

## University of Southampton Research Repository ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

**From Requirement Document to Formal Modelling and  
Decomposition of Control Systems**

by

**Sanaz Yeganehfar**

Thesis for the degree of Doctor of Philosophy

August 2014



UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL SCIENCES AND ENGINEERING

Electronics and Computer Science

Doctor of Philosophy

FROM REQUIREMENT DOCUMENT TO FORMAL MODELLING AND  
DECOMPOSITION OF CONTROL SYSTEMS

by **Sanaz Yeganefard**

Formal modelling of control systems can help with identifying missing requirements and design flaws before implementing them. However, modelling using formal languages can be challenging and time consuming. Therefore intermediate steps may be required to simplify the transition from informal requirements to a formal model.

In this work we firstly provide a four-stage approach for structuring and formalising requirements of a control system. This approach is based on monitored, controlled, mode and commanded (MCMC) phenomena. In this approach, requirements are partitioned into MCMC sub-problems, which then will be formalised as independent sub-models. The formal language used in this thesis is Event-B, although the MCMC approach can be applied to other formal languages. We also provide guidelines and patterns which can be used to facilitate the process of modelling in the Event-B language.

The second contribution of this work is to extend the structure of machines in Event-B language and provide an approach for composing the formal MCMC sub-models in order to obtain the overall specification. The composition deals with phenomena that are shared amongst the formal sub-models. In our third contribution, patterns and guidelines are provided to refine the overall formal specification further in order to define design details. In addition, we discuss the decomposition of a formal model of a control system.

As practical examples, the MCMC approach is applied to the requirements of three automotive control systems, namely a cruise control system, a lane departure warning system, and a lane centering controller.





# Contents

<b>Declaration of Authorship</b>	<b>xix</b>
<b>Acknowledgements</b>	<b>xxi</b>
<b>1 Introduction</b>	<b>1</b>
1.1 First Contribution: Structuring RD and Formalising Control Systems . .	3
1.2 Second Contribution: Composition of Sub-models with Shared Phenomena	4
1.3 Third Contribution: Decomposition of Control Systems . . . . .	4
1.4 Outline of Thesis . . . . .	5
<b>2 Requirement Engineering</b>	<b>7</b>
2.1 Requirement and Requirement Engineering . . . . .	7
2.1.1 Requirement, Domain Property and Assumption . . . . .	8
2.1.2 Categorising Requirements . . . . .	9
2.2 Validation and Verification . . . . .	10
2.2.1 Verification . . . . .	10
2.2.2 Validation . . . . .	11
2.3 Overview of Some Requirement Engineering Methods . . . . .	12
2.3.1 Four-Variable Model . . . . .	12
2.3.2 Problem Frames . . . . .	13
2.3.2.1 Phenomena in PF . . . . .	14
2.3.2.2 Frames in PF . . . . .	15
2.3.3 MCC Guidelines for Modelling Control Systems . . . . .	16
2.3.4 KAOS . . . . .	18
2.3.4.1 Goal Model . . . . .	18
2.3.4.2 Agent Model . . . . .	19
2.3.5 Tabular Methods based on Four-Variable Model . . . . .	19
2.3.6 SCR . . . . .	21
2.3.7 WRSPM and a Requirement Tracing Approach . . . . .	22
2.3.8 HJJ . . . . .	23
2.4 Comparison of Some RE Approaches . . . . .	24
2.5 Conclusion . . . . .	26
<b>3 Formal Methods</b>	<b>29</b>
3.1 Formal Methods: Definition, Standards, Advantages and Limitations . . .	29
3.1.1 Advantages of Applying Formal Methods . . . . .	29
3.1.2 Standards and Certifications . . . . .	30
3.1.3 Limitations of Formal Methods . . . . .	31

3.2	Classification of Formal Methods . . . . .	31
3.2.1	First Classification of Formal Methods . . . . .	32
3.2.2	Second Classification of Formal Methods . . . . .	32
3.2.3	Comparison of the Two Classifications . . . . .	33
3.3	Overview of Some Formal Methods . . . . .	33
3.3.1	Z . . . . .	34
3.3.2	VDM . . . . .	35
3.3.3	Classical B . . . . .	35
3.3.4	Comparison of Z, VDM and B-Method . . . . .	36
3.3.5	Action System . . . . .	37
3.3.6	Hoare Logic . . . . .	37
3.3.7	Temporal Logic . . . . .	38
3.4	Model Checking . . . . .	38
3.5	Refinement . . . . .	39
3.6	Conclusion . . . . .	40
<b>4</b>	<b>Event-B: Structure, Refinement and Decomposition</b>	<b>41</b>
4.1	Event-B . . . . .	41
4.1.1	Context . . . . .	42
4.1.2	Machine . . . . .	42
4.1.3	Event . . . . .	43
4.2	Refinement in Event-B . . . . .	44
4.3	Proof Obligation Rules in Event-B . . . . .	45
4.4	Decomposition in Event-B . . . . .	46
4.4.1	Shared-Variable Decomposition . . . . .	46
4.4.2	Shared-Event Decomposition . . . . .	47
4.4.3	Shared-Event Decomposition with Shared Parameters . . . . .	49
4.4.4	Features of the Two Decomposition Styles . . . . .	50
4.5	Shared-Event Composition in Event-B . . . . .	51
4.5.1	Event Composition . . . . .	51
4.5.2	Structure of Machine Composition . . . . .	52
4.6	Rodin: the Event-B Tool . . . . .	53
4.7	Overview of Some Rodin Plugins . . . . .	54
4.8	Conclusion . . . . .	55
<b>5</b>	<b>Structuring, Formalising and Validating Requirements Documents</b>	<b>57</b>
5.1	Overview of the Four-Stage Approach . . . . .	58
5.2	MCMC Variable Phenomena . . . . .	60
5.2.1	MCMC Variable Phenomena . . . . .	61
5.2.2	Identifying MCMC Phenomena of a CCS . . . . .	62
5.2.3	Commanded phenomena . . . . .	62
5.2.3.1	How Commanded Phenomena Help . . . . .	63
5.2.3.2	Ambiguity of Commanded Phenomena . . . . .	64
5.2.3.3	Distinguishing Monitored and Commanded Phenomena . . . . .	65
5.2.4	Mode Phenomenon . . . . .	66
5.3	Stage 1: Structuring an RD Using MCMC Phenomena . . . . .	66
5.4	MCMC Event Phenomena . . . . .	69

5.4.1	Definition of MCMC Event Phenomena . . . . .	69
5.4.2	Mode Event Phenomenon . . . . .	70
5.5	MCMC Event and Variable Phenomena . . . . .	70
5.5.1	Dependencies between MCMC Event and Variable Phenomena . . . . .	71
5.5.2	Instantiating Event and Variable Phenomena Dependencies . . . . .	72
5.6	Stage 2: Formalising a Structured RD . . . . .	74
5.6.1	Refinement Guidelines by Layering an RD based on Variable Phenomena . . . . .	74
5.6.2	Formalising MCMC Sub-Problems as Sub-Models . . . . .	76
5.7	Patterns for Formalising MCMC Variable and Event Phenomena in Event-B . . . . .	77
5.7.1	Defining Variable and Event Phenomena . . . . .	78
5.7.2	A Pattern for Monitor Events . . . . .	79
5.7.3	A Pattern for Mode Events . . . . .	80
5.7.4	A Pattern for Command Events . . . . .	81
5.7.5	A Pattern for Control Events . . . . .	82
5.7.6	Combining Patterns . . . . .	82
5.8	Stage 3: Validation and Revision of RD and Formal Model . . . . .	84
5.8.1	Validation (Traceability) of Model against RD . . . . .	85
5.8.2	Revising an RD and its Formal Model . . . . .	85
5.9	Discussions and Conclusion . . . . .	87
<b>6</b>	<b>Composition of Sub-Models</b> . . . . .	<b>89</b>
6.1	Overview of Composing Sub-Problems . . . . .	89
6.2	Shared Phenomena of Sub-Models in Event-B . . . . .	90
6.2.1	Reconciliation based on Shared-Variable Composition . . . . .	90
6.2.2	Reconciliation based on Shared-Event Composition . . . . .	92
6.2.3	Introducing Shared Variables in Shared-Event Reconciliation . . . . .	93
6.3	Evaluation and Discussion on Reconciliation . . . . .	94
6.3.1	Evaluation of the Shared-Variable Style Reconciliation . . . . .	95
6.3.2	Evaluation of the Shared-Event Style Reconciliation . . . . .	96
6.4	An Example: Sub-Models and Reconciliation of CCS . . . . .	97
6.4.1	MCMC Sub-Models in CCS . . . . .	97
6.4.1.1	Monitored Sub-Model of the CCS . . . . .	97
6.4.1.2	Mode Sub-Model of the CCS . . . . .	98
6.4.1.3	Commanded Sub-Model of the CCS . . . . .	98
6.4.1.4	Controlled Sub-Model of the CCS . . . . .	99
6.4.2	Variable Phenomena Shared amongst Sub-Models of CCS . . . . .	100
6.4.3	Shared-Variable Style Reconciliation for CCS . . . . .	100
6.4.4	Shared-Event Style Reconciliation for the CCS . . . . .	101
6.4.4.1	Reconciling Shared Variable . . . . .	101
6.4.4.2	Reconciling Shared Event . . . . .	102
6.5	Extending Machine Structure . . . . .	103
6.5.1	Extending the Structure of a Machine . . . . .	103
6.5.2	Consistencies between Internal and External Variables . . . . .	106
6.5.3	Composition of Machines with Extended Structure . . . . .	106
6.5.4	An Example of Composing Machines with Extended Structure . . . . .	107

6.5.5	Sharing Read-Only Variables through Witnesses . . . . .	109
6.5.6	Discussions on Extended Machine Structure . . . . .	109
6.6	MCMC Sub-Models Schemas . . . . .	111
6.6.1	Schemas of Machine Structure in MCMC Sub-Models . . . . .	111
6.6.1.1	Mode Sub-Model Schema . . . . .	111
6.6.1.2	Controlled Sub-Model Schema . . . . .	112
6.6.2	An Example: Formalising CCS Sub-Models using MCMC Schemas . . . . .	113
6.7	Discussions and Conclusion . . . . .	115
<b>7</b>	<b>Case Studies: Lane Departure Warning and Lane Centering Controller</b>	<b>117</b>
7.1	Case Study 1: Lane Departure Warning System . . . . .	118
7.2	Stage 1: Structuring RD of LDWS . . . . .	118
7.2.1	Version 1: RD of LDWS based on Public Domain . . . . .	119
7.2.2	Version 2: RD of LDWS based on Expert Feedback . . . . .	121
7.3	Stage 2: Formalising LDWS . . . . .	121
7.3.1	Layering Requirements for Refinement Levels . . . . .	121
7.3.2	Abstract Level . . . . .	122
7.3.3	First Refinement . . . . .	122
7.3.4	Second Refinement . . . . .	124
7.4	Stage 3: Revision and Validation of LDWS . . . . .	125
7.4.1	Version 3: RD of LDWS after Formal Modelling . . . . .	126
7.4.2	Revising the Formal Model . . . . .	127
7.4.3	Validation (Traceability) of LDWS Model . . . . .	128
7.5	Case Study 2: Lane Centering Controller . . . . .	129
7.6	Stage 1: Structured RD of LCC . . . . .	132
7.6.1	Process of Writing LCC RD . . . . .	132
7.6.2	Variable Phenomena and Structured RD of LCC . . . . .	132
7.7	Stage 2: Formalising LCC using Sub-Models . . . . .	135
7.7.1	Variable Phenomena Shared among LCC Sub-Problems . . . . .	136
7.7.2	Identifying and Formalising MCMC Event Phenomena of LCC . . . . .	137
7.7.3	Monitored Sub-Model of LCC . . . . .	138
7.7.4	Controlled Sub-Model of LCC . . . . .	139
7.7.5	Mode Sub-Model of LCC . . . . .	141
7.7.6	Commanded Sub-Model of LCC . . . . .	143
7.8	Stage 4: Composing Sub-Models of LCC . . . . .	144
7.8.1	Composing Shared Events . . . . .	145
7.8.2	Composing Machines . . . . .	146
7.9	Discussions . . . . .	147
7.9.1	Advantages of Formalising Sub-Problems as Composeable Sub-Models . . . . .	148
7.9.2	Derived Phenomena: A New Category of Phenomena . . . . .	148
7.10	Conclusion . . . . .	149
<b>8</b>	<b>Vertical Refinements and Decomposition of Control Systems</b>	<b>151</b>
8.1	Overview of a Control System Decomposition . . . . .	152
8.2	Controller Area Network Bus . . . . .	153
8.2.1	A Message Frame in CAN . . . . .	154

8.2.2	Arbitration in CAN	154
8.2.3	Modelling CAN: Simplifications and Assumptions	155
8.3	Overview of Formalising Design Details	156
8.4	Tasks, Cycle Variable and their Patterns	157
8.4.1	Tasks and their Events Sequences	157
8.4.2	Patterns for Cycle Variable and Events	158
8.5	Guidelines and Patterns for Vertical Refinement	159
8.5.1	Monitored Variable Refinement Pattern	160
8.5.1.1	Introducing Timing to Monitored Variables	160
8.5.1.2	MCMC Events based on Timed Monitored Variable	161
8.5.2	Sensor Refinement Pattern	162
8.5.3	Button Refinement Pattern	163
8.5.4	Actuator Refinement Pattern	165
8.5.5	CAN Bus Refinement Pattern	165
8.5.6	Events Refinement and their Orders	169
8.6	Case Study: Modelling Design Details of a Simplified CCS	170
8.6.1	Overview of the SCCS	171
8.6.2	Concrete Model of the SCCS	172
8.6.2.1	Speed Sensor Task in SCCS	173
8.6.2.2	Controller Task in SCCS	174
8.6.2.3	Actuator Task in SCCS	175
8.6.2.4	Cycle Events in SCCS	177
8.6.3	Refinement Steps of the SCCS	178
8.6.3.1	Abstract Level	178
8.6.3.2	First Refinement: Introducing Cycle Variable	179
8.6.3.3	Second Refinement: Introducing Timed Monitored Variable	179
8.6.3.4	Third Refinement: Controller Task Receives Speed	180
8.6.3.5	Fourth, Fifth and Sixth Refinement: Modelling Speed Sensor Task	180
8.6.3.6	Seventh Refinement: Modelling Set Button Task	180
8.6.3.7	Eighth Refinement: Modelling Actuator Task	181
8.6.3.8	Ninth Refinement: Introducing Maximum Number of Messages on CAN	181
8.6.3.9	Tenth Refinement: Modelling the CAN Bus	181
8.6.4	Proof Obligations in SCCS	182
8.7	Discussion on Vertical Refinements	183
8.7.1	Coarse-Grained Timer and Invariants in Vertical Refinement	183
8.7.2	Assumptions for Vertical Refinements Patterns	183
8.7.3	Modelling of Buttons Differs from Actuators and Sensors	184
8.8	Decomposition of a Control System	185
8.8.1	Decomposition based on Architecture	185
8.8.2	Cycle Component in Decomposition	186
8.9	Case Study: Decomposition of a Simplified CCS	187
8.10	Discussions on Decomposition of Phenomena of a Control System	188
8.11	Conclusion	189

<b>9</b>	<b>Comparison, Future Work and Conclusion</b>	<b>191</b>
9.1	Overview of the Proposed Approach . . . . .	191
9.2	Comparing MCMC Approach and MCC Guidelines . . . . .	194
9.2.1	Clarifying Commanded Phenomena . . . . .	194
9.2.2	Mode Phenomena . . . . .	194
9.2.3	Structuring RD based on Phenomena . . . . .	195
9.2.4	Guidelines on Horizontal Refinement . . . . .	195
9.2.5	Guidelines on Vertical Refinement and Decomposition . . . . .	195
9.3	Comparing the MCMC and Other RE Approaches . . . . .	196
9.3.1	Four-variable model and MCMC . . . . .	196
9.3.2	Problem Frames and MCMC . . . . .	197
9.3.3	KAOS and MCMC . . . . .	198
9.3.4	Tabular Approach of SDV and MCMC . . . . .	199
9.3.5	SCR and MCMC . . . . .	200
9.3.6	WRSPM and MCMC . . . . .	201
9.3.7	HJJ and MCMC . . . . .	202
9.4	MCMC Approach in other Formal Methods . . . . .	202
9.4.1	MCMC Sub-Models in B-Method . . . . .	203
9.4.2	MCMC Sub-Models in Z . . . . .	204
9.4.3	MCMC Sub-Models in VDM . . . . .	204
9.5	Role of Rodin and its Plugins . . . . .	205
9.6	Generalisability and Scalability of the MCMC Approach . . . . .	206
9.6.1	Generalisability of the Approach . . . . .	207
9.6.2	Scalability of the Approach . . . . .	207
9.7	Conclusion and Future Work . . . . .	208
<b>A</b>	<b>Sub-Models of LCC</b>	<b>211</b>
A.1	Contexts of LCC . . . . .	211
A.1.1	Context C0_MNR . . . . .	211
A.1.2	Context C0_CNT . . . . .	211
A.1.3	Context C0_MOD . . . . .	212
A.1.4	Context C0_CMN . . . . .	212
A.1.5	Context C1_MNR . . . . .	213
A.1.6	Context C1_CNT . . . . .	213
A.1.7	Context C2_CNT . . . . .	213
A.1.8	Context C3_CNT . . . . .	214
A.1.9	Context C4_CNT . . . . .	214
A.1.10	Context C5_CNT . . . . .	214
A.2	Monitored Sub-Model of LCC . . . . .	215
A.2.1	Monitored Sub-Model: Abstract Level . . . . .	215
A.2.2	Monitored Sub-Model: First Refinement . . . . .	217
A.2.3	Monitored Sub-Model: Second Refinement . . . . .	219
A.3	Controlled Sub-Model of LCC . . . . .	222
A.3.1	Controlled Sub-Model: Abstract Level . . . . .	222
A.3.2	Controlled Sub-Model: First Refinement . . . . .	223
A.3.3	Controlled Sub-Model: Second Refinement . . . . .	223
A.3.4	Controlled Sub-Model: Third Refinement . . . . .	225

A.3.5	Controlled Sub-Model: Fourth Refinement . . . . .	227
A.3.6	Controlled Sub-Model: Fifth Refinement . . . . .	229
A.3.7	Controlled Sub-Model: Sixth Refinement . . . . .	232
A.4	Mode Sub-Model of LCC . . . . .	236
A.4.1	Mode Sub-Model: Abstract Level . . . . .	236
A.4.2	Mode Sub-Model: First Refinement . . . . .	237
A.4.3	Mode Sub-Model: Second Refinement . . . . .	239
A.4.4	Mode Sub-Model: Third Refinement . . . . .	242
A.4.5	Mode Sub-Model: Fourth Refinement . . . . .	246
A.5	Commanded Sub-Model of LCC . . . . .	249
A.5.1	Commanded Sub-Model: Abstract Level . . . . .	249
A.5.2	Commanded Sub-Model: First Refinement . . . . .	250
A.5.3	Commanded Sub-Model: Second Refinement . . . . .	251
A.6	Composition of the MCMC Sub-Models of LCC . . . . .	251
<b>B</b>	<b>Vertical Refinement and Decomposition of SCCS</b>	<b>263</b>
B.1	Contexts of SCCS . . . . .	263
B.1.1	Abstract Context: C0 . . . . .	263
B.1.2	First Extended Context: C1 . . . . .	264
B.1.3	Second Extended Context: C2 . . . . .	264
B.1.4	Third Extended Context: C3 . . . . .	265
B.1.5	Fourth Extended Context: C4 . . . . .	265
B.2	Vertical Refinement of SCCS . . . . .	265
B.2.1	Abstract Machine: CC0 . . . . .	265
B.2.2	Refinement 1: Machine CC1 . . . . .	266
B.2.3	Refinement 2: Machine CC2 . . . . .	269
B.2.4	Refinement 3: Machine CC3 . . . . .	272
B.2.5	Refinement 4: Machine CC4 . . . . .	275
B.2.6	Refinement 5: Machine CC5 . . . . .	279
B.2.7	Refinement 6: Machine CC6 . . . . .	283
B.2.8	Refinement 7: Machine CC7 . . . . .	288
B.2.9	Refinement 8: Machine CC8 . . . . .	294
B.2.10	Refinement 9: Machine CC9 . . . . .	302
B.2.11	Refinement 10: Machine CC10 . . . . .	311
B.2.12	Refinement 11: Machine CC11 . . . . .	319
B.2.13	Refinement 12: Machine CC12 . . . . .	327
B.3	Decomposition of SCCS . . . . .	335
B.3.1	Button Machine . . . . .	335
B.3.2	Environment Machine . . . . .	336
B.3.3	Sensor Machine . . . . .	337
B.3.4	Controller Machine . . . . .	338
B.3.5	Actuator Machine . . . . .	341
B.3.6	Cycle Machine . . . . .	342
B.3.7	CAN Machine . . . . .	345
	<b>References</b>	<b>349</b>





# List of Figures

2.1	Four-variable model (based on [MT01]). . . . .	13
2.2	A simple problem diagram [Jac05]. . . . .	13
2.3	Problem frame diagram of a traffic light [Jac01]. . . . .	14
2.4	Required behaviour frame [Jac01]. . . . .	16
2.5	Commanded behaviour frame [Jac01]. . . . .	16
2.6	A commanded control system. . . . .	17
2.7	A goal model for a cruise control system [PD09]. . . . .	19
2.8	Decomposition of proof obligations [LFM04]. . . . .	21
2.9	WRSPM reference model. . . . .	23
2.10	An overview of the HJJ approach [HJJ03, JHJ07]. . . . .	24
2.11	Sluice gate specification in HJJ [JHJ07]. . . . .	25
2.12	Four-variable model in a problem diagram [Jac01]. . . . .	25
3.1	A state space schema in Z [Bow03]. . . . .	34
3.2	An operation schema in Z [Bow03]. . . . .	34
3.3	An overview of a module in VDM. . . . .	36
4.1	Event-B machines and contexts and their relation. . . . .	42
4.2	Structure of an event in Event-B language. . . . .	44
4.3	Shared-variable decomposition style. . . . .	47
4.4	Shared-event decomposition style. . . . .	48
4.5	Event-B machine which is to be decomposed [But09c]. . . . .	48
4.6	Machines VW1 is decomposed into V1 and W1 [But09c]. . . . .	49
4.7	An event with shared parameter [But09c]. . . . .	49
4.8	Decomposition of an event with shared parameter [But09c]. . . . .	50
4.9	Composition of two events with a shared parameter. . . . .	52
4.10	Structure of a shared-event composed machine. . . . .	52
5.1	Overview of the four stages of the MCMC approach. . . . .	60
5.2	A commanded control system. . . . .	61
5.3	Structuring RD based on MCMC phenomena. . . . .	67
5.4	Modes of the CCS and their transitions. . . . .	71
5.5	Formalising CCS using Refinement. . . . .	76
5.6	Formalising MCMC sub-problems as composeable sub-models. . . . .	77
5.7	The pattern of a monitor event updating $MNR_i$ . . . . .	79
5.8	A monitor event updates the actual speed in the CCS. . . . .	79
5.9	The pattern of a mode event updating <i>mode</i> . . . . .	80
5.10	A mode event in the CCS. . . . .	81

5.11	The pattern of a command event updating $CMN_i$ .	81
5.12	A command event for setting the target speed in the CCS.	82
5.13	The pattern of a control event updating $CNT_i$ .	82
5.14	A control event for updating the acceleration in the CCS.	82
5.15	Composition of command and mode event patterns.	83
5.16	Composition of <i>Activate</i> and <i>SetTargetSpeed</i> events.	84
6.1	Overview of the composition of sub-models (Stage 4 of the MCMC approach).	90
6.2	Shared-variable composition style.	91
6.3	Reconciliation of M1 and M2 based on the shared-variable style.	91
6.4	Shared-event composition style.	92
6.5	Shared-event reconciliation with shared variable $v3$ .	93
6.6	Two design decisions for modelling requirements in a single model.	95
6.7	Evaluation of shared-variable style reconciliation.	95
6.8	Shared-event style reconciliation when the requirements are defined concurrently.	96
6.9	Shared-event style reconciliation of <i>mode</i> when the requirements are defined sequentially.	97
6.10	MNR sub-model in CCS.	98
6.11	MOD sub-model in CCS.	99
6.12	CMN sub-model in CCS.	99
6.13	CNT sub-model in CCS.	100
6.14	CNT sub-model includes an abstraction of the mode events.	101
6.15	Mode is captured as a shared variable phenomenon in <i>UpdateAcceleration</i> using the shared-event reconciliation.	102
6.16	<i>Activate</i> and <i>SetTargetSpeed</i> in their corresponding sub-models.	102
6.17	<i>SetTargetSpeed</i> is revised to reconcile the shared <i>mode</i> variable.	103
6.18	Extended structure of a machine.	105
6.19	An example of importing and exporting variables in an extended machine.	108
6.20	Composition of the machines X and Y.	108
6.21	An example of the extended shared variable composition.	110
6.22	A schema for a mode (MOD) sub-model.	112
6.23	A schema for a controlled (CNT) sub-model.	114
6.24	MNR, MOD, CNT and CMN sub-models of the CCS based on the MCMC schemas.	115
6.25	Composition of MNR, MOD, CNT and CMN sub-models of the CCS.	115
7.1	Warning and no-warning zone for an LDWS.	118
7.2	Possible mode changes in LDWS.	120
7.3	Abstract level consists of controlled phenomenon <i>warning</i> and its events.	123
7.4	Two <i>SwitchOff</i> operator mode events are defined in the first refinement.	123
7.5	Control event <i>IssueWarning</i> in the first refinement.	123
7.6	Control event <i>IssueWarning</i> at the second refinement.	125
7.7	Limited vision field of cameras when car is travelling on a lane boundary.	126
7.8	Control events after revising the formal model of LDWS.	128
7.9	Overview of LCC inputs and output.	130
7.10	LCC actuates steering angle using target and predicted path.	130

7.11	Possible mode changes in LCC. . . . .	135
7.12	The mode event <i>Activate</i> in the MOD sub-model. . . . .	138
7.13	Refinement steps in the monitored sub-model of the LCC. . . . .	138
7.14	An overview of the monitored (MNR) sub-model of the LCC. . . . .	139
7.15	Refinement steps in the controlled sub-model of the LCC. . . . .	140
7.16	An overview of the controlled (CNT) sub-model of the LCC. . . . .	142
7.17	Refinement steps in the mode sub-model of the LCC. . . . .	142
7.18	An overview of the mode (MOD) sub-model of the LCC. . . . .	143
7.19	Refinement steps in the commanded sub-model of the LCC. . . . .	144
7.20	An overview of the commanded (CMN) sub-model of the LCC. . . . .	144
7.21	<i>StopWarning_DisplayOff</i> and <i>Override_Indicator</i> can be composed. . .	145
7.22	<i>StopWarning_DisplayOff</i> and <i>SwitchOff_FromActive</i> can be com- posed. . . . .	146
7.23	An overview of the composed model of the LCC. . . . .	147
7.24	An alternative overview of LCC to Figure 7.9. . . . .	149
8.1	A message frame in CAN. . . . .	154
8.2	ECUs can be transmitters, receivers or both. . . . .	155
8.3	An overview of vertical refinement. . . . .	156
8.4	An example of the timed speed variable. . . . .	161
8.5	Sense events. . . . .	162
8.6	An actuator event. . . . .	165
8.7	Transmission of a message between ECUs. . . . .	167
8.8	Vertical refinement of introducing sensors. . . . .	169
8.9	Vertical refinement of introducing buttons. . . . .	170
8.10	Vertical refinement of introducing actuators. . . . .	170
8.11	Overview of the connection in the simplified CCS. . . . .	171
8.12	Sequence of events in speed sensor task in the concrete level. . . . .	173
8.13	The concrete model of the speed sensor task (T2). . . . .	174
8.14	The order of events in the controller task (T1). . . . .	175
8.15	The concrete model of the controller task (T1). . . . .	176
8.16	The final two events in the concrete model of T1. . . . .	176
8.17	The concrete model of the actuator task (T4). . . . .	176
8.18	Updating cycle variable in the concrete level. . . . .	178
8.19	Events of a simplified cruise control system (SCCS). . . . .	179
8.20	The variable <i>speed</i> models the monitored phenomenon car speed. . . . .	180
8.21	An overview of vertical refinement from phenomena point of view. . . . .	184
8.22	An overview of the decomposition based on the architecture. . . . .	186
8.23	An overview of the data flow in the decomposition based on the cycle. . .	187
8.24	An overview of the SCCS decomposition. . . . .	188
8.25	An overview of decomposition from phenomena point of view. . . . .	188
9.1	Overview of refinement steps and decomposition of a control system. . . .	192
9.2	The MCMC approach has more phenomena than the four-variable model. . .	196
9.3	MCMC phenomena and their vertical refinement in PF. . . . .	198
9.4	A representation of agents of the MCMC approach in a context diagram in KAOS. . . . .	199



# List of Tables

2.1	The SRS table of a pressure trip [LFM04]. . . . .	20
2.2	A pressure trip SDD and abstract function [LFM04]. . . . .	21
2.3	Comparison of the four-variable model based requirement engineering approaches. . . . .	27
4.1	Deterministic and nondeterministic assignments in Event-B. . . . .	45
5.1	Some requirements of a CCS (based on [Abr09]). . . . .	62
5.2	MCMC variable phenomena of CCS. . . . .	62
5.3	Brake pedal is an existing interface for CCS. . . . .	66
5.4	MCMC variable phenomena of CCS are listed to structure the RD. . . . .	68
5.5	Structured RD of the CCS. . . . .	68
5.6	MCMC event phenomena of CCS. . . . .	69
5.7	Dependency between MCMC event and variable phenomena. . . . .	72
5.8	Dependency between MCMC event and variable phenomena. . . . .	73
5.9	Dependency between MCMC event and variable phenomena. . . . .	73
5.10	Validation of RD of CCS against its model. . . . .	86
6.1	Shared variables of MCMC sub-models. . . . .	100
7.1	Monitored and controlled variable phenomena of the LDWS. . . . .	119
7.2	RD version 1: Requirements are identified based on public domain. . . . .	120
7.3	RD version 2: Requirements are added based on experts feedback. . . . .	121
7.4	Formal monitored and commanded variables and their events. . . . .	125
7.5	Third RD: Requirements added based on Formal Modelling. . . . .	127
7.6	Validation of the Requirement against the model. . . . .	129
7.7	Structured requirements of LCC - Monitored sub-problem. . . . .	133
7.8	Structured requirements of LCC - Controlled sub-problem . . . . .	134
7.9	Structured requirements of LCC - Commanded sub-problem . . . . .	134
7.10	Structured requirements of LCC - Mode sub-problem . . . . .	136
7.11	Variables shared among the MCMC sub-models of the LCC. . . . .	137
8.1	The MCMC event and variable phenomena based on timed monitored variables. . . . .	161
8.2	MCMC event and variable dependencies when introducing sensors. . . . .	163
8.3	MCMC event and variable dependencies when introducing buttons. . . . .	164
8.4	Control event when introducing actuators. . . . .	166
8.5	MCMC event and variable dependencies when introducing buttons. . . . .	168
8.6	Statistics of the proof obligations in SCCS. . . . .	182

9.1 Agents and their capabilities in the MCMC approach. . . . .	199
---	-----

## Declaration of Authorship

I, **Sanaz Yeganefard** , declare that the thesis entitled *From Requirement Document to Formal Modelling and Decomposition of Control Systems* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
- parts of this work have been published as: [YBR10], [YB11], [YB12] and [YB13].

Signed:.....

Date:.....





## Acknowledgements

I am deeply indebted to my supervisor, Prof. Michael Butler, not just for his expertise and advice but also for his continual support and encouragement and for his trust in me. Especial thanks to Manoranjan Satpathy, S. Ramesh and Silky Arora from GM India Science Lab who have supported the case studies of this research. I would also like to thank members of ESS and the former DSSE group, especially John Colley, Andy Gravell and Mike Poppleton for their useful feedback.

Thanks to my families and friends both in Iran and the UK for their support and encouragement. I would like to thank Jill and Arthur for all their support and kindness. Special thanks to my mother who has been an inspiration to my achievements and to my sister for motivating me and supporting me at every moment of my life. A special thank also to Tim for his patience, love, trust and companionship in pursuing my dream. There are many more people whose words kept me encouraged and I am grateful to every single one of them.



*To my golden triangle: Nasim, Tannaz and Tim.*



# Chapter 1

## Introduction

This thesis focuses on control systems which are reactive systems interacting with and reacting to their evolving environment. We are mainly interested in control systems that can perform complicated functionalities in life-critical situations. Such systems interact with their environment continuously, which makes them complex. This means that the process of constructing and structuring a functional requirements document (RD) of such control systems and designing them can be challenging and time consuming.

Clear and comprehensive requirements are essential prerequisites for developing a control system. It is well-known that errors in requirements can be very costly to fix. It is reported that “a mature requirements specification is an indispensable prerequisite for dependable software systems”, as 40% of the errors in a software-based function can be traced to immature requirements specification [Dep09].

In the automotive sector which has shown a significant increase in the usage of control systems (a premium car in 2006 implemented about 270 functions with which a user interacts, deployed over about 70 embedded platforms [BKPS07]), software errors can leave car manufacturers with bad publicity of their brand and perhaps with a recall of their cars. The latter was the case for Toyota with the “recall of 160,000 of its Prius hybrid vehicles” in which software errors were the cause of the bugs in warning lights, the gasoline engine and the electrical motor [Dep09].

Formal methods are mathematical based techniques used for specification, development and verification of systems [Win90]. Modelling using formal methods is known to improve understanding of a system and thus help to find missing and ambiguous requirements [Hal90]. Furthermore, constructing a formal model supports formal verification of the system properties, which can improve safety and reliability of the system design. However, one of the difficulties in using rigorous formal modelling is the transition from informal requirements to a formal model. In addition, using formal methods is perceived as requiring expertise.

The main aims of this thesis is to firstly facilitate the process of formalisation by defining intermediate steps and secondly, reduce the barriers of formal modelling so it can be used by modellers less expert in formal methods. To achieve the latter goal, we provide patterns, guidelines and schemas for modelling a control system in the Event-B language [Abr10]. Event-B is a refinement-based formal language with simple and extendible notation. We take advantage of the refinement and (de)composition techniques in Event-B to develop approaches for specifying requirements formally and decomposing a model.

The contributions of this thesis are:

- a four-stage approach for formalising an RD of a control system, Section 1.1;
- extending the machine structure in Event-B to facilitate formalisation of a control system as composable sub-models, Section 1.2;
- an approach for decomposing a model of a control system into components which correspond to physical components in the architecture of the control system, Section 1.3. As will be discussed, model decomposition requires further refinement steps through which design details will be introduced to the model.

The proposed four-stage approach and the decomposition approach can be applied using any state-based formal language that supports refinement and (de)composition. However, the patterns provided in this thesis follow the Event-B notations. The proposed patterns are also applied to several case studies to determine their suitability and provide examples of the patterns.

Although a case-study-based approach might have limitations, we have overcome some of these. For instance, the application of patterns depends on the knowledge of the modeller. Therefore, in addition to us, the patterns were used by other modellers, e.g. [Col12]. It is important to notice that a level of expertise is required in order to use the patterns, and they do not constrain a modeller. Another limitation of a case-study-based approach is the range of case studies that are used. The MCMC approach has been applied to several types of control systems, while it was being developed and after the approach presented in this thesis was completed. Examples of the range of case studies are a cruise control system [YBR10], a washing machine controller [Col12], a full authority digital engine controller (FADEC) [Das10] and a sluice gate controller [PS12].

## 1.1 First Contribution: Structuring RD and Formalising Control Systems

Formalising requirements and modelling all variables and events in one step (i.e., flat development) can be complex and time consuming, as functionalities of control systems are usually broad and their RDs are complex. To avoid formalising a long list of requirements in a single step, *refinement* techniques are used. The style of refinement where behaviours and requirements are introduced to the model, is referred to as *horizontal refinement*. However, it is usually “not easy to decide how to organise the construction steps” and to determine the abstract model [Abr06]. This is one reason for approaches such as UML-B [SB06], which combines UML and the B method, being developed, since they can be used as intermediate steps to facilitate the process of modelling.

In our first contribution, we propose a four-stage approach for structuring and formalising an informal RD of a control system. This approach consists of patterns and guidelines that can be used to facilitate the transition between a functional requirements document (RD) and its formal representations in Event-B. The stages of the approach are as follow:

1. *structuring an RD* into sub-problems;
2. *formalising the sub-problems* as sub-models;
3. *composing sub-models* to obtain the overall specification;
4. *validating* the overall model against the requirements.

The guidelines and patterns of the four-stage approach are based on the concept of *monitored, controlled, mode* and *commanded* (MCMC) phenomena. The term phenomena is influenced by Jackson’s problem frames approach [Jac01]. Also, the definition of monitored and controlled phenomena is based on Parnas and Madey’s four-variable [PM95].

The proposed patterns and guidelines for structuring and formalisation are improved and evolved as the result of experimenting with different case studies. The definition of the MCMC phenomena and the approach for formalising requirements are evolved from the monitored, controlled and commanded (MCC) modelling guidelines [But09a]. The MCC modelling guidelines were applied to several case studies [YBR10, YB12, Das10, PS12], which helped us build upon the strengths and weaknesses of the guidelines.

As a practical example, the phenomena of two automotive systems, a lane departure warning system (LDWS) and a lane centering controller (LCC) are identified and their RDs are structured accordingly. These case studies have been supported by industrial partners. In addition to this, a cruise control system (CCS) is used as a running example,



although this system is not explained in detail and it is used to clarify the MCMC phenomena and approach.

## 1.2 Second Contribution: Composition of Sub-models with Shared Phenomena

Formalising an RD as sub-models means there will be phenomena that are shared amongst the sub-models. In our second contribution we investigate the suitability of the *shared-variable* [AH07] and the *shared-event* [But09b] decomposition styles in Event-B for formalising the shared phenomena and also composing the sub-models that share phenomena.

The results of our research will show the advantages and disadvantages of using these two decomposition styles for modelling shared phenomena and composing sub-models. We take advantage of this result in order to extend the structure of machines in Event-B. The extended structure consists of two new clauses, namely *exports variables* and *imports variables*. Using these clauses a sub-model can allow other sub-models to have read-only accessibility to any of its variables.

Based on the extended structure, we define *schemas* which are the framework for modelling a control system as composable sub-models. The schemas provide an overview of the monitored, controlled, mode and commanded sub-models.

## 1.3 Third Contribution: Decomposition of Control Systems

Introducing requirements gradually using refinement techniques usually results in increasing the complexity and the size of the model. Thus, the model can become difficult to manage. One way of overcoming this complexity is to decompose a model into smaller pieces [AH07]. The final focus of this research is on the *decomposition of a formal model* of a control system. Our main objective in decomposing a control system is to extract *software specifications* from system specifications.

In addition, decomposition helps in separating *implementable parts* of the model of a system from other components, such as user interface and sensors. This would firstly help with the process of implementation and secondly, it is suitable for team working since different components can be developed by different engineers.

However, before starting the process of decomposition, it is necessary to decide on the *architecture* of the system. This is because decomposition is not an ad-hoc process,

but it should correspond to the system architecture. Furthermore, it is necessary to introduce *low-level design details* (user interface, sensors, actuators and communication bus) in order to prepare a model for decomposition. Therefore, further refinement levels are required before decomposing a model.

## 1.4 Outline of Thesis

This thesis is organised in nine chapters. After discussing the introduction the following chapters represent the background of this work:

- Chapter 2 discusses requirements and requirement engineering (RE). In addition some of the specification and RE techniques that relate to the proposed MCMC approach are described.
- Chapter 3 provides the definition of formal methods, their advantages and their usage in standards and certificates. Also, brief descriptions of some formal methods are given.
- Chapter 4 describes the Event-B formal method that is used in this research.

Our contributions are discussed in the following chapters:

- Chapter 5 explains our first contribution, which is on structuring and formalising RD of a control system. The case study of a CCS is also discussed here.
- Chapter 6 discusses the second contribution, which is extending the structure of an Event-B machine in order to facilitate composition of sub-models with shared phenomena.
- Chapter 7 describes the case studies. Here the MCMC approach of structuring and formalising is applied to the two case studies of a lane departure warning system (LDWS) and a lane centering controller (LCC). The sub-models of the LCC are defined based on the extended machine structure in Event-B.
- Chapter 8 discusses the third contribution, which is the refinement of a control system in order to introduce design details of sensors, actuators, buttons and a communication bus. This chapter also presents the decomposition of a control model.
- Chapter 9 provides a comparison of the provided approaches with other specification techniques.



## Chapter 2

# Requirement Engineering

In this chapter, we first explain the terms requirement and requirement engineering (RE), as well as other commonly used terms in RE. After that in Section 2.2, we consider the definition of validation and verification. In Section 2.3, some of the requirement modelling and specification techniques which are related to the approach we propose are discussed. A brief comparison of the specification techniques is given in Section 2.4. Section 2.5 provides the conclusion of this chapter.

### 2.1 Requirement and Requirement Engineering

Software applications are developed to solve problems defined by the customer. However, before developing any application, it is essential to understand the problem. The term *requirement* refers to the descriptions of the services that the system should provide for the stakeholder and the constraints on these services.

Receiving the requirements partially or fully from the customer, and understanding them is the first required step in almost any software process model. For instance, in the traditional development process of *waterfall*, which is a sequential approach to software life cycle, a set of requirements are agreed with the customer at the first stage. Also, in an *incremental* development process, which is an iterative and sequential approach, the requirements of one increment at a time are captured in detail.

Informally speaking, the set of activities which helps to define, understand, document and specify the requirements are called *requirement engineering*. More formally, requirement engineering is a process which involves [Som07]:

- Feasibility study: Studying the usefulness of the system.

- Requirements elicitation and analysis: This step includes gathering and classifying requirements as well as prioritising and negotiating them, especially when several stakeholders are involved.
- Requirements specification: The process of writing down the requirements. System requirements specification can be written in different ways, such as natural language, graphical notations or formally using mathematical specification. Notice that while requirements define “the boundaries of the solution space” (requirements describe *what*), the specifications determines “a few solutions within that space” (specifications describe *how*) [BD09].
- Requirements validation: In this stage the requirements are checked to ensure that they represent the customer needs.
- Requirements management: As system requirements can change, it is important to understand and control these changes.

### 2.1.1 Requirement, Domain Property and Assumption

The statements involved in requirement engineering can be either *descriptive* or *prescriptive*. The former represents system properties which hold regardless of the system behaviour, an example is the statement ‘if the system is off, it cannot be active’. However, a prescriptive statement is a desirable system property that should hold. The definition of requirements, domain properties and assumptions are discussed below based on [vL09]:

- **System requirements** are prescriptive statements written in terms of environmental quantities, such as *car speed* in a cruise control system, or *train doors* and *train speed* in a train central controller. So, in a system requirements document, the software which will be designed is treated like a black-box incorporating with other components [PM95]. System requirements are sometimes referred to as high-level requirements or customer requirements [BH05]. Examples of system requirements are “cruise control system shall be activated when the car speed is greater than a certain threshold” or “all train doors shall always remain closed while a train is moving”.
- **Software requirements** are prescriptive statements written in terms of quantities that the software-to-be and the environment share, such as the measured or sensed value of a car speed or a train door and speed. The software requirements will be enforced by the software only. A software requirement document is used by developers. An example of a software requirement is “the doorsState output variable shall always have the value closed when the measuredSpeed input variable has a non-null value”.

- **Domain properties** are descriptive statement about the problem world which is derived from the system domain. Domain properties hold regardless of the behaviour of the system, and even regardless of the existence of the system. These properties cannot be changed or modified. For instance physical laws, such as the relation between speed, acceleration and time, are domain properties. An example of a domain property is “a train is moving if and only if its physical speed is non-null”.
- **Assumptions** are constraints on the environment that should be satisfied by the environment. In comparison with domain properties, assumptions can be changed or modified. An example of an assumption is “a train’s measured speed is non-null if and only if its physical speed is non-null”. So, it is assumed that the speed sensors are fault-free.

The system and software requirements can be linked by defining the four variables *monitored*, *controlled*, *input* and *output* [vL09, PM95]. Monitored (M) and controlled (C) variables are environmental quantities respectively monitored and controlled by the software. Inputs (I) are data that the software receives and outputs (O) are quantities produced by the software. Inputs and outputs respectively correspond to monitored and controlled quantities. These quantities are defined in four variable model [PM95] which will be discussed later on.

