	0	0	54	0	15	15
Mod_EHM.slx	0	0	34	4	0	6
Mod_MHM.slx	0	0	36	3	0	10
Mod_THM.slx	0	0	6	1	0	2
Mod_BHM.slx	0	0	25	2	0	5
Total	**20**	**0**	**2013**	**115**	**162**	**338**

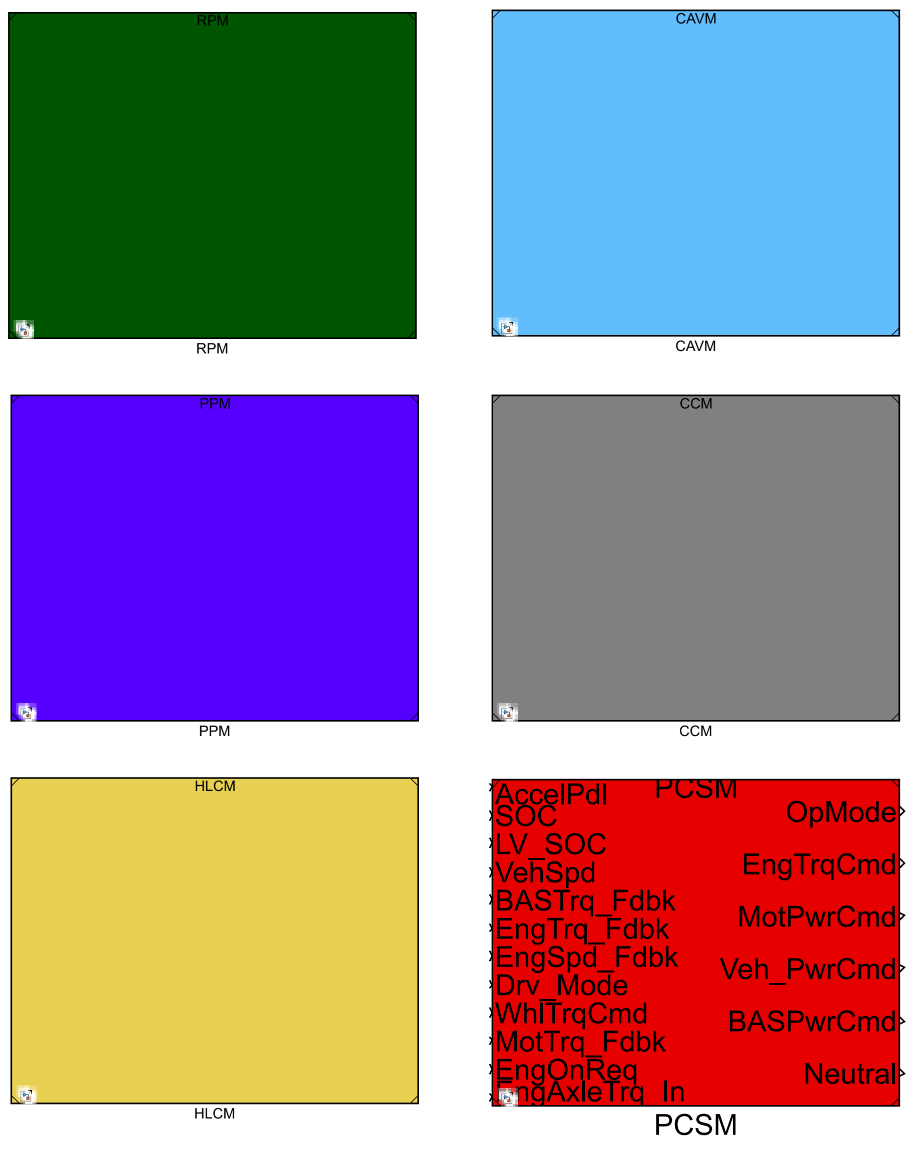


FIGURE 6.35: Modularized Model with High cohesion within Simulink Modules

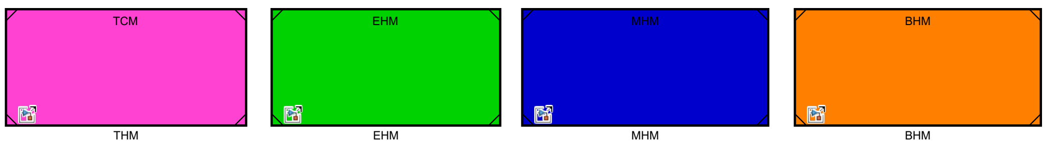
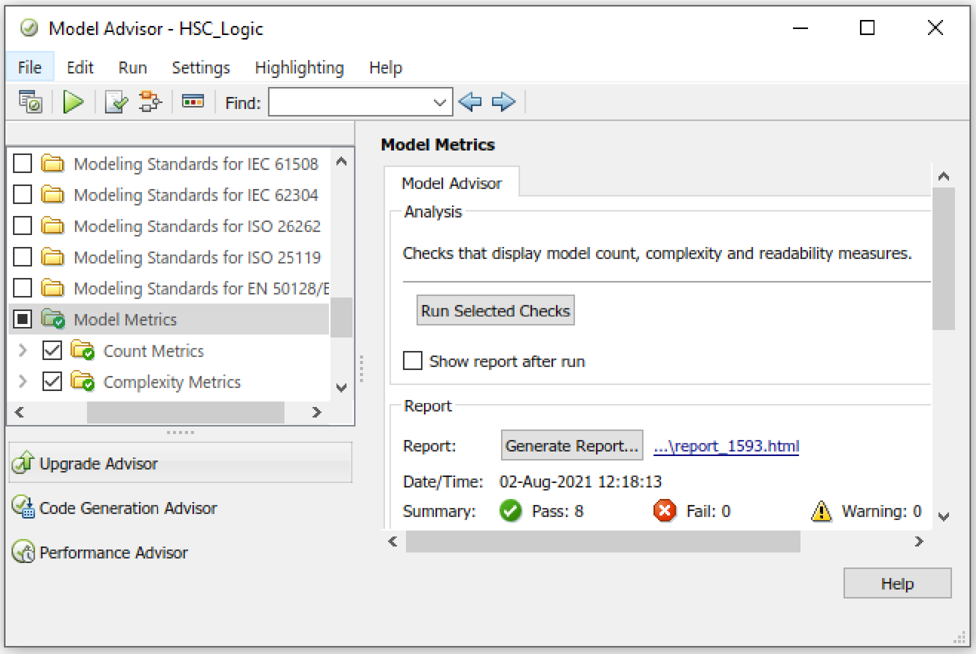


FIGURE 6.36: Modularized Model with High cohesion among Simulink Modules



Checks that display model count, complexity and readability measures.

Report: ...\report_1593.html

Date/Time: 02-Aug-2021 12:18:13

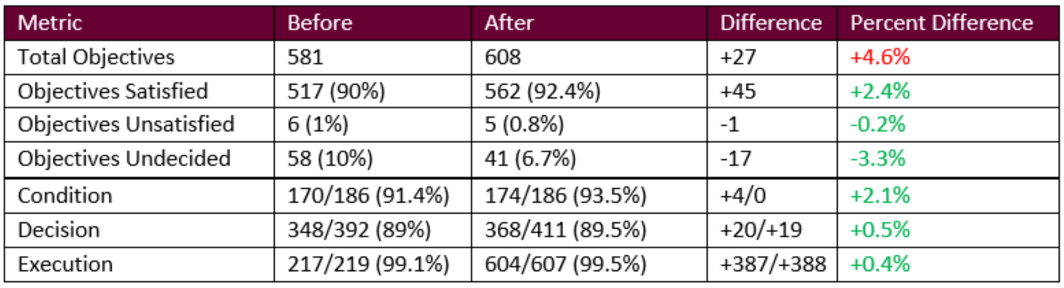
Summary: Pass: 8 Fail: 0 Warning: 0

FIGURE 6.37: Model Advisor showing checks run on both HSC Models

TABLE 6.5: Comparison of Block count and Cyclomatic Complexity before and after changes

Model Advisor Metric	Before	After	Difference	Percent Difference
Model References	6	20	+14	+70.0
Library Links	0	0	0	0.0
Blocks	1612	2013	+401	+19.9
Subsystems	105	115	+10	+8.7
Stateflow Objects	164	162	-2	-1.2
Cyclomatic Complexity	300	338	+38	+11.2

TABLE 6.6: SDV Test Generation Analysis Results Before and After Modularization



Metric	Before	After	Difference	Percent Difference
Total Objectives	581	608	+27	+4.6%
Objectives Satisfied	517 (90%)	562 (92.4%)	+45	+2.4%
Objectives Unsatisfied	6 (1%)	5 (0.8%)	-1	-0.2%
Objectives Undecided	58 (10%)	41 (6.7%)	-17	-3.3%
Condition	170/186 (91.4%)	174/186 (93.5%)	+4/0	+2.1%
Decision	348/392 (89%)	368/411 (89.5%)	+20/+19	+0.5%
Execution	217/219 (99.1%)	604/607 (99.5%)	+387/+388	+0.4%

and Modules increasing the complexity and block count (previously discussed in Section 6.5). Despite the increase in system complexity and total objectives needed, the number of satisfied objectives increased by 2.4%, from 517 to 562, as well as reducing the number of unsatisfied and undecided objectives by 0.2% and 3.3%, respectively. These changes in test generation are small and did not seem to be impacted by changes in modular structure.

Once test objectives were identified, a test harness was created from the original EcoCAR and Modular Models, shown in Figure 6.38. The Simulink Coverage Analyzer was used to provide code coverage analysis and testing completeness using the decision and condition metric (discussed in Section 2.6.3). Simulations were performed for both the root model as well as all its associated model references. The summary of testing coverage results can be seen in Table 6.6, while the detailed breakdown of each model can be seen in Tables 6.7 and 6.8, respectively. The original root model, *Comp_HSC_Logic.slx* had 24 condition, 52 decision, and 132 execution objectives, which can be attributed to the numerous virtual subsystems located in the input and output conversion layers. After restructuring of the model took place, these subsystems were converted to Simulink Functions and placed outside of the root model within an appropriate Simulink Module. As a result, the restructured root model, *Mod_HSC_Logic.slx*, has reduced its total objectives to 0 condition, 0 decision, and 123 executions. These execution objectives represent the Simulink function calls present throughout the input conversion, operation mode, component control, and output conversion layers. Overall, there were no significant changes in testability, other than the total number of objectives increasing. This increase was expected with the addition of new blocks, Simulink functions, and model references. The separation of algorithmic concern led to the redistribution of objectives from within the HSC root level to its associated Model references, but did not significantly impact the level of testability.

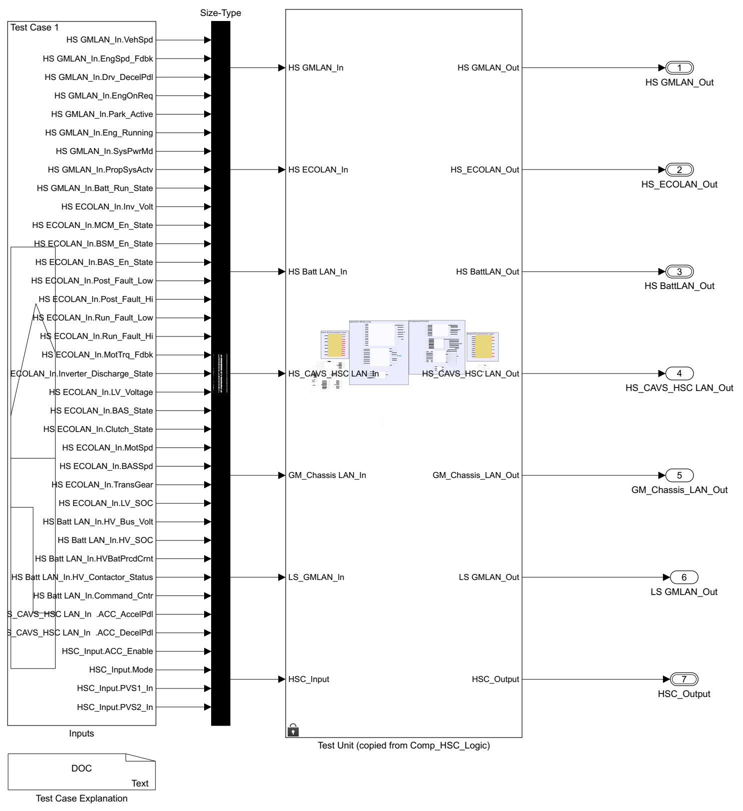
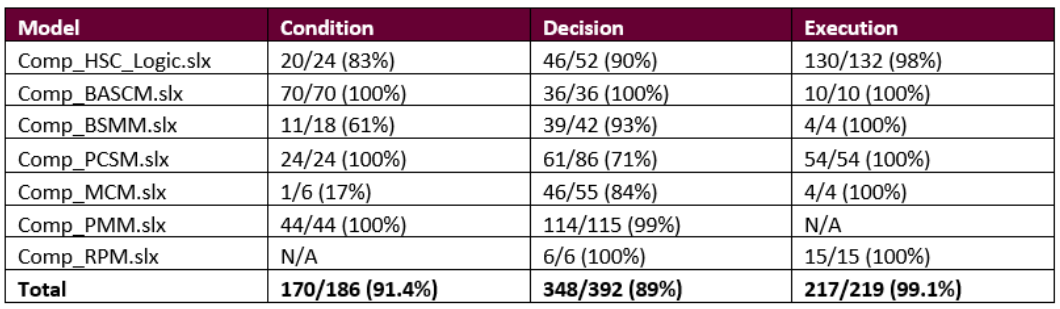


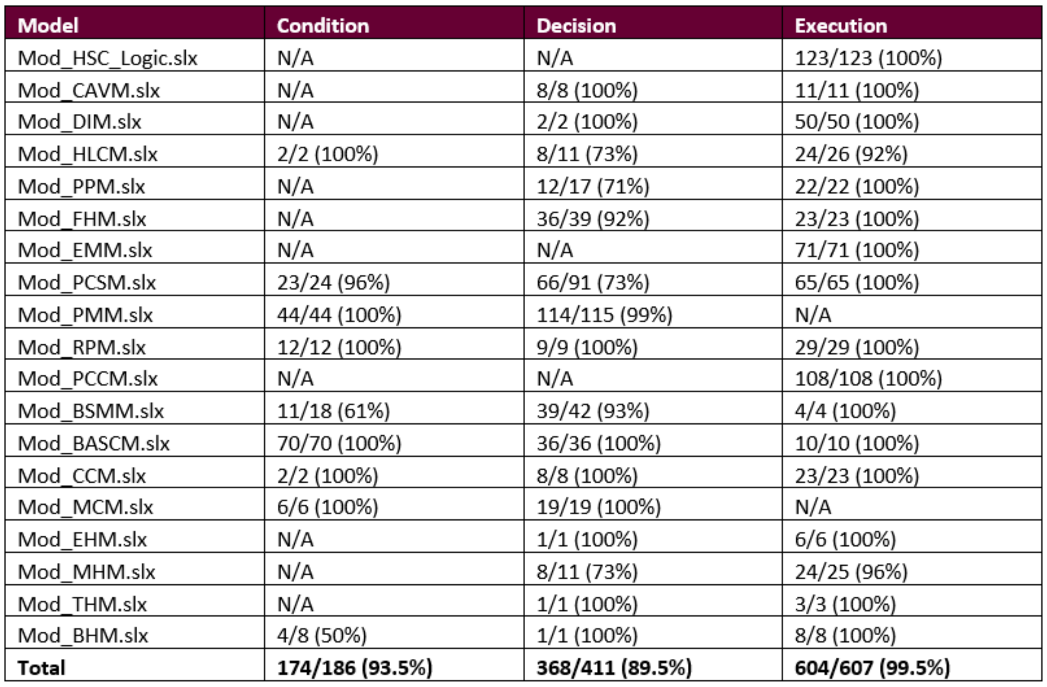
FIGURE 6.38: Test Harness created for coverage analysis