Using these four variables, system requirement can be defined as the relation between M and C (the possible set of monitored and controlled variables for a system). However, software requirements can be defined as the relation between I and O. Parnas and Madey discuss that a software requirements document is a combination of a system requirements document and a *system design document* which captures the communication between I/O devices and software, the description of the devices and the values for input and output variables [PM95].

### 2.1.2 Categorising Requirements

Requirements are broadly categorised into functional and non-functional. Although for some systems the distinction between these two categories is unclear, in this section a common interpretation of functional and non-functional requirements according to Sommerville is explained.

- **Functional Requirements:** These describe the system behaviour and the facilities it should provide. In other words, they represent what the system should do.

- **Non-functional requirements:** These are requirements that do not directly specify the behaviour and the service that the system should provide, but they define system properties such as reliability and response time. Also they can define system constraints, such as the features of I/O devices. Non-functional requirements are classified into performance, usability, safety, security, etc. [Som07].

## 2.2 Validation and Verification

Validation and verification (V&V) are important topics in software engineering. Their objective is to “identify and resolve software problems and high-risk issues early in the software lifecycle” to reduce the costs involved in finding and fixing problems later in the lifecycle [Boe84].

V&V are interpreted differently in various approaches. Sometimes these terms are used interchangeably, although they refer to two different activities in software engineering. In this section, we discuss the most commonly used meanings of these terms based on the IEEE terminology. After that our usage of the terms validation and verification is clarified.

### 2.2.1 Verification

Verification has two definitions in *IEEE Standard Glossary of Software Engineering Terminology*.

Firstly, it is contrasted with validation as “the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase”. The second is “formal proof of program correctness” [IEE90].

Verifying a system means ensuring that “we are building the product right” [Boe84]. Therefore, verification is used to show that a software product conforms to its specification.

We use the term verification to refer to *formal verification*. In a formal verification, mathematical techniques are used to verify that a *model* of a system would satisfy its design properties within the environment the system is supposed to operate in [Bje05]. When the specified design properties are not satisfied, some formal verification techniques provide counterexamples. Generally speaking there are two techniques for verification, namely theorem proving and model checking [CW96].

- **Theorem proving:** In this technique a system and its desired properties are both formulated in some mathematical logic. The logic is given by the used formal language and it consists of some axioms and inference rules. In theorem proving the attempt is to find proofs for the specified properties based on the axioms and proof rules of the system.
- **Model checking:** This is an algorithmic and automatic technique used for verification of concurrent systems. To check the satisfaction of a system property, a model of the system is provided. Also, the property is formulated, for instance in temporal logic. After that the model of the system is checked exhaustively to determine whether the formulated property holds. Generally speaking, the model of the system is translated into an automaton and it is checked whether the formulated property is an acceptable language of the automaton.

While theorem proving can be used for reasoning infinite state systems, the model checking techniques can be used for systems with finite state space. A further discussion on model checking is provided in Section 3.4.

### 2.2.2 Validation

Validation is defined in *IEEE Standard Glossary of Software Engineering Terminology* as

“The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements” [IEE90].

Validating a system is undertaken to ensure that “we are building the right product” [Boe84]. Generally speaking validation refers to checking that a developed system meets its stakeholders’ expectations. However, validation can happen in different stages of a software development lifecycle. In the initial stage of a lifecycle validation means ensuring that requirements and assumptions represent the stakeholders views of the system, while at the final stage of the development, validation can mean checking the final product against the stakeholders expectations. The above definition of validation can be compared to the first definition of verification by *IEEE Standard Glossary of Software Engineering Terminology*.

However, we use the term validation to refer to the process of validating the formal specification or the formal model against its requirements. This interpretation can be compared to the second definition of verification (formal verification). So, by validation we mean ensuring that ‘we are building the right model with respect to its requirement’, while it is assumed that the requirement represent the stakeholder expectations.



## 2.3 Overview of Some Requirement Engineering Methods

In this section an overview of the following requirement techniques is given:

- Four-variable model [PM95];
- Problem frames (PF) [Jac01];
- MCC guidelines [But09a, But09d];
- Keep All Objectives Satisfied (KAOS) [vL09];
- A tabular method based on the four-variable model [LFM04, WL03];
- Software Cost Reduction (SCR) [HJL96];
- WRSPM [GGJZ00b] and requirement tracing based on WRSPM [JHLR10];
- HJJ [HJJ03].

The reason for choosing these techniques is that they have similarities to the MCMC approach which we have developed. For instance, both SCR and the MCMC are based on the four-variable model. WRSPM looks at phenomena (states and transitions) of a system and its environment which is also the case in MCMC. Also, similarly to MCMC, HJJ deals with requirements of control systems.

### 2.3.1 Four-Variable Model

The four-variable model [PM95] is based on the suggestion for identifying four variables in order to document requirements of a system. The first two variables are environmental quantities. These usually represent physical properties of the system and can be shown by mathematical variables. Environmental quantities are categorised to *monitored variables* (MON in Figure 2.1) which the system monitors and responds to, and *controlled variables* (CON in Figure 2.1) whose states are modified by the system.

The relations between controlled and monitored variables are defined by NAT and REQ. NAT is the relation in the environment without the computer system. REQ represents the state of controlled variables in response to monitored variables where the computer system exists. In other words, while NAT defines the behaviour of the environment, i.e. domain properties and assumptions, REQ defines the system requirements.

The two other variables are *input* and *output*. The values of monitored variables are stored in input variables when they are read by the system, while outputs are variables that the system sets in order to change states of controlled variables. The relations between inputs and monitored variables are defined as IN. Also, OUT represents the

relations between outputs and controlled variables. These variables and their relations are shown in Figure 2.1.

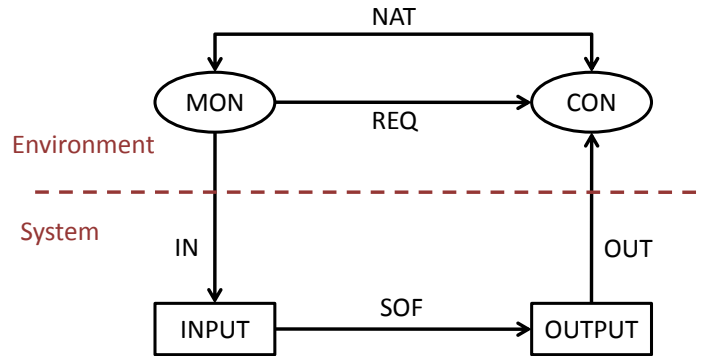


Figure 2.1: Four-variable model (based on [MT01]).

Separating the four variables in a system helps it to be more robust [MT01]. This is because identifying monitored and controlled variables means the system boundaries are defined explicitly. Also when these variables are chosen correctly, IN and OUT relations might change only due to changes to the underlying hardware. Also, REQ may change when customer needs and consequently the requirements change.

### 2.3.2 Problem Frames

The problem frame approach (PF) [Jac01, Jac04, Jac05] distinguishes between the problem and the solution with the aim to focus on the problem domain in requirement analysis. It also provides *problem frames* which are patterns for identifying different problem classes. When the system under development is more complex, PF suggests to decompose the problem into subproblems which can be mapped to a known problem frame.

Figure 2.2 represents a simple PF diagram which consists of *requirements*, a *problem world* and a *machine*. Requirements represent the system's goal. More specifically, they describe the behaviour expected from the world, which is where the problem is located. Satisfaction of requirements is to be ensured by the machine, which is the software to be built. In a PF diagram rectangles denote domains. As shown in Figure 2.2, a rectangle with three vertical stripes denotes a machine domain, while the problem world is denoted by a rectangle with no stripes. This is a given domain, meaning that we do not have the freedom to design the problem world domain.

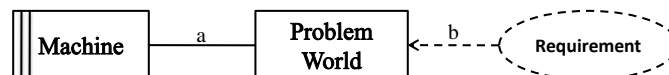


Figure 2.2: A simple problem diagram [Jac05].

Figure 2.2 shows that domains are connected by interfaces of *shared phenomena*, i.e. shared states and events. The term *phenomena* is explained further later on. In Figure 2.2, the interface *a* consists of a set of shared phenomena through which the machine influences and interacts with the problem world. This means both domains can participate in some actions on the shared phenomena.

Requirements are usually conditions on the states and behaviours of the problem world. Requirements most likely refer to phenomena of the problem world. This is shown in Figure 2.2 by linking requirements and the problem world through phenomena *b*.

Phenomena *a* and *b* are respectively referred to as *specification phenomena* and *requirement phenomena*. Usually these phenomena are different and their gap is bridged by *domain properties*. In other words, satisfaction of *requirements* (which refer to phenomena *b*) depends on the *machine specification* (which is the machine behaviour at interface *a*) and *domain properties*.

As an example of a PF diagram a simple traffic light system is shown in Figure 2.3. Here, the interface *b* shows that the requirements refer to the phenomena *Stop* and *Go*. The notation LU! shows that these phenomena are controlled by the *light units* domain. Also, the interface *a* illustrates that the requirements will be satisfied by the machine through modifying *RPulse* and *GPulse*.

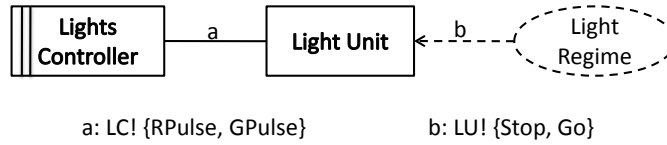


Figure 2.3: Problem frame diagram of a traffic light [Jac01].

### 2.3.2.1 Phenomena in PF

We have used the term *phenomena* to explain the interface between the domains. In PF six types of phenomena are identified. The first three are of kinds *individuals* and the remainder are of kinds *relations*. Individuals share the property that they can be individually named and distinguished from others in this category. Phenomena of kinds *relations* represent relations among individuals. The subcategories and their descriptions are given below.

- **Individuals**

- **Events** are “indivisible and instantaneous” individuals which happen at a particular point in time. An event takes no time to happen.

- **Entities** exist within the system over time, while their properties and states can be changed from time to time. An example can be the car speed in a cruise control system.
- **Values** are fixed and intangible individuals. An example is a number.
- **Relations**
  - **States** represent the relation between entities and values which can change over time. For instance a car speed can be 0 or more.
  - **Truths** are fixed and unchangeable relations between individuals, for instance the relation  $3 < 5$  always holds.
  - **Roles** express arguments of an event. So, a role is a relation between an event and individuals.

In another category, phenomena are divided into *causal* and *symbolic*. The former is directly controlled or caused by domains. Events, roles and states which relate entities are causal phenomena, while values, truth and states which relates values are symbolic phenomena.

### 2.3.2.2 Frames in PF

PF defines five basic *problem frames*, namely required behaviour, commanded behaviour, information display, simple workpieces and transformation. These are patterns which describe problem classes in terms of their domains, interfaces and requirements. In this section two of these frames are discussed.

In PF domains are divided into three types: *causal*, *biddable* and *lexical*. Properties of a causal domain are predictable relationships among the causal phenomena of the domain. Biddable domains usually involve people, and thus they are not predictable. Lexical domains represent symbolic phenomena.

- **Required Behaviour:** This frame, shown in Figure 2.4, captures the following class of problems:

“There is some part of the physical world whose behaviour is to be controlled (controlled domain) so that it satisfies certain conditions (requirement). The problem is to build a machine (control machine) that will impose that control.” [Jac01]

C1 and C2 are specification phenomena which are controlled by the machine and the controlled domain respectively. So, C1 represents phenomena through which the control machine affects the behaviour of the domain and C2 are phenomena

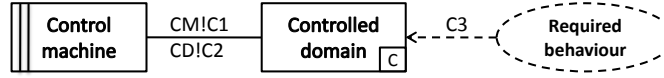


Figure 2.4: Required behaviour frame [Jac01].

which provide feedback. C3 are requirement phenomena (shared between the domain and requirements). Notice that the controlled domain is a causal domain, which is shown by a ‘C’ in a small box.

- **Commanded Behaviour:** The commanded behaviour frame, shown in Figure 2.5, captures the following class of problems:

“There is some part of the physical world whose behaviour is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator’s commands and impose the control accordingly.” [Jac01]

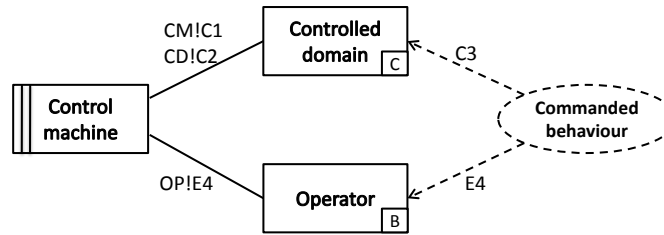


Figure 2.5: Commanded behaviour frame [Jac01].

In comparison to the required behaviour frame, an operator domain is added to this figure. The commands of the operator are represented as events E4. Requirements may refer to operator commands E4 in addition to C3. Note that the operator domain is a biddable domain since it is not predictable. This is shown using a ‘B’ inside the domain box.

### 2.3.3 MCC Guidelines for Modelling Control Systems

The MCC (Monitored, Controlled and Commanded) guidelines [But09a, But09d] can be used for formal modelling of control systems in the Event-B formal language. The MCC guidelines are the basis of the MCMC approach which is developed and discussed in this thesis.

The MCC guidelines describe two types of control systems, firstly autonomous controllers which consist of plants and controllers. The second type is commanded controllers which consist of plants, controllers and operators who can send commands to the controller (shown in Figure 2.6 <sup>1</sup>).

<sup>1</sup>The diagram uses Jackson’s Problem Frame notation [Jac01].

The modelling steps suggested in the MCC guidelines are based on the four-variable model [PM95]. Variables shared between a plant and a controller, labelled as ‘A’ in Figure 2.6, are environment variables and are categorised into *monitored variables* whose values are determined by the plant and *controlled variables* whose values are set by the controller. There are also *environment events* and *control events* which modify monitored and controlled variables respectively.

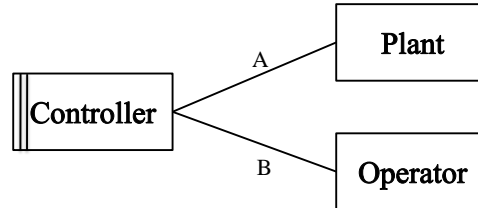


Figure 2.6: A commanded control system.

The other two variable categories of the four-variable model which are *input* variables and *output* variables, are not used in abstract formal model; instead MCMC guidelines provide patterns for introducing them as refinements.

In addition to the variables of the four-variable model, the identification of the phenomena shared between a controller and its operator is suggested for a commanded control system. Such phenomena are labelled as ‘B’ in Figure 2.6. These phenomena are represented by *command events* which are the commands from an operator and *commanded variables* whose values are determined by command events and can affect the way a control system behaves.

MCC guidelines also define a refinement step for monitored, controlled and commanded phenomena. We refer to these refinement steps as vertical refinement, since the model is refined towards its implementation. This is discussed further in Section 3.5. The steps of vertical refinements are as following:

- **Refinement step for monitored variables by adding sensors:** This step defines how a controller receives values of monitored variables and responds to them. This corresponds to introducing *input* variables in the four-variable model.
- **Refinement step for controlled variables by adding actuators:** This is when a controller sets the value of a controlled variable. This corresponds to introducing *output* variables in the four-variable model.
- **Refinement step for commanded variables by adding buttons:** This defines when a controller receives a request from the operator through a button press.

### 2.3.4 KAOS

KAOS (Keep All Objectives Satisfied) [vL08, vL09] is a goal-oriented requirement engineering (RE) which uses the concept of goals in various aspect of RE. *Goals* are prescriptive statements - properties whose satisfaction depends on the system behaviour - which represent objectives of the system. Goals are satisfied by cooperation of their *agents* which are active system components. The focus in KAOS is to capture the problem world by defining models with a specific view of the system. The views and their corresponding models are:

- **Goal models** represent the *intentional* view of the system, as goals capture the objectives of the system. This is explained in more detail in Section 2.3.4.1.
- **Object models** show the *structural* view of the system. This model is an operation-free UML class diagrams. Object models help to identify objects which are used in other models. Thus, they provide a glossary of terms that other models can refer to. Objects can be entities, associations, agents and events.
- **Agent models** represent the *responsibility* view of the system which shows the link between agents and goals. This is discussed further in Section 2.3.4.2.
- **Operation models** represent the *functional* view of the system. This view captures the services that the system should provide and defines operations for goals.

In the remainder of this section the goal model and agent model are discussed in more detail.

#### 2.3.4.1 Goal Model

A goal model represents the contribution of functional and non-functional goals of a system to one another's satisfaction. Here a goal can be refined to finer-grained goals using AND/OR refinement links. A more abstract goal will be satisfied by its refined goals. This refinement helps to structure "complex specifications at different levels of concern".

Figure 2.7 shows a goal model of a cruise control system (CCS) [PD09]. We focus on two parts of the diagram, goals in this section and agents in the next section. A goal is shown by a parallelogram. The abstract goal of maintaining the car speed safely, is decomposed into two sub-goals. For instance, the right sub-goal which deals with safety of the CCS, states that the braking is always allowed.

When a goal is finer-grained the number of cooperating agents to satisfy the goal are less. A *requirement* is defined as a goal satisfied by one agent of the software. If a

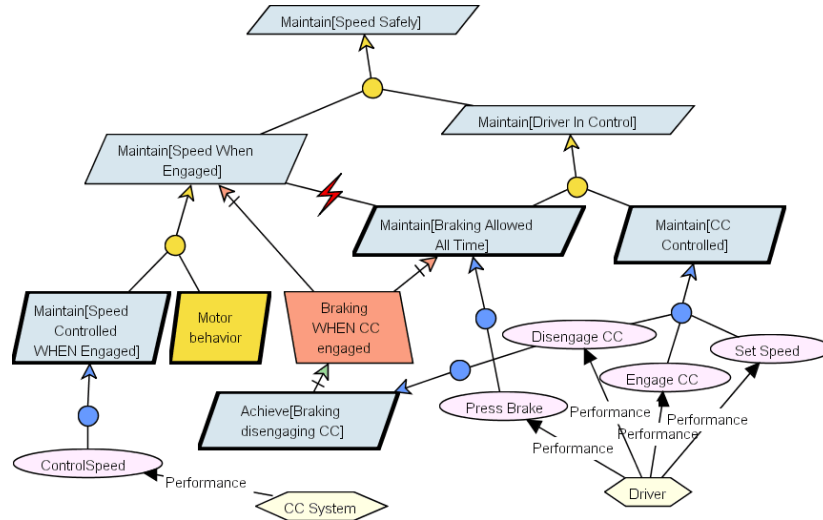


Figure 2.7: A goal model for a cruise control system [PD09].

goal is under the responsibility of an environment agent then it is an *expectation*. A requirement or an expectation appears in a leaf node of a goal model. A goal model is generally linked to other models (views) of the system. For instance, *responsibility links* which are interfaces between goal and agent models can be defined.

#### 2.3.4.2 Agent Model

An agent model links agents to the goals that they are responsible to fulfil. An agents can be a human (e.g. an operator), a device (e.g. a sensor), an existing or a new software component (e.g. an off-the-shelf component or the software-to-be).

The agent model itself can be shown in different kinds of diagrams. Two of these are *agent diagrams* and *context diagrams*. An agent diagram represents the system agents, their capabilities for goal realisation, their responsibilities and the operations they can perform. A context diagram in KAOS shows the relation between the agents. This diagram is explained in Chapter 9 and it is used in order to compare the MCMC approach and KAOS.

In KAOS, a hexagon denotes an agent. Figure 2.7 shows that the sub-goal of ‘braking is always allowed’ is under the responsibility of the driver (shown as an agent). Also, a driver can perform the operation ‘press brake’ which is shown as an oval in order to achieve this sub-goal.

#### 2.3.5 Tabular Methods based on Four-Variable Model

Systematic Design Verification (SDV) procedure was applied to the Darlington Nuclear Generation Station Shutdown System as part of the software development process.



Relevant to this thesis is the verification of functional properties based on the SDV procedure [LFM04, WL03].

In SDV, tabular notations are used to produce requirements and design documents which are called Software Requirement Specification (SRS) and Software Design Description (SDD) respectively. The SDD adds design details such as scheduling, error handling and implementation dependencies to the SRS functionality. “The objective of SDV is to verify that the behaviour of every output defined in the SDD is in compliance with the requirements for the behaviour of that output as specified in the SRS” [LFM04].

In the tabular notation, the first columns of SRS and SDD tables represent the conditions which are predicates that can be true or false. The second columns capture the results corresponding to every condition. The composition of tables provides the overall system specification [WL03].

Table 2.1 provides an example of an SRS table for a pressure sensor trip that is tripped when the pressure sensor value is above a threshold. Note that *Pressure* is a monitored phenomena, while *K\_PressSP* and *k\_DeadBand* are constants defined in the requirements. The reader is referred to [LFM04] for more details.

$Pressure \leq K\_PressSP - k\_DeadBand$	NotTripped
$K\_PressSP - k\_DeadBand < Pressure \wedge Pressure < K\_PressSP$	PreviousVal
$Pressure \geq K\_PressSP$	Tripped

Table 2.1: The SRS table of a pressure trip [LFM04].

To ensure the completeness (a condition is applicable to every input) and disjointness (conditions do not overlap) of the conditions within a table, proving the following two invariants are suggested [LFM04].

**Disjointness:** For every two distinct conditions,  $condition_i \wedge condition_j \Leftrightarrow FALSE$

**Completeness:** For all conditions,  $condition_i \vee \dots \vee condition_j \Leftrightarrow TRUE$

To isolate the verification of hardware interfaces, Figure 2.8 shows that the internal representations of monitored (M) and controlled (C) variables are respectively defined as pseudo-monitored ( $M_p$ ) and pseudo-controlled variables ( $C_p$ ). Abstract functions ( $Abst_M$  and  $Abst_C$ ) are used to map the real value SRS input to the discrete SDD input. An example is the monitored value temperature which belongs to  $M$  and might be 500.3 Kelvin. This value can be read by a temperature sensor and A/D converters to produce a value such as 3.4 volts in  $I$ . This value can be processed by a hardware hiding module to the internal value of  $M_p$  which might be 500 Kelvin.

Table 2.1(a) and 2.1(b) [LFM04] provide examples of an SDD table with pseudo variables and an abstract function for the pressure sensor trip respectively. In Table 2.1(a), PRES

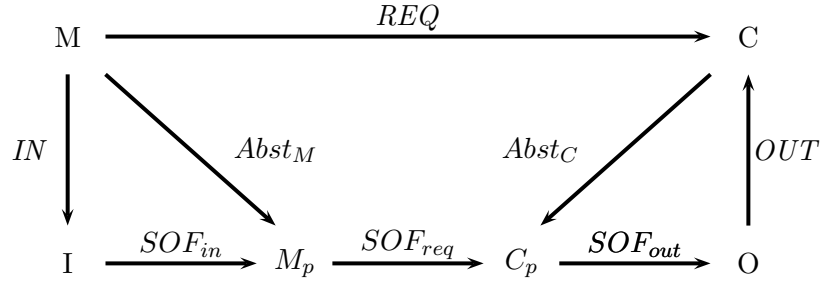


Figure 2.8: Decomposition of proof obligations [LFM04].

is the  $M_p$  whose value is a result of Table 2.1(b). KPSP and KDB are constants of the system. In Table 2.1(b)  $x$  represents the value of the sensed pressure, i.e. the value of  $I$ .

(a) SDD		(b) Abstract function	
$PRES \leq KPSP - KDB$	FALSE	$x \leq 0$	0
$KPSP - KDB < PRES \wedge PRES < KPSP$	PREV	$0 < x \wedge PRES < 5000$	floor(x)
$PRES \geq KPSP$	TRUE	$x \geq 5000$	5000

Table 2.2: A pressure trip SDD and abstract function [LFM04].

### 2.3.6 SCR

SCR (Software Cost Reduction) [Hei07, HJL96] is a formal method for specification of control systems with a tabular notation. SCR is based on the four-variable model, although in addition to the four variables, the SCR uses *mode*, *term*, *condition* and *event* [Hen80].

A *mode* (or classes of system states [Hen80]) is an auxiliary variable representing the state of a monitored variable. A *term* is a “text macro” which can be used to make the specification concise. A term can be a quantity which is not directly obtained from monitored variables. Also it can be an abbreviation of conditions which are used frequently in the specification. Using a term means that the precise definition of a quantity or a condition can be postponed [Hen80].

*Conditions* are predicates on the system state. Finally, an *event* represents changes in system variables and mode. Therefore, an event occurs when the value of a variable changes. A *conditioned event* in SCR is represented as “@T(c) WHEN d” where  $c$  and  $d$  are conditions. This event shows that condition  $c$  changes from FALSE to TRUE, while condition  $d$  holds. Therefore, this conditioned event has the semantic  $\neg c \wedge c' \wedge d$  where  $c$  shows the value of the current state and  $c'$  shows the value of the next state. It is also possible to define a basic event such as “@T(c)” where there is no enabling condition [Hei07, HJL96].

In SCR, the system behaviour is represented using three types of table. These tables define the constraints and relations between the values of mode, terms and controlled variables. The tables are:

- **event tables**, which define controlled variables and terms as functions of modes and events;
- **condition tables**, which define controlled variables and terms as functions of modes and conditions;
- **mode transition tables**, within which a mode is defined as a function of an event and a mode.

The SCR requirement method is supported by a toolset [HKLB98]. Some of the tools that the SCR Toolset consist of are as follow:

- Specification Editor for creating and modifying requirement specifications;
- Simulator which can be run by a user to validate the specification;
- Dependency Graph Browser to show the dependencies of variables;
- Model Checker, the Spin model checker [Hol04] is integrated into SCR Toolset.

### 2.3.7 WRSPM and a Requirement Tracing Approach

WRSPM [GGJZ00a, GGJZ00b] is a reference model for requirements and specifications of systems. WRSPM distinguishes between artifacts and phenomena which are shown in Figure 2.9. Artifacts are broadly classified into groups related to the *system* versus those related to the *environment*. The *interface* of these two groups is the specification, Figure 2.9. The artifacts are as follow:

- domain knowledge or world (W): behaviour of the world;
- requirements (R): how the customer needs the world to behave;
- specification (S): information based on which a system that satisfies requirement can be built;
- program (P): implementation of the specification;
- program platform or machine (M): the basis provided for programming the system.

The environment and the system have phenomena that define their states and events. Phenomena that belong to and are controlled by the *environment* are represented by  $e$  in Figure 2.9. Similarly,  $s$  denotes phenomena which belong to the *system*. In Figure 2.9, phenomena are shaded into two groups to show that they are controlled by either the environment or the system.

Phenomena that are controlled by the system are also visible to it. However not all the environment phenomena are visible to the system. The same is true for the environment. Therefore, phenomena  $e$  and  $s$  are divided into hidden ( $e_h$  and  $s_h$ ) and visible ( $e_v$  and  $s_v$ ). The visibility of phenomena for the artifacts is shown using a visibility oval in Figure 2.9. For instance, phenomena  $e_h$ ,  $e_v$  and  $s_v$  are visible to the environment and can be used in W and R.

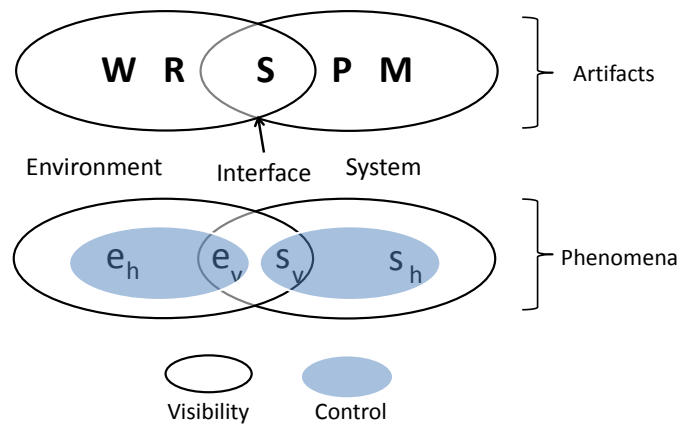


Figure 2.9: WRSPM reference model.

[JHLR10] introduces an approach for tracing requirements to an Event-B formal model. This approach is based on WRSPM. The method of [JHLR10] for traceability involves taking a requirement and identifying phenomena and artifacts of the environment and the system for that requirement. After this, the identified phenomena and artifacts are modelled in Event-B and traceability information between requirements and their Event-B representations are provided.

### 2.3.8 HJJ

The HJJ [HJJ03, CJ05, JHJ07] is an approach to formal specification of “open systems”. These are systems which use sensors and actuators to interact with their physical environment. The specification of a system in the HJJ is initially based on the *system* view rather than the *software* view, Figure 2.10. A system specification can be derived from the physical phenomena which are measurable and starting with system specification is usually easier and more meaningful than jumping into software specification. The steps suggested for obtaining a software specification from its system specification in HJJ are:

1. Specifying the requirements and the environment of the *system*.
2. Capturing *rely conditions*.
3. Deriving the specification of the control system, i.e. the *software* requirements.

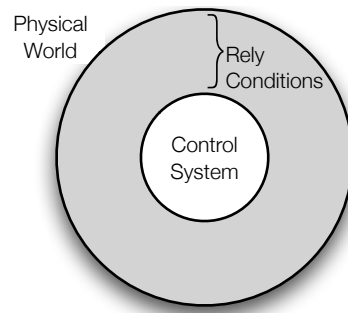


Figure 2.10: An overview of the HJJ approach [HJJ03, JHJ07].

Rely conditions represent the assumptions of non-software components. In other words, they are assumptions made on the components which link the environment to the controller, such as actuators and sensors. For instance, a rely condition can be the accuracy of a sensor. There are also guarantee conditions which are obligations on behaviours of the software-to-be. If a program is used in an environment in which rely conditions are not true, the program has no obligations. So requirements can be fulfilled provided that their corresponding rely conditions hold. This means the satisfaction of requirements relies on the “causal properties of the problem world”, which are captured as rely conditions [JHJ07]. The guarantee conditions and the software specification are derived from the overall specification.

A simplified example of a sluice gate specification is shown in Figure 2.11. The requirement of a sluice gate is that “over the whole time of system operation, the time when the gate is fully closed should be in a certain ratio to the time when it is full open”. The controller is specified as a system with inputs of *top* and *bottom* sensors and the output of *motor* and *direction* (depending on the inputs the controller switches the motor on or off and sets the direction to up or down). *SensorProperty* and *MotorOperation* are formalised to represent the rely conditions under which the controller will behave in such a way that guarantees the *requirement*. Notice that the keyword *external* shows the environment quantity which is the *position* of the gate (closed, open, neither) in this example.

## 2.4 Comparison of Some RE Approaches

The presented requirement engineering and specification approaches have one main characteristic which is at initial stages of a system development, the focus should be on

$Controller \triangleq$ <b>system</b> <b>external</b> <i>position</i> <b>input</b> <i>top, bot</i> <b>output</b> <i>motor, dir</i> <b>rely</b> $SensorProperty \wedge MotorOperation$ <b>guar</b> <i>SluiceGateRequirement</i>
---

Figure 2.11: Sluice gate specification in HJJ [JHJ07].

*system* requirements and not software requirements. So, the main message of these approaches is that a system-level view is simpler and more meaningful for stakeholders and developers than starting the process of requirement analysis and specification at a software level. Besides, deriving software requirements/specifications from system requirements/specification can help to explicitly identify the assumptions on the problem world.

Despite this similarity, these approaches are very different from one another. PF and KAOS cover a broad range of requirement engineering activities, such as requirement gathering, elicitation as well as specification. However, the other approaches are more domain specific (mostly suitable for control systems) and their focus is greatly on formal specification, rather than requirement elicitation. Some comparison between these approaches is discussed in the remainder of this section.

Figure 2.12 illustrates the four-variable model as a problem diagram. As shown, the *environment* and the *system* are connected via *sensors* and *actuators* domains. REQ represents the requirements of the system in terms of monitored and controlled phenomena, which are denoted as *m* and *c* respectively in Figure 2.12. As shown *m* is controlled by the environment and *c* by the actuator. Phenomena *input* and *output* of the four-variable model are represented as *i* and *o* which are controlled by the sensors and the system respectively.

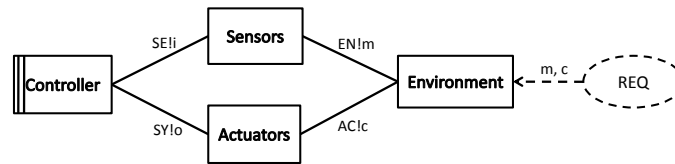


Figure 2.12: Four-variable model in a problem diagram [Jac01].

This comparison shows that requirements refer to *m* and *c* which are also phenomena shared between the environment and, the sensors and actuators. So, if we consider the sensors and actuators as parts of the machine, the four-variable model suggests that the requirement phenomena and the specification phenomena are the same. Also, the

four-variable model assumes that  $m$  and  $c$  phenomena are disjoint and thus the sensors and actuators are “mutually independent in their operations” [Jac01].

Another comparison is between the WRSPM and the four-variable model [GGJZ00a]. Firstly, that the four-variable model does not discuss  $e_h$  phenomena (environment phenomena hidden from the system). The monitored and controlled phenomena in the four-variable model correspond to the phenomena  $e_v$  (environment phenomena visible to the system) and  $s_v$  (system phenomena visible to the environment) in WRSPM respectively. Also,  $s_h$  (system phenomena hidden from the environment) represents both input and output in the four-variable model.

The SDV procedure is also based on the four-variable model, although SDV procedure has extended controlled and monitored variables of the four-variable model by pseudo variables as well as abstract functions [LFM04]. This helps to distinguish the hardware interface.

Using the four-variable model, the proof obligation for verifying the functional equivalence of the SRS and SDD is  $REQ = OUT \circ SOF \circ IN$ . Note that ‘ $\circ$ ’ denotes *functional composition*. For example the functional composition of  $g \circ f(v_1)$  for the functions  $f : V_1 \rightarrow V_2$  and  $g : V_2 \rightarrow V_3$  is  $g(f(v_1))$ .

Introduction of pseudo variables in Figure 2.8 helps with the decomposition of  $REQ = OUT \circ SOF \circ IN$  into following equations where the first one represents “a comparison of the functionality of the system” and the last two show “comparisons of the hardware hiding of the system”.

$$\begin{aligned} Abst_C \circ REQ &= SOF_{req} \circ Abst_M \\ Abst_M &= SOF_{in} \circ IN \\ id_C &= OUT \circ SOF_{out} \circ Abst_C \end{aligned}$$

## 2.5 Conclusion

This chapter explained the terms requirements and requirement engineering. We also discussed some requirement engineering methods that are used to specify a system formally or semi-formally. These methods can be used to specify requirements of a control system in a systematic way. Most methods discussed in this chapter use the four-variable model as the basis of their approach. In particular, the MCC guidelines 2.3.3, the tabular method of SDV 2.3.5 and SCR 2.3.6 have extended the four-variable model in order to provide detailed steps for formalising a control system.

In addition, other approaches such as WRSPM 2.3.7 and HJJ 2.3.8 require understanding the requirements at the system level before focusing on the software requirements. This

is similar to the principle of the four-variable model, which is to identify SOF (SOF is the relation between the system quantities IN and OUT) from REQ and NAT (these are the relations between the environment quantities MON and CON).

In proposing the MCMC approach, Chapter 5, we take a similar approach to the requirement engineering methods of this chapter. The MCMC approach is defined based on the four-variable model, since this model and its extended methods, such as SCR and the MCC guidelines, have been applied to several control systems. Thus, there is a shown record of successful application of the four-variable model. Examples are the Darlington Nuclear Generation Station Shutdown System [LFM04, WL03] and the Safety Injection System [Hei07].

In addition, the MCMC approach is defined based on the MCC guidelines. Table 2.3 represents a comparison of the requirement engineering approaches which have been developed based on the four-variable model. As shown, the role of an operator is captured explicitly in the MCC guidelines. We believe that the special role of an operator should be identified separate to environment and system quantities.

Method	Formal	Environment Quantities (Mon & Con)	System Quantities (In & Out)	Operator Role
MCC Guidelines	✓	✓	✓	✓
SDV	✓	✓	✓	—
SCR	✓	✓	✓	—

Table 2.3: Comparison of the four-variable model based requirement engineering approaches.

As will be discussed the MCMC approach focuses on the system level requirements. Chapter 5 presents guidelines and patterns based on which requirements of a system can be specified formally. In order to obtain the software specification of a system, in Chapter 8 we propose patterns and guidelines to refine and decompose the system specification. Therefore, software specifications can be extracted from their system specifications.

Chapter 9 provides a detailed comparison between the contributions of this thesis, in particular the MCMC approach, and the requirement engineering methods that were explained in this chapter.





## Chapter 3

# Formal Methods

The definition, limitations and advantages of formal methods are discussed in Section 3.1. Also in this section, some standards that recommend using formal methods and examples of real systems where formal methods were successfully applied are given. Section 3.2 explains classifications of formal methods. In Section 3.3, overviews of some existing formal methods are given. We then briefly discuss the terms model checking and refinement in Sections 3.4 and 3.5. The conclusion of this chapter can be found in Section 3.6.

### 3.1 Formal Methods: Definition, Standards, Advantages and Limitations

In computer science, formal methods are mathematically based techniques used for describing the properties of a system. They provide a systematic approach for the specification, design and verification of software and hardware systems. Also, formal methods can be used to reason about properties of a model [Win90].

The mathematical basis for a formal method is usually provided by a formal specification language. This is a language with syntactic (its notation) and semantic (formal models of system behaviours) domains and with precise rules defining the satisfaction of specification for the models. Having this mathematical basis is the main reason for being able to prove that a specification is satisfied [Hal90, Win90].

#### 3.1.1 Advantages of Applying Formal Methods

The cost of correcting an error of a software application is usually higher in later stages of a development lifecycle. For instance, the cost of correcting an error in development stage can be up to 100 times of the cost of correcting it in the requirement stage of lifecycle [BP88]. Also, ‘faulty requirements definition’ is one of the causes of errors in

software applications. Thus, the quality of defined requirements plays an essential role in reducing costs and errors in software development.

Formal methods can be applied in all phases of a system development in order to reveal ambiguity, incompleteness, and inconsistency [Win90]. Using formal methods in requirement elicitation and specification phase helps with identifying missing and ambiguous requirements. Therefore, the changes that otherwise would happen in implementation are identified at an earlier stage. In addition, these changes are documented, whereas usually an implemented system is modified without keeping track of the changes. These factors can contribute to decrease the cost of maintenance [Hal90].

Using formal modelling in the design stage helps to establish model's properties by formal reasoning through proof obligations (POs) [AH07]. This means design flaws can be discovered in the model during design rather than testing. Thus instead of making major changes in the implementation of a system, the model can be manipulated during the design [CW96, Win90]. In addition, using proof enables us to show the absence of bugs whereas testing can only demonstrate the presence of bugs [Hal90]. Some examples of successful applications of formal methods in developing a system are:

1. Z, which is discussed in Section 3.3.1, was used to formalise part of the IBM's Customer Information Control System (CICS). This formalisation improved the overall quality of the product [CW96].
2. [BBFM99] describes the successful process of developing a railway-automated system that used B. The B formal method is discussed in Section 3.3.3.
3. In a product for the UK Civil Aviation Authority, VDM (Section 3.3.2) and CCS [Mil89] were used. The result showed the quality of the product was much higher than "comparable projects carried out using informal methods" [CW96].

### 3.1.2 Standards and Certifications

Formal methods are sometimes not just desirable, but required by standard bodies. The UK Ministry of Defence Standard 00-55, known as DEF STAN 00-55, "mandates the extensive use of formal methods", including formal specification of safety-critical components and analysing completeness and consistency of these components [BH95]. The DEF STAN 00-55 states that [Min97]:

"Where safety is dependent on the safety related software fully meeting its requirements, demonstrating safety is equivalent to demonstrating correctness with respect to the Software Requirement".

These guidelines define formal arguments as one type of safety argument. Formal arguments are defined as “the object code satisfies the formal specification and that the formal specification complies with the safety properties in the software requirement”.

IEC 61508 is a generic safety standard based on which some domain specific standards are defined. The IEC 61508 standards defines SIL (Safety Integrity Level) levels from 1 to 4, where SIL4 is the most dependable system. In this standard use of formal methods is “highly recommended” for SIL4.

One of the domain specific standards related to the case studies of this research is ISO 26262. This standard is defined based on the IEC 61508 for safety critical systems in automotive sector. In this standard formal methods are “recommended”, while semi-formal methods are “highly recommended” [Dep12].

### 3.1.3 Limitations of Formal Methods

Some of the limitations of modelling and proving properties of a system are as follows [Hal90]:

- Usually not all aspects of a system can be modelled.  
Formal modelling of non-functional requirements such as performance is not yet fully understood.
- Mapping real world to a formal description is limited.  
Some of the requirements cannot be modelled as our resources such as the formal language are limited.
- Not every property of a system can be proved.  
Sometimes a property cannot be presented in a formal language or its proof might be difficult. Also, a property might be false which makes it impossible to be proven.
- It is possible to make mistakes during modelling and proving.  
However, using automated theorem provers and modelling tools lessen the possibility of making such mistakes.

## 3.2 Classification of Formal Methods

In this section, two classifications of formal methods are discussed. Section 3.2.1 discusses the first classification, which is based on [Win90]. Here formal methods are broadly categorised into **model-oriented**, **property-oriented** and **visual languages**.

In Section 3.2.2, the second classification is discussed. This is based on [LCY<sup>+</sup>97, LYZ97], which categorises formal methods into **model-based**, **logic-based**, **algebraic**, **process algebra** and **net-based** approaches.

### 3.2.1 First Classification of Formal Methods

Traditionally, formal methods were categorised into three generic groups [Win90]. The first is **model-oriented** where a model of the system's behaviour is built. Some methods of this group are used for specifying the behaviour of a *sequential program*, examples are VDM [Jon90] and Z [Spi92]. Others such as Communicating Sequential Processes (CSP) [Hoa85] and Calculus of Communicating Systems (CCS) are used mainly for *concurrent and distributed systems*.

The second is **property-oriented** in which a model represents a set of properties of the system's behaviour. This group is divided into *axiomatic* and *algebraic* methods. The former uses first order logic pre/postconditions to specify the state-dependent behaviour of a system. An example of this method is Larch [GH93] which is used for sequential programs.

In the algebraic method, algebra is used to define data types and process. Also, properties of the system are defined as axioms. Examples of this method is ActOne and Lamport's transition. The last can be used to specify concurrent and distributed systems' behaviour [Win90].

The third category is **visual languages** which includes any methods with graphical elements in their languages. Examples of this category are Petri Net [Rei85] and Statecharts.

Formal methods that help to specify sequential systems' behaviour use rich mathematical structures such as sets and relations to describe states. Also, state transitions are defined as pre/postconditions. However, in methods that focus on specifying concurrent systems' behaviours, states are defined as simple domains such as integers or are uninterpreted. Behaviours are defined as sequences, trees, or partial orders.

In some formal methods, two different methods are combined to manage rich state spaces and complexity of concurrent systems [CW96, Win90]. Example of such methods are Raise, a combination of VDM and CSP, LOTOS, a combination of ActOne and CCS, Circus [WC02], a combination of Z and CSP, and combined CSP and B [BL05].

### 3.2.2 Second Classification of Formal Methods

In another classification [LCY<sup>+</sup>97, LYZ97], formal methods are divided as follow:

- Model-based approaches: Systems are modelled by defining their states and operations which transform the system. Z, VDM, B-Method and Event-B are some of the examples of this category.
- Logic-based approaches: Desired properties of a system are modelled using logics. These properties can also be validated using the logic's axiom system. Some of the examples are Temporal Logic and Hoare Logic.
- Algebraic approaches: Behaviours of operations and transitions are modelled without defining states. Also, there is no explicit representation for concurrent systems. Such as Larch.
- Process algebra approach: Concurrent processes are modelled explicitly and behaviour of a system is modelled by observing the communication between processes. Examples of this approach are CSP and CCS.
- Net-based approaches: Formal methods with graphical representation, such as Petri Net.

### 3.2.3 Comparison of the Two Classifications

Comparison between the two classifications of Sections 3.2.1 and 3.2.2 shows that the category of model-based and algebraic approaches are respectively equivalent to model-oriented and property-oriented, but only for non-concurrent systems.