TABLE 6.7: Original Model Coverage Analysis



Model	Condition	Decision	Execution
Comp_HSC_Logic.slx	20/24 (83%)	46/52 (90%)	130/132 (98%)
Comp_BASCM.slx	70/70 (100%)	36/36 (100%)	10/10 (100%)
Comp_BSMM.slx	11/18 (61%)	39/42 (93%)	4/4 (100%)
Comp_PCSM.slx	24/24 (100%)	61/86 (71%)	54/54 (100%)
Comp_MCM.slx	1/6 (17%)	46/55 (84%)	4/4 (100%)
Comp_PMM.slx	44/44 (100%)	114/115 (99%)	N/A
Comp_RPM.slx	N/A	6/6 (100%)	15/15 (100%)
Total	**170/186 (91.4%)**	**348/392 (89%)**	**217/219 (99.1%)**

Table 6.8: Modularized Model Coverage Analysis



Model	Condition	Decision	Execution
Mod_HSC_Logic.slx	N/A	N/A	123/123 (100%)
Mod_CAVM.slx	N/A	8/8 (100%)	11/11 (100%)
Mod_DIM.slx	N/A	2/2 (100%)	50/50 (100%)
Mod_HLCM.slx	2/2 (100%)	8/11 (73%)	24/26 (92%)
Mod_PPM.slx	N/A	12/17 (71%)	22/22 (100%)
Mod_FHM.slx	N/A	36/39 (92%)	23/23 (100%)
Mod_EMM.slx	N/A	N/A	71/71 (100%)
Mod_PCSM.slx	23/24 (96%)	66/91 (73%)	65/65 (100%)
Mod_PMM.slx	44/44 (100%)	114/115 (99%)	N/A
Mod_RPM.slx	12/12 (100%)	9/9 (100%)	29/29 (100%)
Mod_PCCM.slx	N/A	N/A	108/108 (100%)
Mod_BSMM.slx	11/18 (61%)	39/42 (93%)	4/4 (100%)
Mod_BASCM.slx	70/70 (100%)	36/36 (100%)	10/10 (100%)
Mod_CCM.slx	2/2 (100%)	8/8 (100%)	23/23 (100%)
Mod_MCM.slx	6/6 (100%)	19/19 (100%)	N/A
Mod_EHM.slx	N/A	1/1 (100%)	6/6 (100%)
Mod_MHM.slx	N/A	8/11 (73%)	24/25 (96%)
Mod_THM.slx	N/A	1/1 (100%)	3/3 (100%)
Mod_BHM.slx	4/8 (50%)	1/1 (100%)	8/8 (100%)
Total	**174/186 (93.5%)**	**368/411 (89.5%)**	**604/607 (99.5%)**

6.7 Summary

Chapter 6 has provided a comparison between the models described in Chapters 4 and 5. An evaluation was performed based on several commonly-used modularity metrics including information hiding, coupling and cohesion, cyclomatic complexity, and testability. In regards to system changeability, information hiding was greatly improved. Likely changes to the system propagate throughout the original EcoCAR Model, but are restricted to within a single module in the improved version. Hidden internal implementation is also increased within the modular model. Parameters defined within the individual model workspaces have a local scope and cannot be accessed in the root model, preventing unwanted changes propagating throughout the model hierarchy. After making changes to the original EcoCAR model, the data coupling between modules was decreased by grouping together subsystems and functions which shared common secrets. The reduction of global data dictionaries also reduced the data coupling and dependency between modules. The cohesion within individual modules was also greatly increased, with functions previously located within the root HSC model transferred to their respective modular groupings. Despite improvements in modular structure, the cyclomatic complexity was slightly increased, which was mainly attributed to the addition of new modules, and Simulink function caller blocks. Finally, a slight change in testability was observed, with the number of objectives increasing proportional to the additional blocks needed within the system. The testing coverage results slightly improved after making model changes, but were mostly insignificant. Overall, the improvements to information hiding as well as coupling and cohesion created a more modular HSC model, while slightly increasing the interface and cyclomatic complexity. These increases in complexity were deemed acceptable and far outweighed by the benefits resulting from increased modularization.

Chapter 7

Conclusions and Future Work

7.1 Summary of Thesis Content

This thesis began with a brief introduction to trends in software complexity within the automotive industry, as well as some shortcomings in the software quality within MBD frameworks. This was followed by a literature review and description of important background information in Chapter 2, which included the prevalent Simulink Constructs used throughout the model development, as well as important modularity and design process principles. An overview of the EcoCAR Mobility Challenge was provided in Chapter 3, giving context to the HSC Model discussed throughout the remaining Chapters, as well as describing the vehicle architecture and VTS targets. The software design process followed during the creation of the original EcoCAR Model is described in Chapter 4, providing relevant examples of modeling components to show the steps of requirement definition, software design, implementation, testing, and validation. Chapter 5 describes the modular system decomposition, using guidelines established in the literature as well as the Simulink Module Tool to apply relevant Simulink modifications. The potential benefits provided by the new modular decomposition are analyzed in Chapter 6, providing a comparison of several important software metrics including coupling, cohesion, interface complexity, and testability. Most importantly, the system changeability is compared, determining the degree to which change propagation was reduced by forming a modular decomposition based on shared secrets as opposed to role in the data flow process.

7.2 Future Work

This thesis has applied modular principles to hybrid vehicle controls development, providing a template for future EcoCAR teams or students to use as a guideline. The model decomposition outlined in Chapter 5 and analyzed in Chapter 6 contains a multitude of confidential data and parameters which are restricted from public access due to competition NDA restrictions. In order for the modular decomposition to be free for public use, a significant amount of model work must be done to remove all instances of confidential information, and implement black box functionality throughout the system hierarchy. Once these changes have been implemented, the HSC modular model will be uploaded to a remote git repository and made available for public use. Providing a real-world example of Simulink Module Tool application will hopefully provide useful insight to future EcoCAR Teams looking to begin vehicle software development, as well as demonstrating the benefits of applying modularity to a Simulink model.

The second task scheduled for future work is the formalization of the established EcoCAR Vehicle design process into a document that can be available to future EcoCAR teams. One of the biggest challenges faced during the author's time as PCM lead was the lack of clearly defined vehicle development process, which resulted in increased development time, and confusion with software version incompatibilities. After establishing a software design process, changes were made more efficient and fewer corrections were needed to the original functional requirements and system architecture. By providing an example of a vehicle software design process, future students can use this as a guiding framework or starting point for adapting to new competition requirements and changes in the automotive design standards. Efforts are being made to remove all NDA sensitive information from the model, and improve the formatting to better illustrate the modularity. Work on both the model and documentation is ongoing and expected to be complete by October 2021.