The category of process algebra is similar to model-oriented for concurrent systems. One of the differences between these two categories is what is observable from the modelling point of view. In a model-oriented approach the *states* of the system are observable, while in a process algebra approach the system *transitions/actions* are observable. Finally, net-based approach which was mentioned in the second classification, is equivalent to visual language of in Section 3.2.1.

## 3.3 Overview of Some Formal Methods

The formal method which is used in this project is the Event-B language. Since this formal language is a model-based approach, in this section we look at some of the other model-based formal methods, namely Z, VDM and B-Method. In addition, an overview of Action Systems is given, since Event-B has evolved from Action Systems. Hoare logic and temporal logic are also briefly discussed. The Event-B formal language is explained in detail in the next chapter.

### 3.3.1 Z

Z [Bow03, Spi92] is a model-oriented specification notation based on the set theory and the first order logic. Construction of a model in Z is by defining *schemas* which are boxed notations. Schemas are mainly used to define firstly the state space of the system and secondly the operations. One of the main advantages gained by using schema is that a Z specification can be built incrementally. Therefore, it helps to manage complexity of the specified system.

As shown in Figure 3.1, each schema is divided into two sections. The top section contains declarations of variables and their types. The bottom section of a *state space schema* represents invariants of the system. An invariant is a relation between variables that hold for all possible states of the system.

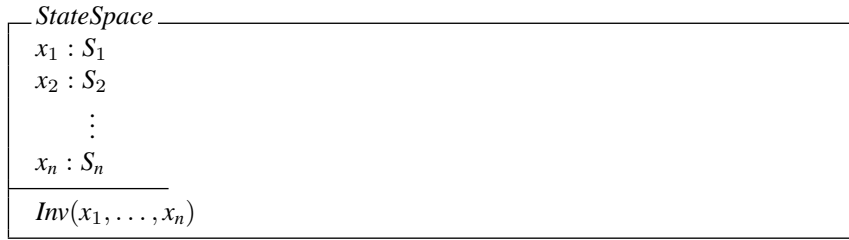


Figure 3.1: A state space schema in Z [Bow03].

In addition to invariants, the bottom section of an *operation schema* contains before-after predicates, Figure 3.2. Conventionally, the state of a variable such as  $x$  before an operation invocation is represented by  $x$ , whereas the state after the operation occurrence is shown as  $x'$ . Preconditions of an operation are derived from the before-after predicates. A precondition represents the state of the system that must hold for the invocation of an operation. If preconditions do not hold, the behaviour of the system is unspecified [Win90].

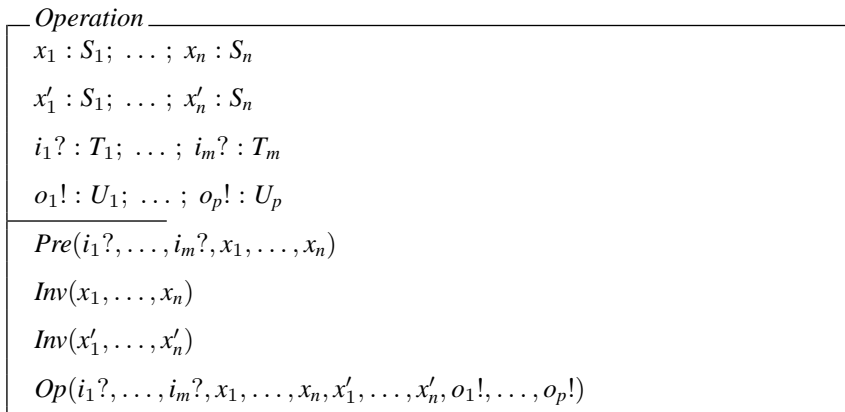


Figure 3.2: An operation schema in Z [Bow03].

### 3.3.2 VDM

Vienna Development Method (VDM) [Jon90, FLV07] originated in the 1960s and 70s in IBM's Vienna Laboratory. Its logic is based on three-valued logic within which predicates can be true, false or undefined.

The definitions of a VDM specification are collected in a *module*. A simple form of module is shown in Figure 3.3. As represented, a module contains a series of definition blocks, namely, *types*, *values* (similar to constants), *state* (similar to variables), *functions* and *operations*.

A function is an implicit specification for manipulating numbers. Functions define rules for obtaining results from zero or more arguments. Also, they must have a result to be meaningful. Operations specify the intended behaviour of a system. They represent *what* the program has to achieve. While functions return the same value for the same arguments, outputs of operators depend on their execution.

One of the differences between VDM and Z is that in VDM keywords are used to express precondition and postcondition clauses in functions or operations, while Z avoids keywords. As mentioned, preconditions represent the state that the system must hold to invoke an operation. Postconditions are predicates representing the state which the system holds on the return of an operation.

In addition, in VDM a variable in an operation is preceded by keywords *rd* or *wr* which respectively represent the operation has read-only or read-write access to the variable. This access modifier expresses whether or not the state of the system may be modified in the operation [Win90, Jon90].

### 3.3.3 Classical B

Classical B [Abr96], also known as B-Method, is a model-oriented formal method. This method is based on Abstract Machine Notation (AMN) which provides a framework for describing specification, refinement and implementation [Sch02]. AMN itself is based on the Generalised Substitution Language (GSL), which is an extended version of Dijkstra's guarded command notation [ALN<sup>+</sup>91].

In the B-Method, a system specification can be constructed by one or more abstract machines. Each abstract machine contains the following elements [Sch02]:

- Variables: to keep the local state of the system;
- Invariants: predicates which represent types of variables and their relations with other variables;



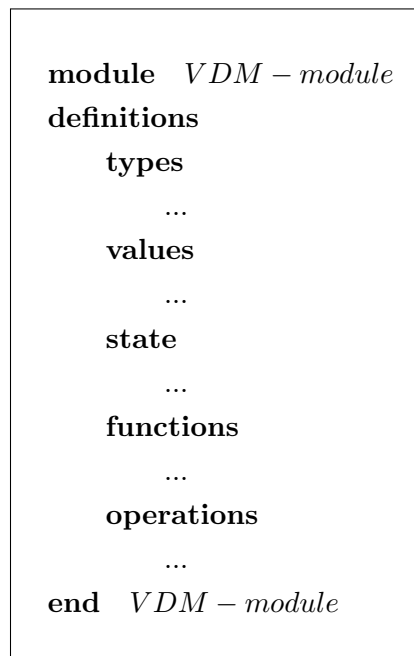


Figure 3.3: An overview of a module in VDM

- Initialisation: to specify the initial state of the system;
- Operations: represent what the system components do. Operations are also interfaces for variables which are encapsulated within the abstract machine.

### 3.3.4 Comparison of Z, VDM and B-Method

In VDM and B-Method it is possible to specify algorithmic information [BR95]. However, in Z before and after predicates are used to specify only the state change of the system [CM03]. This difference makes VDM and B-Method suitable for both specification and design of a system and therefore they are *development methods*, while Z is a *formal specification language* [LH95]. Furthermore, [BR95] explains that the B-Method has “greater programming feel”, since it is possible to express more algorithmic details in B-Method based models.

Another difference is that preconditions are not stated explicitly in Z, while pre/postconditions are explicit in VDM and B. Notice that postconditions in the B-Method show the state change of the system by using substitution, although it has been shown that substitution in B and pre/postconditions in VDM are equivalent.

Furthermore, three-valued logic in VDM allows to model undefinedness explicitly. However this is not the case in Z and the B-Method, and undefinedness is not explicit. In the formal language Event-B, which is used in this work, undefinedness is dealt with by using definedness proof obligations that ensure expressions are well-defined.

### 3.3.5 Action System

Action Systems [Bac90] are a state-based approach to describe distributed systems. An action system is a collection of labelled *actions* on some *state variables*. Also, an action system contains an *initialisation* statement which assigns initial values to the state space.

In an Action System actions determine what can happen in the system during an execution. An action such as  $A_i$  is shown below:

$$A_i = g_i \rightarrow S_i$$

Here,  $g_i$  represents the *guard* of the action and  $S_i$  is the *body* of the action. Guards are boolean conditions, while the body is a sequential statement on the state variables.

An action is said to be enabled if its guards are true. Occurrence of an action modifies the state of the system. Actions are atomic, meaning that an action which is chosen for execution will be executed to completion without any interference from other actions of the system. The execution terminates when there is no enabled action.

### 3.3.6 Hoare Logic

Hoare logic [Hoa69] is an axiomatic proof approach for computer arithmetic. The main idea of Hoare logic is to specify assertions that are general properties of variables and their relation before and after execution of a piece of code.

Validity of the results of a program mostly depends on the values of the variables before the code execution. Therefore, Hoare logic proposes adding general assertions as preconditions. The following notation is introduced to show the relation of a precondition, P, a program, Q, and the result of the execution, R. This notation is also known as Hoare triple.

$$P\{Q\}R$$

This notation can be interpreted as “if the assertion P is true before initiation of a program Q, then the assertion R will be true on its completion” [Hoa69]. Standard Hoare logic deals with proofs related to partial correctness (safety properties - something bad never happens) of a program. In other words termination of the program is not considered in Hoare logic. However, proving the correctness of a sequential program requires termination as well as correct results. Therefore, a better interpretation of the Hoare triple is as following: “provided that the program successfully terminates, the properties of its results are described by R”. Termination of a loop can be proved using variants which are natural numbers that will be decreased at every iteration.

### 3.3.7 Temporal Logic

It is sufficient to describe a sequential program by considering only its states before and after execution. However, in a concurrent program it is sometimes necessary to describe the behaviour of the system *during* the execution.

Properties of a system are broadly classified into *safety* and *liveness*. The former represents that something bad never happens, while the latter property shows that something good must eventually happens. Generally speaking safety properties are defined on states of system whereas liveness properties are defined for events.

In concurrent systems, simple safety properties can be proved using the implicit temporal concept of “at all times”. However, proving liveness properties usually requires the explicit consideration of the temporal concepts [Lam83]. Therefore, temporal logic is an approach for specifying and verifying concurrent and reactive systems formally by defining temporal formulas. A temporal formula is constructed using state formulas (well-formed first-order formulas) and logical and temporal operators [MP92].

As an example of temporal operators, the basic unary temporal operators are shown below [CGP99]:

- **X**: the property holds in the next state;
- **F**: the property holds at some future state;
- **G**: the property always holds.

Generally speaking, there are two views of time which results in two approaches to temporal logic [EH83, Lam80]. Firstly, *linear time logic* which allows to describe events along a time path in a single linear sequence. Here, the underlying time view is linear time where at each instant there is only one future occurring. Secondly, *branching time logic* in which all the possible futures are considered. This type of temporal logic usually results in a tree structure representation of the events along branches of time.

## 3.4 Model Checking

Model checking is an automatic verification technique mainly used in hardware controllers and communication protocols. This technique can be used for verifying concurrent systems with finite state [CGP99].

Model checking is based on building a finite model of a system to check whether desired properties of the system hold. This check is performed exhaustively on the *state space*

and it will terminate as the model is finite [CW96]. A state space consists of all reachable states and all transitions the system can make between these states [Val98].

To apply model checking, a formal model of the design should be provided. Also, the properties that the system is expected to satisfy are formalised using prepositional temporal logic. Then, an automatic search procedure can be used to check whether the model satisfies the specification. The result of this check is either a *true* answer, which indicates that the specification formula is satisfied by the model, or a *counterexample* execution which shows why the specification is not satisfied. So, counterexamples can be used to find errors [CGP99].

Advantages such as being completely automatic and producing a counterexample have made this verification technique attractive. However, its main disadvantage is the *state explosion*. This is problematic since the size of the state space of most systems (number of their states) is huge [Val98]. Also, the greater the size of the state space the more time consuming the process of model checking. Another problem with model checking is that the system should have a finite number of states. Thus, restrictions should be applied in order to model check some systems. For instance, a software application with large and unbounded data structures or recursion should be restricted [CGP99].

### 3.5 Refinement

Modelling the entire system in one step can be very difficult. This is particularly the case in complex systems, where there are greater number of states and transitions. Refinement is a top-down design strategy that enables elaboration of an abstract (high-level) statement. Using refinement allows us to deal with the complexity of a system in a stepwise manner, by working at different abstract levels [Abr10, Win90]. This means the internal behaviour of statements can be defined in detail in refinement steps [Pre10].

Modelling a system based on refinement usually results in the construction of an ordered sequence of models within which every model (concrete model) is a refinement of its preceding (abstract model) [AH07]. This means as the model becomes more concrete our view of the system is more accurate. Thus, there are usually more variables defined in the concrete model than there are in the abstraction. It is also necessary to ensure that the refined model preserves the correctness of its abstract model. Defining the relation between concrete and abstract variables can help to prove that the abstract behaviour is preserved. Relating a state of a concrete model to its abstract state can be defined through a *gluing invariant* which is a predicate such as  $J(v, w)$  where  $v$  and  $w$  are the state variables of the abstract and the concrete machine respectively.

## 3.6 Conclusion

In this chapter, formal methods were described as mathematically based techniques for describing the properties of a system. Advantages of using formal methods, such as improving system understanding and proving system properties formally, have meant that some certification and standard bodies require the use of formal methods in safety-critical system development.

A number of formal methods have been discussed and compared to one another in this chapter. Although formal methods can be used in different phases of a system development lifecycle, in the remainder chapters we use the Event-B formal method mainly in the requirement specification and design phases.

In the next chapter, the Event-B formal language [[Abr10](#)], which is used in this research, is explained in detail. Compared to other state-based formal methods, such as B and VDM, Event-B has a simple and extendible notations. In addition to refining existing events, Event-B allows the introduction of new events to a model in refinement levels. As will be discussed in Chapter 5, we take advantage of the Event-B refinement techniques to break the complexity of modelling requirements by introducing them in a step-wise manner.

In addition, Event-B provides (de)composition techniques which we use to formalise system requirements as sub-models. This is shown in Chapters 5 and 6. Also, Event-B is supported by an open source tool, called Rodin, and a number of plugins which facilitate the process of modelling. A detailed description of the Event-B language and its tool is provided in the next chapter.

## Chapter 4

# Event-B: Structure, Refinement and Decomposition

In this chapter, the Event-B formal language is explained. The structure of Event-B consists of *context* and *machine*. In Section 4.1.1 and 4.1.2, these two structures are discussed. A brief description of refinement in Event-B is presented in Section 4.2. After this, some proof obligation rules for an Event-B model are presented in Section 4.3.

In Section 4.4, two styles of decomposition for an Event-B model, namely *shared-variable* and *shared-event*, are discussed. Since the shared-event composition is used in the later chapters, Section 4.5 explains this style of composition in detail. In Section 4.6 and 4.7, the tool for Event-B, called Rodin, and some of its plugins are explained. Finally, Section 4.8 provides a brief conclusion.

### 4.1 Event-B

Event-B [Eve11, Abr10] is extended from the B-Method [Abr96] and has evolved from action systems [Bac90]. In particular, the notation of guarded actions in Event-B is the same as action systems. In addition, some notations of refinement, such as refining skip to introduce new events, are based on action systems.

Event-B is a refinement-based language and has simple concepts within which a complex and discrete system can be modelled. An Event-B model can consist of two kinds of constructs or components, namely *context* and *machine* [AH07, Abr10, Hal06]. In the remainder of this section these two constructs are discussed.

### 4.1.1 Context

Contexts [Abr10] demonstrate the static part of a model and provide axiomatic properties. A context can contain the following element:

- **Carrier sets:** These are similar to types and are implicitly assumed to be non-empty;
- **Constants:** Constants of the context;
- **Axioms:** These describe the properties of carrier sets and constants;
- **Theorems:** Properties that can be proved from other established statements, such as axioms and other theorems in a context.

A context such as C0 in a model can be referenced by any machines of the same model by using the syntax *sees C0* in the machine. A machine that sees C0 can use the elements defined in this context. Also, a context (C0) can be refined by other contexts such as C1, using the *extends* clause. In this case, if a machine sees C1, it will have access to the elements of both C1 and C0. These are shown in Figure 4.1.

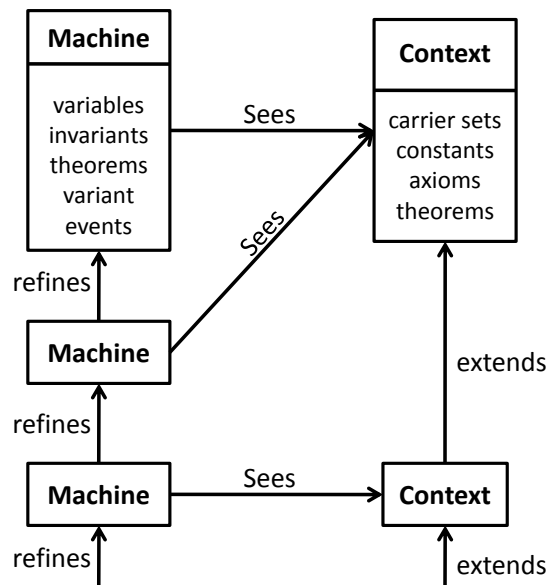


Figure 4.1: Event-B machines and contexts and their relation.

### 4.1.2 Machine

Machines [Abr10] represent the dynamic part of a system. Figure 4.1 represents this construct. An Event-B machine can contain the following clauses:

- **Refines:** This is used to introduce more details to a machine. In this case a more concrete machine, e.g.  $M1$ , refers to its abstract machine, e.g.  $M0$ , using the *refines* clause ( $M1 \text{ refines } M0$ ).
- **Variables:** These represent the system states. Variables are defined based on mathematical objects such as sets, functions and relations.
- **Invariants:** These are defined to constrain variables. They also represent properties of a system. An invariant must be always true. For instance, if  $I(v)$  shows an invariant for variable  $v$ , it must be proved that  $I$  holds despite any change to the value of  $v$ .
- **Theorems:** These are properties which can be proved from other established statements, such as invariants and other theorems of a machine and axioms and theorems of a context which is seen by the machine.
- **Variants:** These are defined to prove that a new event in a refined machine does not *diverge*. This means the new event will not be infinitely enabled. An example of a variant is a natural number which is decreased every time the new event occurs and when it reaches 0 the new event will be disabled.
- **Events:** They are explained further in the next section.

### 4.1.3 Event

An event in the Event-B language is an atomic transition that modifies the states of the system. As shown in Figure 4.2, an event consists of various elements.

These elements are:

- **Name:** The name of an event which should be unique within each machine.
- **Refines:** An event in a refined machine can be a refinement of some abstract events in its preceding abstract machine. For instance, an event such as  $e1$  may refine an abstract event  $e0$  by stating ' $e1 \text{ refines } e0$ '.
- **Parameters:** Parameters of an event are listed in front of the “any” clause.
- **Guards:** These are represented in the “where” clause of an event. Guards are predicates which describe the conditions that need to hold for the occurrence of an event. If there is more than one guard defined for an event the conjunction of guards should hold for the event to be enabled. Also execution of events will stop, if guards of no event holds. This situation causes a deadlock in the model of a system.



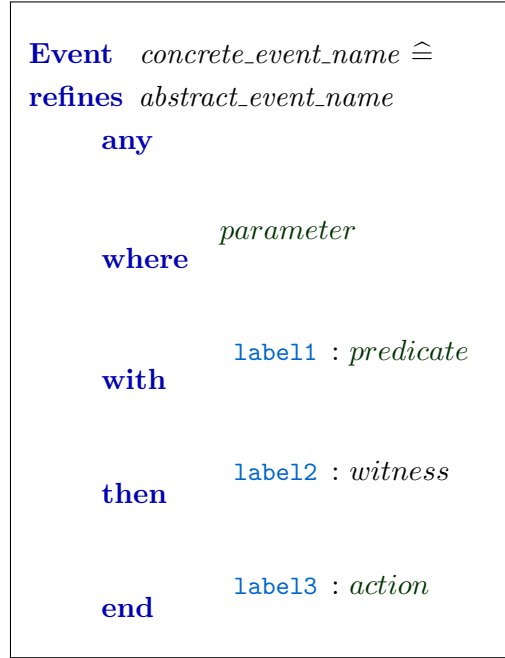


Figure 4.2: Structure of an event in Event-B language.

Since two events cannot happen simultaneously, if guards of more than one event hold at the same time, at most one of these events can be executed. The choice between the events will be made nondeterministically.

- **Witness:** If an event refines a parameterised abstract event, the refined event should provide a value, called witness, for any abstract parameter that does not appear in the concrete event.
- **Actions:** These are shown in the “then” clause and determine how state variables change when the event occurs. Actions of an event will be performed simultaneously to preserve the atomic nature of the event.

The state of a variable can change through deterministic or nondeterministic assignments as shown in the table below. Here  $E$  denotes an expression on parameters  $par$  and variables  $var$ , also  $Q$  represents a predicate. Notice that  $x'$  denotes the state of the variable just after its occurrence. In other words,  $x'$  is the after-value, while  $x$  is the before-value.

## 4.2 Refinement in Event-B

Refinement of a model in Event-B can be categorised into two groups. Firstly, **horizontal refinement** [Abr10] (also known as feature augmentation refinement [DB09]) where in each step new features of the system are introduced. Here a system engineer

Type	Assignment	Description
Deterministic	$x := E(par, var)$	x will be modified based on E
Nondeterministic	$x :\in E(par, var)$	x is assigned to an element of a set
Nondeterministic	$x :  Q(par, var, x')$	x is assigned to a value which satisfies Q

Table 4.1: Deterministic and nondeterministic assignments in Event-B.

explores and models the requirement document and features of the system gradually in refinement steps.

This style of refinement is possible because Event-B allows introducing new events in a concrete model. A newly added event refines a dummy even called *skip* which does nothing in the abstract level. The horizontal refinement is completed when the modelable parts of the system are formalised. This refinement helps designers to acquire better understanding by achieving a more precise model in every refinement step and dealing with requirements in a step-wise manner.

The second form is **vertical refinement** [Abr10] (also known as structural [DB09] or data refinement). This form of refinement is used to introduce details of system design to the formal model of the system. So, the model will become a more accurate representation of the implementation. In other words, vertical refinement closes the gap between modelling and programming languages by adding design details. In vertical refinement gluing invariants are defined to link the concrete and abstract states. However, invariants defined in horizontal refinements mostly define properties and behaviour of the system.

An analogy is that in horizontal refinement the chain of refinement is in the same level of abstraction from the implementation point of view, whereas in vertical refinement every machine in the refinement chain is in a different implementation level.

Notice that the distinction between horizontal and vertical refinements is methodological and it can help with the process of modelling. However, there is no distinction in terms of proof obligations of these two styles of refinement.

### 4.3 Proof Obligation Rules in Event-B

Proof Obligations (POs) define what should be proved for a model of a system. In this section some of the proof obligation rules for Event-B are outlined [Abr10]. The proof obligation rules related to variants, theorems and merging guards are not discussed, as they are not relevant to this thesis.

- **Invariant Preservation: INV**

This PO rule ensures every event preserves every defined invariant. This means whenever a variable which is constrained by an invariant is modified, it should be proved that the invariant is not violated. In other words, when actions of an event modify a variable involved in an invariant, the modified variable should satisfy the invariants.

- **Feasibility: FIS**

This PO states that actions of an event should be feasible. So when guards of an event hold and there is a nondeterministic assignment such as  $x : |Q(par, var, x')$ , it must be proved that this action can be performed. In other words, if  $var$  and  $par$  satisfy the invariants and guards, then there should exist an  $x'$  satisfying  $Q$ .

- **Guard Strengthening: GRD**

This PO helps to ensure that in a chain of refinement when a concrete event is enabled, its corresponding abstract event is also enabled. This is provided by proving that the guards of a concrete event are as strong as the guards of its corresponding abstract event.

- **Simulation: SIM**

The purpose of this PO rule is to ensure that every action of an abstract event is correctly simulated in its corresponding refined event(s). This means that the execution of a concrete event does not contradict the execution of its corresponding abstract event.

## 4.4 Decomposition in Event-B

The process of modelling and introducing new variables and events in refinement steps can increase the complexity of a model and consequently the model can become difficult to manage. At this point decomposition can help to split the developed model of a system into smaller pieces [AH07]. In the remainder of this section, two styles of the shared-variable decomposition [AH07] and the shared-event decomposition [But06, But09c, But09b] which allow to decompose an Event-B model are discussed.

### 4.4.1 Shared-Variable Decomposition

In the shared-variable decomposition technique [AH07], events of a model are split among the subsystems. Figure 4.3 illustrates this style of decomposition [SPHB11]. In this figure A, B and C are events which modify the variables  $v$ ,  $z$  and  $w$ . The dependencies between the events and variables are shown using links between them. For instance, in Figure 4.3 event A depends on variables  $v$  and  $z$ . In terms of Event-B this means that  $v$  and  $z$  appear in guards or actions of A.

In Figure 4.3, machine  $M$  is decomposed into two  $M1$  and  $M2$  sub-machines. The dashed line in machine  $M$  represents the point where  $M$  is decomposed. As shown, event  $A$  is moved to machine  $M1$ , and  $B$  and  $C$  to  $M2$ . Therefore,  $v$  and  $w$  are local (internal) variables of  $M1$  and  $M2$  respectively.

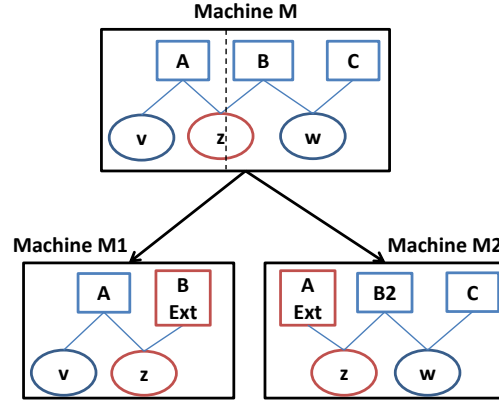


Figure 4.3: Shared-variable decomposition style.

However, variable  $z$  is shared between the two sub-machines and thus it is replicated in both  $M1$  and  $M2$ . In order to mimic the behaviour on the shared variable  $z$  in each of the sub-machines, external events should be defined. Events  $A \text{ Ext}$  and  $B \text{ Ext}$  in Figure 4.3 represent the external events which simulate the behaviour on the shared variable  $z$  in the subcomponents  $M1$  and  $M2$  respectively. Introducing external events gives rise to proof obligations (POs), such as the invariant preservation PO, which ensure the correctness of the sub-models.

#### 4.4.2 Shared-Event Decomposition

The basis of the shared-event decomposition technique “corresponds to the synchronous parallel composition of processes as found in process algebra such as CSP” [But09b]. In order to decompose an Event-B machine based on this technique, variables of the machine are partitioned amongst the sub-machines [But09b, SPHB11].

Figure 4.4 shows this style of decomposition. Here, the variables  $v$  and  $w$  of the machine  $M$  are partitioned amongst  $M1$  and  $M2$ . Therefore, the events of  $M$  are allocated to the appropriate sub-machine based on the dependencies between events and variables. For instance, event  $A$  modifies only variable  $v$ . Thus, this event is moved to the machine  $M1$ . The same is true for the event  $C$  and the variable  $w$ , which are moved to machine  $M2$ .

However,  $B$  depends on both variables  $v$  and  $w$ . In this case the event is divided into two events, namely  $B1$  and  $B2$ , each of which refers to the variable of a single sub-machine. In other words,  $B$  is divided in such a way that  $B1$  depends only on  $v$  and  $B2$  on  $w$ . So,

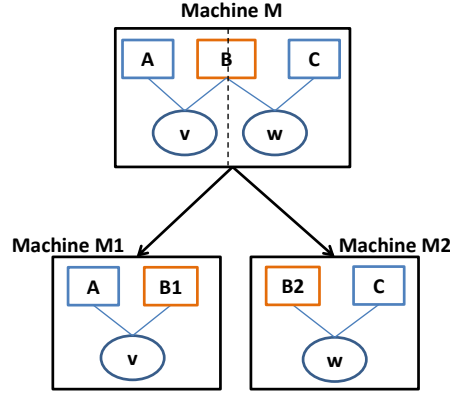


Figure 4.4: Shared-event decomposition style.

B is an event shared between M1 and M2, while events A and C are local (internal) to their machines.

This decomposition is discussed further using a simple example of a machine named VW1 in Event-B as shown in Figure 4.5. This machine is partitioned based on the variables  $v$  and  $w$  into two machines called V1 and W1 respectively, Figure 4.6.

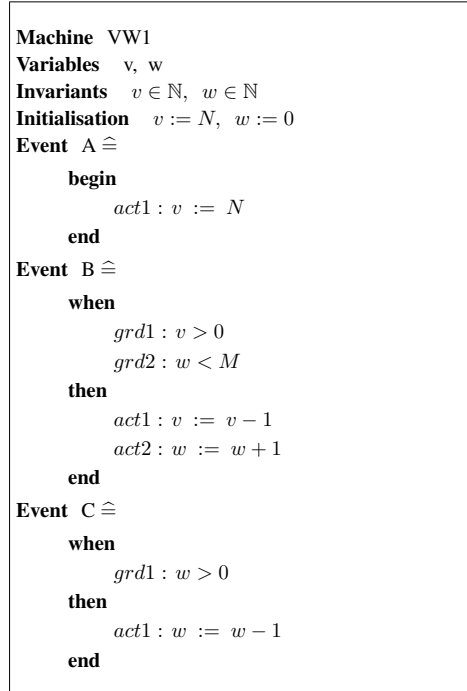


Figure 4.5: Event-B machine which is to be decomposed [But09c].

Since events A and C refer to  $v$  and  $w$ , which are local to V1 and W1 respectively, A is moved to V1, while C is moved to W1. However, event B refers to both variables  $v$  and  $w$ . Therefore, B is shared amongst V1 and W1.

Since *grd1* and *act1* of B depend on variable *v*, they are transferred to the machine V1 as shown in Figure 4.6. The rest of the event which consists of *grd2* and *act2* constructs the event B in W1. Thus, the machines V1 and W1 synchronise through the event B.

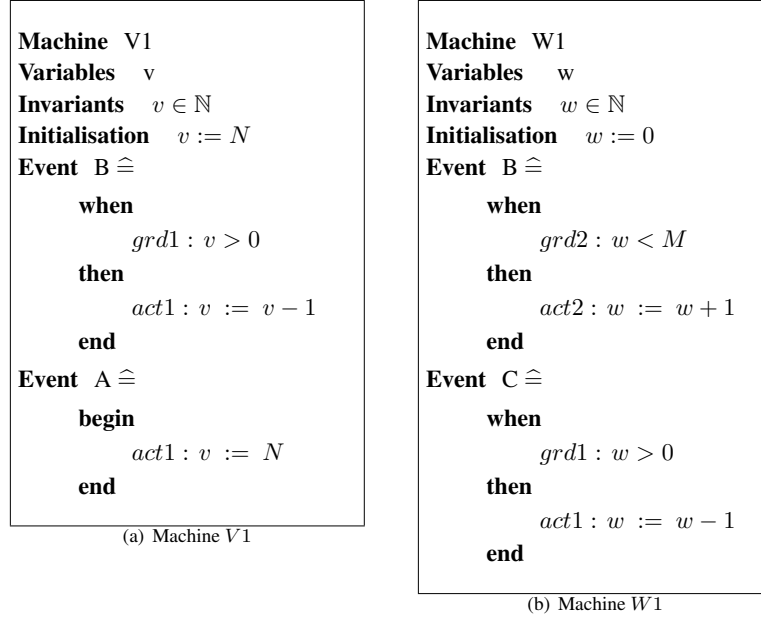


Figure 4.6: Machines VW1 is decomposed into V1 and W1 [But09c].

As will be discussed in Section 4.5, similar techniques are used to compose machines. For instance, the composition of the machines V1 and W1 in Figure 4.6 will result in the machine VW1 in Figure 4.5.

#### 4.4.3 Shared-Event Decomposition with Shared Parameters

It is possible for guards and actions with disjoint variables to have common parameters. As an example to decompose an event such as B in Figure 4.7, two sub-events can be constructed where one refers to the variable *v* and the other to *w*. In this case, the former event has the action *act1* and the latter contains *act2*. However, both events share the parameter *i*.

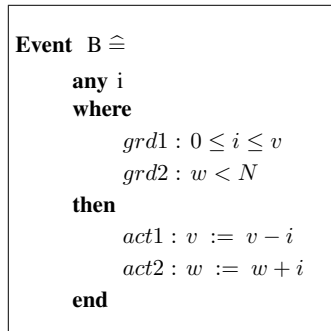


Figure 4.7: An event with shared parameter [But09c].

Figure 4.8 shows the decomposition of B to two events which share the parameter  $i$ . This parameter is constrained by  $grd1$  in the left sub-event, while it has a loose typing guard in right sub-event. This indicates that  $i$  is an output in the event B on the left, whereas it is an input in the other event.

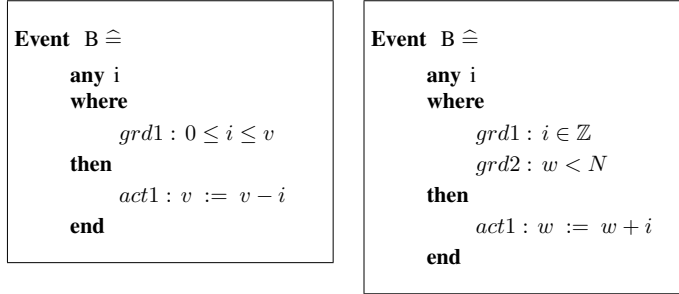


Figure 4.8: Decomposition of an event with shared parameter [But09c].

#### 4.4.4 Features of the Two Decomposition Styles

Sub-machines (or sub-models) generated as a result of a decomposition based on the shared-variable or the shared-event style can be refined further independently [AH07, But09b]. This allows team development, as each sub-machines can be developed further independently and in parallel. In addition, refining sub-machines independently, rather than refining the entire model, results in proof obligations of the model to be partitioned among the sub-machines. Therefore, proof obligations can be studies and discharged easier [SPHB11].

The following provides a comparison between the two styles of decomposition:

- In the shared-event decomposition it might be necessary to prepare a model for a decomposition by refining it to solve complex predicates and actions. For instance, in order to decompose an action such as  $x := y$  into two sub-machines, where the variable  $x$  is moved to one sub-machine and the variable  $y$  to another, we require a parameter which would act as an output in one sub-machine and as an input in another.

However, in the shared-variable style it is possible to partition a model at any point, so no preparation is required. However, sub-machines which result from a shared-variable decomposition might have a large number of shared variables if the decomposition happens at some arbitrary refinement level of a model [SPHB11].

- There is a restriction on the refinement of sub-machines which are decomposed using the shared-variable style. This restriction is that shared variables and external events cannot be refined [AH07]. However, there is no restriction on the refinement of sub-machines which result from a shared-event style decomposition.

- In the shared-event decomposition the interactions between sub-machines is by synchronising through shared events, whereas the shared-variable decomposition is based on sharing resources of the system. This makes the shared-event decomposition style more suitable for developing message-passing distributed systems [But97], while the shared-variable style is more suitable for designing parallel algorithms operating on some shared variables [SPHB11].

## 4.5 Shared-Event Composition in Event-B

In the shared-event composition style two Event-B machines which have *no variable in common* can be composed. Composing machines means that the elements of individual sub-machines will be conjoined to construct a single Event-B machine. This involves *conjoining* individual invariants, *conjoining* variables and *synchronising* events [SB10].

In the remainder of this section, shared-event composition is discussed in detail. In Section 4.5.1, parallel composition of events is explained. After that, Section 4.5.2 shows the structure of a composed machine.

### 4.5.1 Event Composition

To compose events, their variables should be independent of each other. However, events can share parameters. The definition of the composition of events  $e_a$  and  $e_b$  with a common parameter  $p$  is shown below.

$$\begin{aligned}
 e_a &\triangleq \text{ANY } p, x \text{ WHERE } G(p, x, m) \text{ THEN } S(p, x, m) \\
 e_b &\triangleq \text{ANY } p, y \text{ WHERE } H(p, y, n) \text{ THEN } T(p, y, n) \\
 e_a \parallel e_b &\triangleq \text{ANY } p, x, y \text{ WHERE } G(p, x, m) \wedge H(p, y, n) \\
 &\quad \text{THEN } S(p, x, m) \parallel T(p, y, n)
 \end{aligned} \tag{4.1}$$

The parameters  $x$  and  $y$  are local to their corresponding events, i.e.  $e_a$  and  $e_b$  respectively. Also,  $m$  and  $n$ , which are disjoint, respectively denote the variables of  $e_a$  and  $e_b$ .  $G$  and  $H$  represent the guards and actions are shown by  $S$  and  $T$ . As  $e_a \parallel e_b$  shows, the composition of the two events mean that their guards ( $G$  and  $H$ ) are conjoined and their actions ( $S$  and  $T$ ) happen simultaneously. As an example the composition of the events B1 and B2 is shown in Figure 4.9. This figure is similar to the decomposition example discussed in the previous section.



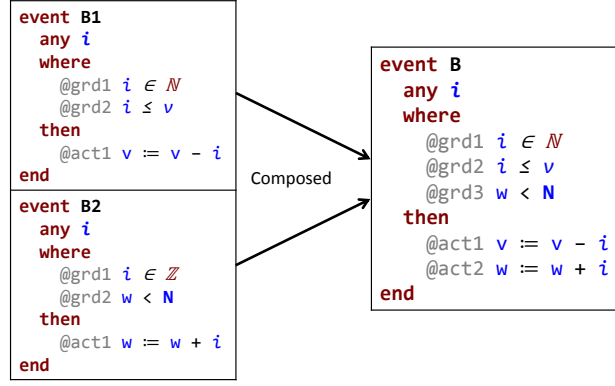


Figure 4.9: Composition of two events with a shared parameter.

#### 4.5.2 Structure of Machine Composition

Figure 4.10 represents a machine which consists of the parallel composition of machines  $M_1$  to  $M_n$ . As shown, variables of the composed machine are variables of machines  $M_1$  to  $M_n$ , i.e.  $v_1$  to  $v_n$ . Note that these variables are disjoint. Events in a composed machine are defined according to Definition 4.1 which was discussed in the previous section. As an example, event  $e_{11}$  from  $M_1$  can be composed with event  $e_{n1}$  from  $M_n$ , as well as some other events, to construct the event  $e_{cm1}$  which belongs to the composed machine  $CM$ .

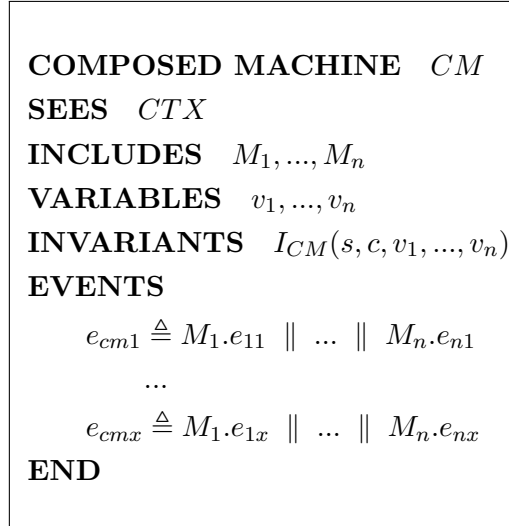


Figure 4.10: Structure of a shared-event composed machine.

Invariants of the parallel composition of  $M_1$  to  $M_n$  is shown in Definition 4.2. This invariant is defined as the conjunction of invariants in individual machines ( $I_1(s, c, v_1)$  to  $I_n(s, c, v_n)$ ) and the *composition invariant*, i.e.  $I_{CM}$ . The *composition invariant* captures properties that relate individual machines. In other words this invariant represents properties which involve variables  $v_1$  to  $v_n$ . In addition,  $I_{CM}$  can be defined based on the sets  $s$  and the constants  $c$  from the context  $CTX$  which is seen by  $CM$  [SB10].

$$I(M_1 \parallel \dots \parallel M_n) \triangleq I_1(s, c, v_1) \wedge \dots \wedge I_n(s, c, v_n) \wedge I_{CM}(s, c, v_1, \dots, v_n) \quad (4.2)$$

## 4.6 Rodin: the Event-B Tool

Unlike programming languages, formal methods may not be supported by tools [Win90]. However, one of the advantages of Event-B is the availability of an open source tool, known as Rodin, which is implemented on top of an extension of the Eclipse platform [ABH<sup>+</sup>10, ABHV06]. Some of the development ideas in the Event-B tool, such as performing tasks automatically in the background and fast feedback, come from modern integrated development environments for programming.

The three major components of the Rodin tool are the:

- *static checker*, which provides feedback about syntactical and typing errors in contexts and machines of an Event-B model.
- *proof obligation generator*, which generates proof obligations automatically for a model based on the Event-B proof obligations rules.
- *proof obligation manager*, which invokes automatic prover and keeps track of proofs related to an obligation and the status of each proof. Also, this component provides an interface for the proofs which are not discharged automatically and need to be discharged interactively.

Advantages of this tool are that firstly, the POs are generated automatically, whereas in theorem provers usually the user has to state what the theorems are to be proved. Secondly, POs are easily related to the model. So, if a PO cannot be discharged, it will be obvious which part of the model it relates to. This helps the user to find errors and inconsistencies of the model easier and quicker [Hal06]. Thirdly, since the Rodin tool is not just a theorem prover, it provides an interface for modelling and specifying a system. Fourthly, Rodin can be extended and integrated with other tools. This allows modelling various applications in Event-B and developing a wider usage of Event-B notation in industry as well as research [BH07].

Four plugins that have been used in this work regularly are ProB, Model Decomposition, Shared-Event Composition and the Camille Text Editor. These plugins are discussed in the next section.

## 4.7 Overview of Some Rodin Plugins

In this work, four Rodin plugins have mainly been used. In this section, an outline of these plugins is given.

- **ProB** [Pro11, LB03]: ProB is a model checker and an animator of B machines. ProB can be used for a non-exhaustive model checking. Alternatively an exhaustive model checking can be performed by restricting the sets and integer variables of the model. ProB also provides counterexamples if it discovers any violation of invariants.

Using ProB as an animator can help with debugging and testing a model. In this work, ProB is used primarily as an animator. In addition, the enabledness of events is tested using this plugin. In other words, models are checked for deadlock freeness by running the ProB model checker.

- **Model Decomposition** [Dec10, SPHB10, SPHB11]: The input of this plugin is an Event-B machine that is going to be decomposed. The user of this plugin can define the subcomponents and the style of decomposition (shared-event or shared-variable). After this, for a shared-variable (shared-event) decomposition style the user should specify events (variables) that are allocated to each subcomponent.

In addition to facilitate the process of decomposition, this plugin can guide the modeller if further preparation before decomposition is required. This is because the tool flags any variable, action or invariant that is too complex to be decomposed. As will be discussed in Chapter 8, we used this plugin for decomposing our model automatically.

- **Shared-Event Composition** [Com11]: This plugin can be used to combine sub-machines (or sub-models) into a single machine (model). This plugin is developed based on the shared-event composition style which was discussed in Section 4.5. A composition file, which looks similar to Figure 4.10, consists of five clauses, namely *refines*, *sees*, *includes*, *invariants* and *composes events*. When a composition file is created, the sub-machines which are to be composed can be added under the *include* clause. The composed events can also be introduced under the *composes events* clause.

This plugin is used in the future chapters to generate the model of a system from its sub-models. In other words, we specify a system as sub-models that can be composed to provide the overall specification.

- **Camille** [Cam11, BFJ<sup>+</sup>11]: Camille is a syntax-aware and semantic-aware text editor for Rodin. This plugin is used in order to specify the models.

## 4.8 Conclusion

In this chapter, we have discussed the Event-B formal language in detail. The constructs of the language, i.e. context and machines, have been explained. In addition, the elements of an event have been described in detail. We also discussed refinement as well as the shared-variable and the shared-event decomposition techniques in Event-B.

The constructs of the Event-B language and the discussed techniques are used in the remainder chapters to develop an approach for structuring, formalising and validating a requirement document. Simple notations of Event-B and its refinement notation - particularly, introducing new events and variables by refining the skip event - as well as the decomposition support in Event-B has made it a suitable formal language for the purpose of this thesis, which is to provide a structure to the process of modelling by defining patterns that use refinement and decomposition techniques.

The remaining of this thesis presents our contributions. Chapter 5 explains the four-stage approach of the MCMC which is developed to facilitate the transition from informal requirements documents to formal models. In Chapter 6, we discuss that requirements can be formalised as composeable sub-models. In this chapter, the structure of an Event-B machine is extended to facilitate the composition of sub-models as well as the formalisation of phenomena shared amongst sub-models.

In Chapter 7, we will use the Rodin tool and its plugins to apply the proposed approach to two automotive case studies. The patterns and guidelines for refining a control system into subcomponents are discussed in Chapter 8. Finally, Chapter 9 provides a detailed comparison between the contributions of this thesis and other approaches which were described in the background chapters, i.e. Chapters 2 to 4.



## Chapter 5

# Structuring, Formalising and Validating Requirements Documents

This chapter describes our first contribution, which is a four-stage approach for structuring, formalising and validating requirements of a control system. An overview of this approach is given in Section 5.1. The approach is based on variable and event phenomena of a control system which are broadly categorised into *monitored*, *controlled*, *mode* and *commanded* (MCMC). Section 5.2 and 5.4 discuss MCMC variable and event phenomena respectively. The relation between the MCMC event and variable phenomena is discussed in Section 5.5.

In the first stage of the approach, Section 5.3, MCMC variable phenomena are used to *structure a requirements document*. The second stage is to *formalise* the MCMC by modelling the structured requirements document (RD). This is explained in Section 5.6, where we firstly provide guidance on layering an RD in order to construct refinement chains, and secondly, represent an approach for modelling an RD as composable MCMC sub-models. Section 5.7 focuses on formalisation of MCMC variable and event phenomena in Event-B. This section also provides some patterns for formalising event phenomena.

At the third stage, discussed in Section 5.8, the formal model is *validated* against its RD. The fourth stage, which is *composing sub-models*, is explained in Chapter 6. Finally, in Section 5.9 we will conclude that this approach can facilitate the transition from informal requirements to a formal model.

## 5.1 Overview of the Four-Stage Approach

Our aim is to facilitate the process of formalisation of informal requirements documents (RDs) of control systems by providing guidelines and patterns. To do this a four-stage approach based on *system phenomena* is proposed. The concept of phenomena is taken from problem frames (PF) [Jac01]. This four-stage approach can be applied when modelling requirements of a control system using any state-based formal language with refinement and composition techniques. A further discussion on this is provided in Chapter 9. In this research we use the Event-B language. The stages are shown in Figure 5.1 on the page 60. they are summarised as follow:

- **Stage 1 - Structuring RD:** The Parnas and Madey's four-variable model [PM95] suggests the identification of *monitored* and *controlled* variables in system requirements documents. In addition, the four-variable model introduces *input* and *output* variables which can be identified for software requirements documents.

Influenced by the first category of variables proposed in the four-variable model, we suggest the identification of *monitored*, *controlled*, *mode* and *commanded* (MCMC) variable phenomena for system requirements document of a commanded control system (consists of plant, controller and operator) which is mainly the type of control systems we discuss. Section 5.2 explains each MCMC variable phenomenon in detail.

After this, Section 5.3 discusses the structuring process of a control system RD. Here, an RD is structured into monitored, controlled, mode and commanded (MCMC) sections, each accommodating the requirements about their corresponding variable phenomena. For instance, the monitored section of a structured RD represents requirements related to monitored variable phenomena of the system.

- **Stage 2 - Formalising RD:** The aim of this stage is to map a structured RD to its formal representation in a step-wise manner using refinement. To do this the MCMC variable phenomena should be modelled alongside their corresponding event phenomena, since events provide means of updating and modifying the variables. Thus, before commencing the process of formalisation, it is important to identify event phenomena of the system.

The MCMC event phenomena are explained in Section 5.4. After that in Section 5.5 the variable and event phenomena are related. We start the process of modelling a structured RD in Section 5.6. To reduce the complexity and the effort required for formalising an RD we take advantage of *refinement* and *composition* techniques.

We propose to model a control system as composeable sub-models, where each sub-model is formalised using refinement. Section 5.6.1 provides some guidelines on layering requirements where each layer will be modelled in a distinct refinement

step. These layering guidelines are based on the refinement techniques in the Event-B language. Section 5.6.2 describes how the MCMC sections of a structured RD can be treated as sub-problems which will be mapped to their corresponding sub-models. The composition of the sub-models provides the overall specification. Formalising sub-problems as sub-models is discussed further in Chapter 6.

Section 5.7 describes the activities for formalising sub-problems in Event-B. This section discusses the formalisation of the MCMC variables and events in Event-B. Also, it provides patterns for formalising the MCMC event phenomena in the Event-B formal language. These patterns are developed based on the relations between variable and event phenomena, Section 5.5.

In Figure 5.1, the sub-models are shown using a dotted rectangular around their corresponding refinement chain. Also, in this figure, the rectangles with solid lines in this stage (as well as Stage 3) represent machines in Event-B. The solid upwards arrows that connect machines in Stage 2 and Stage 3 represent refinement of an abstract machine to a concrete machine. As an example,  $Mntr\_1 - V_0$  is the abstract machine in the monitored sub-model. This machine is refined in several steps to the concrete machine  $Mntr\_p - V_0$ .

Note that it is possible to formalise the entire RD of a system as a single model rather than sub-models. This is explained in Section 5.6.2. Advantages of formalising an RD as sub-models is discussed in Chapter 7 after describing the case studies.

- **Stage 3 - Validating RD against its Formal Model and Revision Step:** At this stage we validate the model against its RD by relating each requirement to its representative formal elements of the model. This allows us to trace between the RD and the model as well as validate and justify that the requirements are modelled correctly. In addition, the processes of modelling and validating help to improve our understanding of the system and new or ambiguous behaviour might be identified. At this point it might be necessary to revise the structured RD and the model to incorporate the new/unambiguous behaviour. Validation and revision of RDs and their formal models are discussed in Section 5.8.
- **Stage 4 - Composition of Sub-Models:** When an RD is formalised as sub-models, they can be composed in order to obtain the overall specification. However, sub-models may share phenomena. Thus, composition techniques can be used to allow us to deal with the shared phenomena. In Stage 4 of Figure 5.1, the downwards arrows represent the composition of the sub-models. This stage of the approach is discussed in Chapter 6.

Note that the requirement structuring and formalising guidelines are proposed to improve the efficiency of the modelling process. We acknowledge that other ways of modelling requirements exists. Examples are UML to B [SB06], where requirements are



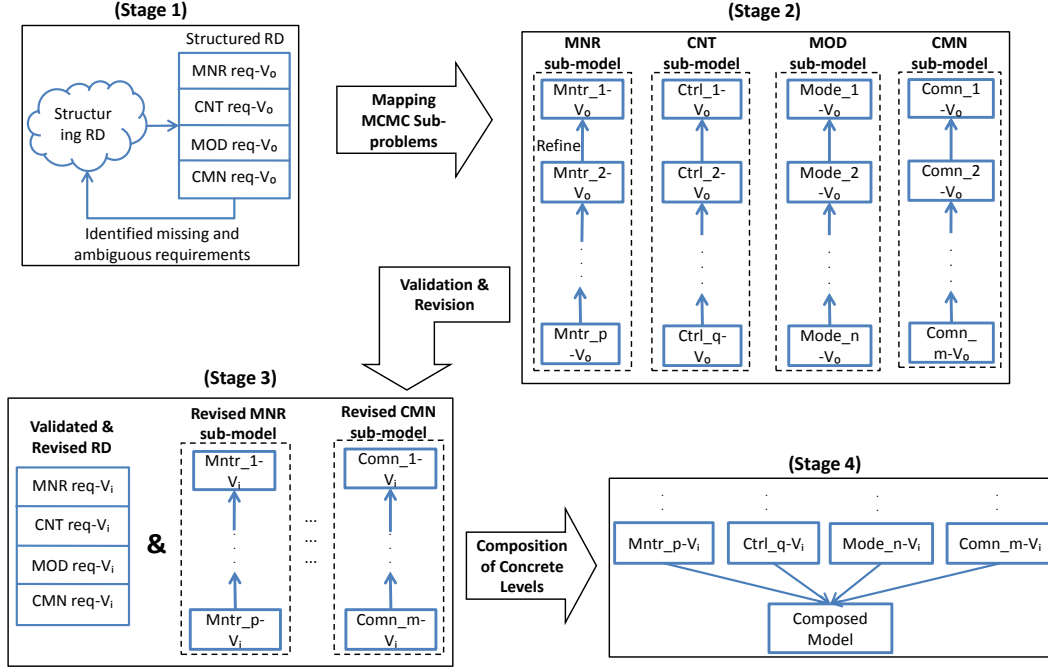


Figure 5.1: Overview of the four stages of the MCMC approach.

captured as UML-like diagrams that can be translated into Event-B, and KAOS to Event-B [PD11] where requirements models are connected to formal design models in Event-B.

To clarify the approach, in this chapter we use the example of a cruise control system [YBR10] at every stage. In Chapter 7, the proposed approach is applied in more detail to two automotive case studies of a lane departure warning system and a lane centering controller.

## 5.2 MCMC Variable Phenomena

The first stage of the MCMC approach is to structure requirements document. Since, this stage is based on the *monitored*, *controlled*, *mode* and *commanded* (MCMC) variable phenomena, in this section we discuss them in detail. The structuring approach is explained in the next section.

Section 5.2.1 defines the MCMC variable phenomena. As an example Section 5.2.2 provides the MCMC phenomena of a cruise control system (CCS). In Sections 5.2.3 and 5.2.4 we will discuss the role of a commanded and a mode variable phenomenon respectively. Note that in this section as well as the next section where the structuring stage is discussed, the term ‘phenomena’ and ‘variable phenomena’ are used interchangeably.

### 5.2.1 MCMC Variable Phenomena

An *autonomous control system* consists of plant and controllers, whereas a *commanded control system* consists of plant, controllers and operators who can send commands to the controller [But09a, But09d]. This is shown in Figure 5.2, which is based on Problem Frames diagrams, Section 2.3.2. In this figure, the controller domain is shown as the machine domain (a rectangle with three stripes) and the plant and operator domains form the problem world.

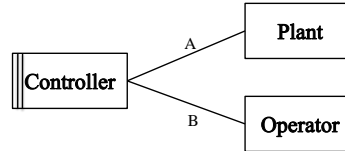


Figure 5.2: A commanded control system.

The initial step of the approach to formalising RDs is to identify control system phenomena. These are *monitored*, *controlled* and *mode* variable phenomena for an autonomous control system. For commanded control systems, we propose the identification of *monitored*, *controlled*, *mode* and *commanded* (MCMC) variable phenomena. Monitored and controlled phenomena represent *environment* quantities, as they belong to the environment or the plant. Whereas, mode and commanded phenomena are *controller* quantities, meaning that they are phenomena internal to the controller. This is discussed further in Chapter 8, where design details are introduced.

- *Monitored variable phenomena* are environment phenomena whose values are determined by the environment or the plant.
- *Controlled variable phenomena* represent phenomena in the environment whose values are set by the controller.
- The *mode variable phenomenon* represents a controller phenomenon that captures distinct manner of system operation. A control system has one mode variable phenomenon whose value represents the state of the system at every moment in time.
- *Commanded variable phenomena* are controller phenomena whose values are determined by the operator and that influence controlled and mode phenomena.

To identify the MCMC phenomena of a system their definitions and the system requirements will be used. Note that for the mode phenomenon we identify its possible values. However, for the other categories we identify distinct variable phenomena. The identification of the MCMC phenomena is discussed further in the next section using the example of a cruise control system (CCS).

### 5.2.2 Identifying MCMC Phenomena of a CCS

A CCS receives the *actual speed* of the car from the environment and the *target speed* from the driver and its role is to minimise the difference between these two. This is done by setting the *acceleration* of the car via a speed regulation mechanism [Abr09]. Some of the requirements of the CCS are demonstrated in Table 5.1 [Abr09].

1	CCS can be switched <b>on</b> or <b>off</b> by the driver.
2	CCS shall monitor the vehicle's actual speed.
3	Once CCS is switched on, the driver can determine the target speed by setting it to the value of actual speed.
4	When CCS is on, it will be <b>activated</b> as soon as the driver sets the target speed.
5	The target speed is always within a specific range.
6	Once CCS is active, the speed regulation mechanism will be automatically invoked.
7	When CCS is active, the speed regulation mechanism will maintain the difference between actual speed and target speed as close to 0 as possible by correcting the acceleration according to speed control laws.
8	Once CCS is active, if the driver uses the brake pedal, the speed regulation mechanism will be <b>suspended</b> .

Table 5.1: Some requirements of a CCS (based on [Abr09]).

Based on the definition of the MCMC variable phenomena and the CCS requirements, Table 5.2 shows some of the monitored, controlled and commanded variable phenomena of the CCS as well as values of the mode variable.

Monitored Variable	Controlled Variable
actual speed	acceleration
Commanded Variable	Values of Mode
target speed	on, off, active, suspend

Table 5.2: MCMC variable phenomena of CCS.

### 5.2.3 Commanded phenomena

In this section, the concept of commanded phenomena is explained in more detail. Firstly, we discuss why it is useful to have commanded phenomena. After that, the ambiguity between this and monitored phenomena is explained. Section 5.2.3.3 suggests how to clarify this ambiguity.

### 5.2.3.1 How Commanded Phenomena Help

The special role of the operator in a commanded control system is captured as *commanded* phenomena whose values are stored and maintained by the controller. The benefit of identifying commanded phenomena might be questionable for the readers as it might seem possible to capture operator commands as *monitored* phenomena. This view is mainly the result of perceiving the system from the controller (or the software to be implemented) point of view, where any input to the system is treated as a monitored quantity.

However, a system-level view where inputs are differentiated and operator actions are taken into account can provide a better understanding and a more accurate perception of the controller behaviour. Furthermore, requirements should be considered in the system-level view, as they represent details of the controller within an environment which may also consists of operators. In addition, conducting several case studies has shown that monitored and commanded phenomena are different in the following ways:

1. Monitored phenomena are part of the environment, which is usually *predictable*. However, commanded phenomena involve people, who are *unpredictable*. Problem frames [Jac01] distinguishes between these as causal and biddable domains.
2. A controller usually receives values of monitored phenomena *periodically* and through means such as sensors. For instance, a CCS receives the car speed every  $x$  milliseconds through the speed sensors. However, operator commands are *sporadic* requests sent through a user interface, such as a button. For example, in a CCS, a driver can set the target speed using the set button at any point after switching on the CCS.
3. Usually values of commanded phenomena have to be preserved by the controller, while values of monitored phenomena are received at every control cycle and responded to by the controller. In other words, commanded phenomena are maintained *internally* by the controller and has no representation outside the controller (i.e., quantities of the controller), whereas the monitored phenomena are part of the environment (i.e., quantities of the environment) and usually their values are stored temporarily.

The first difference shows that there should be minimal assumptions on operator behaviours, or even possibly none. For instance, it cannot be assumed that an operator will switch off a controller as soon as they are notified of a fault in the system. Therefore, to achieve a well-behaved system (a system with no surprising results) the requirements and the design of the system should include situations where the operator does not follow the expected routine.

The second difference shows that usually for a commanded phenomena a user interface should be designed. Whereas it is more likely that sensors which are used for transferring values of monitored phenomena will be configured or modified to fit their purposes, rather than designed. We recognise that some control systems may require specialised sensors which should be designed according to their specification. However, this discussion is avoided as requirements related to design of sensors are beyond our current work.

As an example of the third difference the target speed in the CCS can be increased or decreased by sporadic operator requests. This means while the controller is active, it requires to maintain the value of the target speed. So this value can be increased or decreased as soon as the controller receives a corresponding operator request. In contrast, monitored phenomena are sensed at the start of each control cycle and then temporarily stored to be responded to by the controller.

### 5.2.3.2 Ambiguity of Commanded Phenomena

As mentioned, in order to model a system using the MCMC guidelines, an engineer is required to identify the monitored, controlled and commanded variable phenomena as well as possible modes of the controller. While the distinction between monitored, controlled and mode variable phenomena is usually straightforward, identifying commanded phenomena and sometimes distinguishing them with monitored phenomena can be more challenging. We use the example of the CCS to show this ambiguity.

Commanded phenomena are defined as phenomena whose values are determined by the operator. Based on this definition and also Requirements 3 and 8 of Table 5.1, *target speed* and *brake pedal* can be identified as commanded phenomena of the CCS. However, after modelling and examining these phenomena, we realised that the behaviour of *brake* and *target speed* are different. The latter is defined specifically to serve the CCS, while the former is a more generic phenomenon. This is because the primary role of the brake pedal is to reduce the car speed, while its secondary role is to suspend the CCS. Also, the brake is part of the plant (the given environment), whereas a user interface (such as an arrangement of buttons) needs to be implemented to allow interactions between the driver and the target speed.

This distinct characteristic can cause potential confusion, since it is possible to treat the brake as either a commanded phenomenon, because it is a form of interaction between the operator and the controller, or a monitored phenomenon, because it is part of the environment. Identifying the type of a phenomenon is important since it can affect the modelling process, particularly in vertical refinements where implementation details are modelled. In the next section we provide guidelines which clarify the distinction between monitored and commanded variables.

### 5.2.3.3 Distinguishing Monitored and Commanded Phenomena

The conducted case studies have shown that one way of differentiating between the monitored and commanded phenomena is to distinguish between their user interfaces (UI). We categorise the UI of a control system as:

- *Existing interface*: This is the UI whose primary role is to serve the given environment. The existence of such UI does **not** rely on the existence of the control system. For instance pedals in cars exist even without a CCS. Another example, is the indicator lever whose primary role is to inform other road users of the driver's intention. However, a lane departure warning system receives data from the indicator to determine intentional lane departure.
- *Specialised interface*: This is the UI whose primary role is to serve the controller. This UI will exist only when its corresponding control system is implemented. An example is an increase or a decrease button in a CCS which changes the value of the target speed.

Another difference between these two UI is in the way they are connected to a control system. An *existing interface* may be connected to one or more controllers in a plant. Also this type of UI usually *broadcasts* operator interactions as messages which can be received by any controller connected to the UI. For instance, a brake pedal can be connected to a CCS as well as an anti-lock braking system (ABS). In this case, if the brake is pressed, the message will be broadcast and consequently received by both systems. However, a *specialised interface* is usually designed for and directly connected to one control system.

The case studies that we have considered showed that in automotive systems an *existing interface* is usually connected to one or more controllers via a Controller Area Network (CAN bus) [CAN13] which uses broadcasting mechanism to transfer data. However, a *specialised interface* is usually connected to its corresponding control system using the Local Interconnect Network (LIN bus) [LIN13]. Note that at system level requirements we are not concerned with such design details.

We use the distinction between an existing and a specialised interface as one way of distinguishing monitored and commanded variable phenomena. We redefine monitored phenomena as phenomena whose values are determined by existing interfaces of a plant/environment. Whereas, phenomena whose values are determined by the operator via *specialised interfaces* are commanded phenomena. Based on this definition, the brake pedal in the CCS is a monitored phenomenon, while the target speed is a commanded phenomenon. Thus, the requirement of Table 5.3 can be added to the RD of the CCS shown in Table 5.1 in order to clarify that brake pedal is a monitored phenomenon.

..	...
8	...
9	CCS shall monitor any pressure applied to the brake pedal.

Table 5.3: Brake pedal is an existing interface for CCS.

### 5.2.4 Mode Phenomenon

Experimenting with the requirements and models of the conducted case studies showed that these systems usually have one particular phenomenon in common. This phenomenon which is called *mode* is internal to a controller (i.e., a controller phenomenon) and its values represent states or modes of the controller. Examples of the mode values in the CCS are **on**, **off**, and **suspended**.

Mode is also a special type of phenomenon, since both the *controller* and its *operator* can update its value. For instance, in the example of the CCS, Table 5.1, the driver can switch the CCS **on** or **off** (requirement 1). In addition, the controller itself can update the mode to **suspend**, when the brake pedal is pressed (requirement 8). Section 5.4.2 explains these two ways of updating the mode variable phenomenon (via an operator or a controller request) in more detail.

Because of this characteristic, separating the mode variable phenomenon from commanded or controlled can greatly clarify the process of identifying the MCMC phenomena and subsequently structuring and formalising the RD.

Another characteristic of the mode variable phenomenon is that it can be used to deal with faults and errors of the system. This is especially the case when modelling a system. If an error is detected, the control system can change its mode, for instance to a **recovery** or a **fault** state. This way the controller avoids performing its normal role. An example of this is shown in the next chapter.

## 5.3 Stage 1: Structuring an RD Using MCMC Phenomena

This section explains the first stage towards the formalisation of a textual RD. We propose to structure the functional requirements of a control system according to its MCMC variable phenomena [YB11, YB12]. Therefore, the very first step is to identify the *monitored*, *controlled* and *commanded* variable phenomena of the control system as well as its possible *modes*.

After the identification of the variable phenomena, the requirements are divided into monitored, controlled, mode and commanded (MCMC) sections as shown in Figure 5.3.

Each requirement is then placed into its appropriate section, according to the phenomenon it represents. Note that structuring an RD is an incremental process where the most obvious MCMC phenomena and their requirements are structured in the initial step and then the RD is elaborated with new and unambiguous requirements.

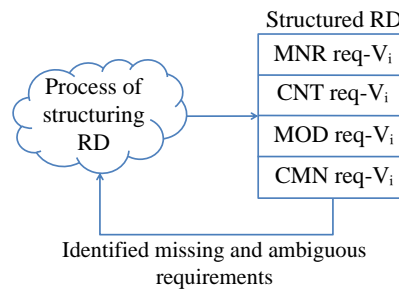


Figure 5.3: Structuring RD based on MCMC phenomena.

The steps for structuring an RD based on the MCMC phenomena are as below:

1. *List the system's monitored, controlled and commanded phenomena.* **Actual speed** and **target speed** are examples of a monitored and a commanded phenomena in the CCS.
2. *List the values of the mode phenomenon.* Examples of mode values in the CCS are **on**, **off**, and **suspend**.
3. *Organise the RD into monitored (MNR), commanded (CMN), controlled (CNT) and mode (MOD) sections.* Take every requirement and based on the list of phenomena determine the **main variable phenomenon** that the requirement refers to. This is the phenomenon for which the requirement describes *how* it should be modified or updated. The requirement is then placed in the section which represents the type of its main variable phenomenon.

An example in the CCS is the requirement “when CC is on, it will be activated as soon as the driver sets the target speed” which defines changes of the mode phenomenon (from **on** to **active**) under certain conditions. Thus, we place this requirement in the MOD section. Another example is the requirement “when CCS is active, the speed regulation mechanism maintains the difference between actual speed and target speed as close to 0 as possible by correcting the acceleration according to speed control laws”. This defines the modification of the controlled variable *acceleration* so we place it in the CNT section.

4. *Add unique ID labels.* Every ID starts with the section the requirement belongs to (i.e. MNR, CNT, MOD or CMN), followed by a unique number. This ID is used for traceability.
5. *Revise the RD* to accommodate any identified missing or ambiguous behaviour of the system. The revision step involves going back to Step 1 to identify any



phenomena that the new requirement represents and adding the requirement to an appropriate section.

Notice that requirements of the monitored section represent the *assumptions* on the environment, since they describe environment changes which are outside the control of the controller.

Applying the first two steps to the textual requirements of the CCS will result in listing the phenomena and mode values as shown in Table 5.4.

MNR	CNT	CMN	MOD
actual speed, brake pedal	acceleration	target speed	on, off, active, suspend

Table 5.4: MCMC variable phenomena of CCS are listed to structure the RD.

After this we take the final three steps to categorise the requirements into the MCMC sections. The structured requirements are represented in Table 5.5.

Phen.	ID	Requirement Description
Actual speed	MNR1	CCS shall monitor the vehicle's actual speed.
Brake pedal	MNR2	CCS shall monitor any pressure applied to the brake pedal.
Acceleration	CNT1	Once CCS is <b>active</b> , the speed regulation mechanism will be automatically invoked.
	CNT2	When CCS is <b>active</b> , the speed regulation mechanism will maintain the difference between actual speed and target speed as close to 0 as possible by correcting the acceleration according to speed control laws.
Mode	MOD1	CCS can be switched <b>on</b> or <b>off</b> by the driver.
	MOD2	When CCS is on, it will be <b>activated</b> as soon as the driver sets the target speed.
	MOD3	Once CCS is <b>active</b> , if the driver uses the brake pedal, the speed regulation mechanism will be <b>suspended</b> .
Target speed	CMN1	Once CCS is switched <b>on</b> , the driver can determine the target speed by setting it to the value of actual speed.
	CMN2	The target speed is always within a specific range.

Table 5.5: Structured RD of the CCS.

## 5.4 MCMC Event Phenomena

As mentioned, the second stage of the approach is to formalise the structured RD of a control system. To do this it is necessary to identify the transitions (or the operations) of the control system in addition to its variable phenomena. Therefore, before discussing the formalisation process, Section 5.4.1 explains the MCMC *event phenomena*. Also, a discussion on mode events is given in Section 5.4.2.

### 5.4.1 Definition of MCMC Event Phenomena

For every variable phenomenon at least one event phenomenon which modifies and updates the variable is required. The following defines the event phenomena corresponding to each category of the MCMC variable phenomena.

- *Monitor event phenomena* update monitored variable phenomena.
- *Control event phenomena* update controlled variable phenomena.
- Two types of *mode event phenomena* are defined, as they can be triggered by the operator or by the controller:
  1. *Operator mode event (OME) phenomena*, which update the mode based on the operator requests, such as switch on.
  2. *Controller mode event (CME) phenomena*, which update the mode based on the values of monitored variables.

These two event phenomena are explained further in the next section.

- *Command event phenomena* are controller's responses to operator requests for modifying commanded variable phenomena.

Some of the monitor, control, mode and command event phenomena of the CCS requirements are shown below.

Monitor Event	Control Event
Update actual speed	Update acceleration
Command Event	Mode Event
Set target speed	Switch on, Suspend

Table 5.6: MCMC event phenomena of CCS.

As discussed, the requirement structuring steps are based on the variable phenomena. However, listing command and mode event phenomena can be beneficial in structuring an RD, since they represent the *specialised interface* and can provide a better understanding of the requirements.

### 5.4.2 Mode Event Phenomenon

As mentioned, mode is a special phenomenon, since it can be updated by an operator or by the controller. *Operator mode event* (OME) and *controller mode event* (CME) phenomena are respectively requested by the operator and by the controller. Based on the requirements of the CCS in Table 5.5 respective examples of OME and CME are **switch on** and **suspend** events.

The decision to trigger an OME takes place *externally* by the operator (similar to command events), whereas performing a CME is an *internal* decision that the controller makes according to the monitored variable phenomena (similar to control events). Differentiating between these two events is important, since their modelling processes can be different.

In particular, during the vertical refinement, Chapter 8, where the design decisions are modelled, the refinement of an OME will be different from the refinement of a CME. An OME most likely requires a UI of the category *specialised interface* to be implemented to provide means of interactions between the operator and the controller, whereas a CME requires the introduction of sensors to the model.

These two events impose different requirements on the system design. For instance, some of the issues that should be considered when modelling the **switch on** and **off** OMEs in a CCS are:

- What kind of interface to define? A toggle button or two separate on and off buttons?
- What should happen if the initial conditions for switching the system on do not hold, but the driver requests to switch the CCS on?
- What is the priority of responding to the switch button in comparison to other buttons?

To identify the mode event phenomena, one possibility is to use a state machine diagram whose transitions represent CME and OME phenomena. Figure 5.4 illustrates a state machine for mode events in a CCS. Here **off** is the initial state of the CCS. Notice that not all mode transitions of Figure 5.4 are discussed in the CCS RD.

## 5.5 MCMC Event and Variable Phenomena

In this section, the dependencies between the MCMC variable and event phenomena are discussed. Section 5.5.1 provides details on the dependencies of the variable and event phenomena.

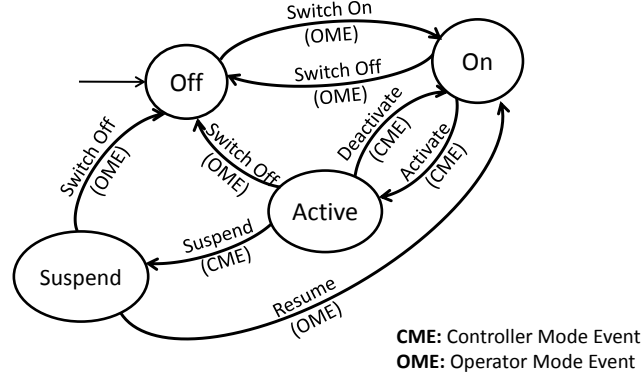


Figure 5.4: Modes of the CCS and their transitions.

Examples of the instantiated dependencies based on the CCS phenomena (Table 5.4 and 5.6) are given in Section 5.5.2. These dependencies can then be simplified based on the requirements of the CCS. The simplification involves eliminating any variable phenomena irrelevant to the specified event phenomenon.

Later on in Section 5.7, these dependencies are used to provide elaborated *guidelines and patterns* for formalising event phenomena in the Event-B language. In the remainder of this section the terms ‘variables (events)’ are sometimes used to refer to the informal variable (event) phenomena.

### 5.5.1 Dependencies between MCMC Event and Variable Phenomena

Table 5.7 shows the dependencies between MCMC event and variable phenomena. In this table the columns from left to right represent firstly, the type of events, secondly, the variable phenomena on which an event depends and thirdly, the type of a variable phenomenon which is modified by the event.

In Table 5.7, MNT, CNT and CMN denote the set of all monitored, all controlled and all commanded variable phenomena respectively. However, *mode* represents the single variable phenomenon mode. This variable can be set to possible states of the control system.

As shown in the table, monitor events modify monitored variables based on the values of CNT and MNT. The mode and commanded variables have no direct influence on the environment, since they are internal to the controller (i.e. controller quantities). The effects of these variables are on controlled phenomena. So, they can influence the value of a monitored variable indirectly.

As illustrated, in the control and mode events any MCMC phenomena can affect the value of a controlled or a mode variable. This is because a controller can update controlled or mode variables based on the state of the environment (MNR and CNT) as well as its internal state (MOD and CMN).

Event Phenomenon	Depends on	Modifies
Monitor Event	MNR CNT	MNR
Control Event	MNR CNT mode CMN	CNT
Mode Event	MNR CNT mode CMN	mode
Command Event	MNR mode CMN	CMN

Table 5.7: Dependency between MCMC event and variable phenomena.

A commanded phenomenon can be affected by the mode and by the set of monitored and commanded variable phenomena (MNT and CMN). CNT is not present in a command event, as the value of a commanded variable can change in response to an operator request only. So, the controlled variable phenomena should have no role in a command event. However, it is usually the case that an operator request can be responded only when the controller is in a specific mode. Therefore, the mode variable can influence the occurrence of a command event.

### 5.5.2 Instantiating Event and Variable Phenomena Dependencies

Table 5.8 shows instantiations of the MCMC event and variable dependencies based on the phenomena of the CCS (Table 5.4 and 5.6).

Table 5.8 can be simplified based on the requirements of the CCS by eliminating irrelevant variable phenomena. The simplified dependencies are shown in Table 5.9.

The monitor event phenomenon *Update Actual Speed* is simplified based the requirements MNR1 of the CCS (Table 5.5) to show that *actual speed* can be updated based on its previous value as well as the value of the *acceleration*.

Based on the requirements of the CCS (CNT2 - Table 5.5), the *Update Acceleration* event phenomenon is simplified to represent that the value of the *acceleration* depends on the *actual speed*, the *target speed*, the previous value of the *acceleration* and the *mode*.

Event Phenomenon	Depends on	Modifies
Update Actual Speed	$actual\ speed \times brake\ pedal$ $acceleration$	$actual\ speed \times brake\ pedal$
Update Acceleration	$actual\ speed \times brake\ pedal$ $acceleration$ $mode$ $target\ speed$	$acceleration$
Suspend	$actual\ speed \times brake\ pedal$ $acceleration$ $mode$ $target\ speed$	$mode$
Set Target Speed	$actual\ speed \times brake\ pedal$ $mode$ $target\ speed$	$target\ speed$

Table 5.8: Dependency between MCMC event and variable phenomena.

In addition, the *Suspend* and *Set Target Speed* events are simplified based on the requirements MOD3 and CMN1 in Table 5.5 respectively. The *Suspend* event shows when the value of the brake pedal changes (from not pressed to pressed), the *mode* is updated, while the *Set Target Speed* event states that the value of the *target speed* depends on the *actual speed* and the *mode*.

Event Phenomenon	Depends on	Modifies
Update Actual Speed	$actual\ speed$ $acceleration$	$actual\ speed$
Update Acceleration	$actual\ speed$ $acceleration$ $mode$ $target\ speed$	$acceleration$
Suspend	$brake\ pedal$ $mode$	$mode$
Set Target Speed	$actual\ speed$ $mode$	$target\ speed$

Table 5.9: Dependency between MCMC event and variable phenomena.

## 5.6 Stage 2: Formalising a Structured RD

Formalising requirements and modelling all variables and events in a single flat development step can be difficult and time consuming, because functionalities of control systems are usually broad and their RDs are complex. We avoid formalising a long list of requirements in a single step by using *refinement* techniques. We use the term *horizontal refinement* to refer to refinement steps where behaviours and requirements are introduced to a model.

It is usually “not easy to decide how to organise the construction steps” or determine the requirements which are modelled at the abstract level [Abr06]. In this section we propose to organise a horizontal refinements chain of a model based on the MCMC phenomena. Section 5.6.1 provides guidelines on layering requirements to establish the level of refinement where they can be modelled at.

To break the complexity of modelling even further, we propose to decompose a requirements document into sub-problems which can be formalised as composable sub-models. This is discussed briefly in Section 5.6.2. A thorough discussion is given in Chapter 6.

### 5.6.1 Refinement Guidelines by Layering an RD based on Variable Phenomena

To simplify the construction of refinement steps, we suggest layering requirements of a structured RD and modelling each layer in one level of refinement. The layering is based on variable phenomena of a control system. We propose three guidelines for layering the requirements and subsequently defining the order of the refinement chain.

Note that these refinement guidelines are based on the refinement techniques in Event-B which allow the introduction of new behaviours (new variables and events) in refinement. Also, these guidelines require the modeller’s judgment. Human judgment is important in the development of a formal model, e.g. in validating a model against its requirements. Human judgment can be contrasted with a mathematical judgment which provides the means for proofs and verification.

1. Minimising the number of variable phenomena of every layer (or every refinement step). Our suggestion is to aim for formalising *one* variable phenomenon and its event phenomena that update the variable at every step. If the variable and its behaviour are straightforward, one might decide to introduce several variables in one refinement step. However, we avoid complicating a model by introducing too many variables.

In some cases a phenomenon does not stand independently, meaning that it is related to other variable phenomena. We propose to identify the relation between the phenomena and model them as follow:

- *Simultaneous Relations* where several phenomena contribute to the specification of an expression. In this relation, the meaning of the expression depends on a number of phenomena which should be modelled either before or at the same time as specifying the expression. An example is when phenomena are compared to one another. For instance we suggest modelling a requirement which includes a clause such as “when  $x$  is greater than  $y$ ” as one of the following ways: either formalise  $x$  and  $y$  in any order in refinement levels prior to modelling the expression  $x > y$ ; or model the expression simultaneously as formalising  $x$  and  $y$ .
- *Dependant-Dependee Relations* where one phenomenon is dependent on other phenomena. For instance the requirement CMN1 in the CCS which states the driver can set the target speed to actual speed by pressing a button, shows that the value of the target speed depends on the value of the actual speed. We propose to model such phenomena by either introducing the dependee phenomena prior to the dependant; or modelling them simultaneously in one refinement level. A dependant is a phenomenon whose value depends on a dependee, e.g.  $dependant \in R[\{dependee\}]$ . In CMN1, target speed is a dependant phenomenon. So, to model the requirement CMN1 we can either introduce the two phenomena in one level or formalise actual speed in a level prior to introducing target speed.

2. Postponing the modelling of the conditions under which a variable phenomenon can be updated to refinement levels later than the level in which the variable is modelled. In other words, we try to model the *guards* of events after modelling their *actions*. Following this guidance helps to minimise the number of phenomena introduced at each level. An example is the requirement CMN1 which shows that the condition for updating the target speed variable is that the CCS should be switched on, e.g.  $mode = ON$ . In this example, the introduction of the mode phenomenon can be delayed to a post-target speed refinement level.
3. Defining the *main phenomenon* (or the main role) in the most abstract level. If the system has more than one role, it is the modeller’s judgment to choose the most important one.

An example of a refinement chain which is constructed using the above guidelines is shown in Figure 5.5. This figure also shows the aim at every refinement level, as well as the phenomena and requirements which are modelled at each level. The main behaviour of the CCS is to set the acceleration of the car. This behaviour which corresponds to



the controlled phenomenon *acceleration* is modelled in the abstract level. In addition, acceleration depends on *actual* and *target speed* phenomena, since its value is determined by them. Thus, these three phenomena are modelled in the abstract level. In the second level we modelled the *mode* phenomenon and its transitions. After this, the *brake pedal* is modelled, as it represents a condition under which *mode* can become suspended.

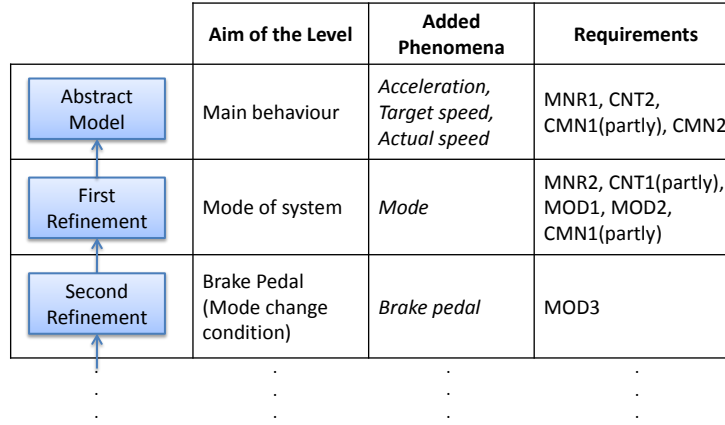


Figure 5.5: Formalising CCS using Refinement.

A model such as Figure 5.5 where the entire RD is formalised as a single refinement chain can have a high number of refinement steps. Therefore, in the next section we propose to *decompose* an RD based on the structured MCMC sections in order to formalise each section as a separate sub-model.

### 5.6.2 Formalising MCMC Sub-Problems as Sub-Models

One way of decomposing the complexity and the effort required for formalising a structured RD is to model a control system as *composable sub-models*. In this section we propose an approach for decomposing an RD into *sub-problems* which can be formalised independently. The decomposition of an RD is inspired by the PF approach [Jac01] which discusses that a problem can be divided into sub-problems each with a degree of unity.

Our proposed sub-problems are the *monitored* (MNR), *controlled* (CNT), *mode* (MOD) and *commanded* (CMN) sections of the structured RD. To formalise each sub-problem, its requirements are layered based on the refinement guidelines defined in the previous section. So, refinement techniques are used to formalise every sub-model gradually. This results in four sets of refinement chains each representing a distinct sub-model.

The number of refinement steps in each sub-model is less than the overall number of refinement steps, which means it is easier to determine the refinement order within every chain. Also, the number of proof obligations are decomposed amongst the sub-models. Therefore, the complexity of the modelling process becomes more manageable. A more

detailed explanation of the advantages of formalising an RD as sub-models is discussed in Chapter 7 after describing the case studies.

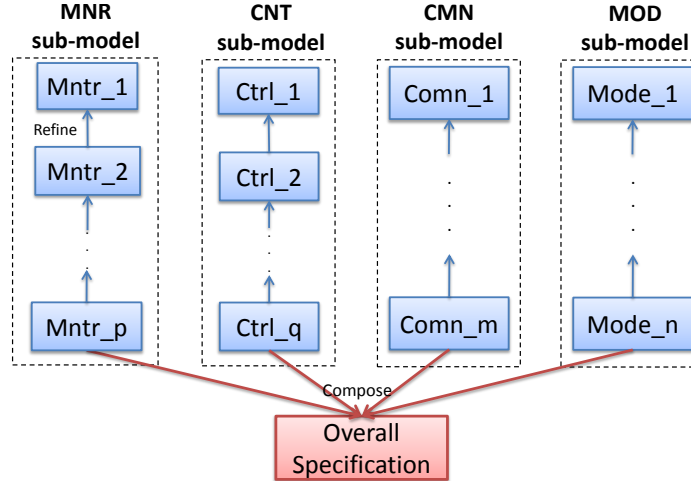


Figure 5.6: Formalising MCMC sub-problems as composable sub-models.

The composition of the most concrete refinement levels of each sub-models provides the *overall specification*. This is shown in Figure 5.6. It is possible that the sub-models of a system share phenomena, as requirements of a sub-problem may refer to phenomena in other sub-problems. Such shared phenomena can be formalised as *shared variables* and *shared events* amongst the sub-models. This means the cross-cutting invariants and the consistency of the sub-models can be proved.

Further discussions and examples of formalising sub-models with shared phenomena are provided in Chapter 6. In addition, in Chapter 6 the structure of machines in Event-B is extended in order to simplify the formalisation of shared phenomena.

## 5.7 Patterns for Formalising MCMC Variable and Event Phenomena in Event-B

While the approach of mapping sub-problems to composable sub-models can be applied to any state-based formal languages with refinement and composition mechanisms, in this work we chose the Event-B formal language. This language is mainly chosen because of the simplicity of its notation, its support for incremental refinement-based development and decomposition, and also because of the availability of a modelling tool with a range of plugins and provers.

The formalisation of the MCMC sub-problems in Event-B includes two sets of activities:

1. Formalising each sub-problem as a sub-model. This is discussed in this section.

2. Dealing with the phenomena shared amongst the sub-models in order to compose them. This is discussed in the next chapter where sub-models are reconciled based on shared-variable and shared-event decomposition styles.

Section 5.7.1 explains the formalisation of variable and event phenomena of sub-problems. Also, Sections 5.7.2 to 5.7.6 provide *event patterns* which guide a modeller in formalising monitor, control, mode and command event phenomena. These patterns provide a generalised means of formalising the MCMC events. The patterns represent recurring templates and can equip a modeller with an insight into the expected outcome of the formalisation process.

These patterns are defined based on the dependencies between the MCMC variable and event phenomena which were discussed earlier in Section 5.5. As well as a modelling guideline, the patterns can be used to *validate* formalised MCMC events.

Note that generally speaking we use the term ‘phenomena’ to refer to the informal (pre-formalisation) MCMC variables and events. The terms ‘variable’ and ‘event’ in this section refer to the *formal* representation of the variables and events.

### 5.7.1 Defining Variable and Event Phenomena

To formalise a structured RD the first step is to define the MCMC variable and event phenomena formally. The basis of the formalisation process is to model each variable (event) phenomenon as a formal variable (event) in Event-B. The formalisation of a variable phenomenon involves defining a *name* and a *type* for the variable. For instance the variable phenomena of actual speed and target speed are modelled as formal variables of *actualSpd* and *targetSpd*.

Each formal variable requires at least one invariant to define its type. The typing invariant is determined based on the system requirements. For example the *actualSpd* variable is always a value between 0 (when the car is stationary) and the maximum car speed. So, this variable is defined as  $actualSpd \in 0..MaxSpd$ .

To formalise an event, the event *name* and its *body* need to be defined. For instance events which update *actualSpd* and *targetSpd* are named *UpdateActualSpeed* and *Set-TargetSpeed*. To define the body of the *monitor*, *control*, *mode* and *command* events in Event-B, we have provided formal patterns in next sections.

Note that in addition to variables and events a model of an RD may require other elements, such as constants or sets. It is modeller’s judgment to introduce further elements to a model when required.

### 5.7.2 A Pattern for Monitor Events

The pattern for modelling a monitor event which updates a single monitored variable called  $MNR_i$  is shown in Figure 5.7. Here,  $x$  denotes some nondeterministically chosen values.  $G$  is a meta-guard which represents the conjunction of all guards in the event  $Update\_MNR_i$ . The sets and constants of the context seen by the machine are shown by  $s$  and  $c$ , while  $MNR_{sub}$  and  $CNT_{sub}$  represent subsets of all monitored and controlled variables (MNR and CNT) respectively.