7.3 Conclusion

The Hybrid vehicle controller model developed during the McMaster EcoCAR Mobility Challenge has been presented in this thesis, giving both an overview of the model functionality as well as the software design process followed. An application of the Simulink Module Tool has also been demonstrated, restructuring the system decomposition based on separation of concerns as opposed to execution order. The two system decompositions were then analyzed and compared according to commonly used Software indicators . It was determined that new modular decomposition significantly improved the system's ability to gracefully handle likely changes and consolidate them to as few modules as possible. A primary example is provided in Chapter 6 to illustrate this improvement, but the complete list of analyzed changes can be seen in Table A3.1 - A3.3. Furthermore, the information hiding within the HSC system was greatly improved, with additional modules created to separate component hardware and behaviour secrets. Simulink Design Verification demonstrated a relatively acceptable increase in the software complexity, as well as minimal changes to test generation and coverage.

Overall, the research presented in this thesis provides evidence to suggest that the application of the Simulink Module Tool resulted in an improved system decomposition of the HSC model. This evaluation also provides further evidence that application of the Simulink Module Tool can be successfully applied to vehicle controller Simulink models, and improve the information hiding and system changeability. As software complexity continues to grow within the automotive industry, the need for proper software design will increase; larger systems will require a greater level of modularization. By following a unified software design process as closely as possible, as well as applying separation of concerns during system decomposition, the information hiding and changeability of vehicle software can be improved dramatically. Developers of automotive controller models must step up to the task of adapting to changes in system complexity and adopt new decomposition methods to greatly improve the efficiency and quality of vehicle software.

Appendix A

Vehicle Power Moding

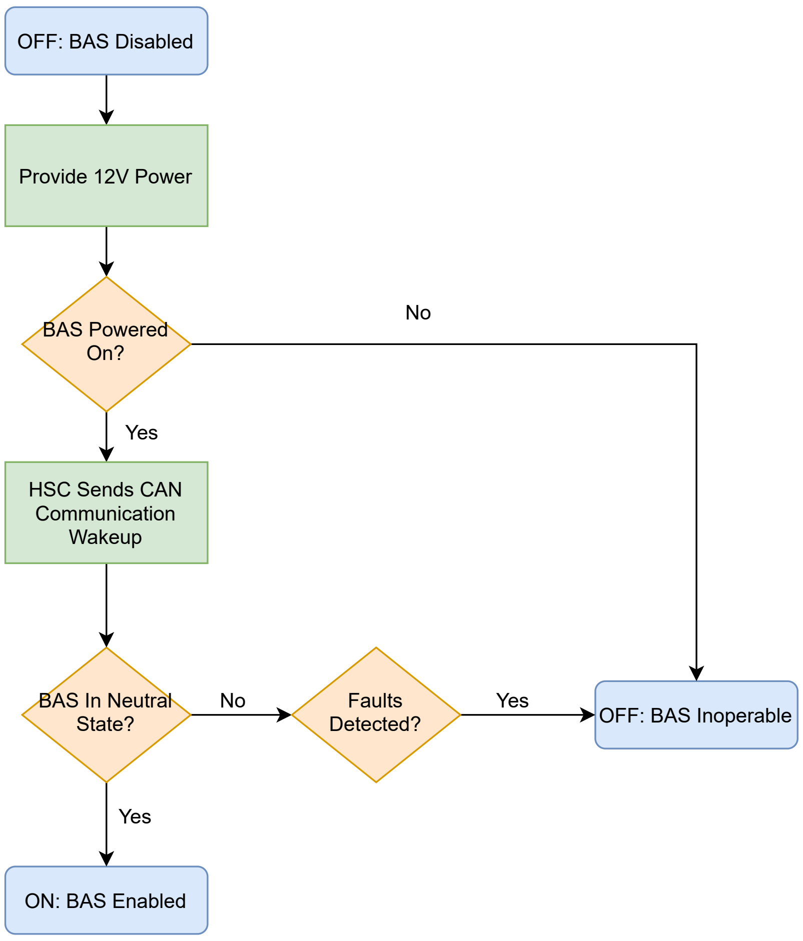


Figure A1.1: BAS Power Moding Algorithm

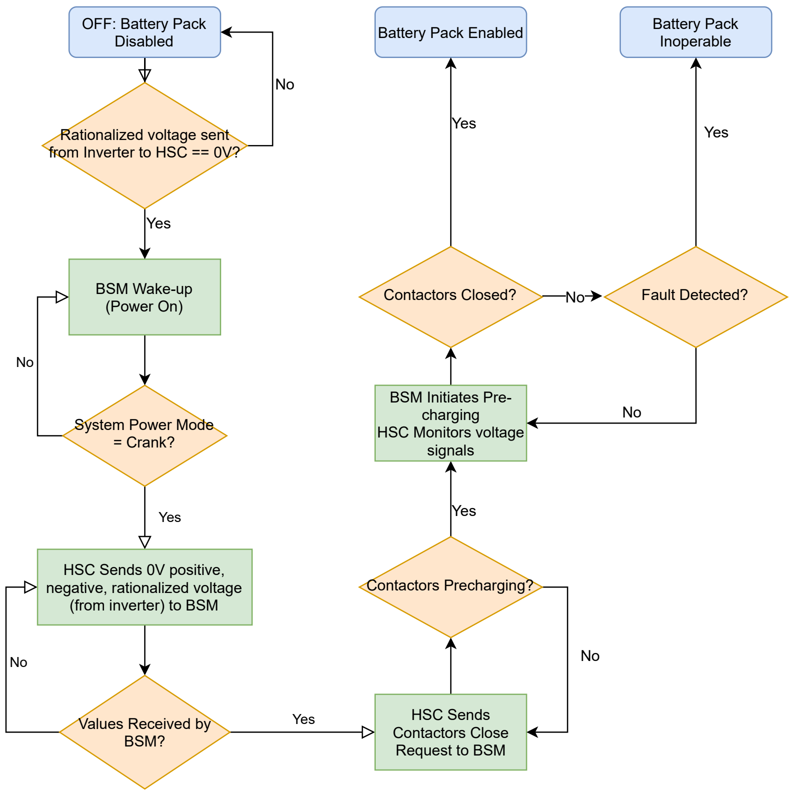

OFF: Battery Pack Disabled
No
Rationalized voltage sent from Inverter to HSC == 0V?
Yes
BSM Wake-up (Power On)
No
System Power Mode = Crank?
Yes
HSC Sends 0V positive, negative, rationalized voltage (from inverter) to BSM
No
Values Received by BSM?
Yes
HSC Sends Contactors Close Request to BSM
Contactors Precharging?
No
Yes
BSM Initiates Pre-charging HSC Monitors voltage signals
Contactors Closed?
No
Fault Detected?
No
Yes
Yes
Battery Pack Enabled
Battery Pack Inoperable