$F$  shows that the variable  $MNR_i$  can be updated based on  $x$ ,  $s$ ,  $c$  as well as the sets  $MNR_{sub}$  and  $CNT_{sub}$ . Note that  $MNR_{sub}$  or  $CNT_{sub}$  can be empty sets. For instance the action of a monitor event can be  $MNR_i := F(MNR_{sub})$ .

```

Update_MNRi  $\triangleq$   any  $x$ 
                   when  $G(MNR_{sub}, CNT_{sub}, x, s, c)$ 
                   then  $MNR_i := F(MNR_{sub}, CNT_{sub}, x, s, c)$ 
                   end

```

Figure 5.7: The pattern of a monitor event updating  $MNR_i$ .

When both  $MNR_{sub}$  and  $CNT_{sub}$  are empty sets, the action of the monitor event is  $MNR_i := F(x, s, c)$ . This action means that a monitored variable can be set *nondeterministically* based on some parameters as well as some sets and constants. An example of this is shown in Figure 5.8 where the *UpdateActualSpeed* event is defined nondeterministically. This event models the requirement MNR1 of Table 5.5. Note that the difference in the style of Figures 5.7 and 5.5 is because the former represents the pattern, while the latter is an example of the pattern in a cruise control system. This difference of the style can be seen in the remainder of the chapter.

```

event UpdateActualSpeed
  any actSpd
  where
    @grd1 actSpd  $\in$  0.. $maxCarSpd$ 
  then
    @act1 actualSpd := actSpd
  end

```

Figure 5.8: A monitor event updates the actual speed in the CCS.

It is important to notice that formal models of monitor events should not have strong guards. Since these events update monitored variables which should not be restricted any stronger than represented by system assumptions which are captured in the MNR section of the RD. We usually formalise monitor events nondeterministically and delay defining stronger guards to refinement steps.

### 5.7.3 A Pattern for Mode Events

During the horizontal refinement, i.e., the formalisation of the requirements, the controller mode events (CME) and the operator mode events (OME) are both modelled in the same way. Therefore, one pattern for defining both mode events is provided in this section. This is because the difference between CME and OME emerges when the UI and sensors are added to the model. However, we postpone introducing such design details to future refinement steps (in vertical refinement - Chapter 8). Delaying the introduction of UI allows us to focus on system functionalities at horizontal refinement, and decide on design details later at vertical refinement levels.

The mode event pattern which updates the formal *mode* variable is shown by the event *Update\_Mode<sub>st</sub>* in Figure 5.9. This event updates the variable *mode* to a new state called *st*. In Figure 5.9, *x* denotes some nondeterministical values. *G* is a meta-guard which represents the conjunction of all guards of the event *Update\_Mode<sub>st</sub>*. The sets and constants are shown as *s* and *c* respectively. The sets *MNR<sub>sub</sub>*, *CNT<sub>sub</sub>* and *CMN<sub>sub</sub>* are subsets of all monitored, controlled and commanded variables (MNR, CNT and CMN) respectively.

*ModeSet* in *Update\_Mode<sub>st</sub>* is a subset of a set called STATES which represents all possible values of the *mode*. As an example, in the CCS, the set STATES is {*on*, *off*, *active*, *suspend*}, while *ModeSet* might be {*off*, *suspend*}, meaning that an event can occur when *mode* is either *off* or *suspend*. The variable *mode* is an element of the set STATES. The action in *Update\_Mode<sub>st</sub>* sets *mode* to an element of STATES which is represented as *st* (i.e.  $st \in STATES$ ).

```

Update_Modest  $\triangleq$   any x
                   when mode  $\in$  ModeSet
                      $\wedge$  G(MNRsub, CNTsub, mode, CMNsub, x, s, c)
                   then mode := st
                   end

```

Figure 5.9: The pattern of a mode event updating *mode*.

An example of a mode event that suspends the controller when the brake pedal is pressed is shown in Figure 5.10. This event captures the requirement MOD3 of Table 5.5. Here, *MNR<sub>sub</sub>* = {*brakePedal*}, while the *CNT<sub>sub</sub>* and the *CMN<sub>sub</sub>* are empty sets. Also, *mode* is changed from *active* to *suspend*.

```

event Suspend
  where
    @grd1 mode = active
    @grd2 brakePedal = TRUE
  then
    @act1 mode := suspend
  end

```

Figure 5.10: A mode event in the CCS.

#### 5.7.4 A Pattern for Command Events

Figure 5.11 represents the pattern for modelling a command event that updates a commanded variable named  $CMN_i$ . Similarly to the previous patterns,  $x$  denotes some nondeterministically chosen value.  $G$  represents the conjunction of all guards of  $Update\_CMN_i$ . The sets and constants of the context are shown as  $s$  and  $c$ .  $ModeSet$  is a subset of all possible values of the mode variable. The action of this event updates the  $CMN_i$  based on  $x$ ,  $s$ ,  $c$ ,  $mode$  as well as monitored and commanded variables ( $MNR_{sub}$  and  $CMN_{sub}$ ).

```

 $Update\_CMN_i \triangleq$  any  $x$ 
  when  $mode \in ModeSet$ 
     $\wedge G(MNR_{sub}, mode, CMN_{sub}, x, s, c)$ 
  then  $CMN_i := F(MNR_{sub}, mode, CMN_{sub}, x, s, c)$ 
  end

```

Figure 5.11: The pattern of a command event updating  $CMN_i$ .

Note that the means for requesting a command event to take place should be defined as a UI. We delay the introduction of a UI to future refinement steps (vertical refinement in Chapter 8). This means, at this level we focus on responses provided by the control system to operator requests.

An example of a command event in the CCS is shown in Figure 5.12 where the target speed is updated according to the requirements CMN1 and CMN2 in Table 5.5. The former requirement is captured by  $@act1$  (the target speed is set to the actual speed) and  $@grd1$  (the target speed can be set only if the controller is **on**). The latter is shown in  $@grd2$  which defines a specific range within the lower bound  $lb$  and upper bound  $ub$  for the actual speed. Here the  $MNR_{sub}$  is the set  $\{actualSpd\}$  and  $ModeSet$  is  $\{on\}$ , while the  $CMN_{sub}$  is an empty set.

```

event SetTargetSpeed
  where
    @grd1 mode = on
    @grd2 actualSpd ∈ lb..ub
  then
    @act1 targetSpd := actualSpd
end

```

Figure 5.12: A command event for setting the target speed in the CCS.

### 5.7.5 A Pattern for Control Events

Figure 5.13 shows the pattern of a control event which updates a controlled variable named  $CNT_i$ . As before,  $x$  denotes some nondeterministically chosen value.  $G$  represents the conjunction of all guards of the event  $Update\_CNT_i$ .  $ModeSet$  is a subset of all possible mode values. The action of this event updates the  $CNT_i$  variable based on all the variables as well as  $x$ ,  $s$  and  $c$ .

```

 $Update\_CNT_i \triangleq$  any  $x$ 
  when  $mode \in ModeSet$ 
     $\wedge G(MNR_{sub}, CNT_{sub}, mode, CMN_{sub}, x, s, c)$ 
  then  $CNT_i := F(MNR_{sub}, CNT_{sub}, mode, CMN_{sub}, x, s, c)$ 
  end

```

Figure 5.13: The pattern of a control event updating  $CNT_i$ .

An example of a control event that updates the acceleration variable and models the requirement CNT1 and CNT2 in Table 5.5 is shown in Figure 5.14. The action  $@act1$  shows that acceleration is a function of the actual speed and the target speed which are elements of the sets  $MNR_{sub}$  and  $CMN_{sub}$  respectively. We assume that this function is provided, which means for every possible tuple of actual speed and target speed, the value for the acceleration is given.

```

event UpdateAcceleration
  where
    @grd1 mode = ACTIVE
  then
    @act1 acceleration := Func(targetSpd→actualSpd)
end

```

Figure 5.14: A control event for updating the acceleration in the CCS.

### 5.7.6 Combining Patterns

To update multiple variables using a single event, it is possible to combine the MCMC event patterns. Combinations are suggested for *control*, *mode* and *command* events.

Multiple *monitor* events can be combined together to form a single monitored event which updates several monitored variable phenomena simultaneously. However, we avoid composing a monitor event with any other type of events, such as a mode event. This is because monitor events belong to the environment and the controller has no role in their occurrence. Whereas the controller has an active role in performing control, mode or command events.

Notice that combining patterns is useful when a control system is modelled as a single refinement chain rather than composeable sub-models. This is because when formalising an RD as sub-models, MCMC events will be defined in their corresponding sub-models, so patterns combinations are not required.

An example of a combined pattern is shown in Figure 5.15. Here, the command and mode event patterns (Figure 5.11 and 5.9) are combined to represent an event that updates a commanded variable ( $CMN_i$ ) and the *mode* variable simultaneously. In the combined pattern the guards of the original patterns are conjoined and their actions occur simultaneously. For instance in Figure 5.15, the guards of the command and the mode event patterns are conjoined and their actions occur simultaneously. In this figure  $x$  represents some nondeterministic values. The sets of the MNR, CNT and CMN variables and ModeSet which influence  $CMN_i$  are shown with the lower index of *sub1*, while *sub2* represents the sets that affect the value of *mode*.

$$Update\_CMN_i\_mode_{st} \triangleq$$

```

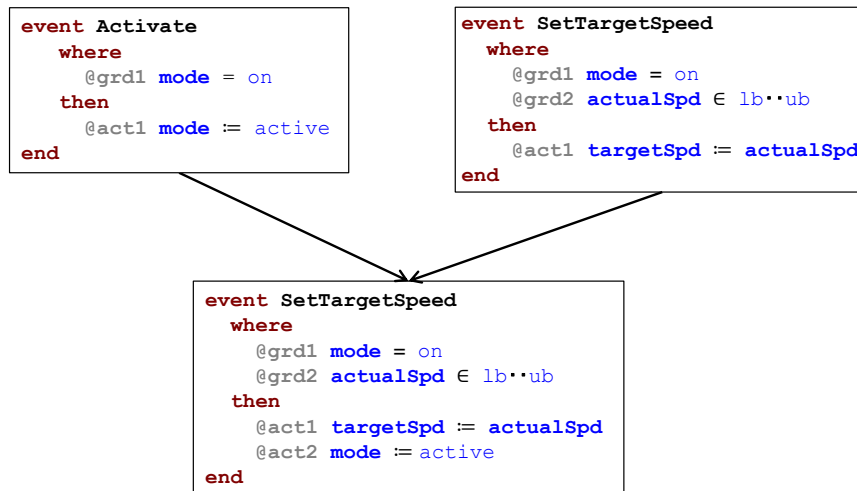
any  $x$ 
when  $mode \in ModeSet_{sub1}$ 
       $\wedge G1(MNR_{sub1}, mode, CMN_{sub1}, x, s, c)$ 
       $\wedge mode \in ModeSet_{sub2}$ 
       $\wedge G2(MNR_{sub2}, CNT_{sub2}, mode, CMN_{sub2}, x, s, c)$ 
then  $CMN_i := F(MNR_{sub1}, mode, CMN_{sub1}, x, s, c)$ 
       $\parallel mode := st$ 
end

```

Figure 5.15: Composition of command and mode event patterns.

Figure 5.16 provides an example of a combined event which updates the *mode* variable and the commanded variable *targetSpd*. This event is the combination of the mode event *Activate*, which models the requirement MOD2 in Table 5.5, and the command event *SetTargetSpeed* which models the requirement CMN1. The composition of these two events is an example of the pattern in Figure 5.15.



Figure 5.16: Composition of *Activate* and *SetTargetSpeed* events.

The combination of events should be sensible from the requirement document point of view. For instance, in the example of Figure 5.16, it might be unreasonable to combine *Suspend* and *SetTargetSpeed* events, as their requirements are not related to one another.

In addition, it is important to ensure that a combined event is reachable. In other words, the guards of a combined event should be enabled at some point. As an example the combination of two events with contradictory guards on a shared variable (e.g., guards  $0 < x$  and  $0 > x$ ) results in an unreachable event. Since contradictory guards can never be satisfied, they block the combined event from firing.

Reachability of multiple tasks have been considered by others [ACM05]. In task reachability, the execution of a task which may consist of one or more events, should not postpone executions of others infinitely. Such reachability problems can be solved by using variants [ACM05]. However, the reachability that we referred to here is local to a single event. This means to ensure that every event can be enabled at some point.

## 5.8 Stage 3: Validation and Revision of RD and Formal Model

This section explains the third stage, where a model is traced and validated against its RD. This process is discussed in Section 5.8.1 using the example of the CCS. Section 5.8.2 discusses the need for revising the RD, its model and the validation column, as new or ambiguous requirements might be identified.

### 5.8.1 Validation (Traceability) of Model against RD

Validation of the model against its RD can be done by adding a *validation column* to the right hand side of the tables of a structured requirement document. Every requirement is then validated by adding the elements of the model, such as events or variables, which capture the requirement to its validation column. If the formal representation is not self-explanatory, then adding justifications can be helpful, as the main purpose of this column is to justify that the formal elements of the model represent the requirement correctly.

Table 5.10 shows validation of some of the CCS requirements against their model. For instance, requirement CNT2 is modelled using the following elements:

1. the controlled variable *acceleration*;
2. the control event *UpdateAcceleration*;
3. the acceleration is set according to the received values of the actual speed and the target speed. This is shown in the action of *UpdateAcceleration* as  $acceleration := accFun(targetSpd \mapsto actualSpd)$ . The function  $accFun \in lb..ub \times 0..n \rightarrow \mathbb{Z}$  is defined to return the value of acceleration.

It also might be useful to add another column to the validation table which would refer to the sub-model and the refinement level in which the requirement was modelled. It is important to mention that the process of validation should take place at the end of every modelling step. This means as well as modelling, validation is a gradual and incremental process. Thus, if a requirement is modified, the model and the validation column both should be updated. Note that the entire RD is not always modelable, which means the validation column may only contain the reason for not modelling the requirement rather than the elements of the model.

Validating the model against the RD helps to ensure that the model is an accurate representation of the RD. In addition, validation helps with the traceability of requirements, since we can easily identify which parts of a model represent a specific requirement. Therefore, changes can be managed easier and quicker.

### 5.8.2 Revising an RD and its Formal Model

The stages of structuring RD, formal modelling and validation of the model against its RD can result in finding missing and ambiguous requirements of the system. These requirements can be handled by taking the requirement structuring steps. This means identifying the main phenomenon each requirement represents and then placing the requirement in an appropriate MCMC section.

Phen.	ID	Requirement Description	Validation Column
Actual speed	MNR1	CCS shall monitor the vehicle's actual speed.	Monitored variable: <i>actualSpd</i> ; Monitor event: <i>UpdateActualSpd</i>
Acceleration	CNT1	Once CCS is active, the speed regulation mechanism will be automatically invoked.	Guard (on control event <i>UpdateAcceleration</i> ): <i>mode = active</i> ; The invocation of speed regulation mechanism is not modelled.
	CNT2	When CCS is active, the speed regulation mechanism will maintain the difference between actual speed and target speed as close to 0 as possible by correcting the acceleration according to speed control laws.	Constant: <i>accFun</i> ; Axiom: $accFun \in lb..ub \times 0..n \rightarrow \mathbb{Z}$ ; Controlled variable: <i>acceleration</i> ; Control event: <i>UpdateAcceleration</i> ; Action: $acceleration := accFun(targetSpd \mapsto actualSpd)$
Mode	MOD1	CCS can be switched <b>on</b> or <b>off</b> by the driver.	Set: $STATE = \{on, off\}$ ; Mode variable: <i>mode</i> ; Operator mode events: <i>SwitchOn</i> & <i>SwitchOff</i>
	MOD2	When CCS is on, it will be <b>activated</b> as soon as the driver sets the target speed.	Mode Event (Command Event): <i>SetTargetSpeed</i> ; Action: <i>mode := active</i>
Target speed	CMN1	Once CCS is switched on, the driver can determine the target speed by setting it to the value of actual speed.	Command Event: <i>SetTargetSpeed</i> ; Guard: <i>mode = on</i> ; Action: <i>targetSpd := actualSpd</i>
	CMN2	The target speed is always within a specific range.	Constants: <i>lb</i> , <i>ub</i> ; Axiom: $ub > lb$ ; Invariant: $mode = active \Rightarrow targetSpd \in lb..ub$

Table 5.10: Validation of RD of CCS against its model.

After updating the structured RD, it is necessary to also revise the formal model. To do this we firstly identify the sub-model in which the requirement should be formalised, and secondly introduce the requirement either by modifying existing refinement levels to accommodate the new requirement or by modelling the requirement in a new level of the refinement.

For instance, if the newly identified requirement is about a controlled phenomenon, we take the CNT sub-model and check whether or not this phenomenon has already been modelled in one of the sub-model's refinement levels. If it has been modelled, it might be sufficient to modify that refinement level to accommodate the new requirement.

Alternatively, if the requirement represents a new system behaviour as a result of which a new phenomenon is identified, the requirement can be modelled in a new refinement level of the corresponding sub-model.

To avoid modifying the refinement chains significantly and to make the least amount of change to every refinement step, we suggest the introduction of the new requirements in a new refinement level or in a more concrete level within the corresponding sub-model. An example of this is discussed in the LDWS case study, in Section 7.4.2.

After the modelling of the requirement, it is necessary to validate the modified parts of the model against the RD. This will allow us to trace between elements of the model and the RD. This way the RD, its formal model and the validation column remain consistent.

## 5.9 Discussions and Conclusion

In this chapter we proposed a four-stage approach to facilitate the transition from an informal RD of a control system to a formal model. In the first stage, system requirements are structured into four sub-problems based on *monitored*, *controlled*, *mode* and *commanded* (MCMC) phenomena. Differences between the MCMC phenomena were explained in detail. In particular, commanded phenomena, which are determined by operators and have no representations outside a controller, were introduced.

In the second stage, we proposed to formalise each sub-problem independently. The formalisation starts with modelling variables and events. We proposed patterns for modelling MCMC events in the Event-B language. Also, we provided refinement guidelines which can be used to structure a refinement chain for every sub-model.

The third stage involves validating a formal model against its structured RD. This stage can be used for traceability between a model and its requirements. Finally, the fourth stage involves composing the sub-model in order to obtain the overall specification. Further discussion on the formalisation of sub-problems and composition of sub-models is provided in the next chapter.

This four-stage approach can be applied when any state-based formal language that supports refinement and composition techniques is used. In this research we applied the MCMC approach to the Event-B language and we provided patterns and guidelines for modellers who wish to use this language.

The MCMC approach and the proposed patterns and guidelines are the result of experimenting with various control systems case studies, some of which are supported by industrial partners. Some of the case studies that have been modelled by following an early version of the MCMC approach are a FADEC system [Das10] and a sluice gate [PS12], which were conducted as MSc projects by students, and others, such as a

cruise control system (CCS) [YBR10], a lane departure warning system (LDWS) [YB11], and a lane centering controller (LCC) [YB12], as part of this work by us.

The MCMC approach was also used to model a washing machine control system in Event-B [Col12]. The feedback we received from this work was that the identification of the MCMC phenomena and structuring the requirements helped greatly with identifying and organising the RD. In addition, a safety analysis approach based on the MCMC approach has been developed [Col12].

Our experience as well as others with control systems have demonstrated that clear identification of the MCMC phenomena and structuring of requirements based on these phenomena can facilitate formalisation. Furthermore, the requirement structuring approach can help to improve system understanding.

## Chapter 6

# Composition of Sub-Models

The fourth stage of the MCMC four-stage approach which is *composition of the sub-models* is explained in this chapter. An overview of the stage is provided in Section 6.1.

Section 6.2 discusses how to revise and reconcile sub-models to formalise phenomena shared amongst them in Event-B. Sections 6.2.1, 6.2.2 and 6.2.3 provide approaches for reconciling shared phenomena based on the *shared-variable* and the *shared-event* composition styles. These two reconciliation approaches are evaluated in Section 6.3 using a simple example. Although sub-models of any RD may be composed based on the proposed approaches, in this thesis we focus on sub-models of a control system which are formalised based on the MCMC guidelines. In Section 6.4, the case study of the CCS is formalised as MCMC sub-models which are then revised in order to reconcile and formalise their shared phenomena.

In Section 6.5, we propose to extend the structure of a machine in Event-B in order to simplify the composition of sub-models which share phenomena. The extended machine structure can be used to formalise any sub-models that share variables or events. However, our ultimate goal is to develop guidelines for modelling control systems. Therefore, in Section 6.6 we define schemas for modelling monitored, controlled, mode and commanded (MCMC) sub-models in Event-B based on the extended machine structure. Also, in this section the CCS is formalised based on the MCMC schemas. Section 6.7 provides the conclusion of this chapter.

### 6.1 Overview of Composing Sub-Problems

The fourth stage of the MCMC four-stage approach is shown in Figure 6.1. This stage is to compose sub-models which provide formal representations of the MCMC sub-problems.

It is possible for sub-models to share phenomena. In this chapter, we discuss how to revise formal sub-models in order to accommodate shared phenomena. To do this we define reconciliation techniques based on the composition styles in Event-B, namely the *shared-variable* and the *shared-event* composition. Note that these composition styles were originally developed for *model decomposition*. Here, we adapt them to make them suitable for developing a system as composable sub-models.

Sub-models reconciliation ensures the shared phenomena defined in several sub-models are consistent and cross-cutting invariants are preserved. Note that reconciliation based on the *shared-variable* and the *shared-event* styles can be used in order to deal with phenomena shared amongst any sub-models which are composable. However, the examples provided in this thesis represent reconciliation and composition of the MCMC sub-models, since our ultimate goal is to provide guidelines for formalising a control system.

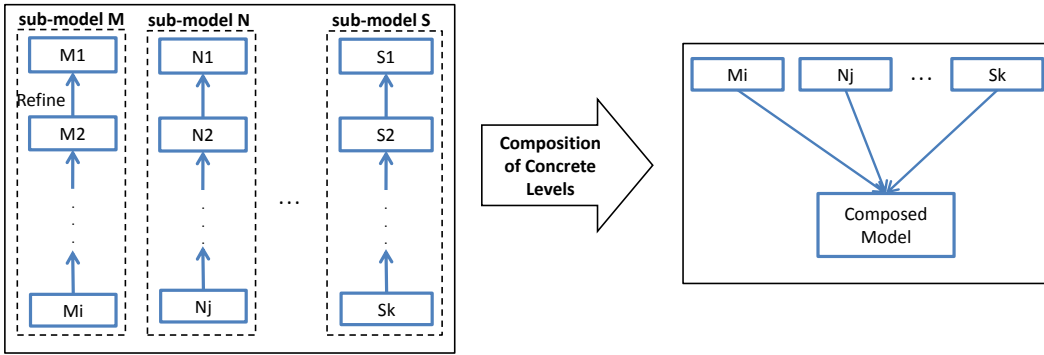


Figure 6.1: Overview of the composition of sub-models (Stage 4 of the MCMC approach).

## 6.2 Shared Phenomena of Sub-Models in Event-B

Sections 6.2.1 and 6.2.2 discuss the reconciliation based on the *shared-variable* and the *shared-event* composition styles respectively [YB13]. Section 6.3 evaluates these two styles for the reconciliation of shared phenomena. The reconciliation techniques are applied in Section 6.4 to the example of the CCS.

### 6.2.1 Reconciliation based on Shared-Variable Composition

Shared-variable composition [YB13] is based on the decomposition style of [AH07] that was explained in Section 4.4.1. This composition is shown in the example of Figure 6.2, where M is a composition of the machines M1 and M2. M1 consists of the events E1 and E2 whose dependencies to the variables  $v1$  and  $v2$  are shown using links. Similarly,

E3 and E4 in M2 depend on  $v2$  and  $v3$ . While  $v1$  and  $v3$  are *local* (internal) to M1 and M2 respectively,  $v2$  is *shared* among them.

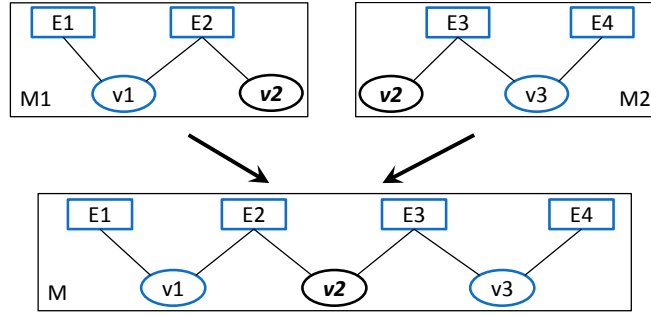


Figure 6.2: Shared-variable composition style.

We can treat M1 and M2 as composable sub-models that share the variable phenomenon  $v2$ . To ensure that the behaviours on  $v2$  in M1 and M2 are consistent and also the cross-cutting invariants are preserved the sub-models can be reconciled. To do this, *external events* which simulate the behaviour on the *shared variable phenomena* are added to the sub-models. Introducing external events gives rise to proof obligations (POs), such as the invariant preservation PO, which ensure the correctness of the sub-models. Thus, after adding external events the sub-models which were independent of each other, may require modifications to preserve the invariants.

Figure 6.3 shows the external events  $E2'$  and  $E3'$ .  $E3'$  ( $E2'$ ) of M1 (M2) simulates the way in which its environment, i.e. M2 (M1), may modify the shared variable  $v2$ .

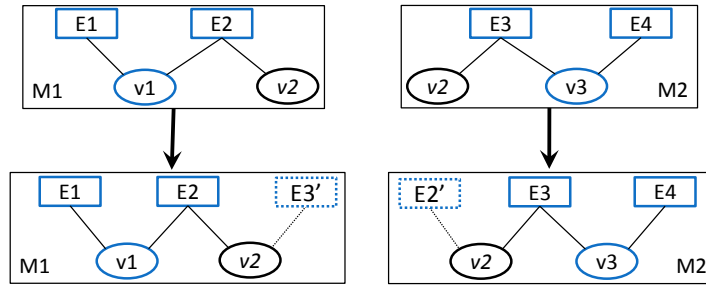


Figure 6.3: Reconciliation of M1 and M2 based on the shared-variable style.

Assume  $E3$  has the following form:

$$E3 \triangleq \text{Any } p \text{ Where } G(v2, p) \wedge H(v3, p) \\ \text{Then } S(v2, p) \parallel T(v3, p)$$

$E3'$  can be constructed from  $E3$  by projecting out the non-shared variable  $v3$  resulting in the following:

$$E3' \triangleq \text{Any } p \text{ Where } G(v2, p) \text{ Then } S(v2, p) \quad (6.1)$$



Note that the definition provided for  $E3$  is not the most general form for events in Event-B. It is possible for the variables  $v2$  and  $v3$  to depend on one another, e.g.  $S(v2, v3, p)$ . Such actions can be simplified by defining extra parameters.

External events cannot be refined, as refining an external event can break its consistency with other sub-models. However, refining internal events is allowed.

The nature of this style allows the reconciliation of *shared variable phenomena*. As will be discussed later, shared event phenomena cannot be reconciled in this style.

### 6.2.2 Reconciliation based on Shared-Event Composition

The shared-event composition style was discussed in Section 4.5. We briefly remind the reader of this composition approach using the Figure 6.4 which shows the composition of  $M1$  and  $M2$  with the shared event  $E2$ . The event  $E2a$  ( $E2b$ ) in  $M1$  ( $M2$ ) represents parts of  $E2$  which refer to the variable  $v1$  ( $v2$ ). The variable  $v1$  and the event  $E1$  are *local* (internal) to  $M1$ . Similarly,  $v2$ ,  $v3$  and  $E3$  are local to  $M2$ . So,  $M1$  and  $M2$  can be treated as sub-models which share the event phenomenon  $E2$ .

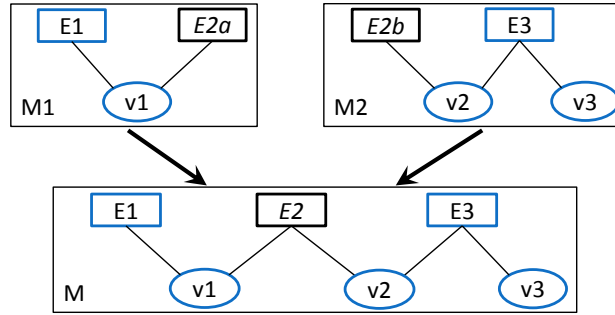


Figure 6.4: Shared-event composition style.

Definition 6.2 below shows that two events can be composed by conjoining their guards and combining their actions so they happen simultaneously. It is possible for guards and actions with disjoint variables to have common parameters. In Definition 6.2, the events  $E2a$  and  $E2b$  share the parameter  $p$ .

$$\begin{aligned}
 E2a &\triangleq \text{Any } p \text{ Where } G(v1, p) \text{ Then } S(v1, p) \\
 E2b &\triangleq \text{Any } p \text{ Where } H(v2, p) \text{ Then } T(v2, p) \\
 E2 &\triangleq \text{Any } p \text{ Where } G(v1, p) \wedge H(v2, p) \\
 &\quad \text{Then } S(v1, p) \parallel T(v2, p)
 \end{aligned} \tag{6.2}$$

### 6.2.3 Introducing Shared Variables in Shared-Event Reconciliation

Using the shared-event composition style means that the sub-models cannot explicitly share variable phenomena. However, sub-models may share variable phenomena as well as event phenomena [YB13]. As an example consider a requirement such as “ $v1$  shall be increased when  $v3$  is greater than 0” for Figure 6.4 where  $v1$  and  $v3$  reside in two different sub-problems (M1 and M2 respectively). This requirement belongs to the M1 sub-problem, since it defines a condition under which  $v1$  can be modified. This means  $v3$  is a phenomena shared amongst M1 and M2. Note that our aim is to model every requirement in a single sub-model. In other words, we avoid modelling the part of the requirement which refers to  $v1$  in M1 and the rest in M2.

To model such requirements using shared-event reconciliation, variable sharing in addition to event sharing is required. This is shown in Figure 6.5. Here, the variable  $v3$  which is shared amongst M1 and M2 is duplicated in M1 as  $v3'$ . Also, the event  $E3'$  is defined in M1 to simulate the behaviour on  $v3$  in M2. To some extent this event is similar to an *external event* in the shared-variable reconciliation. Thus, it can be defined similarly to Definition 6.1. However here, events on shared phenomena synchronise, which means they are *shared* amongst the sub-models. For instance in Figure 6.5,  $E3$  and  $E3'$  are shared events.

In this reconciliation, shared variables can be modified only in one sub-model. In Figure 6.5,  $v3$  can be modified by events in M2, whereas E1 can only read the value of  $v3$ . In other words, a single sub-model has read and write access to the shared variable, while other sub-models have a read-only access. This is shown by the dependency link between  $E1$  and  $v3'$  which is named *rd*. So, in this style, sub-models can have *shared event phenomena* as well as *read-only shared variable phenomena*.

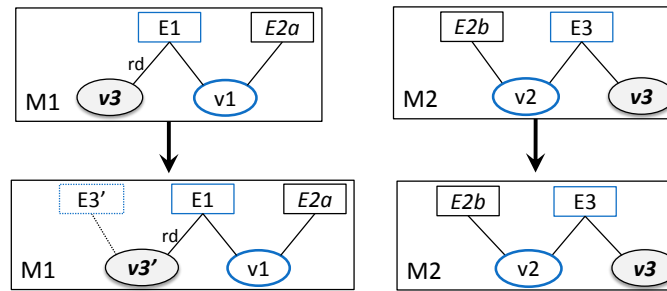


Figure 6.5: Shared-event reconciliation with shared variable  $v3$ .

As discussed a copy of  $v3$ , called  $v3'$ , is defined in M1. Shared variables have to be renamed as the current semantics of the shared-event style does not allow the introduction of variables with the same name in different sub-models. This semantic approach emphasises that variables are *local* to a sub-model. To prove the correctness of invariants of a sub-model with regards to all possible values for a shared phenomenon and to prove the consistency between the sub-models, invariants which equalise shared variables are

defined in the composed model, e.g., the invariant  $v\beta = v\beta'$ . Such invariants give rise to consistency and correctness related POs.

### 6.3 Evaluation and Discussion on Reconciliation

The advantages and disadvantages of the shared-variable and the shared-event reconciliation techniques are respectively discussed in Section 6.3.1 and 6.3.2 [YB13]. These sections evaluate the suitability of the two reconciliation techniques in designing a control system in Event-B. This evaluation is explained using an example which is described below.

Let us consider a requirement on a controlled phenomenon  $x$  and the *mode* phenomenon that states “ $x$  should be set to 0 as soon as the *mode* becomes **off**”. This requirement belongs to the CNT sub-problem, as it describes the modification of the controlled phenomenon  $x$ . So, we call this requirement  $CNT_x1$ . There also exists a requirement in the MOD sub-problem that complements  $CNT_x1$ . This requirement, which is called  $MOD_x1$ , states that “*mode* becomes **off**” under certain conditions which are not of interest for the purpose of the evaluation.

To formalise these two requirements in *a single model* rather than two distinct sub-models, two design decisions can be made. Firstly, the two requirements can be modelled *concurrently* by defining an atomic event with two actions which update the  $x$  and the *mode* variables simultaneously. This event which is shown in Figure 6.6(a), is a combination of a mode and a control event that capture the requirement  $MOD_x1$  and  $CNT_x1$  respectively.

The second design decision is to define these requirements *sequentially* by introducing each requirement in a separate event. We refer to this design decision as sequential, as the sequence in which the events take place affects their enabledness. As shown in Figure 6.6(b), there exists two distinct mode and control events which are called *UpdateMode* and *UpdateX* respectively. The second part of the requirement CNT1 which is about mode (“as soon as the *mode* becomes **off**”), is modelled as a guard in *UpdateX*. This guard represents that *UpdateX* can occur at some point after the occurrence of *UpdateMode*, since the latter event sets the *mode* to **off**.

In the remainder of this section, we will show that shared-variable reconciliation style always forces the second design decision (Figure 6.6(b)), while the shared-event reconciliation style allows both ways of modelling. However, this costs overhead in shared-event reconciliation. In Section 6.5 we propose an extension to the shared-event composition style which allows the freedom in the design decision, while the overheads are lessened.

<pre> event UpdateMode_X where   @grd1 mode ≠ Off   @grd2 x ≠ 0 then   @act1 mode := Off   @act2 x := 0 end </pre>	<pre> event UpdateMode where   @grd1 mode ≠ Off then   @act1 mode := Off end </pre>	<pre> event UpdateX where   @grd1 mode = Off   @grd2 x ≠ 0 then   @act1 x := 0 end </pre>
--	---	---

(a) Combined mode and control events. (b) Mode and control events are defined separately.

Figure 6.6: Two design decisions for modelling requirements in a single model.

### 6.3.1 Evaluation of the Shared-Variable Style Reconciliation

The main disadvantage of the shared-variable reconciliation is that sub-models cannot share *event phenomena*. This means events of different sub-models cannot contribute to the specification of a requirement, since shared phenomena can be reconciled only as *variables*. When modelling the requirements  $CNT_x1$  and  $MOD_x1$  in the controlled (CNT) and the mode (MOD) sub-models respectively, their shared phenomenon is the *mode* variable.

Figure 6.7 shows the model of the requirements  $CNT_x1$  and  $MOD_x1$  in their sub-models after the reconciliation. Here, behaviours on the *mode* are captured as events of the MOD sub-model. Thus, the requirement  $MOD_x1$  is modelled as the event *Off*. However, the variable  $x$  and consequently the requirement  $CNT_x1$  are modelled in the CNT sub-model. The event *UpdateX* in the CNT sub-model represents  $CNT_x1$ . In addition to this event, an external event, called *UpdateMode*, which is an abstraction of mode events in MOD is defined. This event simulates the variable *mode* in the CNT sub-model.

MOD Sub-Model	CNT Sub-Model	
<pre> event Off where   @grd1 mode ≠ Off then   @act1 mode := Off end </pre>	<pre> event UpdateMode any st where   @grd1 st ∈ STATUS then   @act1 mode := st end </pre>	<pre> event UpdateX where   @grd1 mode = Off   @grd2 x ≠ 0 then   @act1 x := 0 end </pre>

Figure 6.7: Evaluation of shared-variable style reconciliation.

The style of modelling of the CNT sub-model in Figure 6.7 is similar to the sequential design decision which was shown in Figure 6.6(b). We were forced by the shared-variable reconciliation to model the requirement  $CNT_x1$  in a sequential style, since shared phenomena can be modelled as *shared variables* only. Therefore, this reconciliation imposes a design decision. To avoid this restriction sub-models should be allowed to share events as well as variables. This is similar to the shared-event style which is evaluated next.

### 6.3.2 Evaluation of the Shared-Event Style Reconciliation

The main advantage of using the shared-event style is that sub-models can share *event* and *variable phenomena*. However, variable sharing in the shared-event reconciliation style is more restrictive than shared-variable style, since the latter allows all sub-models to read and write to shared variables, while in the former one sub-model has read-write access to a shared variable and others have read-only access.

Figure 6.8 shows that the requirement  $CNT_x1$  can be modelled by defining an event in the CNT sub-model which synchronises with a mode event in MOD that captures the requirement  $MOD_x1$ . Thus, these requirements can be modelled as a composition of the mode event *Off* and the control event *UpdateX*, i.e.  $MOD.Off \parallel CNT.UpdateX$ .

The composition of these events means that  $CNT_x1$  and  $MOD_x1$  are modelled concurrently. This is similar to Figure 6.6(a) where the concurrent design decision was taken. However, Figure 6.6(a) represented the formalisation of requirements in a single model, while in Figure 6.8 requirements are modelled in separate sub-models. Because the semantics of the shared-event style does not allow the introduction of variables with the same name in different sub-models, the variable *mode* in the CNT sub-model is called *cnt\_mode*. To ensure that the CNT and MOD sub-models are consistent the invariant  $mode = cnt\_mode$  will be added when CNT and MOD are composed.

MOD Sub-Model	CNT Sub-Model
<pre> <b>event</b> Off   <b>where</b>     @grd1 mode ≠ Off   <b>then</b>     @act1 mode := Off   <b>end</b> </pre>	<pre> <b>event</b> UpdateX   <b>where</b>     @grd1 cnt_mode ≠ Off     @grd2 x ≠ 0   <b>then</b>     @act1 x := 0   <b>end</b> </pre>

Figure 6.8: Shared-event style reconciliation when the requirements are defined concurrently.

As discussed, Figure 6.8 represents the formalisation of the requirements  $CNT_x1$  and  $MOD_x1$  in a concurrent design style. We can also model these requirements sequentially, similarly to Figure 6.6(b). This is shown in Figure 6.9. The sequential design requires reconciling the variable *mode* which is shared between the MOD and CNT sub-models.

Thus, a duplication of the *mode* variable, called *cnt\_mode*, is defined in the CNT sub-model. After adding this shared variable, events which simulate behaviours on the shared phenomenon should be defined explicitly in CNT. As Figure 6.9 shows the event *UpdateMode* in CNT simulates the behaviour on *mode* in MOD. This event synchronises with its concrete representations in MOD, the event *Off*. This synchronisation happens through the parameter *st*. The guard @grd2 in the event *Off* ensures that the right value of *mode* will be passed to *UpdateMode*. Note that to prove the consistencies between

the sub-models the invariant  $mode = cnt\_mode$  will be introduced to the composition of the CNT and MOD sub-models.

MOD Sub-Model	CNT Sub-Model	
<pre> event Off any st where   @grd1 mode ≠ Off   @grd2 st = Off then   @act1 mode := Off end </pre>	<pre> event UpdateMode any st where   @grd1 st ∈ STATUS then   @act1 cnt_mode := st end </pre>	<pre> event UpdateX where   @grd1 cnt_mode = Off   @grd2 x ≠ 0 then   @act1 x := 0 end </pre>

Figure 6.9: Shared-event style reconciliation of  $mode$  when the requirements are defined sequentially.

Defining the extra events in this reconciliation is a drawback, as it can impose an overhead to the modelling process. Another disadvantage of this style is that because of the restrictions of its semantics, shared variable phenomena should be renamed in different sub-models. This means to prove the correctness and consistency of the sub-models with respect to the shared variables, invariants which equalise the renamed shared variables should be defined in the composed model. In Section 6.5, we propose an extension to the shared-event composition which simplifies the shared-event reconciliation.

## 6.4 An Example: Sub-Models and Reconciliation of CCS

In Section 6.4.1 the example of the CCS which was discussed in Table 5.5 of Chapter 5 is formalised using the monitored, controlled, mode and commanded (MCMC) sub-models. These sub-models are formalised using refinement guidelines of Section 5.6.1.

The variables shared among the MCMC sub-models of the CCS are represented in Section 6.4.2. Sections 6.4.3 and 6.4.4 represent examples of the revision and reconciliation of the sub-models based on the shared-variable and the shared-event styles respectively. Not all shared phenomena of the CCS are discussed here. Other shared phenomena can be added to sub-models by revising and reconciling them similarly.

### 6.4.1 MCMC Sub-Models in CCS

The sub-models of the CCS are explained in this section. Sections 6.4.1.1 to 6.4.1.4 represent monitored, mode, commanded and controlled sub-models of the CCS.

#### 6.4.1.1 Monitored Sub-Model of the CCS

The MNR sub-model is shown in Figure 6.10. This figure also represents the phenomena and the requirement ID numbers which are introduced at every refinement level. As

illustrated, the monitored variable *actual speed* is modelled in the most abstract level, i.e. *Mnrt\_1*. After that, the variable *brake pedal* is modelled in *Mnrt\_2*. Since in this work we do not consider the behaviour of the environment in great detail, it is sufficient to define monitor events nondeterministically. Monitor events can be elaborated in future refinement levels if necessary.

The refinement steps are organised according to the guidelines of Section 5.6.1. Based on Rule 1 we minimised the number of phenomena chosen for each refinement step. As shown, one variable phenomenon is modelled at every level. However, the refinement levels can be arranged in any order (e.g. brake pedal can be formalised before actual speed), since these variable phenomena are independent of each other.

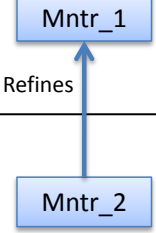
MNR Sub-Model	Phen.	Req ID	Examples of Event-B Model
	<ul style="list-style-type: none"> <li>Actual Speed (local)</li> </ul>	MNR1	<pre> event UpdateActualSpeed any actSpd where   @grd1 actSpd ∈ 0..maxCarSpd then   @act1 actualSpd := actSpd end </pre>
	<ul style="list-style-type: none"> <li>Brake Pedal (local)</li> </ul>	MNR2	<pre> event UpdateBrake any bp where   @grd1 bp ∈ BOOL then   @act1 brakePedal := bp end </pre>

Figure 6.10: MNR sub-model in CCS.

#### 6.4.1.2 Mode Sub-Model of the CCS

Figure 6.11 shows parts of the refinement chain of the MOD sub-model. To arrange the refinement steps the guidelines of Section 5.6.1 are used. Based on the Rule 3 the main phenomenon of the MOD sub-problem which is *mode*, is modelled at the most abstract level, *Mode\_1*. So, the transitions between values of the *mode* variable are modelled in *Mode\_1*. For instance, the *Suspend* event at this level simply shows that the CCS can be suspended from any other state.

Also based on Rule 2, the conditions for modifying mode (guards on mode events) are modelled after defining transitions (actions). For instance, the CCS should be suspended when brake pedal is pressed. This is shown in the guard *brakePedal = TRUE* in *Mode\_2*.

#### 6.4.1.3 Commanded Sub-Model of the CCS

Figure 6.12 represents the CMN sub-model where the requirements of the commanded sub-problem are formalised. Since *target speed* is the main phenomenon of this sub-problem, it is modelled in the most abstract level, i.e. *Cmnd\_1*. At this level the

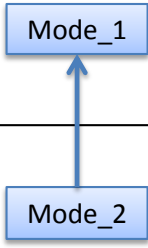
MOD Sub-Model	Phen.	Req ID	Examples of Event-B Model
	<ul style="list-style-type: none"> <li>• Mode (local)</li> </ul>	Abstract transition	<pre> <b>event</b> Suspend <b>where</b>   @grd1 mode ≠ SUSPEND <b>then</b>   @act1 mode := SUSPEND <b>end</b> </pre>
	<ul style="list-style-type: none"> <li>• Brake Pedal (shared)</li> </ul>	MOD3	<pre> <b>event</b> Suspend <b>refines</b> Suspend <b>where</b>   @grd1 mode = ACTIVE   @grd2 brakePedal = TRUE <b>then</b>   @act1 mode := SUSPEND <b>end</b> </pre>

Figure 6.11: MOD sub-model in CCS.

requirement CMN1 is modelled by defining the variable *targetSpd*. This variable should be updated to the value of the actual speed which is modelled in the MNR sub-model. Therefore, the shared variable *actualSpd* is introduced in *Cmnd\_1*. The variable *mode* is defined in *Cmnd\_2* to show that the event *SetTargetSpeed* can happen only when the CCS is on.

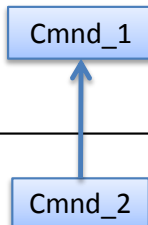
CMN sub-model	Phen.	Req ID	Examples of Event-B Model
	<ul style="list-style-type: none"> <li>• Target Speed (local)</li> <li>• Actual Speed (shared)</li> </ul>	CMN1, CMN2	<pre> <b>event</b> SetTargetSpeed <b>where</b>   @grd1 actualSpd ∈ lb..ub <b>then</b>   @act1 targetSpd := actualSpd <b>end</b> </pre>
	<ul style="list-style-type: none"> <li>• Mode (shared)</li> </ul>	CMN1	<pre> <b>event</b> SetTargetSpeed <b>extends</b> SetTargetSpeed <b>where</b>   @grd2 mode = ON <b>end</b> </pre>

Figure 6.12: CMN sub-model in CCS.

#### 6.4.1.4 Controlled Sub-Model of the CCS

The CNT sub-model of the CCS is shown in Figure 6.13. Based on Rule 3 of the refinement guidelines, the main phenomenon of the CNT sub-problems which is *acceleration* is modelled in the abstract level. However, the value of *acceleration* depends on the *actual speed* and the *target speed* which are modelled in MNR and CMN sub-models respectively. Therefore, based on Rule 1 in addition to acceleration, the shared phenomena *actual speed* and *target speed* are modelled in *Cntrl\_1*. The remaining requirements of the CNT sub-problem are introduced to the model gradually in refinement steps. The *mode* phenomenon, which is shared amongst MOD and CNT sub-models, is modelled in *Cntrl\_2*. The reconciliation step for defining *mode* is discussed in the remainder.



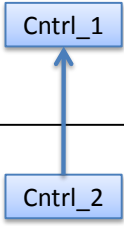
CNT Sub-Model	Phen.	Req ID	Examples of Event-B Model
	<ul style="list-style-type: none"> <li>• Acceleration (local)</li> <li>• Target speed (Shared)</li> <li>• Actual Speed (Shared)</li> </ul>	CNT2	<pre> event UpdateAcceleration then   @act1 acceleration :=     accFun(targetSpd ↦ actualSpd) end </pre>
	<ul style="list-style-type: none"> <li>• Mode (Shared)</li> </ul>	CNT1, CNT2	<pre> event UpdateAcceleration extends UpdateAcceleration where   @grd1 mode = ACTIVE end </pre>

Figure 6.13: CNT sub-model in CCS.

#### 6.4.2 Variable Phenomena Shared amongst Sub-Models of CCS

The variable phenomena shared amongst the MCMC sub-models of the CCS are shown in Table 6.1. Note that sub-models share variables with other sub-models, which means the main diagonal of the table has no entries. Also, to avoid redundancy, it is sufficient to fill in the cells above (or below) the main diagonal.

The entries in Table 6.1 are decided based on the sub-problems of the CCS which were discussed in Table 5.5. As an example, the requirement CNT1 (CCS operates when it is *active*) denote that the CNT and MOD sub-problems share the *mode* variable phenomenon. This is because the *mode* phenomenon belongs to the MOD sub-problem, however, CNT1 and CNT2 refer to *mode*. Also, the requirement CMN1 (once CCS is *on*, the driver can set the target speed) shows that *mode* is shared amongst the CMN and MOD sub-problems. Thus, the sub-models of the CCS need to be reconciled to deal with the shared variable *mode*. Similarly, the four sub-models are reconciled to model other shared phenomena, such as *brake pedal* which is shared between MNR and MOD sub-problems. However, they are not discussed here.

Sub-Models	MNR	CNT	MOD	CMN
MNR	-	Actual speed	Brake pedal	Actual speed
CNT	-	-	Mode	Target speed
MOD	-	-	-	Mode
CMN	-	-	-	-

Table 6.1: Shared variables of MCMC sub-models.

#### 6.4.3 Shared-Variable Style Reconciliation for CCS

In this section the sub-models are revised and reconciled using the shared-variable style in order to manage the shared phenomena. To do this, events which update shared

variables need to be simulated as *external events*. As an example, Figure 6.14 illustrates that the mode events which modify the shared variable *mode* are modelled in detail in the MOD sub-model. However, an abstraction of these mode events is defined as an external event in the CMN and CNT sub-models. The guards of the mode events are eliminated in order to construct a simple abstraction. The external event *UpdateMode* will be added to the refinement levels *Cmnd\_2* and *Cntrl\_2* shown in Figures 6.12 and 6.13 respectively. Revising the CMN and CNT sub-models to include the external event means that a representation of the shared variable will be added to them. Other shared variables can be reconciled in a similar manner.

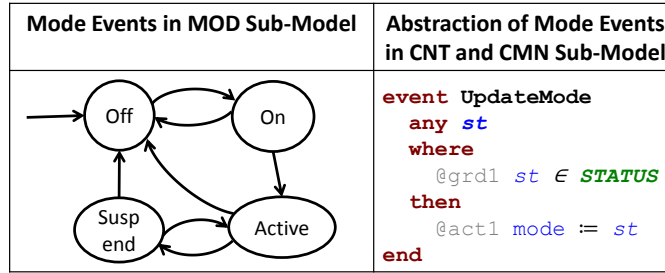


Figure 6.14: CNT sub-model includes an abstraction of the mode events.

#### 6.4.4 Shared-Event Style Reconciliation for the CCS

Using the shared-event style in reconciliation means that sub-models can share *variable* or *event* phenomena. In this section the sub-models of the CCS are revised based on shared-event reconciliation in order to handle their shared phenomena. The shared variable phenomenon *mode* is discussed in Section 6.4.4.1. After that Section 6.4.4.2 represents an event which is shared between the MOD and the CMN sub-models.

##### 6.4.4.1 Reconciling Shared Variable

To reconcile the shared variable *mode*, a copy of this variable is introduced in sub-models which need to include this variable. The current semantics of shared-event style does not allow variable with the same name to be introduced in different sub-models, therefore, this variable is renamed in other sub-models. In this section we consider the mode variable in the CNT sub-model. This variable is named *cnt\_mode*, while its counterpart in the MOD sub-model is named *mode*.

An abstraction of the mode events in MOD is added to CNT in order to update *cnt\_mode*. This is represented in Figure 6.15 as the event *UpdateMode*. As mentioned before, this event can be defined in a similar way to an external event in the shared-variable style. The *UpdateMode* event synchronises with the mode events in MOD. For instance, Figure 6.15 illustrates that *Suspend* in MOD and *UpdateMode* in CNT synchronise through the parameter *st*.

The consistency of the shared variable can be proved by adding the invariant  $mode = cnt\_mode$  to the composed model. Adding this invariant also means that in the composed model the variable  $cnt\_mode$  can be replaced by  $mode$ .

MOD Sub-Model	CNT Sub-Model
<pre> event Suspend refines Suspend any st where   @grd1 mode = ACTIVE   @grd2 brakePedal = TRUE   @grd3 st = SUSPEND then   @act1 mode := SUSPEND end </pre>	<pre> event UpdateMode any st where   @grd1 st ∈ STATUS then   @act1 cnt_mode := st end  event UpdateAcceleration extends UpdateAcceleration when   @grd1 cnt_mode = ACTIVE end </pre>

Figure 6.15: Mode is captured as a shared variable phenomenon in *UpdateAcceleration* using the shared-event reconciliation.

#### 6.4.4.2 Reconciling Shared Event

Shared-event reconciliation allows sub-models to share event phenomena. Therefore, the requirements CMN1 (Once the CCS is switched on, the driver can set the target speed to the actual speed) and MOD2 (When CCS is on, it will be **activated** as soon as the driver sets the target speed) can be modelled as shared events whose composition will provide the model of both requirements. This means requirements of different sub-models can contribute to specifications of a single event.

As a reminder, the formal representations of the requirements CMN1 and MOD2 are respectively given as the *SetTargetSpeed* and *Activate* events in Figure 6.16. This figure represents the concrete events, i.e. the event *SetTargetSpeed* in *Cmnd\_2* and the event *Activate* in *Mode\_2*. Thus, some guards are not shown here. The composition of *SetTargetSpeed* and *Activate* will result in an event that updates the commanded variable *targetSpd* and the *mode* variable simultaneously.

MOD Sub-Model	CMN Sub-Model
<pre> event Activate refines Activate where   @grd1 mode = ON   @grd2 ... then   @act1 mode := ACTIVE end </pre>	<pre> event SetTargetSpeed refines SetTargetSpeed where   @grd1 actualSpd ∈ 1b..ub   @grd2 mode = ON then   @act1 targetSpd := actualSpd end </pre>

Figure 6.16: *Activate* and *SetTargetSpeed* in their corresponding sub-models.

However, before composing the MOD and the CMN sub-models, *mode* needs to be modelled in CMN. This is necessary because the requirement CMN1 refers to *mode*.

Therefore, the CMN sub-model should be revised to accommodate this variable. The mode variable in CMN is named *cmn\_mode*. Figure 6.17 represents the revised SetTargetSpeed. Therefore, the guard @grd2 is renamed. In addition, the action @act2 is added to update *cmn\_mode* consistently with the MOD sub-model. Other mode events which update *cmn\_mode* should also be defined in the CMN sub-model. The consistency of the shared variable can be proved by adding the invariant *mode = cmn\_mode* to the composed model.

```

event SetTargetSpeed
refines SetTargetSpeed
  where
    @grd1 actualSpd ∈ lb..ub
    @grd2 cmn_mode = ON
  then
    @act1 targetSpd := actualSpd
    @act2 cmn_mode := ACTIVE
End

```

Figure 6.17: *SetTargetSpeed* is revised to reconcile the shared *mode* variable.

## 6.5 Extending Machine Structure

As discussed in the shared-event composition in Section 4.5, two machines can be composed provided that they have *no variable in common*. Therefore, when this style was used to formalise an RD as sub-models which share variable phenomena, Section 6.2.2, we renamed the shared variables and simulated their behaviour using extra events. Also, equaliser invariants were introduced to prove the consistency of shared variables amongst the sub-models. Introducing *renamed variables*, *simulating events* and *equaliser invariants* when modelling shared variables cost extra time and effort.

In Section 6.5.1 we propose a simple extension to the structure of an Event-B machine which makes variable sharing among sub-models possible. This extension involves defining *internal* and *external* variables for every machine. The consistency checks between the internal and external variables is discussed in Section 6.5.2.

Composition of machines with extended structure is explained in Section 6.5.3. After this, in Section 6.5.4 we discuss a simple example for composing machines. An alternative to extending machine structures in order for sub-models to share variables is to use *witnesses*. This is discussed further in Section 6.5.5.

### 6.5.1 Extending the Structure of a Machine

Structures of Event-B machines were discussed in Section 4.1.2, where we mentioned machines can contain *refines*, *variables*, *invariants*, *theorems*, *variants* and *events* clauses.

In this section we propose a simple extension to this structure which allows a machine to access variables in other machines. The extension for the machine structure is defined in a way that it enables machines with extended structures to be composed using the shared-event style.

The extended structure of a machine is shown in Figure 6.18. The extended structure contains two extra clauses, namely *exports variables* and *imports variables*. This extension is based on the concept of *internal* and *external* variables for an Event-B machine. A machine has read-write access to its internal variables, while it has read-only access to its external variables.

A detailed explanation of the clauses *variables*, *exports variables* and *imports variables* are provided below:

- **Variables:** The *internal* variables of a machine are defined in this clause. These are variables to which the machine has read and write access. In Figure 6.18 internal variables are shown as  $v_1, \dots, v_n$ . Events of a machine can update variables internal to the machine in their actions (write access). In addition, internal variables can be referred to in events guards (read access). This is shown in the action and guard of the event  $e_1$  in Figure 6.18.
- **Exports Variables:** These are *internal* variables which a machine can export in order for other machines to access them. In other words, a machine makes an internal variable externally available. A variable of a machine can be exported only if it is defined in the *variables* clause of the same machine.

As shown in Figure 6.18, every exported variable of  $v_p, \dots, v_q$  is also an internal variable,  $(\forall x. x \in p, \dots, q \Rightarrow x \in 1, \dots, n)$ . The type of an exported variable is determined by its internal definition.

- **Imports Variables:** These are *external* variables which a machine can import in order to have read-only accessibility to them. This in terms of Event-B means that imported variables cannot be modified in actions of events of the importing machine. However, they can be used for instance in guards, invariables and theorems.

Figure 6.18 shows that machine M has imported the external variables  $ev_1, \dots, ev_m$ . These variables are exported by other machines which are not shown here. The types of external variables are defined by machines which export them. The imported variables which are read-only can be used in the guard G in event  $e_1$ . In addition, these variables can influence the expression E through which an internal variable such as  $v_i$  is updated.

```

MACHINE  M
REFINES  N
SEES    C
VARIABLES   $v_1, \dots, v_n$ 
EXPORTS VARIABLES   $v_p, \dots, v_q$   (where  $\forall x. x \in p, \dots, q \Rightarrow x \in 1, \dots, n$ )
IMPORTS VARIABLES   $ev_1, \dots, ev_m$ 
INVARIANTS   $I(s, c, v_1, \dots, v_n, ev_1, \dots, ev_m)$ 
EVENTS
  event  $e_1$ 
  refines  $e_1$ 
    any  $p$ 
    where  $G(p, s, c, v_1, \dots, v_n, ev_1, \dots, ev_m)$ 
    then  $v_i := E(p, s, c, v_1, \dots, v_n, ev_1, \dots, ev_m)$   (where  $i \in 1, \dots, n$ )
  end
  ...
END

```

Figure 6.18: Extended structure of a machine.

Note that other elements of an Event-B machine, such as *variants* and *theorems* can be added to the extended structure of Figure 6.18. We have not included them in this figure as their definitions have not been modified.

Invariants (or theorems) of a machine may contain external variables. Such cross-cutting invariants ( $I$  in Figure 6.18) will not be proved at this stage, but they are defined to be used in proving the correctness of other invariants as well as refinement steps. Invariants which refer to external variables will be proved after composing all machines. It is also possible to define further cross-cutting invariants in the composed machine. The composition of machines is discussed in Section 6.5.3.

The implementation of importing and exporting variables can be realised by using a global variable repository for every project which may consist of several machines. When a machine exports a variable, the variable will be added to the global variable repository of the project. Variables of this repository are accessible by other machines within the same project. A machine can import any variable from the repository provided that the variable is not internal to it. In other words, every machine can import variables which are exported by other machines.

### 6.5.2 Consistencies between Internal and External Variables

Based on the definition of the *variables*, *exports variables* and *imports variables* clauses which were discussed in the previous section, the following consistency statements should hold when machines are defined.

1. For every exported variable there should exist at least a single machine which imports the variable.
2. For every imported variable there should exist one and only one machine which exports the variable.
3. Internal variables of machines which represent the requirements must have unique names globally within a project.
4. A machine cannot import any of its internal variables.
5. Imported variables of a machine cannot be exported.
6. As part of static checking external variables can be type checked against their counterpart internal variable. Any typing difference then can be flagged out.

### 6.5.3 Composition of Machines with Extended Structure

This section discusses the composition of sub-machines with extended structure using the shared-event composition style. Event composition and sub-machine composition in the shared-event style were described in detail in Section 4.5. Composing sub-machines with extended structure is very similar to the original descriptions of Section 4.5.

A composition may involve several sub-machines with extended structure. Variables of the composed machine consist of all the variables of the sub-machines which are included in the composition. This means variables which are defined in the *variables* clauses of the included sub-machines will be composed. The *exports* and *imports variables* clauses are eliminated. This is because they are defined to allow sub-machines (or sub-models) to access variables in other sub-machines. When sub-machines are composed they are merged into a single model, so, all variables become internal to the composed machine and the concept of external and internal variables (or *imported* and *exported variables* clauses) is no longer meaningful.

Events of sub-machines are composed according to the description of Section 4.5. As a reminder, to compose several events their guards are conjoint and their actions take place simultaneously. Invariants of the composed machine are defined as a conjunction of all invariants of the included sub-machined.

One difference between the share-event composition approach of Section 4.5 and composition of sub-machines with extended structure is that invariants which contain external variables must be added to the composed machine. As discussed, cross-cutting invariants which refer to external variables, are not proved in sub-models. This is because the definition of external variables in a sub-model is incomplete, since the states of external variables are not modified. Therefore, in order to prove the preservation of cross-cutting invariants they must be added to the composed machine.

It is also possible to introduce new invariants in a composed machine. Therefore, the introduction of cross-cutting invariants can be postponed to after composition. Generally speaking, unless cross-cutting invariants can improve a sub-model (e.g. facilitating automatic proof), we try to introduce them in the composed machine.

The current shared-event composition plugin allows the user to determine whether or not they wish to include the invariants of sub-models into the composed machine. However, extending the structure of machines means that this plugin will need modifications to reflect that cross-cutting invariants should be included in the composed model. An example of a composed machine is shown in the next section.

#### 6.5.4 An Example of Composing Machines with Extended Structure

Figure 6.19 represents parts of a simple specification which is modelled using two sub-models. These sub-models are defined using the extended structure. As shown, machine  $X$  contains an internal variable called  $x$  which is updated in the event  $DecX$ . In addition, this machine imports the external variable  $y$  which is exported by machine  $Y$ . Based on the type defined in machine  $Y$ , the variable  $y$  in  $X$  is implicitly a natural number. In  $DecX$ , variable  $y$  is used in order to update  $x$ . In addition, this event has a guard which shows the relation between  $x$  and  $y$ .

Consistency checks which were mentioned in Section 6.5.2 ensure that the internal representation of  $y$  in the machine  $Y$  matches its external representation in the machine  $X$ . For instance, if  $y$  in machine  $Y$  was not a number, the guards  $x \geq y$  in  $X$  would need to be flagged as inconsistent with the type of  $y$ . The consistency checks also guarantee that the exported variable  $y$  is imported in at least one other machine. In addition, it is checked that the imported  $y$  is not an internal variable of  $X$ .

Figure 6.20 represents the composition of machines  $X$  and  $Y$ . As shown, the composed machine has the same structure as a normal Event-B machine. The variables of the composed machine are internal variables of  $X$  and  $Y$ . Also, the invariant of the composed machine is a conjunction of the invariants in  $X$  and  $Y$ . Cross-cutting invariants can also be added to the composed machine. For instance the invariant  $x + y \geq 1$  can be added to the composed machine. Note that  $X$  and  $Y$  share the event INITIALISATION which is composed based on the shared-event composition techniques.



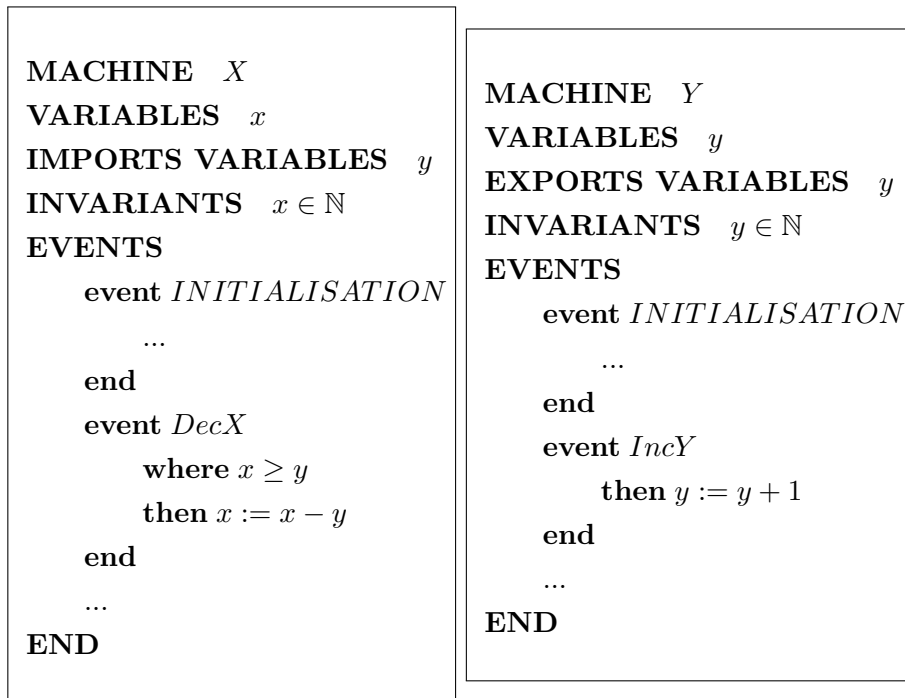


Figure 6.19: An example of importing and exporting variables in a extended machine.

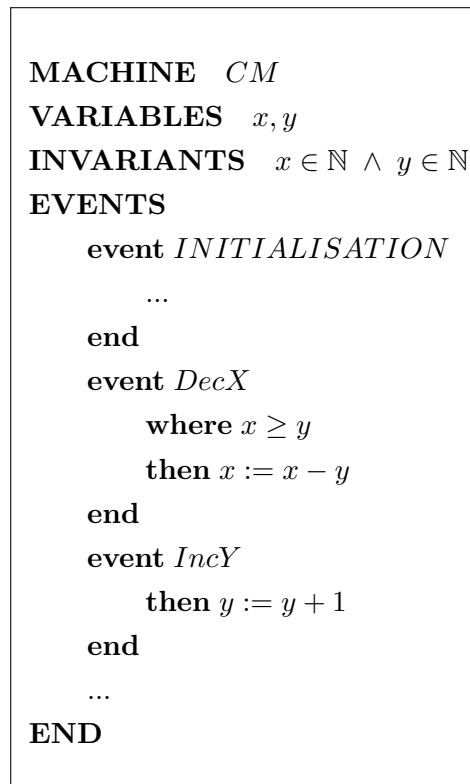


Figure 6.20: Composition of the machines X and Y.

### 6.5.5 Sharing Read-Only Variables through Witnesses

The extended structure for machines allows us to define variables of a machine as read-only accessible to other machines. The effect of the extension is also achievable through *parameters* and *witnesses*. In this section we discuss how parameters can be used to formalise requirements as sub-models which share read-only variables.

To do this parameters can be used in order for events of a sub-model to refer to external variables (variables of other sub-models). After the composition of sub-models, parameters which represent external variables can be concretised through witnesses. This is explained further using the example illustrated in Figure 6.21.

Figure 6.21(a) represents two sub-models which are modelled as machines M1 and M2. Variables  $n$  and  $m$  are respectively internal to M1 and M2. Consider a requirement such as “ $m$  shall be set to true when  $n$  is above 10”. This requirement shows that the event *UpdateM* in M2 should refer to  $n$  which is internal to M1. To model this requirement a parameter which is called  $x$  is defined in *UpdateM* to capture the variable  $n$  in M1. Note that  $x$  has the same type as  $n$ .

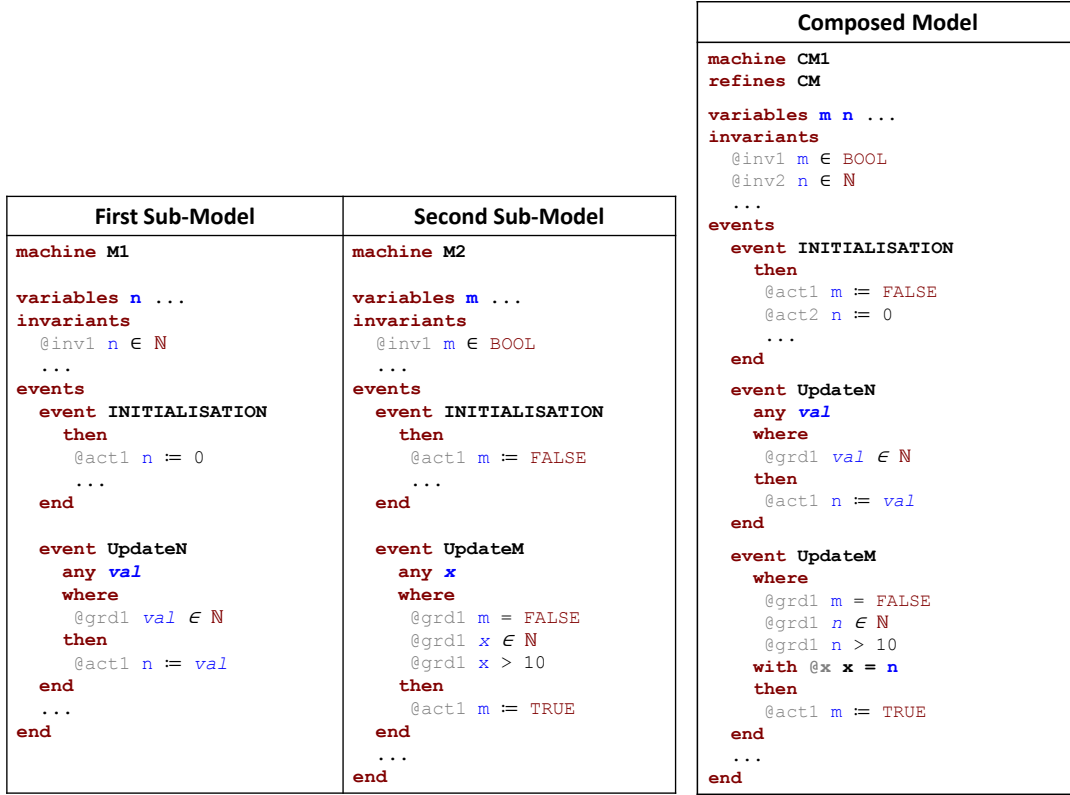
M1 and M2 can be composed based on the shared-event composition. The variables of the composed machine are  $n$  and  $m$ . This means in the composed machine cross-cutting invariants which contain both variables  $n$  and  $m$  can be defined. Events of the composed machine are defined as below:

$$\begin{aligned}
 CM.INITIALISATION &\triangleq M1.INITIALISATION \parallel M2.INITIALISATION \\
 CM.UpdateN &\triangleq M1.UpdateN \\
 CM.UpdateM &\triangleq M2.UpdateM
 \end{aligned}$$

Figure 6.21(b) represents that the composed machine is refined in order to replace the parameter  $x$  in *UpdateM* with its concrete representation, which is  $n$ . This example shows a way of formalising shared variables without extending the structure of a machine.

### 6.5.6 Discussions on Extended Machine Structure

Our objective for extending the structure of machines in Event-B was to simplify formalising a system as sub-models which might share events or read-only variables. Without extending machine structure, requirements can be specified as sub-models using the shared-event composition. This was discussed in Section 6.2.3 where extra events, variables and invariants were defined in order for sub-models to share read-only variables.



(a) The parameter  $x$  in COMP2 represents the variable  $n$  in COMP1. (b) Composition of the parameter  $x$  and the variable  $n$ .

Figure 6.21: An example of the extended shared variable composition.

Extending machine structure simplifies the formalisation of an RD, since behaviours on shared variables can be specified in one sub-model, while others have read-only access to them. The proposed extended machine structure and the shared-event composition can be used to specify requirements of any system in Event-B as composable sub-models which share events and read-only variables. In other words, the usage of the extended machine structure is not limited to the formalisation of a control system based on the MCMC sub-models.

One main advantage of formalising requirements as sub-models which share events and read-only variables is the *separation of concerns*. In other words, while formalising a sub-model, the focus is on variables and events which belong to the sub-model (internal variables and events), as well as interfaces of the sub-model with other sub-models through importing variables. It is only during the composition that the sub-models are connected via firstly synchronising events and secondly relating imported and exported variables. In addition, it is after composition of sub-models that cross-cutting invariants are considered. Therefore, before composition the modeller can focus on local properties of a single sub-model, while during composition the global aspects of the model are of interest.

## 6.6 MCMC Sub-Models Schemas

The extended machine structure can be used to formalise sub-models which share events or read-only variables. However, our ultimate goal in this thesis is to develop guidelines for modelling control systems. To do this, Section 6.6.1 defines schemas for modelling monitored, controlled, mode and commanded (MCMC) sub-models in Event-B using the extended machine structure. As an example of a formalisation based on the MCMC schemas, Section 6.6.2 represents the sub-models of the CCS.

### 6.6.1 Schemas of Machine Structure in MCMC Sub-Models

Sections 6.6.1.1 and 6.6.1.2 respectively represent schemas of mode and controlled sub-models which are defined based on the extended machine structure. To avoid repetition, schemas for monitored and commanded sub-models are not represented, as they can be defined similarly to the mode and the controlled schemas. The MCMC schemas provide guidelines and patterns which simplify the process of modelling. In addition, they can be used to validate sub-models which are defined based on the MCMC approach.

#### 6.6.1.1 Mode Sub-Model Schema

Figure 6.22 illustrates the schema for a mode sub-model which is modelled as a machine called *MOD-MCHN*. This machine contains a single variable *mode* which represents the mode phenomenon. The variable *mode* can be exported by MOD-MCHN. Also, this machine may import any number of monitored, commanded or controlled variables, which are represented by  $MNR_{sub}$ ,  $CMN_{sub}$  and  $CNT_{sub}$  respectively. Invariants in MOD-MCHN may refer to:

- sets ( $S$ ) and constants ( $C$ ) in the context  $C_x$ ;
- the internal variable *mode*;
- any imported variable ( $MNR_{sub}$ ,  $CMN_{sub}$  and  $CNT_{sub}$ ).

Note that invariants on imported variables will be proved after the composition of the MCMC sub-models.

Mode events in MOD-MCHN in Figure 6.22 update the internal variable *mode*. These events are defined based on the pattern in Figure 5.9. All possible modes can be captured as a set such as STATES, i.e.,  $STATES = \{st_1, \dots, st_n\}$ . In Figure 6.22, *ModeSet* is a subset of STATES. A mode event such as *Update\_Mode<sub>st<sub>1</sub></sub>* updates *mode* to the new state  $st_1$ . The guards of *Update\_Mode<sub>st<sub>1</sub></sub>* are represented by a meta-guard called  $G_1$ .

As shown, guards may refer to the local parameters ( $x$ ), the internal variable  $mode$ , any exported variable, as well as sets and constants of the context.

```

MACHINE  MOD – MCHN
SEES     $C_x$ 
VARIABLES   $mode$ 
EXPORTS VARIABLES   $mode$ 
IMPORTS VARIABLES   $MNR_{sub}, CMN_{sub}, CNT_{sub}$ 
INVARIANTS   $I(S, C, mode, MNR_{sub}, CMN_{sub}, CNT_{sub})$ 
EVENTS
  event INITIALISATION
    ...
  end
  event Update_Mode $_{st_1}$ 
    any  $x$ 
    when  $mode \in ModeSet$ 
       $\wedge G_1(MNR_{sub}, CNT_{sub}, mode, CMN_{sub}, S, C, x)$ 
    then  $mode := st_1$ 
  end
  ...
  event Update_Mode $_{st_n}$ 
    any  $x$ 
    when  $mode \in ModeSet$ 
       $\wedge G_n(MNR_{sub}, CNT_{sub}, mode, CMN_{sub}, S, C, x)$ 
    then  $mode := st_n$ 
  end
END

```

Figure 6.22: A schema for a mode (MOD) sub-model.

### 6.6.1.2 Controlled Sub-Model Schema

The schema for a controlled sub-model which is modelled as the CNT-MCHN machine is shown in Figure 6.23. The internal variables of the CNT-MCHN machine are controlled variables represented by the set  $CNT$ . In this schema the set  $CNT$  is defined as  $CNT = \{cnt1, cnt2, \dots, cntX\}$ . A subset of controlled variables can be exported. This is shown as  $CNT_{sub}$  (i.e.  $CNT_{sub} \subseteq CNT$ ). CNT-MCHN may import the  $mode$  variable and any number of monitored and commanded variables ( $MNR_{sub}, CMN_{sub}$ ). Invariants in CNT-MCHN may refer to sets ( $S$ ) and constants ( $C$ ) in the context  $C_x$ ; the internal variables in the set  $CNT$ ; or any imported variable ( $mode, MNR_{sub}$  and  $CMN_{sub}$ ).

Control events in CNT-MCHN are defined based on the pattern of Figure 5.13. As shown in Figure 6.23, a control event such as  $Update\_Cnt1_1$  updates the controlled variable  $cnt1$  based on parameters ( $x$ ), monitored, controlled, mode and commanded variables ( $MNR_{sub}$ ,  $CNT_{sub}$ ,  $mode$ ,  $CMN_{sub}$ ) as well as sets and constants ( $C$ ,  $S$ ). Similarly, the guards of a control event can refer to parameters, monitored, controlled, mode and commanded variables as well as sets and constants. Since a controlled variable, such as  $cnt1$ , may be updated by different events, there exists  $n$  number of events. This is shown by the events  $Update\_Cnt1_1$  to  $Update\_Cnt1_n$ .

Note that it is possible to define control events so that a number of controlled variables are updated simultaneously in one event. This event can be constructed by combining several control events as was discussed in Section 5.7.6. An example can be an event which is a combination of  $Update\_Cnt1_1$  and  $Update\_CntX_m$ . This event will have two actions which update  $cnt1$  and  $cntX$  simultaneously.

Commanded and monitored sub-models will have a very similar structure to Figure 6.23, although their events will be similar to the patterns which were given in Section 5.7.1.

### 6.6.2 An Example: Formalising CCS Sub-Models using MCMC Schemas

Figure 6.24 represents an example of the formalisation of the monitored (MNR), mode (MOD), controlled (CNT) and commanded (CMN) sub-models of the CCS based on the schemas which were discussed in the previous section. The formalisation in this section is similar to Section 6.4. However, here we use the schemas which were defined in the previous section based on the extended machine structure, while in Section 6.4 the MCMC sub-models of the CCS were defined based on the original machine structure in Event-B.

In Figure 6.24, the *mode* and *actual speed* variables are respectively internal to the MOD and the MNR sub-model where they can be modified. These variables are imported by the CNT and CMN sub-models where they are available as read-only variables.

The composition of the MNR, MOD, CNT and CMN sub-models is shown in Figure 6.25. As represented, the composed machine has the same structure as an original Event-B machine. The variables of the composed machine are the variables of all sub-models. The events are composed as shown below.

$$\begin{aligned} Comp.Activate\_SetTargetSpeed &\triangleq MOD.Activate \parallel CMN.SetTargetSpeed \\ Comp.UpdateAcceleration &\triangleq CNT.UpdateAcceleration \end{aligned}$$

```

MACHINE  CNT – MCHN
SEES     $C_y$ 
VARIABLES  CNT
EXPORTS VARIABLES   $CNT_{sub}$ 
IMPORTS VARIABLES   $MNR_{sub}, CMN_{sub}, mode$ 
INVARIANTS   $I(S, C, mode, MNR_{sub}, CMN_{sub}, CNT)$ 
EVENTS
  event INITIALISATION
    ...
  end
  event Update_Cnt11
    any  $x$ 
    when  $mode \in ModeSet$ 
       $\wedge G_{11}(MNR_{sub}, CNT_{sub}, mode, CMN_{sub}, S, C, x)$ 
    then  $cnt1 := F(MNR_{sub}, CNT_{sub}, mode, CMN_{sub}, S, C, x)$ 
  end
  ...
  event Update_Cnt1n
    any  $x$ 
    when  $mode \in ModeSet$ 
       $\wedge G_{1n}(MNR_{sub}, CNT_{sub}, mode, CMN_{sub}, S, C, x)$ 
    then  $cnt1 := F(MNR_{sub}, CNT_{sub}, mode, CMN_{sub}, S, C, x)$ 
  end
  ...
  event Update_CntXm
    any  $x$ 
    when  $mode \in ModeSet$ 
       $\wedge G_{xm}(MNR_{sub}, CNT_{sub}, mode, CMN_{sub}, S, C, x)$ 
    then  $cntX := F(MNR_{sub}, CNT_{sub}, mode, CMN_{sub}, S, C, x)$ 
  end
END

```

Figure 6.23: A schema for a controlled (CNT) sub-model.

The first composition shows that the event *Activate* and *SetTargetSpeed* both contribute to the specification of the event *Activate\_SetTargetSpeed*. In other words, the sub-models MOD and CMN share events.

MNR Sub-Model	MOD Sub-Model	CNT Sub-Model	CMN Sub-Model
<pre> <b>machine</b> MNR <b>sees</b> C_MNR <b>variables</b> actualSpd <b>exports</b> variables <b>actualSpd</b> <b>invariants</b>   @inv1 actualSpd ∈ 0..maxCarSpd   ... <b>events</b>   ... <b>end</b> </pre>	<pre> <b>machine</b> MOD <b>sees</b> C_MOD <b>variables</b> mode <b>exports</b> variables <b>mode</b> <b>imports</b> variables ... <b>invariants</b>   @inv1 mode ∈ STATUS   ... <b>events</b>   ... <b>event</b> Activate   <b>where</b>     @grd1 mode = ON     @grd2 ...   <b>then</b>     @act1 mode := ACTIVE   <b>end</b>   ... <b>end</b> </pre>	<pre> <b>machine</b> CNT <b>sees</b> C_CNT <b>variables</b> acceleration <b>imports</b> variables mode <b>actualSpd</b> <b>targetSpd</b> <b>invariants</b>   @inv1 acceleration ∈ ℤ   ... <b>events</b>   ... <b>event</b> UpdateAcceleration   <b>where</b>     @grd1 mode = ACTIVE   <b>then</b>     @act1 acceleration :=       accFun(targetSpd ↦ actualSpd)   <b>end</b>   ... <b>end</b> </pre>	<pre> <b>machine</b> CMN <b>sees</b> C_CMN <b>variables</b> targetSpd <b>exports</b> variables <b>targetSpd</b> <b>imports</b> variables mode <b>actualSpd</b> <b>invariants</b>   @inv1 targetSpd ∈ lb..ub   ... <b>events</b>   ... <b>event</b> SetTargetSpeed   <b>where</b>     @grd1 actualSpd ∈ lb..ub     @grd2 mode = ON   <b>then</b>     @act1       targetSpd := actualSpd   <b>end</b>   ... <b>end</b> </pre>

Figure 6.24: MNR, MOD, CNT and CMN sub-models of the CCS based on the MCMC schemas.

Composed Model
<pre> <b>machine</b> Comp <b>sees</b> C_MNR C_MOD C_CNT C_CMN <b>variables</b> mode acceleration actualSpd targetSpd <b>invariants</b>   @inv1 mode ∈ STATUS   @inv2 targetSpd ∈ lb..ub   @inv3 acceleration ∈ ℤ   ... <b>events</b>   ... <b>event</b> Activate_SetTargetSpeed   <b>where</b>     @grd1 actualSpd ∈ lb..ub     @grd2 mode = ON   <b>then</b>     @act1 targetSpd := actualSpd     @act1 mode := ACTIVE   <b>end</b> <b>event</b> UpdateAcceleration   <b>where</b>     @grd1 mode = ACTIVE   <b>then</b>     @act1 acceleration :=       accFun(targetSpd ↦ actualSpd)   <b>end</b> <b>end</b> </pre>

Figure 6.25: Composition of MNR, MOD, CNT and CMN sub-models of the CCS.

## 6.7 Discussions and Conclusion

The main motivation of this chapter was to manage the complexity of a formal design process by breaking the requirements into sub-problems that can be modelled independently. This goal is achieved by proposing a systematic approach for modelling requirements of a control system. The approach suggested formalising requirements as



composeable *monitored*, *controlled*, *mode* and *commanded* (MCMC) sub-models that share events and read-only variables.

One drawback of formalising sub-problems as sub-models is that a mistake in modelling shared phenomena can be reflected in other sub-models that share the phenomena. For instance, if the type of a shared variable needs modification, this change will have to be reflected in every sub-model that shares the variable. Therefore, understanding the sub-problems and the phenomena they share before formalising them is essential.

The requirement structuring approach which was proposed in the previous chapter results in requirements which update a specific variable to appear in a single sub-problem. In other words, taking the requirement structuring steps means every variable will be modified in one sub-problem, while it can be referred to in other sub-problems. This nature of the MCMC structuring approach makes it suitable for read-only variable sharing as well as event sharing between sub-models.

To reconcile shared variables and events we adjusted the shared-event composition approach. However, extra variables, events and invariants were required. To eliminate this additional effort, we proposed to extend the structure of Event-B machines. The extended structure includes *export variables* and *imports variables* clauses. ‘Export variables’ defines internal variables of a sub-model which can be accessed by other sub-models, while ‘import variables’ defines external variables which belong to other sub-models but can be accessed by the sub-model.

The extended machine structure means that modifications of variables become local to a sub-model. Therefore, rectifying errors when modelling shared phenomena becomes easier. The extended machine structure in conjunction with the shared-event composition can be used to specify requirements of any system as composeable sub-models which share read-only variables as well as events. We also provided schemas as guidelines for formalising MCMC sub-problems. These schemas outlined the appearance of the MCMC sub-models, so, they can be used to validate an MCMC sub-model.

The case study of a CCS was formalised using reconciliation as well as the extended machine structure. In the next chapter we formalise a lane centering controller using the extended structure and shared-event composing.

## Chapter 7

# Case Studies: Lane Departure Warning and Lane Centering Controller

In this chapter we apply the MCMC four-stage approach of structuring an RD, formalising as sub-models, composing the sub-models and validating them against the RD, to two automotive case studies of a lane departure warning system (LDWS) and a lane centering controller (LCC). The LDWS is discussed briefly and its RD is represented partially, while the LCC is explained in more detail. In the LDWS case study we focus mainly on the first and the last stages, i.e. structuring the RD and validating the model against the RD.

However, in the LCC case study our main focus is on the second and the fourth stages, i.e., formalising a structured RD as sub-models and composing the sub-models. To do this we use the guidelines and patterns of the previous chapters. In particular, the extended structure of machines that was proposed in Chapter 6 is used in order to simplify the composition process.

In Section 7.1 an overview of the LDWS is discussed and in Section 7.2 parts of its RD are structured. After that, in Section 7.3 the formalisation of the LDWS is discussed briefly. This is done using a single chain of refinement in contrast to the LCC which is formalised as sub-models. Therefore, the LDWS does not require the fourth stage which is composing sub-models. Section 7.4 explains the validation of the LDWS RD.

Section 7.5 discusses an overview of the second case study which is the LCC. After that in Section 7.6 the process of writing requirements of the LCC and the structured RD of the LCC are discussed. The main focus in this case study is the formalisation process. Since examples of Stage 3 (validation and revision) have already been given, the validation of the LCC model against its RD is not provided. The LCC is then formalised

in Section 7.7 using the MCMC sub-models. Section 7.8 provides the composition of the sub-models of the LCC.

A brief discussion on the advantages of formalising an RD as composable sub-models and a possible extension of the MCMC approach are discussed in Section 7.9. The conclusion of this chapter can be found in Section 7.10.

## 7.1 Case Study 1: Lane Departure Warning System

LDWS is a driver assistance system which receives camera observations of the lane on the motorway and uses this information to warn the driver of a lane departure. One way to detect the car departing the lane is by checking the car's current position in the lane. This can be estimated by lane detection algorithms based on the camera observation [RME00].

In order to warn the driver before the vehicle crosses the lane, a virtual lane width which is inside the lane boundaries is assumed. This virtual lane, called the earliest warning lines (EWL), is determined based on the speed of the car. When the vehicle is within the earliest warning lines, the LDWS does not issue any warnings. This area is called the “no-warning zone”, and the area beyond EWL is the “warning zone” [Fed05], shown in Figure 7.1.

To ensure that the driver receives useful warnings, the warning should be issued only when the vehicle is leaving the lane unintentionally. Lane changing is one of the situations where the driver is intentionally leaving the lane and it can be monitored through the indicator. So, whenever the indicator is on and the car is about to leave the lane on the same direction as indicated, LDWS should not issue warnings.

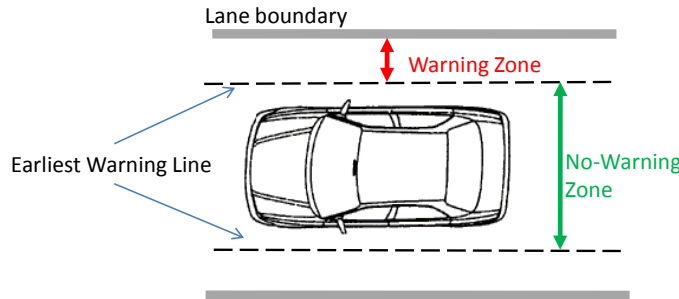


Figure 7.1: Warning and no-warning zone for an LDWS.