FIGURE A1.2: Battery Power Moding Algorithm

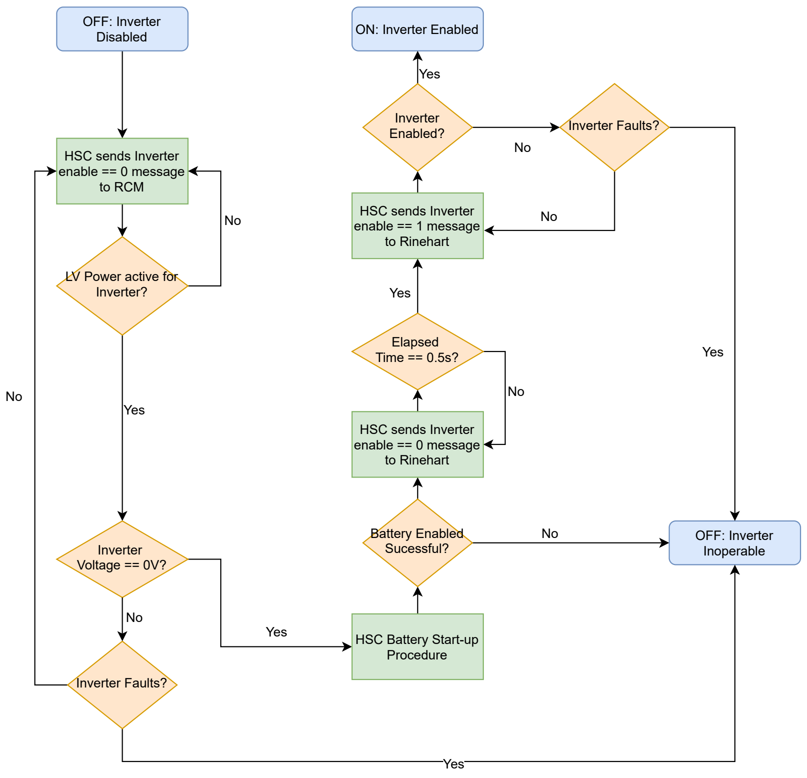

OFF: Inverter Disabled
HSC sends Inverter enable == 0 message to RCM
LV Power active for Inverter?
No
Yes
Inverter Voltage == 0V?
No
Inverter Faults?
No
Yes
Yes
HSC Battery Start-up Procedure
Battery Enabled Sucessful?
No
OFF: Inverter Inoperable
HSC sends Inverter enable == 0 message to Rinehart
Elapsed Time == 0.5s?
No
Yes
HSC sends Inverter enable == 1 message to Rinehart
Inverter Enabled?
Yes
ON: Inverter Enabled
No
Inverter Faults?
No
Yes

FIGURE A1.3: Motor Power Moding Algorithm

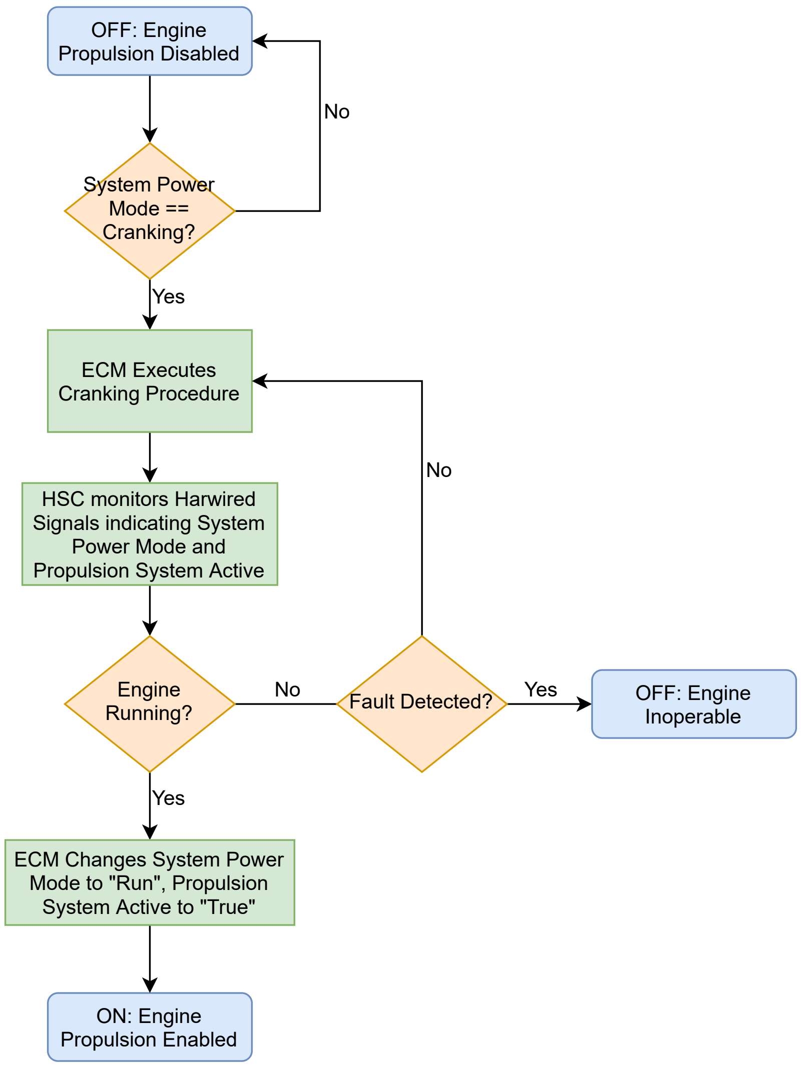

OFF: Engine Propulsion Disabled
System Power Mode == Cranking?
No
Yes
ECM Executes Cranking Procedure
HSC monitors Harwired Signals indicating System Power Mode and Propulsion System Active
Engine Running?
No
Fault Detected?
No
Yes
OFF: Engine Inoperable
Yes
ECM Changes System Power Mode to "Run", Propulsion System Active to "True"
ON: Engine Propulsion Enabled