## 7.2 Stage 1: Structuring RD of LDWS

In this section we discuss some of the requirements of LDWS which are structured according to the MCMC approach of Section 5.3. The textual requirements of the

LDWS evolved in three phases. In the first phase we produced and structured the RD based on information in the *public domain*, such as [For10, Fed05, PMGB05, RME00].

In the second phase, we sought feedback on the structured RD from *domain experts*. This resulted in identifying the commanded phenomenon of *offset* which represents the distance the driver wishes to have from the boundaries. The requirements of these two phases and their structuring process are explained in the remainder of this section.

The third version of the LDWS's RD is discussed in Section 7.4.1. This is because the third RD was produced as the result of the *formal modelling* of the second version of the RD. Notice that we take the structuring steps for every version of the RD. The requirements of the LDWS are shown partially and some requirements, such as the ones related to the indicator, are not discussed here.

### 7.2.1 Version 1: RD of LDWS based on Public Domain

The commencement of the requirement structuring process is to identify the MCMC variable phenomena. In the case of the LDWS, the MCMC phenomena are identified based on material available in the public domain.

At the first step of the structuring process the monitored, controlled and commanded variable phenomena are listed. A summary of the identified phenomena is shown in Table 7.1. Note that we did not identify any commanded variable phenomenon based on the information available in public domain. A discussion on the controlled phenomenon EWL is given in Section 7.9.2.

Monitored Phenomena		Controlled Phenomena
<i>Speed</i> (to determine the location of the EWL)	<i>Car position</i> (difference between the car centre and the lane centre)	<i>EWL</i>
<i>Lane width</i>	<i>Car width</i>	<i>Warning</i>

Table 7.1: Monitored and controlled variable phenomena of the LDWS.

At the second step the values of the mode variable phenomenon are listed. To do this we use the state machine diagram shown in Figure 7.2. As illustrated *off* is the initial state from which the LDWS can be switched *on* and then *activated*. This figure also shows the controller mode events (CME) and the operator mode events (OME) which result in a mode change in the LDWS. Although the identification of the mode transitions is not required for structuring the requirements, understanding these transitions can help with identifying the values of the mode phenomenon. The identified mode events are used in the formalisation process that is described in Section 7.3.

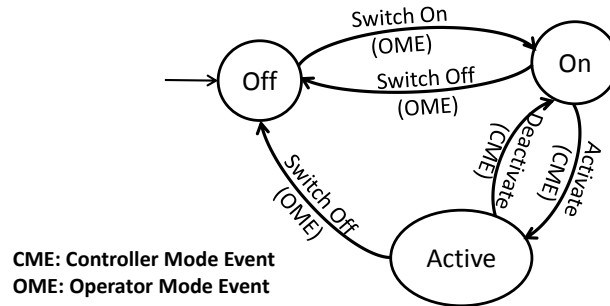


Figure 7.2: Possible mode changes in LDWS.

Finally, the requirements are identified and placed in the MCMC sections which correspond to their phenomena. An identifier is also defined for each requirement. Table 7.2 shows some of the LDWS requirements.

Req ID	Requirement Description
MNR1	LDWS shall detect vehicle position relative to visible lane boundaries based on the lane width and car width.
MNR2	LDWS shall detect the car speed.
MNR3	LDWS shall track lane boundaries where lane markings are clearly visible in daylight, night times and twilight lighting conditions.
CNT1	LDWS will determine the location of the earliest warning line (EWL) based on the car speed.
CNT2	LDWS will issue a warning when the vehicle has left the no warning zone (has crossed the EWL), and is entering the warning zone.
CNT3	If the LDWS is issuing warnings, it will stop the warning signal as soon as the driver steers away from the boundaries and the car is back in the no warning zone.
CNT4	LDWS starts its role when it is activated.
CNT5	If LDWS is issuing the warning and the driver switches the system off, the warning signal will stop.
MOD1	LDWS can be switched on and off by the driver.
MOD2	When LDWS is on it will be activated and if the car speed is greater than or equal to a certain threshold.
MOD3	LDWS become inactive as soon as the car speed goes below the threshold.

Table 7.2: RD version 1: Requirements are identified based on public domain.

### 7.2.2 Version 2: RD of LDWS based on Expert Feedback

Discussing the first version of the LDWS's RD with domain experts from GM India Science Lab resulted in identifying some missing requirements. One of these requirements is that the driver can change the EWL by setting the offset they wish to have from the lane boundaries. This requirement is important, since some people prefer to drive closer to the boundaries.

Retaking the steps for structuring requirements results in *offset* being identified as a commanded variable phenomenon. The requirements related to the commanded phenomenon of *offset* are added to CMN section. Also, as shown in Table 7.3, the requirement CNT1 is modified to reflect the fact that EWL can be influenced by the offset.

Req ID	Requirement Description
CMN1	The driver can set the offset they want to have from either side of the lane boundaries by increasing and decreasing the distance.
CMN2	The offset is always within a certain positive range (it cannot be outside the lane boundaries).
CMN3	The greater the determined offset the closer the EWL to the centre of the lane.
CNT1	LDWS will determine the location of the earliest warning line (EWL) based on the car speed and the offset.

Table 7.3: RD version 2: Requirements are added based on experts feedback.

## 7.3 Stage 2: Formalising LDWS

The guidelines and event patterns given in Section 5.7 for formalisation of variable and event phenomena of a structured RD in Event-B are used to model the second version of the LDWS RD. The identified event phenomena for every variable phenomena are discussed during the modelling.

In this case study we avoid modelling the MCMC sub-problems as separate sub-models. This is firstly because only parts of the requirements of the LDWS are discussed here, and secondly, in this case study we focus on the structuring and validation of the requirements. Therefore, the LDWS is formalised as a single chain of refinement. The refinement guidelines which were provided in Section 5.6.1 are used to define the refinement chain.

### 7.3.1 Layering Requirements for Refinement Levels

The process of modelling starts with identifying requirements necessary for constructing the abstract level. After this, we model each phenomenon and its interrelated phenomena

in one level of refinement. In the most abstract level the *main behaviour of the system* will be modelled. This behaviour is a controlled phenomenon, as controlled phenomena represent the role of the system.

Examining the structured RD of the LDWS shows that the requirement CNT2 represents the main role of this system which is issuing *warnings* when it is necessary. Thus, the controlled phenomenon *warning* will be modelled at the most abstract level. At this level we focus only on this variable and its corresponding events which define the transitions on the formal variable *warning*.

After this, in the first refinement level, the *mode* phenomenon and the requirements corresponding to the mode transitions are modelled. The second level of refinement involves modelling requirements which represent the condition under which the warning will be issued. As will be discussed, at this level of refinement multiple phenomena are modelled simultaneously, since they are interrelated and have simultaneous relations, which was discussed in the refinement guidelines.

### 7.3.2 Abstract Level

As mentioned, the phenomenon *warning* which represents the main role of the LDWS is captured at this level. We identify the requirements corresponding to this phenomenon by examining the CNT section (sub-problem) of the LDWS RD. These are requirements CNT2, CNT3 and CNT5. One of the refinement guidelines was to model a variable phenomenon and its transitions (the actions in event phenomena) before focusing on the conditions under which the events can occur (the guards in event phenomena). Therefore, a requirement such as CNT4 (LDWS starts its role when it is activated) is introduced in refinement levels when its phenomenon is modelled.

We start the modelling by representing the variable phenomenon *warning* as a formal variable in Event-B. This is modelled here as a boolean formal variable, called *warning*, which is set to TRUE when the warning is issued and reset to FALSE when the LDWS stops issuing the warning.

Also, based on the requirements two control event phenomena *stop warning* and *issue warning* which define how the warning variable is updated are identified. These are formalised as the control events, *IssueWarning* and *StopWarning* and are shown in the figure below.

### 7.3.3 First Refinement

At this level of refinement we focus on the *mode* phenomenon. To identify the requirements of this phenomenon, we refer to the MOD section of the structured RD. So, the requirements MOD1, 2 and 3 can be modelled at this level.

<pre> <b>event</b> IssueWarning   <b>where</b>     @grd1 warning = FALSE   <b>then</b>     @act1 warning := TRUE   <b>end</b> </pre>	<pre> <b>event</b> StopWarning   <b>where</b>     @grd1 warning = TRUE   <b>then</b>     @act1 warning := FALSE   <b>end</b> </pre>
--	---

Figure 7.3: Abstract level consists of controlled phenomenon *warning* and its events.

The variable phenomenon of mode is modelled by defining a formal variable named *status*. Also, a number of formal events are defined to capture the transitions among the system modes. For instance, the formal *SwitchOn* and *SwitchOff* operator mode events represent the requirement MOD1 which states that “the LDWS can be switched on and off by the driver”.

To incorporate the requirement CNT5 (warning will stop when the LDWS is switched off) into the *SwitchOff* event, two versions of this event are defined. As illustrated in Figure 7.4, the event *SwitchOff\_WarningOFF*, switches the LDWS off provided that no warning was being issued. However, if the LDWS is issuing warning, the event *SwitchOff\_WarningON* will simultaneously switch off the LDWS and reset the warning to FALSE. This is why this event refines (extends) *StopWarning*.

<pre> <b>event</b> SwitchOff_WarningON <b>extends</b> StopWarning   <b>where</b>     @grd2 status ∈ {ON, ACTIVE}   <b>then</b>     @act2 status := OFF   <b>end</b> </pre>	<pre> <b>event</b> SwitchOff_WarningOFF   <b>where</b>     @grd1 warning = FALSE     @grd2 status ∈ {ON, ACTIVE}   <b>then</b>     @act1 status := OFF   <b>end</b> </pre>
--	--

Figure 7.4: Two *SwitchOff* operator mode events are defined in the first refinement.

The requirement MOD2 is modelled by defining a controller mode event called *Activate*, which updates the mode from *on* to *active*. Since the mode variable is modelled, the requirement CNT3 which states “LDWS starts its role when it is activated” can be formalised. As shown in Figure 7.5, this is modelled by introducing a guard to the control event *IssueWarning*.

<pre> <b>event</b> IssueWarning <b>extends</b> IssueWarning   <b>where</b>     @grd2 status = ACTIVE   <b>end</b> </pre>
--

Figure 7.5: Control event *IssueWarning* in the first refinement.



### 7.3.4 Second Refinement

The warning variable should be set to TRUE or FALSE by comparing the *car position* to the *EWL* position. To do this comparison, the *car position* and the *EWL* can be introduced simultaneously. The value of the former depends on the *lane width* and *car width* phenomenon, while the value of the latter can be determined based on the *speed* and *offset* variable phenomena. So, at this level, multiple phenomena are defined, as they are interrelated. These phenomena capture the conditions for issuing and stopping the warning signal. The requirements related to the *EWL* and the *car position* are as follow:

- MNR1: LDWS detects the *car position* based on the *lane width* and *car width*.
- MNR2: Car *speed* is required for determining the *EWL*.
- CNT1: The *EWL* depends on the *speed* and *offset*.
- CNT2 and CNT3: Express the conditions under which the warning will be issued or stopped respectively. The conditions are defined based on the comparison of *car position* and the *EWL*.
- CMN1: *Offset* can be modified by the driver.
- CMN2: *Offset* is within a certain range.
- CMN3: Explains the relation between *offset* and the *EWL*.

The formalised monitored variables are *speed*, *laneWidth*, and *carPosition*. The formal monitor events which are responsible for modifying these variables are defined to simply set the monitored variables nondeterministically through a parameter. Also, the LDWS knows the width of the car which is a fixed value. Thus, this is modelled as a constant called *carWidth*.

In addition, we assume that the *EWL* is given to the LDWS, so it is modelled as a constant which is a function. This function is named *EWL\_Func* and returns the distance of the *EWL* from the lane boundaries based on the car *speed* and the *offset*. The type of the function is  $EWL\_Func \in \mathbb{N} \times offset\_LB..offset\_UB \rightarrow \mathbb{N}$ , meaning that for every possible tuple of *speed* (of type  $\mathbb{N}$ ) and *offset* (within the range  $offset\_LB..offset\_UB$ ) there is a value for the *EWL\_Func*. This value is a natural number which represents the distance *EWL* has from the lane boundaries.

The formal commanded variable is named *offset*. Since the value of *offset* is within a specific range, two constants called *offset\_LB* and *offset\_UB* are defined to represent the lower and upper bounds of the range. The formal command events which modify the

Monitored Variable	Monitor Event
carPosition	UpdateCarPosition
laneWidth	UpdateLaneWidth
Speed	UpdateCarSpeed
Commanded Variable	Command Event
offset	IncreaseOffset & DecreaseOffset

Table 7.4: Formal monitored and commanded variables and their events.

value of *offset* are *IncreaseOffset* and *DecreaseOffset*. A summary of the variables and events are shown in Table 7.4.

To model the condition for issuing the warning signal, the *EWL\_Func* function helps us to define *@grd6* in Figure 7.6. This guard models the requirement that when the car passes the EWL, event *IssueWarning* will be enabled. Also, as represented by *@grd4*, we modelled the *carPosition* as the difference between the centre of the car and the centre of the lane. Therefore, *carPosition* is a value between  $-(\text{laneWidth} \div 2)$  and  $+(\text{laneWidth} \div 2)$ . We assume when *carPosition* is '+' the car has moved to the right and if it is '-' the car has moved to the left. The *StopWarning* event can be refined in a similar style.

```

event IssueWarning extends IssueWarning
  where
    @grd3 laneWidth > carWidth
    @grd4 carPosition ∈  $-(\text{laneWidth} \div 2) \dots \text{laneWidth} \div 2$ 
    @grd5 offset ∈ offset_LB .. offset_UB
    @grd6  $(\text{carWidth} \div 2) + \text{carPosition} \geq (\text{laneWidth} \div 2) - \text{EWL\_Func}(\text{speed} \mapsto \text{offset})$ 
  end

```

Figure 7.6: Control event *IssueWarning* at the second refinement.

This step of refinement is more complex than the first step, since several phenomena are added to the model simultaneously. As mentioned in the refinement guidelines, it was possible to break this step by introducing phenomena which affect the EWL (i.e. *speed* and *offset*) or the car position (i.e. *lane width* and *car width*) before starting this level. We decided to model all these phenomena in one level to show that layering requirements for formal modelling is usually a matter of judgment and patterns can be used in a flexible manner.

## 7.4 Stage 3: Revision and Validation of LDWS

The formalisation process improves our understanding of the system. Thus, missing and ambiguous requirements can be identified. At this stage, we revise the RD and the

model to accommodate identified missing behaviours and to clarify ambiguous requirements. In Section 7.4.1 some of the missing and ambiguous requirements of the LDWS are discussed. After that, in Section 7.4.2 the formal model of the LDWS is adjusted accordingly.

In Section 7.4.3 the RD of the LDWS is validated against its model to justify the formal representations of each requirement. Notice, the validation discussed here involves the revised version of the RD and the model. However, we suggest that the validation column be constructed at every step of formal modelling. This means the validation column may also need revision.

### 7.4.1 Version 3: RD of LDWS after Formal Modelling

The main identified missing behaviour was that we realised the case where a car is travelling on the actual lane boundary should be differentiated from the case where the car has entered the warning zone (has crossed the EWL), but is yet to cross the actual lane boundary. As shown in Figure 7.7, this is mainly because of the limitations of the camera's vision field which can result in detection of only one boundary.

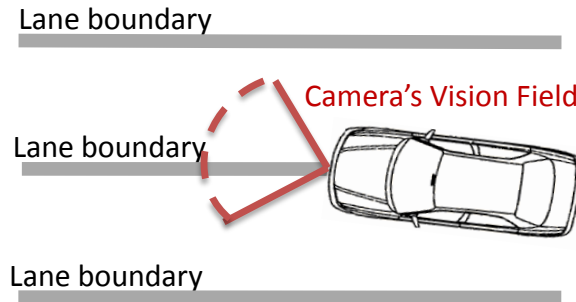


Figure 7.7: Limited vision field of cameras when car is travelling on a lane boundary.

This missing behaviour was identified as the result of the formalisation process which can improve our understanding of the requirements. One way of identifying missing behaviours is to consider what should happen when guards of events do not hold. In other words, whether or not the requirements of a system are logically complete with respect to the guards of events which model the requirements.

For instance, the guards of the event *IssueWarning* which was shown in Figure 7.6 can help us to identify a missing behaviour. By negating the guard  $@grd4$ , i.e.  $carPosition \in -(laneWidth \div 2).. + (laneWidth \div 2)$ , we can consider the situation where the car position is not within the lane. So, we posed the question of what should happen when the car is travelling on the lane boundary.

When a car travels on the actual lane boundary, it has already crossed the EWL. Therefore, if the LDWS is active it should warn the driver. To incorporate the requirements

related to this case and to distinguish them from when the car is in the warning zone and has not crossed the lane boundary, we take the requirement structuring steps of Stage 1. Since this requirement is related to the controlled variable *warning*, it should be added to the CNT sub-problem, using an appropriate ID. As shown in Table 7.5, this requirement is labelled as CNT6.

Also, when the LDWS detects that the car is travelling on the actual lane boundary, it no longer requires to detect the car position or the lane width, since the decision of issuing warnings is independent from the distance of the car from EWL. This resulted in finding a monitored variable, called *crossing lane* which is shown in the requirement MNR4.

Req ID	Requirement Description
MNR4	LDWS shall detect when the car has crossed the lane boundaries and is travelling on the boundary. In this case, the LDWS will no longer detect the position of the car relative to the lane width.
CNT6	LDWS will issue a warning when it detects that the car has crossed the lane boundaries and is travelling on the boundary.

Table 7.5: Third RD: Requirements added based on Formal Modelling.

#### 7.4.2 Revising the Formal Model

To identify which newly found requirement should be added to which level of the LDWS model, we consider the refinement layering guidelines of Stage 2, Section 5.6.1, as well as the guidelines given for the revision step, Section 5.8.2, where we proposed to introduce the new requirements in a new level of refinement or in a more concrete level when it is possible. This means the majority of refinement steps in a refinement chain can remain untouched.

As mentioned, the variable *warning* and its transitions were modelled in the abstract level. Looking at Table 7.5, the requirement CNT6 is about the warning phenomenon. However, this requirement defines a condition under which the warning should be set to TRUE. Therefore, we can postpone modelling this requirement to future refinement levels and we will proceed to the next refinement level.

At the first level of refinement the phenomenon *mode* was introduced. As the requirements of Table 7.5 are not about this phenomenon, we do not consider this refinement level any further. So, we progress to the second level of refinement.

The conditions of issuing and stopping warnings were introduced at the second level of refinement. Therefore, this can be a right level for modelling CNT6. Alternatively, CNT6 and the phenomenon *crossing lane* can be introduced in a new refinement level.

To model CNT6 it is also necessary to formalise MNR4, as the monitored phenomena *crossing lane* is required for modelling CNT6. A boolean variable, called *crossingLane*, whose value is *TRUE* when the car is travelling on the lane boundary is defined. Based on MNR4, if *crossingLane = TRUE*, variables *carPosition* and *laneWidth* will not be detected. This is modelled by adding the guard *crossingLane = FALSE* to both *UpdateCarPosition* and *UpdateLaneWidth* events.

Requirement CNT2 and CNT6 show that there are two cases where an activated LDWS should decide on issuing warnings. Firstly, when the car is *about* to cross the lane boundaries because it has crossed the *EWL* (CNT2). The second is when the car has crossed the *lane boundary* (CNT6). As shown in Figure 7.8, these are modelled in the control events *IssueWarning\_CloseToBound* and *IssueWarning\_CrossingLane* respectively. *IssueWarning\_CloseToBound* event is the same as the event *IssueWarning* in the second refinement level, Figure 7.6, but it also has the additional guard of *crossingLane = FALSE* which shows this event will not be enabled when the actual lane boundary is crossed.

<pre> event IssueWarning_CrossingLane extends IssueWarning    where     @grd3 crossingLane = TRUE   end </pre>	<pre> event IssueWarning_CloseToBound extends IssueWarning   where     @grd3 laneWidth &gt; carWidth     .     .     @grd7 crossingLane = FALSE   end </pre>
--	--

Figure 7.8: Control events after revising the formal model of LDWS.

### 7.4.3 Validation (Traceability) of LDWS Model

As mentioned, in order to validate a model against its RD we add a *validation column* to the structured requirement tables. Validation of some of the LDWS requirements against their model is shown in Table 7.6.

For instance, the requirement CNT1 is modelled by the controlled variable *warning*, the control event *IssueWarning\_CloseToBound* and showing that the car has entered the warning zone by the guard below in the control event *IssueWarning\_CloseToBound*:

$$carWidth + carPosition \geq laneWidth - EWL\_Func(speed \mapsto offset)$$

Another example is the requirement MOD1 which is modelled by defining a set called *STATUS*, a mode variable called *status* and two mode events, namely *SwitchOn* and *SwitchOff*. The set *STATUS* consists of *ON* and *OFF*. The variable *status* is defined as  $status \in STATUS$ .

We may decide not to model the entire RD formally or it may not be possible to represent a requirement formally. For instance the requirement MNR3 is not modelled here.

Req ID	Requirement Description	Validation Column
MNR1	LDWS shall detect vehicle position relative to visible lane boundaries based on the lane width and car width.	Monitored variable: <i>carPosition</i> & <i>laneWidth</i> ; Constant: <i>carWidth</i> ; Monitor event: <i>UpdateCarPosition</i> & <i>UpdateLaneWidth</i> .
MNR3	LDWS shall track lane boundaries where lane markings are clearly visible in daylight, night times and twilight lighting conditions.	Not Considered - Requirements related to the performance of the camera and image processing unit are not modelled.
CNT1	LDWS shall receive the location of the earliest warning line (EWL) based on the car speed and the offset.	Function: $EWL\_Func \in \mathbb{N} \times offset\_LB..offset\_UB \rightarrow \mathbb{N}$ .
CNT2	LDWS will issue a warning when the vehicle has left the no warning zone (has crossed the EWL), and is entering the warning zone.	Controlled variable: <i>warning</i> ; Control event: <i>IssueWarning_CloseToBound</i> ; Guard in event <i>IssueWarning_CloseToBound</i> : $carWidth \div 2 + carPosition \geq laneWidth \div 2 - EWL\_Func(speed \mapsto offset)$ .
CNT6	LDWS will issue a warning when it detects that the car has crossed the lane boundaries and is travelling on the boundary.	Control event: <i>IssueWarning_CrossingLane</i> ; Guards in event <i>IssueWarning_CloseToBound</i> : <i>crossingLane</i> = <i>FALSE</i> .
MOD1	LDWS can be switched on and off by the driver.	Set: <i>STATUS</i> = { <i>ON</i> , <i>OFF</i> }; Mode variable: <i>status</i> ; Operator mode events: <i>SwitchOn</i> & <i>SwitchOff</i> .
CMN1	The driver can set the offset they want to have from either side of the lane boundaries by increasing and decreasing the distance.	Commanded variable: <i>offset</i> ; Command event: <i>DecreaseOffset</i> & <i>IncreaseOffset</i> .

Table 7.6: Validation of the Requirement against the model.

Therefore, it cannot be validated, as no elements of the model represent this requirement. Thus, the validation column of MNR3 provides the justification for not modelling this requirement.

## 7.5 Case Study 2: Lane Centering Controller

A lane centering controller (LCC) is responsible for “automated lane-centering” (we do not consider automated lane-change manoeuvres). This system aims to maintain a vehicle either on the centre line or on a line close to the centre (at an offset chosen by

the driver) within a lane [LL09, YB12]. As shown in Figure 7.9, the LCC consists of a path generator unit and a controller. The path generator provides *predicted* and *target* paths based on the data it receives from the environment and the driver. The role of the LCC is to maintain the angle between these two paths as close to zero as possible by calculating the steering correction angle accordingly.

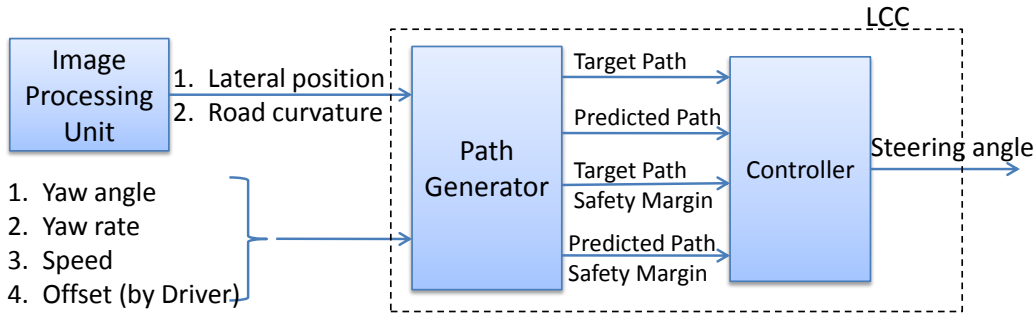


Figure 7.9: Overview of LCC inputs and output.

The predicted path is an estimate of the intended vehicle path. This is generated using the *speed*, the *yaw angle* and the *yaw rate* of the car. The predicted path also depends on the *lateral position*<sup>1</sup> of the car which is received from the image processing unit.

The target path<sup>2</sup> shows the path the vehicle should take to maintain its position within the lane. This path is determined based on the *lateral position* and the *roadway curvature* received from the image processing unit. These paths are shown in Figure 7.10.

The target path can also be influenced by the *offset* the driver wishes to have from the centre of the lane (central line). The offset is within a specific range, such as -2 to +2, where ‘-’ represents the offset from the left of the central line and ‘+’ represents the offset from the right. If the offset is set to 0, the central line will be used as the reference for generating the target path. For instance, the offset in Figure 7.10 is set to ‘-1’.

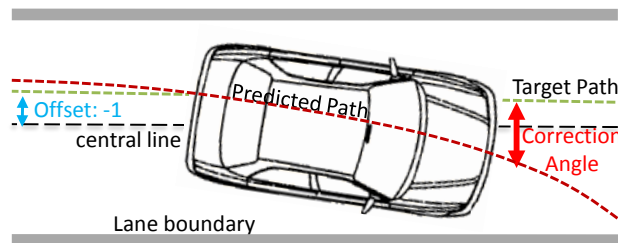


Figure 7.10: LCC actuates steering angle using target and predicted path.

In addition to target path and predicted path, the path generator component provides the LCC with a *safety margin* for each path. The safety margin represents the minimum time within which the path generator can provide either the target or predicted path for the LCC. When the safety margin of a path,  $x$ , is greater than 0, the path can be

<sup>1</sup>Lateral position is sometimes referred to as “lateral displacement” [Jin10].

<sup>2</sup>Target path is sometimes referred to as “desired path” [Jin10].

determined for the next  $x$  milliseconds, while 0 means the path cannot be determined any longer.

This margin is necessary, so that in the cases where the image processing unit fails to operate, the path generator can notify the LCC in advance by reducing the safety margin of the relevant path. In this case the LCC will issue warnings which mean the driver should take over the responsibility of steering. Notice that this is an assistance system, and not a safety system, thus it is the driver's responsibility to resume steering whenever they deem it necessary, including when LCC issues a warning.

Generally speaking an LCC is accompanied by an Adaptive Cruise Control (ACC) which is responsible for the longitudinal control of the car (the speed and the spacing from vehicles at the front), whereas the LCC controls the lateral car position (relevant to the lane boundaries) [PTG97]. Here, we assume that the ACC exists and consider its communication with the LCC. For instance the activation of the LCC requires the ACC to be active. The LCC will issue warnings as soon as faults and errors in ACC are detected.

A driver can also interact with the LCC. As mentioned, the UI is categorised into existing and specialised. Distinguishing these UIs helps us with identifying the system phenomena. The *existing UI* of the car which can override (suspend) the LCC are:

- **Steering wheel** where the applied torque is above a certain threshold, the LCC will be overridden. The pressure on the steering wheel indicates that the driver is overriding the LCC, such as in an emergency. The defined torque threshold allows the LCC to detect the driver's intention without any ambiguity.
- **Indicating** shows an intentional lane-change manoeuvre by the driver and will override the LCC.

The *specialised UI* of the LCC are defined using the requirements and with the help of domain experts. These are:

- **Switching on or off:** The driver can switch the LCC on or off.
- **Increasing and decreasing offset:** The offset the driver wishes to have from the central line can be set by increasing or decreasing its value.
- **Resuming the LCC:** When the controller is overridden, such as when the indicator is used, the LCC can be resumed by the driver.



## 7.6 Stage 1: Structured RD of LCC

Similarly to the LDWS, the RD of the LCC was evolved in several steps. These steps are briefly discussed in Section 7.6.1. After that, in Section 7.6.2 the final version of the RD and the MCMC variable phenomena of the LCC are explained.

### 7.6.1 Process of Writing LCC RD

In comparison to other case studies, such as the CCS [YBR10] and the LDWS [YB11], the LCC has less information available in the public domain. Thus, the feedback we received from domain experts and the process of formalisation influenced the RD considerably.

Initially, the RD of the LDWS was evolved to include the requirements of an LCC. Thus, the reference points based on which the LCC would correct the steering angle were the *lane boundaries* which are the reference points in the EWL. However, discussions with domain experts and papers such as [LL09] and [Jin10], showed that the reference point in the LCC for correcting the steering angle is the *centre of the lane*. Therefore, the RD was adjusted accordingly. Using the central line as the reference point for correcting steering angle meant that the *target path* and *predicted path* were also required. So these paths and the monitored phenomena which influenced them were identified.

After formally modelling the LCC and representing the model to domain experts, some further adjustments to the RD of the LCC were made. One of these was the concept of ‘safety margin’ which was discussed earlier. The concept of safety margin is necessary as in the situations where the predicted or the target path cannot be determined by the path generator, for safety reasons the LCC should not immediately stop correcting the steering. Therefore, the path generator provides the minimum time that the target and the predicted paths can be determined by returning their safety margins.

### 7.6.2 Variable Phenomena and Structured RD of LCC

As mentioned the textual RD of the LCC was evolved according to firstly the information in the public domain [LL09], [Jin10], secondly the feedback from domain experts and thirdly the formal modelling. The requirements presented in this section are from the final version of the LCC RD. To structure these requirements we take the MCMC structuring steps of Section 5.3.

At first the monitored variable phenomena *lateral position*, *roadway curvature*, *yaw angle*, *yaw rate*, *speed*, *indicator*, *steer*, *status of ACC* and *error detection* were identified and listed. These phenomena and their corresponding requirements are shown in the monitored (MNR) sub-problem in Table 7.7. As MNR10 shows, we treat detection of faults and errors in sensors and actuators as monitored phenomena and we are not

concerned with how they have been detected. The error handling and treatments of faults are beyond this work. However, as will be shown in the controlled (CNT) sub-problem, when an error arises the driver will be informed. It is also assumed that errors are manually rectified.

Phen.	ID	Requirement Description
Lateral Position	MNR1	LCC shall receive the lateral position of the car with respect to the lane boundaries.
Road Curvature	MNR2	LCC shall receive the road curvature from the cameras.
Yaw Angle	MNR3	LCC shall receive the yaw angle from the relevant sensor.
Yaw Rate	MNR4	LCC shall receive the yaw rate from the relevant sensor.
Speed	MNR5	LCC shall receive the car speed from the relevant sensor.
Indicator	MNR6	LCC shall monitor the indicator.
Steer	MNR7	LCC shall monitor forces applied to the steering by the driver.
	MNR8	LCC monitors the steering only if the applied torque is above a minimal amount.
ACC Status	MNR9	LCC shall monitor the changes in the status of ACC.
Error detection	MNR10	LCC shall detect faults and errors of sensors and the steering actuator.

Table 7.7: Structured requirements of LCC - Monitored sub-problem.

After this, *target path* (TP), *predicted path* (PP), *target path safety margin*, *predicted path safety margin*, *steering angle*, *display unit* and *warning* were identified and listed as the controlled variable phenomena. Table 7.8 shows the requirements of these phenomena. Notice that while the steering angle represents the main output of the LCC, the display unit and warning phenomena are used to inform the driver of the LCC's mode. In addition, target path, predicted path and their safety margins are used by the controller internally.

The only commanded variable phenomenon of the LCC is *offset*. The requirements corresponding to this phenomenon are shown in Table 7.9.

To construct the mode sub-problem we identify and list the possible states of the LCC using a state machine. An overview of the LCC state machine is shown in Figure 7.11. Although, the LCC modes are shown in a box, there is also an **unavailable** mode which represents a situation in which the LCC cannot perform any functionality. This is because the LCC operates only when the ACC is active.

The state machine in Figure 7.11 also represents the transitions between the states. An example of a mode event is the transition from **active** to **error** mode which will be

Phen.	ID	Requirement Description
Target Path	CNT1	The path generator will determine the target path based on the lateral position of the car, road curvature and the offset it receives from the environment and the driver.
Predicted Path	CNT2	The path generator will determine the predicted path based on the lateral position, yaw angle, yaw rate and speed of the car.
TP Safety Margin	CNT3	The path generator will calculate the safety margin of the target path. This calculation depends on the car speed. The higher the speed the lower the safety margin.
PP Safety Margin	CNT4	The path generator will calculate the safety margin of the predicted path. This calculation depends on the car speed. The higher the speed the lower the safety margin.
Steering Angle	CNT5	The role of LCC is to minimise the difference between the target and predicted paths by correcting the steering angle.
	CNT6	LCC will start correcting steering as soon as it becomes <b>active</b> .
	CNT7	LCC will stop correcting steering as soon as it is switched <b>off</b> , <b>standby</b> , <b>overridden</b> , <b>unavailable</b> or detects <b>errors</b> .
Display Unit	CNT8	The display unit may be green, yellow, red or off depending on the state of the system.
	CNT9	The display unit will be green when LCC is <b>active</b> and performs normally. It will turn yellow when LCC is <b>active</b> and is issuing warning. It will be red when LCC detects <b>errors</b> . It will be off when LCC is <b>off</b> , <b>standby</b> , <b>overridden</b> or <b>unavailable</b> .
Warning	CNT10	If LCC detects <b>errors</b> or if it is <b>active</b> , but either of the safety margins (or both) is below a certain threshold and above 0, LCC will issue warnings.
	CNT11	The warnings should stop as soon as LCC becomes <b>off</b> , <b>standby</b> , <b>overridden</b> or <b>unavailable</b> .

Table 7.8: Structured requirements of LCC - Controlled sub-problem

Phen.	ID	Requirement Description
Offset	CMN1	Driver can determine the offset they wish to have from the centre of the lane when the LCC is <b>active</b> .
	CMN2	Driver chooses the value of the offset through 2 buttons which increase and decrease the offset by one unit every time they are pressed.
	CMN3	The offset is within a specific range.

Table 7.9: Structured requirements of LCC - Commanded sub-problem

performed as soon as the LCC detects an error or fault in the system (e.g. when one of the safety margins is 0). These transitions will be formalised as mode events later on in

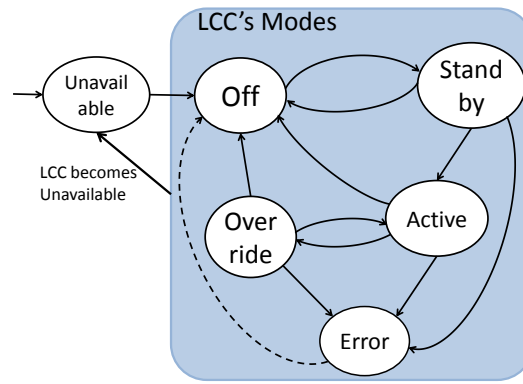


Figure 7.11: Possible mode changes in LCC.

this chapter.

Notice that the dashed arrow in Figure 7.11 represents the assumption that faults will be resolved and the LCC will be reset from the **error** to the **off** state manually. This manual reset is not discussed in this RD. Also notice that there is a transition from every mode (except **unavailable**) to the **unavailable** mode, which means when the ACC is inactivated, the LCC will no longer be available. This is shown using an arrow from the box to the **unavailable** mode. Requirements of the mode sub-problem are shown in Table 7.10.

## 7.7 Stage 2: Formalising LCC using Sub-Models

The proposed approach for horizontal refinement of a control system using composable sub-models, Section 5.6.2, is used here to model the LCC as *monitored* (MNR), *controlled* (CNT), *mode* (MOD) and *commanded* (CMN) sub-models. The MCMC sub-models are defined based on the extended structure of machines in Event-B. Therefore, the MCMC schemas which were proposed in Section 6.6 are used in order to formalise the LCC.

Within each sub-model, the refinement guidelines of Section 5.6.1 are used to introduce the requirements of a sub-problem gradually using refinement steps. The formalisation of the variable and event phenomena is based on the guidelines and patterns of Section 5.7. The formal model of the LCC is given in Appendix A. Note that the model presented in the appendix uses the original machine structure, while the model discussed in this chapter uses the extended machine structure. However, the refinement steps for both structures are similar.

Section 7.7.1 lists the variables shared amongst the MCMC sub-problems of the LCC. This will help us to identify the external variables of every sub-model. Section 7.7.2 briefly discusses examples of the formalisation of event phenomena. The MCMC sub-models of the LCC are represented in Sections 7.7.3 to 7.7.6.

ID	Requirement Description
MOD1	LCC can be switched <b>on</b> and <b>off</b> by the driver.
MOD2	If LCC is <b>off</b> , it will become <b>standby</b> as soon as it is switched on.
MOD3	When LCC is <b>standby</b> , it will become <b>active</b> as soon as the following starting conditions hold (not all conditions are discussed here): <ul style="list-style-type: none"> <li>• The indicator must be off.</li> <li>• No steering torque above the threshold must be detected.</li> <li>• The ACC must be active.</li> </ul>
MOD4	When LCC is <b>active</b> , it will be <b>overridden</b> as soon as either the driver uses the vehicle's indicator or a steering torque above the threshold is detected.
MOD5	If LCC is <b>overridden</b> , it will become <b>active</b> when the resume button is pressed and the starting conditions hold. These conditions include: <ul style="list-style-type: none"> <li>• The indicator must be off.</li> <li>• No steering torque above the threshold must be detected.</li> <li>• The ACC must be active.</li> </ul>
MOD6	If the ACC is not in an active state, the LCC will be <b>unavailable</b> .
MOD7	LCC will be in an <b>error</b> state as soon as it detects an error or when one (or both) of the safety margins is 0.
MOD8	The LCC will become <b>off</b> as soon as it is switched off by the driver.

Table 7.10: Structured requirements of LCC - Mode sub-problem

### 7.7.1 Variable Phenomena Shared among LCC Sub-Problems

The MCMC sub-problems of the LCC were represented in Table 7.7 to 7.10. The variable phenomena shared amongst these sub-problems are shown below in Table 7.11. As mentioned before to avoid redundancy, it is sufficient to fill in the cells above the main diagonal.

From the requirement CNT1 we can conclude that the controller sub-problem refers to *lateral position* and *road curvature*, which are monitored phenomena, as well as the commanded variable phenomenon *offset*. Thus, the first two phenomena are added to the cell where the MNR row and the CNT column intersect, while the last one is added to the cell in the CNT and CMN intersection. The remaining of the requirements are checked in the same way to identify the shared phenomena.

Sub-Models	MNR	CNT	MOD	CMN
MNR	-	Lateral position, Road curvature, Yaw angle, Yaw rate, Speed	Indicator, Steer, ACC status	<i>NON</i>
CNT	-	-	Mode	Offset
MOD	-	-	-	Mode
CMN	-	-	-	-

Table 7.11: Variables shared among the MCMC sub-models of the LCC.

This table will be used to identify the phenomena which should be modelled as external variables.

### 7.7.2 Identifying and Formalising MCMC Event Phenomena of LCC

Based on the structured RD of the LCC, we can identify the MCMC event phenomena. Generally speaking, at least one event for every variable phenomenon is required. Examples of event phenomena in the LCC are as follow:

- **Monitor event phenomena:** Update car speed, Update position, Update road curvature. Each event phenomenon will modify one monitored variable phenomenon nondeterministically and abstractly.
- **Control event phenomena:** Update target path, Update steering angle, Issue warning, Stop warning, Set display colour.
- **Mode event phenomena:** Switch on, Activate, Resume, Override. These are some of the transitions which were illustrated in Figure 7.11.
- **Command event phenomena:** Increase offset, Decrease offset. The value of the offset can be modified by the driver through these event phenomena.

To formalise an MCMC event we use the patterns which were proposed in Section 5.7. An example of a formalised event is shown in Figure 7.12. This event models the requirement MOD3 in Table 7.10. *Grd1* and *act1* in Figure 7.12 show that the variable mode which is modelled as *status* becomes **active** from **standby**. The conditions under which *Activate* event can take place depend on the monitored phenomena *indicator*, *steer* and *ACC status*.

As will be discussed in Section 7.7.5, the mode event *Activate* belongs to the mode (MOD) sub-model. However, the monitored phenomena which are shown in its guards are variables of the monitored (MNR) sub-model. Therefore, based on the extended

```

event Activate
  where
    @grd1 status = Standby
    @grd2 ACC_status = ACC_Active
    @grd3 indicator = FALSE
    @grd4 steer = FALSE
  then
    @act1 status := Active
end

```

Figure 7.12: The mode event *Activate* in the MOD sub-model.

structure of machines and the MCMC schemas which were explained in the previous chapter, *indicator*, *steer* and *ACC\_status* are external variables which will be imported by the MOD sub-model.

Note that the example of Figure 7.12 represents the *Activate* event in the most concrete level. However, we will use refinement techniques to introduce variables in a stepwise manner. This is discussed in the remainder.

### 7.7.3 Monitored Sub-Model of LCC

The MNR sub-model of the LCC is the formal representation of its sub-problem. Figure 7.13 shows the refinement steps in this sub-model. It also shows the phenomena and requirement IDs which are modelled at each step. As we model monitored variable phenomena abstractly and nondeterministically, their formalisation is straightforward. Therefore, in this sub-model we introduced several phenomena at every refinement level. Note that the detection of errors and their handling by the LCC are not modelled, as we do not consider fault tolerance.

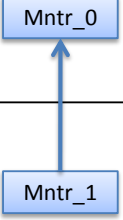
MNR sub-model	Introduced Phenomena	Formal Variables	Req ID	Examples of Event-B Model
	1. Lateral position 2. Roadway curvature 3. yaw angle 4. yaw rate 5. Speed	<ul style="list-style-type: none"> <li>• lateralPos</li> <li>• roadCurve</li> <li>• yawAngl</li> <li>• yawRate</li> <li>• speed</li> </ul>	MNR1, MNR2, MNR3, MNR4, MNR5	<pre> <b>event</b> UpdateLateralPosition   <b>any</b> lp   <b>where</b>     @grd1 lp <math>\subseteq</math> LP   <b>then</b>     @act1 lateralPos := lp   <b>end</b> </pre>
	1. Indicator 2. Steer 3. ACC status	<ul style="list-style-type: none"> <li>• indicator</li> <li>• steerApplied</li> <li>• ACC_status</li> </ul>	MNR6, MNR7, MNR8, MNR9	<pre> <b>event</b> UpdateSteerApplied   <b>any</b> str   <b>where</b>     @grd1 str <math>\in</math> BOOL   <b>then</b>     @act1 steerApplied := str   <b>end</b> </pre>

Figure 7.13: Refinement steps in the monitored sub-model of the LCC.

At the most abstract level, machine *Mntr\_0*, the monitored phenomena corresponding to the requirements MNR1 to MNR5 are introduced. The remaining monitored phenomena,

namely, *indicator*, *steering* and *status of the ACC*, and their requirements (MNR6 to MNR9) are introduced to the model in the first refinement.

An example of a monitored event called *UpdateLateralPosition* which updates the variable *lateralPos* is illustrated in machine *Mntr\_0* in Figure 7.13. This event corresponds to the requirement MNR1 in Table 7.7. The variable *lateralPos* is modelled as a set which estimates values of the lateral position variable within a certain interval. This means *lateralPos* is assigned to a set of possible values rather than a signal value.

Figure 7.14 provides an overview of the most concrete machine in the MNR sub-model. The structure of this machine is defined based on the extended structure and the MNR schema. As shown this machine exports all monitored variables, while no variable is imported. This is usually the case, since the role of the MNR sub-model is to sense the environment and supply the controller with a view of the environment.

```

MACHINE  Mntr_1
SEES    ..
VARIABLES  lateralPos, roadCurve, yawAngl,
               yawRate, speed, indicator, steerApplied, ACC_status
EXPORTS VARIABLES  lateralPos, roadCurve, yawAngl,
               yawRate, speed, indicator, steerApplied, ACC_status
INVARIANTS  ..
EVENTS
    event INITIALISATION
        ..
    end
    event UpdateLateralPosition
        ..
    end
    event UpdateSteerApplied
        ..
    end
    ...
END

```

Figure 7.14: An overview of the monitored (MNR) sub-model of the LCC.