FIGURE A1.4: Engine Power Moding Algorithm

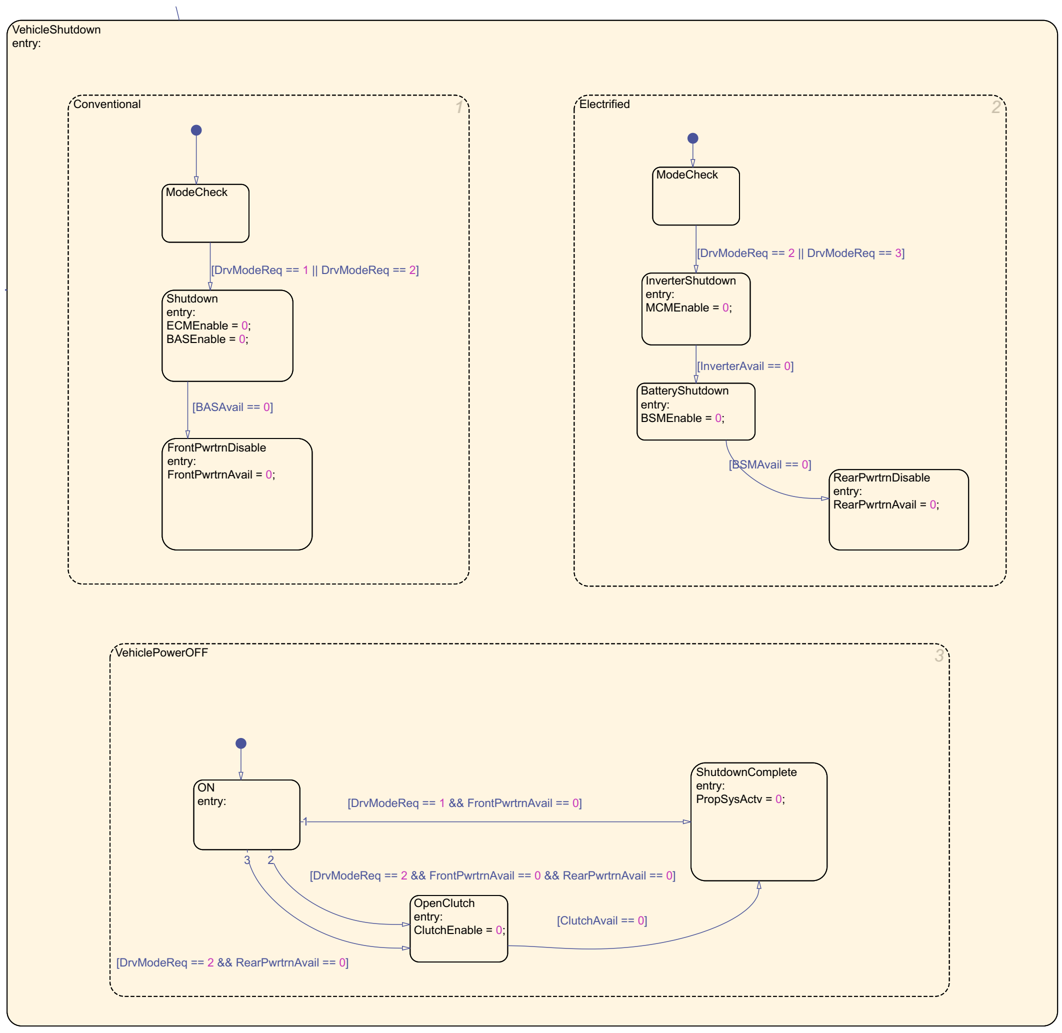


FIGURE A1.5: Vehicle Shutdown Stateflow

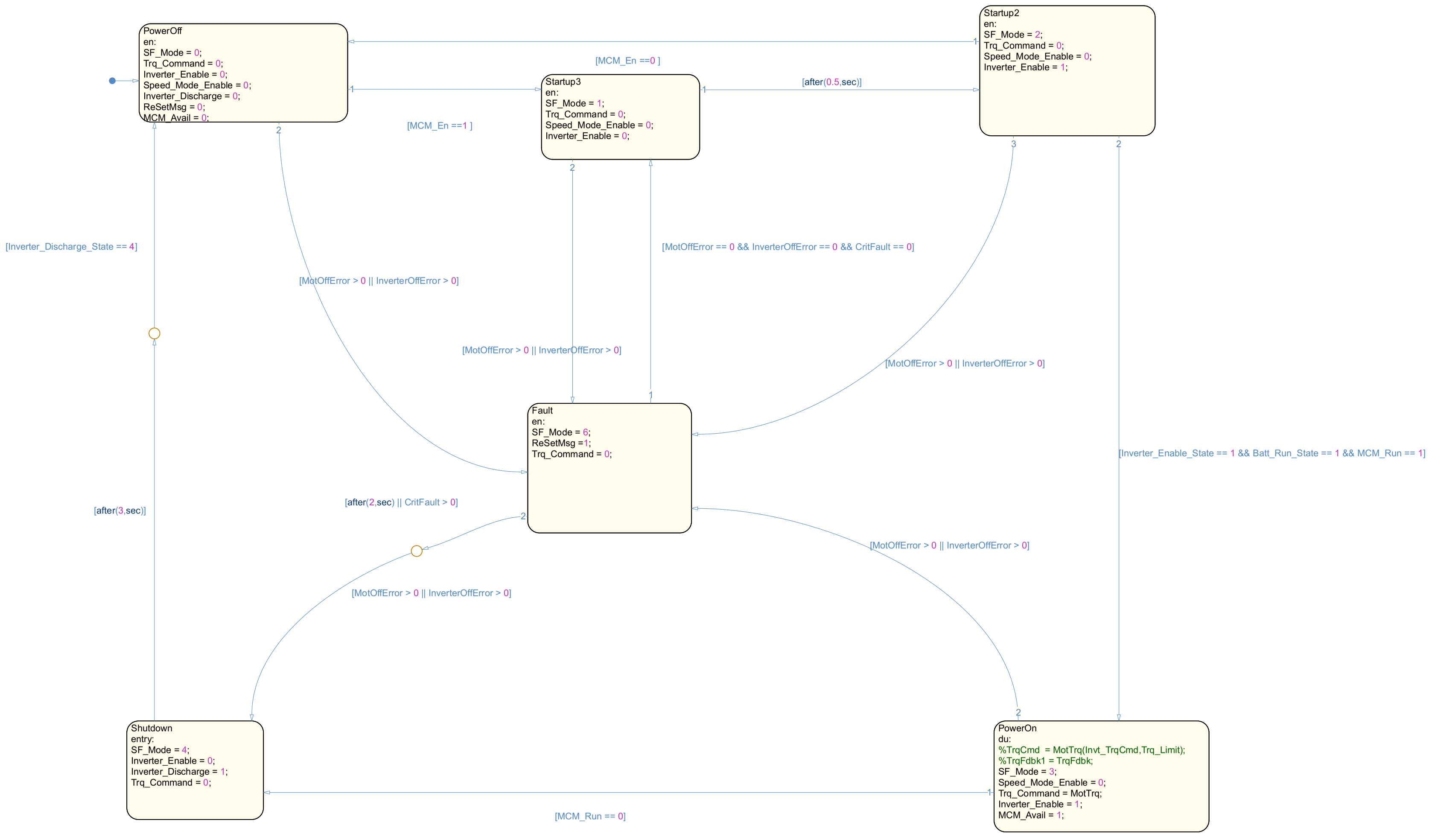


FIGURE A1.6: Motor Control Ring Stateflow

Appendix B

Likely System Changes

Change ID	Likely Change	System Module
1	Data Network Technology (Using a different Protocol than CAN)	Data Interface Module
2	Data Representation (Changing data types or representations within existing Data interface	Data Interface Module
3	Serial Data Interface (Changing the CAN DB, modifying input and output signals.	Data Interface Module
4	Pedal Interpretation Algorithm (changing pedal arbitration method)	Pedal Processing Module
5	Pedal Sensors (change to the voltage to pedal position conversion)	Pedal Processing Module
6	Engine Hardware (swap to a different engine component)	Engine Hardware Module
7	Engine Hardware parameters (more accurate limit data obtained)	Engine Hardware Module
8	Pedal Torque Map (mapping of pedal position to engine axle torque	Engine Hardware Module
9	Battery Hardware (swap to a different battery component)	Battery Hardware Module
10	Battery Hardware parameters (more accurate limit data obtained)	Battery Hardware Module
11	Motor Hardware (swap to a different motor component)	Motor Hardware Module
12	Motor Hardware parameters (more accurate limit data obtained)	Motor Hardware Module
13	Transmission Hardware (swap to a different transmission component)	Transmission Hardware Module
14	Transmission Hardware parameters (Modified Gear ratio conversion factor)	Transmission Hardware Module
15	Vehicle Architecture (design for a plug-in hybrid)	Propulsion System Control Module, Energy Management Module
16	Propulsion Control Strategy Algorithm (Change method of Torque Splitting)	Propulsion Control Strategy Module
17	Power Moding Algorithm (Change order of ECU startup)	Power Moding Module

Table A2.1: List of likely system changes and the Modules they affect

Change ID	Likely Change	System Module
18	Regen Power Calculation Algorithm (BAS is implemented in Generation Mode, adding to the Regenerative Strategy)	Regen Power Module
19	Motor Control Algorithm (Change method of requesting torque from the inverter)	Motor Control Module
20	Battery Control Algorithm (Change bus bleed down procedure)	Battery System Manager Module
21	BAS Control Algorithm (Implement Start/Stop Functionality)	BAS Control Module
22	Change in Clutch Control Algorithm (Requiring additional enable conditions to engage)	Clutch Control Module
23	CAVs Arbitration Algorithm (ACC software is modified to send Torque requests instead of Pedal position)	CAVs Module
24	CAVs Lateral Control Algorithm (CAVs Team develops lane keep assist, requiring HSC arbitration and translation into steering wheel requests)	CAVs Module
25	Power Limit Algorithms (Increase in algorithm fidelity, considering past drive cycles to predict future power limit use)	Hardware Limit Control Module
26	Fault Detection Algorithms (Development of component diagnostic procedures, early detection of faults)	Fault Handler Module
27	Fault Mitigation Algorithms (Introduction of Figures of Merit, development of remedial action algorithms)	Fault Handler Module

TABLE A2.2: List of likely system changes and the Modules they affect, continued

Appendix C

System Changeability

Change ID	Likely Change	Modules Affected (Original)	Components Affected (Original)	Modules Affected (New)	Components Affected (New)	Module Propagation (Difference)	Component Propagation (Difference)
1	Data Network Technology (Using a different Protocol than CAN)	3	3	1	1	-2	-2
2	Data Representation (Changing data types or representations within existing Data interface	3	3	1	1	-2	-2
3	Serial Data Interface (Changing the CAN DB, modifying input and output signals).	3	3	1	1	-2	-2
4	Pedal Interpretation Algorithm (changing pedal arbitration method)	3	2	1	1	-2	-1
5	Pedal Sensors (change to the voltage to pedal position conversion)	3	2	1	2	-2	0
6	Engine Hardware (swap to a different engine component)	3	7	2	6	-1	-1
7	Engine Hardware parameters (more accurate limit data obtained)	3	5	1	4	-2	-1
8	Pedal Torque Map (mapping of pedal position to engine axle torque	3	2	1	2	-2	0
9	Battery Hardware (swap to a different battery component)	4	7	2	4	-2	-3
10	Battery Hardware parameters (more accurate limit data obtained)	3	5	1	2	-2	-3
11	Motor Hardware (swap to a different motor component)	4	9	2	5	-2	-4

Table A3.1: List of likely system changes and the improvements in system changeability, part 1.

Change ID	Likely Change	Modules Affected (Original)	Components Affected (Original)	Modules Affected (New)	Components Affected (New)	Module Propagation (Difference)	Component Propagation (Difference)
12	Motor Hardware parameters (more accurate limit data obtained)	3	7	1	3	-2	-4
13	Transmission Hardware (swap to a different transmission component)	3	6	2	3	-1	-3
14	Transmission Hardware parameters (Modified Gear ratio conversion factor)	2	4	1	1	-1	-3
15	Vehicle Architecture (design for a plug-in hybrid)	8	8	2	7	-6	-1
16	Propulsion Control Strategy Algorithm (Change method of Torque Splitting)	2	2	1	2	-1	0
17	Power Moding Algorithm (Change order of ECU startup)	1	1	1	1	0	0
18	Regen Power Calculation Algorithm (BAS is implemented in Generation Mode, adding to the Regenerative Strategy)	2	2	1	2	-1	0
19	Motor Control Algorithm (Change method of requesting torque from the inverter)	1	1	1	1	0	0
20	Battery Control Algorithm (Change bus bleedown procedure)	1	1	1	1	0	0

Table A3.2: List of likely system changes and the improvements in system changeability, part 2.

Change ID	Likely Change	Modules Affected (Original)	Components Affected (Original)	Modules Affected (New)	Components Affected (New)	Module Propagation (Difference)	Component Propagation (Difference)
21	BAS Control Algorithm (Implement Start/Stop Functionality)	1	1	1	1	0	0
22	Change in Clutch Control Algorithm (Requiring additional enable conditions to engage)	2	2	2	2	0	0
23	CAVs Arbitration Algorithm (ACC software is modified to send Torque requests instead of Pedal position)	2	2	1	1	-1	-1
24	CAVs Lateral Control Algorithm (CAVs Team develops lane keep assist, requiring HSC arbitration and translation into steering wheel requests)	2	2	1	1	-1	-1
25	Power Limit Algorithms (Increase in algorithm fidelity, considering past drive cycles to predict future power limit use)	2	5	1	3	-1	-2
26	Fault Detection Algorithms (Development of component diagnostic procedures, early detection of faults)	6	6	1	1	-5	-5
27	Fault Mitigation Algorithms (Introduction of Figures of Merit, development of remedial action algorithms)	6	6	1	1	-5	-5

Table A3.3: List of likely system changes and the improvements in system changeability, part 3.

Bibliography

[1] Jaswinder Ahluwalia, Ingolf H. Krüger, Walter Phillips, and Michael Meisinger. "Model-Based Run-Time Monitoring of End-to-End Deadlines". In: *Proceedings of the 5th ACM International Conference on Embedded Software*. EMSOFT '05. Jersey City, NJ, USA: Association for Computing Machinery, 2005, 100–109. ISBN: 1595930914. DOI: 10.1145/1086228.1086248. URL: https://doi.org/10.1145/1086228.1086248.

[2] GM Authority. *GM 1.5 Liter Turbo I-4 Ecotec LYX Engine*. 2021. URL: https://gmauthority.com/blog/gm/gm-engines/lyx/ (visited on 08/10/2021).

[3] GM Authority. *GM Hydra-Matic 9T50 9-Speed Automatic Transmission*. 2021. URL: https://gmauthority.com/blog/gm/gm-transmissions/9txx/9t50/ (visited on 08/10/2021).

[4] AVTC. *EcoCAR Mobility Challenge*. 2021. URL: https://avtcseries.org/ecocar-mobility-challenge/ (visited on 08/10/2021).

[5] M. Broy. "Automotive software engineering". In: *25th International Conference on Software Engineering, 2003. Proceedings.* 2003, pp. 719–720. DOI: 10.1109/ICSE.2003.1201259.

[6] Manfred Broy, Ingolf H. Kruger, Alexander Pretschner, and Christian Salzmann. "Engineering Automotive Software". In: *Proceedings of the IEEE* 95.2 (2007), pp. 356–373. DOI: 10.1109/JPROC.2006.888386.

[7] The MathWorks: MATLAB Central. *Simulink Module Tool*. 2021. URL: https://www.mathworks.com/matlabcentral/fileexchange/71952-simulink-module-tool (visited on 08/10/2021).

[8] Advanced Vehicle Technology Competitions. *EcoCAR Mobility Challenge*. 2021. URL: https://avtcseries.org/ecocar-mobility-challenge/ (visited on 08/10/2021).

[9] dSPACE. *ControlDesk*. 2021. URL: https://www.dspace.com/en/inc/home/products/sw/experimentandvisualization/controldesk.cfm (visited on 08/10/2021).

[10] dSPACE. *MicroAutoBox II*. 2021. URL: https://www.dspace.com/en/inc/home/products/hw/micautob/microautobox2.cfm (visited on 08/10/2021).

[11] U.S Department of Energy. *2013 Chevrolet Malibu ECO Hybrid Testing Results*. 2013. URL: \url{www.energy.gov\%2Fsites\%2Fprod\%2Ffiles\%2F2014\%2F02\%2Ff7\%2Fbattery_malibu_3800_0.pdf\&usg=AOvVaw1CC3LhWVlbVYNx-dHORtr5} (visited on 08/10/2021).

[12] U.S. EPA. *Vehicle and Fuel Emissions Testing: Dynamometer Drive Schedules*. 2021. URL: https://www.epa.gov/vehicle-and-fuel-emissions-testing/dynamometer-drive-schedules (visited on 08/10/2021).

[13] GitLab. *GitLab Docs*. 2021. URL: https://docs.gitlab.com/ee/ (visited on 08/10/2021).

[14] K. Grimm. “Software technology in an automotive company - major challenges”. In: *25th International Conference on Software Engineering, 2003. Proceedings.* 2003, pp. 498–503. DOI: 10.1109/ICSE.2003.1201228.

[15] Bernd Hardung, Thorsten Kölzow, and Andreas Krüger. “Reuse of Software in Distributed Embedded Automotive Systems”. In: *Proceedings of the 4th ACM International Conference on Embedded Software*. EMSOFT ’04. Pisa, Italy: Association for Computing Machinery, 2004, 203–210. ISBN: 1581138601. DOI: 10.1145/1017753.1017787. URL: https://doi.org/10.1145/1017753.1017787.

[16] Mike Haussmann. *Development of a Control System for a P4 Parallel-Through-The-Road Hybrid Electric Vehicle*. 2019. URL: http://hdl.handle.net/11375/24786 (visited on 08/10/2021).

[17] Monika Jaskolka, Vera Pantelic, Jason Jaskolka, Alexander Schaap, Lucian Patcas, Mark Lawford, and Alan Wassyng. “Software Engineering for Model-Based Development by Domain Experts”. In: Oct. 2016, pp. 39–64. ISBN: 978-0-12-803773-7. DOI: 10.1016/B978-0-12-803773-7.00003-6.

[18] Monika Jaskolka, Vera Pantelic, Alan Wassyng, and Mark Lawford. “A Comparison of Componentization Constructs in Simulink”. In: Apr. 2020, p. 16. DOI: 10.4271/2020-01-1290.

[19] Monika Jaskolka, Stephen Scott, Vera Pantelic, Alan Wassyng, and Mark Lawford. “Applying Modular Decomposition in Simulink”. In: *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2020, pp. 31–36. DOI: 10.1109/ISSREW51248.2020.00033.

[20] Argonne National Laboratory. “EcoCAR Non-Year Specific Rules - Revision E”. In: (2019).

[21] The MathWorks: Mathworks automotive advisory board (MAAB). *Control algorithm modeling guidelines using MATLAB, Simulink, and Stateflow.* Version 5.0 [Online; accessed Jul 2021]. 2020. URL: https://www.mathworks.com/solutions/mab-guidelines.html.

[22] The MathWorks. *Base and Function Workspaces*. 2021. URL: https://www.mathworks.com/help/matlab/matlab_prog/base-and-function-workspaces.html (visited on 08/10/2021).

[23] The MathWorks. *Check Your Model Using the Model Advisor*. 2021. URL: https://www.mathworks.com/help/simulink/ug/select-and-run-model-advisor-checks.html (visited on 08/10/2021).

[24] The MathWorks. *Goto Tag Visibility*. 2021. URL: https://www.mathworks.com/help/simulink/slref/gototagvisibility.html (visited on 08/10/2021).

[25] The MathWorks. *Goto Tags*. 2021. URL: https://www.mathworks.com/help/simulink/slref/goto.html (visited on 08/10/2021).

[26] The MathWorks. *Longitudinal Driver*. 2021. URL: https://www.mathworks.com/help/vdynblks/ref/longitudinaldriver.html (visited on 08/10/2021).

[27] The MathWorks. *Mapped Motor*. 2021. URL: https://www.mathworks.com/help/autoblks/ref/mappedmotor.html (visited on 08/10/2021).

[28] The MathWorks. *Powertrain Blockset User's Guide*. Version 2021a [Online; accessed Jul 2021]. 2021. URL: https://www.mathworks.com/help/pdf_doc/autoblks/autoblks_ug.pdf.

[29] The MathWorks. *Simulink Design Verifier User's Guide*. Version 2021a [Online; accessed Jul 2021]. 2021. URL: https://www.mathworks.com/help/pdf_doc/simulink/sldv.pdf.

[30] The MathWorks. *Simulink Requirements User's Guide*. Version 2021a [Online; accessed Jul 2021]. 2021. URL: https://www.mathworks.com/help/pdf_doc/slrequirements/slrequirements_ug.pdf.

[31] The MathWorks. *Simulink User's Guide*. Version 2020b [Online; accessed Jul 2021]. 2020. URL: https://www.mathworks.com/help/releases/R2020b/pdf_doc/simulink/simulink_ug.pdf.

[32] The MathWorks. *Stateflow User's Guide*. Version 2021a [Online; accessed Jul 2021]. 2021. URL: https://www.mathworks.com/help/pdf_doc/stateflow/stateflow_ug.pdf.

[33] The MathWorks. *Trace Connections Using Interface Display*. 2021. URL: https://www.mathworks.com/help/simulink/ug/interface-feature-detailed-example-2.html (visited on 08/10/2021).

[34] M.Jaskolka. "Information Hiding for MATLAB Simulink". In: *Research Proposal* (2017).

[35] M.Jaskolka. "Making MATLAB Simulink Models Robust with Respect to Change". In: *Doctoral Thesis* (2020).

[36] M.Jaskolka, V.Pantelic, A.Wassyng, and M.Lawford. "Supporting modularity in Simulink models". In: *arXiv:2007.10120* ().

[37] D. L. Parnas, P.C. Clements, and D.M. Weiss. "The modular structure of complex systems". In: *IEEE Transactions on Software Engineering* SE-11(3) (1985), pp. 259–266.

[38] David Lorge Parnas and Paul C. Clements. "A rational design process: How and why to fake it". In: *IEEE Transactions on Software Engineering* SE-12.2 (1986), pp. 251–257. DOI: 10.1109/TSE.1986.6312940.

[39] D.L. Parnas. "A Technique for Software Module Specification with Examples". In: *Communications of the ACM* 15.5 (May 1972), pp. 330–336.

[40] D.L. Parnas. "On the Criteria To Be Used in Decomposing Systems into Modules". In: *Communications of the ACM* 15.12 (Dec. 1972), pp. 1053–1058.

[41] Alberto Sangiovanni-Vincentelli and Marco Di Natale. "Embedded System Design for Automotive Applications". In: *Computer* 40.10 (2007), pp. 42–51. DOI: 10.1109/MC.2007.344.

[42] Antonio Sciarretta and Lino Guzzella. "Control of hybrid electric vehicles". In: *IEEE Control Systems Magazine* 27.2 (2007), pp. 60–70. DOI: 10.1109/MCS.2007.338280.

[43] McMaster Centre for Software Certification (McSCert). *Simulink Module Tool.* Version 2021 [Online; accessed Jul 2021]. 2021. URL: https://github.com/McSCert/Simulink-Module/blob/master/doc/SimulinkModule_UserGuide.pdf.

[44] W. P. Stevens, G. J. Myers, and L. L. Constantine. "Structured design". In: *IBM Systems Journal* 13.2 (1974), pp. 115–139. DOI: 10.1147/sj.132.0115.

[45] Tilton. *4.5 inch OT-V Carbon Racing Clutches.* 2021. URL: https://tiltonracing.com/product/4-5-ot-v-carbon-racing-clutches/ (visited on 08/10/2021).

[46] Valeo. *Valeo Start-Stop Systems.* 2021. URL: https://www.valeo.com/en/systems-stop-start/ (visited on 08/10/2021).

[47] W Eric Wong, Vidroha Debroy, and Andrew Restrepo. "The role of software in recent catastrophic accidents". In: *IEEE reliability society 2009 annual technology report* 59.3 (2009).

[48] EGAS Workgroup. *Standardized E-Gas Monitoring Concept for Gasonline and Diesel Engine Control Units.* Version Version 6.0 [Online; accessed Jul 2021]. 2015. URL: https://nanopdf.com/download/standardized-e-gas-monitoring-concept-for-gasoline-and_pdf.

[49] YASA. *YASA P400 Product Sheet.* Version 2018 [Online; accessed Jul 2021]. 2018. URL: https://www.yasa.com/products/yasa-p400/.