#### 7.7.4 Controlled Sub-Model of LCC

The CNT sub-model of the LCC is the formal representation of its sub-problem. Figure 7.15 shows the refinement steps in the CNT sub-model.



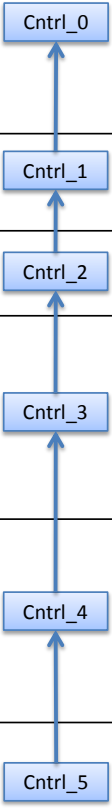
CNT sub-model	Introduced Phenomena	Formal Variables	Req ID	Examples of Event-B Model
 Cntrl_0	1. Target path Shared phenomena: 1. Lateral position 2. Road curvature 3. Offset	• targetPath • lateralPos • roadCurve • offset	CNT1	<pre> event UpdateTargetPath   any pos rc   where     @grd1 pos ∈ lateralPos     @grd2 rc ∈ roadCurve   then     @act1 targetPath :=       TargPathFunc(pos ↦ rc ↦ offset)   end </pre>
Cntrl_1	1. TP safety margin Shared phenomena: 1. Speed	• safeMrgTarg • speed	CNT3	<pre> event UpdateTargetPath extends UpdateTargetPath   then     @act2 safeMrgTarg := SMTargFunc(speed)   end </pre>
Cntrl_2	1. Predicted path 2. PP safety margin Shared phenomena: ...	• predictPath • safeMrgPred	CNT2, CNT4	...
Cntrl_3	1. Steering angle Shared phenomena: 1. Mode	• steeringAngle • status	CNT5, CNT6, CNT7	<pre> event UpdateSteeringAngle   where     @grd1 status = Active     @grd2 safeMrgTarg &gt; 0     @grd3 safeMrgPred &gt; 0   then     @act1 steeringAngle :=       SteerAnglFunc(targetPath ↦ predictPath)   end </pre>
Cntrl_4	1. Warning	• warning	CNT10, CNT11	<pre> event SetWarning_NoTP   where     @grd1 status = Active     @grd2 warning = FALSE     @grd3 safeMrgTarg &lt; SMTargThreshold     @grd4 safeMrgTarg &gt; 0   then     @act1 warning := TRUE   end </pre>
Cntrl_5	1. Display unit colour	• displayClr	CNT8, CNT9	<pre> event Set_Warning_Display_NoTP   extends SetWarning_NoTP   then     @act2 displayClr := YELLOW   end </pre>

Figure 7.15: Refinement steps in the controlled sub-model of the LCC.

Based on the refinement guidelines we aim to model the main controlled phenomenon at the abstract level of this sub-problem. We identified *steering angle* as the main phenomena, since it captures the main control behaviour of the LCC (correcting the steering angle). However, the value of steering angle depends on the target path, predicted path and their safety margins. Thus, before introducing the steering angle, we formalise the paths and their safety margins. The formalisation of these four phenomena, i.e. target path (TP), predicted path (PP), TP safety margin and PP safety margin, can be done in any order, since these phenomena are independent of one another.

As shown in Figure 7.15 at the most abstract level, machine *Cntrl\_0*, target path is introduced by defining the variable *targetPath* and an event named *UpdateTargetPath* which modifies this variable. According to the requirement CNT1 in Table 7.8 the value of the target path depends on the monitored variables *lateralPos* and *roadCurve*, as well as the commanded variable *offset*. Since these variables belong to sub-models other than CNT, they are modelled as external variables which are imported by the CNT sub-model. A function called *TargPathFunc* which returns a target path for all possible values of *lateralPos*, *roadCurve* and *offset* is also defined.

In the MNR sub-model, *lateralPos* was defined as a set of possible positions for the car. However, the *TargPathFunc* function returns the target path based on one lateral position at every time. To model this, the parameter *pos* which is an element of *lateralPos* is defined. The same is true for the parameter *rc* which captures a single value of the set *roadCurve*.

The target path safety margin is added at the first refinement, *Cntrl\_1*, by extending the event *UpdateTargetPath* and defining a variable named *safeMrgTarg*. Based on the requirement CNT3, this variable is a function of the car speed. Therefore, a function called *SMTargFunc* which determines the safety margin is defined. Note that speed is a monitored variable, so it is imported by the CNT sub-model as an external variable. In the second refinement level the predicted path and its safety margin are modelled in a similar style as was discussed for the target path and its safety margin.

At the third refinement, machine *Cntrl\_3*, the requirements CNT5, 6 and 7 are modelled. To do this a variable called *steeringAngle* and an event called *UpdateSteeringAngle* are defined. The value of *steeringAngle* is determined according to the predicted and target paths. This is shown by the function *SteerAnglFunc*. Also, the event *UpdateSteeringAngle* can happen only when the safety margins of both paths are greater than 0 (*@grd2*, *@grd3*). The remaining controlled phenomena and their requirements are modelled in *Cntrl\_4* and *Cntrl\_5*.

Figure 7.16 is given in order to provide the reader with an overview of the most concrete machine in the CNT sub-model. The machine represented in this figure is defined based on the extended structure and the CNT schema which was illustrated in Figure 6.23.

### 7.7.5 Mode Sub-Model of LCC

The MOD sub-problem which was represented in Table 7.10, is also modelled as a distinct sub-model. In this sub-model which is shown in Figure 7.17, we define a formal variable, called *status*, to model the variable phenomenon mode in the LCC.

Abstractions of the transitions on the mode variable, shown in the state machine of Figure 7.11, are modelled in the most abstract level (machine *Mode\_1* in Figure 7.17). As an example of an abstract mode event, the *Activate* event captures two cases of activation firstly from **Standby** and secondly from **Overridden**. At this level this event represents the requirements MOD3 and MOD5 in Table 7.10.

The conditions under which the mode may change are added in refinement steps. However, these conditions involve monitored variable phenomena. Therefore, they are modelled in the MOD sub-model as external variables. For instance based on the requirements MOD3 and MOD5 activating the LCC requires the ACC to be active. This

```

MACHINE  Cntrl_5
SEES    ..
VARIABLES  targetPath, safeMrgTarg, predictpath,
              safeMrgPred, steeringAngle, warning, displayClr
IMPORTS VARIABLES  lateralPos, roadCurve, offset,
              yawAngl, yawRate, speed, status
INVARIANTS  ..
EVENTS
  event INITIALISATION
    ..
  end
  event UpdateTargetPath
    ..
  end
  event UpdateSteeringAngle
    ..
  end
  ...
END

```

Figure 7.16: An overview of the controlled (CNT) sub-model of the LCC.

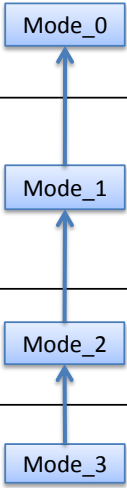
MOD sub-model	Introduced Phenomena	Formal Variable/Parameter	Req ID	Examples of Event-B Model
 Mode_0	1. Mode	• Variable: status	Abstraction of all transitions	<pre> <b>event</b> Activate   <b>where</b>     @grd1 status ∈ {Standby, Overridden}   <b>then</b>     @act1 status := Active   <b>end</b> </pre>
Mode_1	Shared phenomenon: 1. Status of ACC	• External variable: ACC_status	MOD3, MOD5, MOD6	<pre> <b>event</b> Activate <b>refines</b> Activate   <b>where</b>     @grd1 status = Standby     @grd2 ACC_status = ACC_Active   <b>then</b>     @act1 status := Active   <b>end</b> </pre>
Mode_2	Shared phenomenon: 1. Indicator	• External variable: indicator	MOD3, MOD4, MOD5	<pre> <b>event</b> Activate <b>extends</b> Activate   <b>where</b>     @grd3 indicator = FALSE   <b>end</b> </pre>
Mode_3	Shared phenomenon: 1. Steering	• External variable: steerApplied	MOD3, MOD4, MOD5	<pre> <b>event</b> Activate <b>extends</b> Activate   <b>where</b>     @grd4 steerApplied = FALSE   <b>end</b> </pre>

Figure 7.17: Refinement steps in the mode sub-model of the LCC.

condition is modelled using the external variable *ACC\_status* in the guard *@grd2* of the *Activate* event in *Mode\_1*.

Notice that in *Mode\_1*, the *Activate* event is divided into two events, one representing the *activation* of the LCC from **Standby** to **Active** mode (requirement MOD3) and another the *resume* from **Overridden** to **Active** (requirement MOD5). The second event is not shown here. The remaining conditions under which the mode can change are defined in the refinement steps. Also Figure 7.18 represents an overview of the most concrete machine in the MOD sub-model.

```

MACHINE  Mode_3
SEES    ..
VARIABLES  status
EXPORTS VARIABLES  status
IMPORTS VARIABLES  ACC_status, indicator, steerApplied
INVARIANTS  ..
EVENTS
    event INITIALISATION
        ..
    end
    event Activate
        ..
    end
    ...
END

```

Figure 7.18: An overview of the mode (MOD) sub-model of the LCC.

### 7.7.6 Commanded Sub-Model of LCC

The refinement chain of CMN sub-model which is shown in Figure 7.19 formalises the CMN sub-problem of Table 7.9.

As Figure 7.19 represents, the variable phenomenon *offset*, is modelled in the abstract level *Cmnd\_0*. The command event *IncreaseOffset* is also modelled at this level. This event models the requirements CMN1 and CMN2 in Table 7.9. However, CMN1 also refers to the mode variable phenomenon. Since mode belongs to the MOD sub-problem, it is imported by the CMN sub-model. Then the guard *@grd2* in *IncreaseOffset* is defined to show that the offset can change only when the LCC is active (CMN1).

Figure 7.20 represents an overview of the most concrete machine in the CMN sub-model.

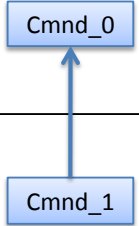
CMN sub-model	Introduced Phenomena	Formal Variable	Req ID	Examples of Event-B Model
	1. Offset	• offset	CMN2, CMN3	<pre> <b>event</b> IncreaseOffset   <b>where</b>     @grd1 offset &lt; offsetUB   <b>then</b>     @act1 offset := offset + 1   <b>end</b> </pre>
	1. Shared Phenomenon : Mode	• External variable: status	CMN1	<pre> <b>event</b> IncreaseOffset <b>extends</b> IncreaseOffset   <b>where</b>     @grd2 status = Active   <b>end</b> </pre>

Figure 7.19: Refinement steps in the commanded sub-model of the LCC.

```

MACHINE Cmnd_1
SEES ..
VARIABLES offset
EXPORTS VARIABLES offset
IMPORTS VARIABLES status
INVARIANTS ..
EVENTS
  event INITIALISATION
    ..
  end
  event IncreaseOffset
    ..
  end
  ...
END

```

Figure 7.20: An overview of the commanded (CMN) sub-model of the LCC.

## 7.8 Stage 4: Composing Sub-Models of LCC

The composition of the MCMC sub-models of the LCC is explained in this section. To do this we use the shared-event composition style which was discussed in Section 6.5.

Section 7.8.1 describes the composition of events. Here examples of the composition of shared events (events which belong to two or more sub-models) as well as local events (events which belong to a single sub-model) are provided. After this in Section 7.8.2 the composition of the most concrete machines of the sub-models is described.

### 7.8.1 Composing Shared Events

When sub-models are composed their events are included in the composed model. Events which are local to sub-models remain unchanged in the composed model, while events which are shared amongst the sub-models are composed according to the shared-event composition techniques.

As an example the event *UpdateLateralPosition* is local to the MNR sub-model. This event is defined in the composed model as below, which means it is the same as the *UpdateLateralPosition* event in *Mntr\_1* (the most concrete level in the MNR sub-model).

$$\text{Comp.UpdateLateralPosition} \triangleq \text{Mntr\_1.UpdateLateralPosition}$$

As an example of a shared event, the composition of a control event and a mode event is shown below.

$$\begin{aligned} \text{Comp.StopWarning\_DisplayOff\_Override\_Indicator} &\triangleq \\ \text{Cntrl\_5.StopWarning\_DisplayOff} \parallel \text{Mode\_3.Override\_Indicator} \end{aligned}$$

Figure 7.21 represents the control event *StopWarning\_DisplayOff* in *Cntrl\_5* (the most concrete level in the CNT sub-model) and the mode event *Override\_Indicator* in *Mode\_3* (the most concrete level in the MOD sub-model). The former event models the requirements CNT9 and CNT11 of Table 7.8 by resetting the warning signal and switching off the display unit, while the latter event models a part of the requirement MOD4 by overriding the mode variable, i.e. *status*, when the indicator is in use. In the composed event the guards of the control event *StopWarning\_DisplayOff* and the mode event *Override\_Indicator* are conjoint, while their actions take place simultaneously.

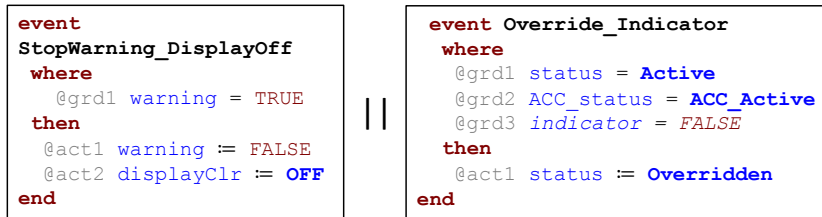


Figure 7.21: *StopWarning\_DisplayOff* and *Override\_Indicator* can be composed.

Events *StopWarning\_DisplayOff* and *Override\_Indicator* can synchronise, since the actions in *StopWarning\_DisplayOff* can occur simultaneously to updating the LCC mode to the *overridden* state. So, when the composed event happens as well as *status*, the controlled variables *warning* and *displayClr* will be updated. Thus, the composed event represents the requirements MOD4, CNT9 and CNT11.

Another example of event composition is shown in Figure 7.22. Here the control event *StopWarning\_DisplayOff* and the mode event *SwitchOff\_FromActive* synchronise. Note that in order for the composition to be valid we defined two switch off events, one

of which is *SwitchOff\_FromActive*. This is because the *StopWarning\_DisplayOff* can be composed with a switch off event which sets the mode from **active** to **off**, since the guard  $warning = TRUE$  can hold only when  $status = Active$ .

Composing the event *StopWarning\_DisplayOff* with any mode event whose guards represent  $status \neq Active$  will result in an event with contradictory guards. Therefore, the composed event will never be enabled. This was discussed in Section 5.7.6 where we argued that a combined event should be reachable.

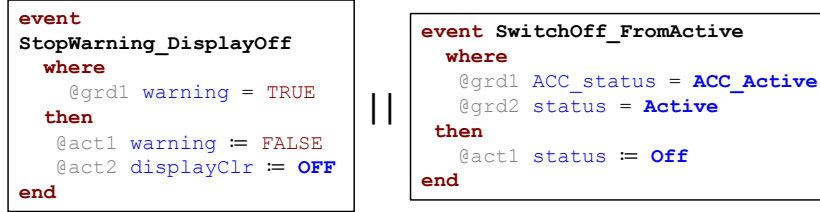


Figure 7.22: *StopWarning\_DisplayOff* and *SwitchOff\_FromActive* can be composed.

### 7.8.2 Composing Machines

The MCMC sub-models of the LCC are composed based on the techniques of Section 6.5.3 which discussed composition of machines with extended structures. The composition techniques are similar to the original shared-event composition, although when machines with extended structures are composed their *imported variables* and *exported variables* clauses will be eliminated, since all variables are internal to the composed model, which means there is no need for importing or exporting variables. This results in the structure of the composed machine to be the same as the original structure of an Event-B machine.

Overviews of the most concrete machines of the MNR, CNT, MOD and CMN sub-models of the LCC were respectively shown in Figures 7.14, 7.16, 7.18 and 7.20. The composition of these machines is shown in Figure 7.23. This figure represents that the variables of the MCMC sub-models are composed to create the state space of the composed machine. In addition, events of the composed machine are defined based on the events of the sub-models as was discussed in the previous section.

Figure 7.23 also represents that it is possible to introduce cross-cutting invariants in the composed machine. For instance the invariant  $warning = TRUE \Rightarrow status = Active$  which refers to the controlled variable *warning* and the mode variable *status* can be proved in the composed model. Note that it was possible to define this invariant in the CNT sub-model, since *warning* is an internal variable of this sub-model and the variable *status* is imported by CNT. However, it is only after the composition that cross-cutting invariants will be proved. This is because imported variables, e.g. *status*, cannot be updated by the machine which imports them.

```

MACHINE  Comp
SEES    ..
VARIABLES  lateralPos, roadCurve, yawAngl, yawRate,
               speed, indicator, steerApplied, ACC_status, targetPath,
               safeMrgTarg, predictpath, safeMrgPred, steeringAngle,
               warning, displayClr, status, offset
INVARIANTS
    @inv warning = TRUE  $\Rightarrow$  status = Active
    ..
EVENTS
    event INITIALISATION
        ..
    end
    event UpdateLateralPosition
        ..
    end
    event UpdateTargetPath
        ..
    end
    event UpdateSteeringAngle
        ..
    end
    event StopWarning_DisplayOff_Override_Indicator
        ..
    end
    ...
END

```

Figure 7.23: An overview of the composed model of the LCC.

## 7.9 Discussions

This section provides a brief discussion on the results of the case study. In Section 7.9.1 we discuss that formalising an RD as composeable sub-models can simplify the modelling process. Section 7.9.2 represents that the MCMC approach can be extended by defining a new category of phenomena called *derived phenomena*.



### 7.9.1 Advantages of Formalising Sub-Problems as Composeable Sub-Models

The usage of sub-problem composition can facilitate the formalisation process. The main reason is that when modelling a sub-problem, the number of its requirements is less than the overall number of requirements, which means the modelling process can become manageable.

Comparing formalising sub-problems as sub-models by using (de)composition and refinement techniques to formalising the entire RD using only refinement shows that the number of variables and events captured in every sub-model is lessened.

Furthermore, the refinement chain in a sub-model can be constructed in a more deterministic manner. This is because the number of refinement steps when sub-problems are modelled is less than the number of refinement steps when the entire RD is formalised.

In addition, the refinement order when the entire RD is modelled is arbitrary, since at every level of refinement there are many requirements to choose from. Modelling an RD as sub-models means less number of refinement steps in every sub-model. Using the refinement guidelines and patterns can also help order the refinement chain in a deterministic manner.

Formalising an RD as sub-models also means the number of proof obligation in each sub-model is less than the overall number of proof obligations. Thus, proof complexity can be dealt with in a step-wise manner. Another advantage is that sub-models are modularised. This will localise changes and encapsulate the formal representation of a set of requirements in a sub-model.

### 7.9.2 Derived Phenomena: A New Category of Phenomena

In the case study of the LDWS we categorised the earliest warning line (EWL) as a controlled phenomenon. This decision was mainly because we assumed that the component responsible for determining the EWL was part of the controller. However, it is possible to assume that the EWL is provided to the controller by a component which is not part of the controller.

A similar decision was taken in the LCC where the target path, the predicted path and their safety margins were defined within the controller. This was shown in Figure 7.9. However, this figure can be modified as illustrated by Figure 7.24. In other words, the system might be designed in a way that the path generator will not be a part of the controller, but an intermediate component which processes the environment inputs before providing the controller with some values.

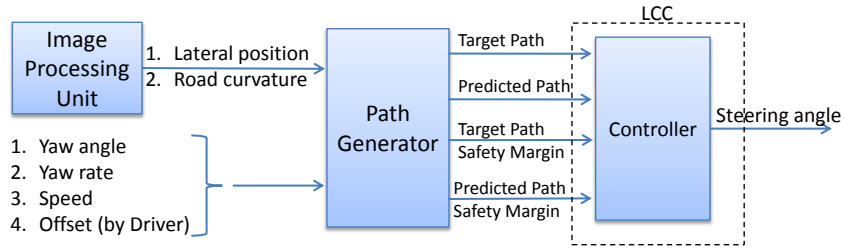


Figure 7.24: An alternative overview of LCC to Figure 7.9.

This view suggests there might be a need for a new category of phenomena which represent *derived phenomena*. These are phenomena whose values are determined based on the MCMC phenomena of a control system. This means derived phenomena can be defined in a separate component as a combination of the MCMC variables. As will be discussed in Chapter 9, in the future work the MCMC guidelines can be expanded to include derived phenomena.

## 7.10 Conclusion

In this chapter, two case studies were discussed, a lane departure warning system (LDWS) and a lane centering system (LCC), both of which were supported by our industrial partner.

The requirement document of the LDWS was structured and its formal model was discussed briefly. After this, the LDWS model was validated against its structured requirement document. The LDWS was formally modelled using one chain of refinement, rather than modelling its MCMC sub-problems as composable sub-models. This is because in this case study we focused mostly on the first and the last stages (structuring an RD and validating the model) of the proposed four-stage MCMC approach.

However, in the LCC case study, the focus was on the first three stages, i.e., structuring the LCC RD, formalising it as sub-models, and composing the sub-models. In the formalisation stage, we used the extended structure of machines that was proposed in the previous chapter. The extended structure simplified the formalisation of the LCC as sub-models, since variables shared among the sub-models were imported by sub-models, which required read-only access to the variables.

In the formalisation of both systems, refinement was used to introduce the requirements to the model gradually. To do this, the refinement guidelines of Section 5.6.1 were used. In the LCC case study, the complexity of the modelling process was reduced further by mapping the MCMC sub-problems to their corresponding MCMC sub-models. In addition, the event and the machines in each sub-model of the LCC were defined, based on the event patterns of Section 5.7 and the MCMC schemas of Section 6.6.1.



## Chapter 8

# Vertical Refinements and Decomposition of Control Systems

In this chapter, an approach for decomposing a control system based on its intended architecture is discussed. The overview of the components of a decomposed control system is explained in Section 8.1. However, in order to decompose a control system it is necessary to define design details, namely, sensors, actuators, buttons and a communication bus. The communication bus used in this research is a Controller Area Network (CAN) bus, which is explained in Section 8.2.

Section 8.3 provides an overview of the approach of vertical refinement of a control system. To formalise design details, we define tasks which are sequences of events, and a cycle variable. These concepts are discussed in Section 8.4.

The approach for formalising sensors, actuators, buttons and a CAN bus is explained in Section 8.5. This approach is applied to a case study of a simplified cruise control system in Section 8.6. Discussions on the vertical refinement steps are provided in Section 8.7.

The approach for decomposing a control system is defined in Section 8.8. This approach is applied to the case study in Section 8.9. A discussion on the decomposition is given in Section 8.10. Finally, we discuss advantages of the proposed approaches and our future work in Section 8.11.

Notice that not all aspects of vertical refinement can be generalised. This is because vertical refinement steps of a control system depend on its architecture and different control systems have different architecture. As an example, the communication bus, CAN, that is modelled in this thesis, is mainly used in the automotive industry, while other communication buses might be used in the aviation. Further discussions can be found in Section 9.6.

## 8.1 Overview of a Control System Decomposition

Decomposing the model of a control system provides a method for differentiating the software and the hardware components and identifying the implementable components. Model decomposition also helps with extracting the *software requirements* from the *system requirements*. This means the formal specification of the software can be derived from the overall system specifications.

In order for a control system model decomposition to be meaningful, it is necessary to decompose the model based on the *system architecture*. Since model decomposition based on the system architecture results in individual models which reflect the individual components of the architecture. To do this we use the shared-event decomposition style which was discussed in Section 4.4.2. This style of decomposition is chosen over the shared-variable style, as it is suitable for message-passing in distributed systems which is similar to the architecture of a control system. Generally speaking, a typical architecture of a commanded control system can consist of the following components:

- **Environment:** This is the physical environment of a control system (excluding the operator). Examples are the vehicle indicator in an LDWS or the car speed in a CCS. The environment contains monitored and controlled phenomena.
- **Sensor:** This is the means through which a controller observes the state of the environment. An example is a speed sensor.
- **Controller:** This is the component which is responsible for the calculations and decision making process. An example is a cruise controller.
- **Actuator:** The decisions of a controller are received by the environment through actuators. An example is actuating the acceleration in the CCS.
- **Operator:** An operator can request a specific action to take place by the controller. A driver is an example of an operator in automotive control systems such as a CCS and a LDWS. In this work we do not consider the behaviour of an operator in detail.
- **User interface:** This provides a means for an operator to interact with a controller. In this thesis we consider the formalisation of buttons as a form of UI. Examples of buttons are switch on or reset.
- **Communication infrastructure:** As the above components are usually distributed, they communicate through some forms of a bus. An example of a bus in automotive systems is CAN (Control Area Network).
- **Global Timer:** This provides a way of synchronisation between the components.

However, before decomposing a control system model into the above components it is important to incorporate the design details to the model. A formal model of a requirement document needs to be refined further to introduce *sensors*, *actuators*, *user interface* (buttons), the *communication bus* and the *timer* to the model. Thus, a realisation and understanding of the design details, such as message prioritisations, is necessary.

We refer to these refinement steps as *vertical refinements*, since they close the gap between the model and the real implementation of the system. In the remainder of this chapter we will focus on the vertical refinement steps and provide guidelines and patterns for introducing *design details*. After that we will discuss the model decomposition of a control systems into sensors, actuators, buttons, a controller, a communication bus and a timer.

Note that we distinguish between horizontal and vertical refinements as it provides a better understanding of the modelling process. However, the formalisation notations and the generated proof obligations in both refinements are exactly the same.

## 8.2 Controller Area Network Bus

The communication bus chosen in this research is the Controller Area Network (CAN). This bus is popular in automotive as it provides a “simple, efficient and robust communications for in-vehicle networks” [DBBL07]. CAN which was developed by BOSCH [Gmb91] is a serial bus communications protocol. It can be used in real-time applications for communication between sensors, actuators, controllers and other nodes.

In the automotive industry, the use of software and embedded control systems have grown. This is mainly because software allows designing more intelligent and safer cars. The first software-based solutions were local and isolated. They had dedicated controllers (Electronic control units or ECUs), sensors and actuators. However, communication buses allowed components to connect to one another. Therefore, functionalities can be distributed over ECUs which are connected. As an example a speed sensor ECU and a cruise controller ECU can be connected to a CAN bus in order for their applications (*tasks*) to transmit messages on the bus.

More details are given on the CAN bus in the remainder. Section 8.2.1 will discuss that every message is made of a frame which consists of a data field and overhead as well as some control fields. In Section 8.2.2, we will explain that any node can request to transmit a message on the CAN bus and therefore, the CAN protocol is designed to resolve conflicts by listening to the network. Section 8.2.3 discusses the simplifications made in order to model the CAN bus.

### 8.2.1 A Message Frame in CAN

A message frame in CAN which is shown in Figure 8.1 [JTN05], begins with the start-of-frame (SOF) bit. Then there is a 11-bit identifier which establishes the priority of the message during the bus access. The lower the binary value of an identifier, the higher the message priority. Message identifiers are predefined and every identifier is unique in the entire system.

There is also a control field which indicates how many bytes of data exist in the data field. The data field may contain zero to eight bytes. This field is followed by the cyclic redundancy check (CRC) which is used for error detection. All receivers of a message will acknowledge the consistency of the message (or flag out any inconsistency) by setting the acknowledgement (ACK) field. Finally, the end of a message frame is shown by the end-of-frame (EOF) field.

SOF	Identifier	RTR	Control	Data	CRC	ACK	EOF
-----	------------	-----	---------	------	-----	-----	-----

Figure 8.1: A message frame in CAN.

### 8.2.2 Arbitration in CAN

CAN resolves access conflicts of ECUs using a bit-wise arbitration mechanism on message identifiers. When the CAN is idle, any node is able to start transmitting a message. Any node which requires to access the CAN bus will be a transmitter which means it starts sending its message bit by bit. However, only one node wins the arbitration and only this node's message will be transmitted on CAN.

The arbitration mechanism is that each transmitter will transmit a bit and then it will listen to the CAN bus. If the transmitter detects an equal value to the transmitted bit, it will continue with the consecutive bit. However, if it detects a dominant value (0) on the bus, while it is sending a recessive value (1), then the transmitter will stop and it will become a receiver.

A receiver is a node which either has lost the arbitration, or does not require access to the CAN and thus is not sending any message. Receiver nodes can filter the messages coming through the CAN bus based on the whole identifier of the messages. This means an ECU will receive messages which are of interest to it.

The arbitration mechanism of the CAN bus guarantees that when two or more messages are being sent, the message with the lowest identifier will be transmitted first. Thus, priority of every message should be decided when designing a system.

### 8.2.3 Modelling CAN: Simplifications and Assumptions

The CAN bus has several methods for error detection, such as the CRC field. The error handling in CAN is that when an error is detected the message will be retransmitted immediately and automatically. In this work we have assumed an error-free control system. Thus, the error detection and handling are not discussed.

Also, we focus on modelling the *effect* of CAN. Therefore, details such as a message frame and the bit-wise arbitration are not considered. The effect of the CAN bus is modelled by taking into account the priority of a message (the identifier) and its content (the data field).

We also categorise ECUs as a transmitter, a receiver or both. This will help with modelling the buffers of an ECU, since if an ECU can transmit a message, it will have a transmitter buffer (*Tx buffer*) for that message. In addition, if an ECU is a receiver of a message, it will have a receiver buffer (*Rx buffer*) for that message. Different case studies have shown that generally speaking:

- user interfaces (or buttons) and sensors are *transmitters*;
- controllers can be both *transmitters* and *receivers*;
- and actuators are *receivers*.

Figure 8.2 shows a button, a sensor, an actuator and a controller ECU which are connected via the CAN. These ECUs can request to broadcast their messages. As mentioned, according to the CAN protocol the message with the lowest identifier will be chosen. The communication directions represent whether an ECU is a transmitter, a receiver or both.

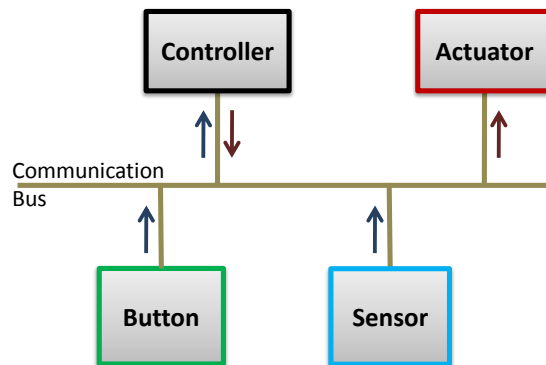


Figure 8.2: ECUs can be transmitters, receivers or both.



### 8.3 Overview of Formalising Design Details

Before decomposing a model of a control system it is necessary to formalise its *design details*. This allows localising the states of components and formalising their state exchange. An overview of the approach for introducing design details to a model of a control system is shown in Figure 8.3. As illustrated the starting point for the vertical refinement is the most concrete model, called *Composed Model*, of the horizontal refinement chain where all modelable requirements are introduced.

Introducing design details in one step can make the modelling process complex. Therefore, refinement techniques are used to elaborate the model in a step-wise manner. As Figure 8.3 represents in the first vertical refinement untimed monitored variables are refined to *timed variables*. The introduction of timed monitored variables allows us to take delays between the environment and the controller into account. This refinement step is explained further in Section 8.5.1.

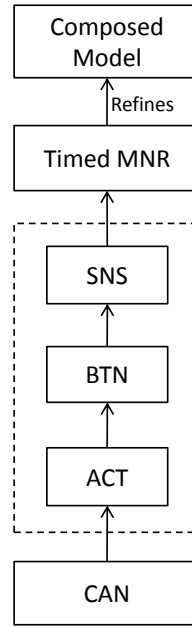


Figure 8.3: An overview of vertical refinement.

Sensors (SNS), buttons (BTN) and actuators (ACT) are then introduced to the model. Sensors are defined to sense a monitored variable. Buttons provide the means for requesting the occurrence of an operator mode event or a command event in the controller. Actuators set controlled phenomena based on controller decisions.

The dashed box around these three refinement levels represents the fact that they can be incorporated to the model in any order, e.g., buttons may be introduced before sensors. The refinement patterns for introducing sensors, buttons and actuators are discussed in Sections 8.5.2, 8.5.3 and 8.5.4 respectively. After introducing sensors, buttons and

actuators, the priorities between messages will be defined using a CAN bus. This is explained in Section 8.5.5.

Note that Figure 8.3 is an overview of the refinement order between the components. In other words every box in Figure 8.3 may incorporate several refinement steps. For instance, a modeller might use several refinement steps to introduce timed monitored variables, or a model might contain several sensors each of which can be modelled in a single refinement step.

## 8.4 Tasks, Cycle Variable and their Patterns

This section explains two concepts which are used in vertical refinement patterns. The first is *task* which is explained in Section 8.4.1 and the second which is described in Section 8.4.2 is *cycle variable*.

### 8.4.1 Tasks and their Events Sequences

To introduce the design details we model a control system as a collection of *tasks* where every task belongs to an ECU. Tasks are cyclic computations with a specific *periodicity*. A task must take place once and only once within its cycle. For instance, a task such as T with the periodicity 10 means that T should happen once and only once every 10 time units.

We model an ECU task as a *sequence of events* which happen in a particular order within the task cycle. A task consists of a set of *start events* ( $S = s_1, \dots, s_n$ ) and a set of *end events* ( $E = e_1, \dots, e_n$ ). In addition, it may contain some *middle events* ( $m_1, \dots, m_n$ ) which can happen only after a start event and before an end event occur.

A task is *enabled* when one of its start events such as  $s_i \in S$  is enabled. A task is *complete* when one of its end events such as  $e_i \in E$  occurs. Also, the execution of the middle events of a task should not cause deadlock or divergence. In other words, for every task it should always be possible to get to an end event from any of the start events.

The periodicity or cycle of a task specifies that the task must be enabled and completed within its cycle once and only once. Task cycles are determined based on timing analysis, such as worst-case execution time. We assume such timing analysis are done and task periods are long enough to allow for the following:

- a *receiver task* to receive messages and act upon them;
- a *transmitter task* to do the necessary calculations and transmit its results;

- a *receiver and transmitter task* to receive messages, act upon those messages and transmit its results.

We categorise tasks of a control system into *environment*, *sensor*, *button*, *controller* and *actuator tasks*. Environment tasks simulate the behaviour of the environment, and are not part of the final implementation of a control system. Environment tasks consist of monitored variables and monitor events, since monitored phenomena provide a picture of the environment. The cycle or periodicity of an environment task depends on the rate of change of its monitored variables and events. The higher the rate of the change, the lower the cycle of the environment task.

However, for simplicity we define the cycle of an environment task as the greatest common division (GCD) of all tasks cycles. This allows to have a coarse-grained timing approach which is discussed further in Section 8.7.1. As an example consider a control system which has a sensor task, a button task and a controller task with cycles 10, 10 and 20 respectively. The cycle of the environment task for this control system can be 10.

The sensor, button, controller and actuator tasks represent tasks corresponding to their ECUs. An ECU may contain one or several tasks, although in this thesis we have assumed that every ECU has a single task. For instance, a system with 4 sensors will contain 4 sensor tasks. Examples of tasks are given using a case study in Section 8.6.2.

#### 8.4.2 Patterns for Cycle Variable and Events

To abstract away from the timing and scheduling of ECUs and CAN, the design details are modelled using a *cycle-based approach*. This style of modelling is inspired by the cycle-based techniques used in simulation [PP95]. In this approach the effect of the system is observed at every cycle. To do this a *cycle variable* which is an abstract representation of the global timer is defined. This variable models the time synchronisation between ECUs.

We also define *cycle events* which increase the cycle variable by the greatest common division (GCD) of all tasks cycles. For instance, consider the cyclic tasks A, B and C with cycles 10, 10 and 20 respectively. As a reminder a cycle 10 for A means that A occurs once and only once every 10 time units. The cycle variable which is defined for the tasks A, B and C will be increased by 10 time units at every occurrence of a cycle event.

By progressing the cycle variable based on the GCD, we observe effects of tasks at time points. In other words, the events which happen within a *time range* are mapped to a *time point*. This means we are not interested in the exact moment every event happens, but we observe their effects and orders at time points. As an example, events of task

A that happen any time between 0 and 10 will be observed at the time point 10, and again events which happen between the time 10 and 20 will be mapped to the time point 20. The advantage of this style of modelling is that a fine-grained model of time is abstracted away. Therefore, we focus mainly on functionalities of tasks, the order of their occurrences and the communication between tasks.

The cycle variable progresses when tasks that should take place during a particular cycle are completed. For instance, in the case of tasks A, B and C when the effect of A and B are observed at the time point 10 (i.e. an end event of A and an end event of B happen), the cycle variable can progress to the point 20. Task C may occur at either the time point 10 or 20, but it must be completed before the cycle variable progresses to 30. Note that we assume that task cycles are longer than the duration of tasks. For instance, in the given example we assumed that it is possible for both tasks A and B to be completed within 10 time units.

As a guideline we suggest defining  $k$  cycle events for a system which consists of several tasks whose greatest cycle equates to  $k \times GCD$ . We define a cycle event for every set of tasks which can be completed by  $i \times GCD$  where  $i \in 1, \dots, k$ . Note that we aim to have at most one cycle event enabled at every moment in time. This is because the cycle variable can increase only once before other events of the system take place. Also, cycle events interact with *start* and *end* events of tasks to model whether a task has completed. This is shown in detail in the future sections.

Based on this guideline in the example of the system containing tasks A, B and C, we define two cycle events, since  $2 \times GCD$  is 20 which is the greatest task periodicity for A, B and C. The first cycle event represents the scheduling of A and B whose events should take place every  $1 \times GCD$  (10) time unit. Therefore, when the behaviour of A and B are observed at time point 10 (the behaviour at this time point represents the events between the time 0 to 10) the cycle variable can increase from 10 to 20. The first cycle event therefore is responsible for time increase from 10 to 20 or from 30 to 40 and so on. The second cycle event is defined for tasks which are completed within  $2 \times GCD$ . In this case when A, B and C are complete the cycle can progress, e.g. from 20 to 30, or from 40 to 50.

## 8.5 Guidelines and Patterns for Vertical Refinement

This section provides patterns and guidelines for vertical refinement steps. These patterns are influenced by the style of decomposition which is shared-event style. Section 8.5.1 explains the monitored variable refinement pattern. Sections 8.5.2 to 8.5.5 respectively discuss the refinement patterns which are used to introduce sensors, buttons, actuators and the CAN bus. A discussion on the events order in vertical refinements is given in Section 8.5.6.

### 8.5.1 Monitored Variable Refinement Pattern

In this section we first look at refining monitored variables to introduce time. This will allow us to express the environment task using the cycle variable. After that the dependencies between the MCMC event and variable phenomena are shown based on the timed monitored variables.

#### 8.5.1.1 Introducing Timing to Monitored Variables

Chapters 5 and 6 assumed that during horizontal refinements monitored variable phenomena which represent the environment were modelled as untimed variables. However, to introduce design details we refine untimed monitored variables as functions of time. Gluing invariants are used to connect timed monitored variables to their counterpart untimed variables.

As an example, in order to vertically refine a CCS, the variable *actualSpd* which represents the set of actual car speed will be replaced by a timed variable called *speed*. The definitions of these variables are as follow:

$$actualSpd \subseteq 0..maxCarSpd$$

$$speed \in 0..cycle \rightarrow 0..maxCarSpd \text{ (where } cycle \text{ represents the current time)}$$

The gluing invariant which relates the untimed variable *actualSpd* to the timed variable *speed* is shown below. This invariant shows that the untimed variable represents the set of car speed values for a specific portion of time, i.e. between 10 time units prior to the current time and the current time.

$$actualSpd = speed[cycle - 10..cycle]$$

Defining monitored phenomena as timed variables will allow us to firstly model the delay between sensors and the physical environment. This is because monitored variable phenomena are environment quantities whose values change as time progresses. In other words, at a moment in time the value of a monitored variable (in the environment) and its sensed value (in the controller) may be different. Secondly, this style of modelling helps us reason about the time range within which sensed values can be received by the controller. As will be shown in the case study of a simplified CCS in Section 8.6, invariants on the time range of a sensed monitored variable are defined.

### 8.5.1.2 MCMC Events based on Timed Monitored Variable

Table 8.1 updates the dependencies between the MCMC event and variable phenomena which were represented in Chapter 5, Table 5.7. This table reflects the step where un-timed monitored variables, i.e., the set  $MNR$ , are refined to timed monitored variables, i.e., the set  $MNR_t$ . As an example, a monitor event which modifies timed monitored variable phenomena ( $MNR_t$ ), depends on controlled variable phenomena ( $CNT$ ) as well as  $MNR_t$ . Note that the modified  $MNR_t$ , which is represented in the last column, captures monitored variable phenomena when time progresses. In other words, this column represents after-values of  $MNR_t$ , while the ‘Depends on’ column represents the before-values for  $MNR_t$ .

Event Phenomenon	Depends on	Modifies
Monitor Event	$MNR_t$ CNT	$MNR_t$
Control Event	$MNR_t$ CNT mode CMN	CNT
Mode Event	$MNR_t$ CNT mode CMN	mode
Command Event	$MNR_t$ mode CMN	CMN

Table 8.1: The MCMC event and variable phenomena based on timed monitored variables.

An example of a monitor event in Event-B based on the above table is shown in Figure 8.4. This event which is called *UpdateSpeed* updates the timed monitored variable *speed*.

```

event UpdateSpeed
  any spd
  where
    @grd1 spd ∈ (cycle+1..cycle+5) → (0..maxSpd)
  then
    @act1 speed := speed ∪ spd
  end

```

Figure 8.4: An example of the timed speed variable.

Note that we model controlled, mode and commanded variable phenomena as *untimed* variables. In the case of controlled phenomena, this is because we assume these phenomena will be updated by actuators immediately after the controller sets their values. This means the value of a controlled phenomenon will be set in the controller and it will be updated in the environment within one cycle. Also, we do not model mode and commanded variables as functions of time, since they are controller phenomena which means the controller accesses them locally and delay is not correlated to these variables.

### 8.5.2 Sensor Refinement Pattern

To model sensors we suggest defining at least one *sense event* for every timed monitored variable (TMV) phenomenon. As shown in Figure 8.5, the role of a sense event is to sample a TMV in order to produce a relevant sensed value. The sensed value is then stored as a *sensed variable* which will eventually be transmitted to the controller. A sense event might simply copy the value of a TMV into a sensed variable, or perform a function on the TMV before sensing it. Examples of  $F(TMV)$  are rounding a monitored variable or sensing its absolute value.

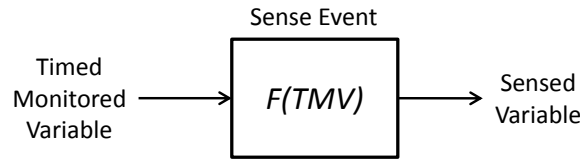


Figure 8.5: Sense events.

In this thesis we model one sensor per monitored variable, although the approach is extendible if modelling several sensors for a monitored variable is required, for example when reliability of the sensed value is very important. In such cases, an event which fuses the sensed values of the sensors can be introduced. The output of the fuse event is the sensed value which will be transmitted to the controller.

A sensed event is shown in Table 8.2. Here,  $MNR_t$  represents the set of all timed monitored variables, while  $SNS$  represents the set of all *sensed variables*. As a guideline, when introducing sensors at least one sensed variable for every monitored variable should be defined.

Based on the definition of a sense event, the dependencies of the MCMC events which were shown in Table 8.1 are modified to reflect that the references to monitored variables in the control, mode and command events are replaced by the corresponding sensed variable. These modifications are shown in Table 8.2.

Note that when modelling the MCMC events formally in a step-wise manner, it is necessary to prove that the MCMC events of Table 8.2 refine the events of Table 8.1. In other words, when a formal language such as Event-B is used, the MCMC events are

Event Phenomenon	Depends on	Modifies
Monitor Event	$MNR_t$ CNT	$MNR_t$
Sense Event	$MNR_t$	SNS
Control Event	SNS CNT mode CMN	CNT
Mode Event	SNS CNT mode CMN	mode
Command Event	SNS mode CMN	CMN

Table 8.2: MCMC event and variable dependencies when introducing sensors.

constructed using refinement and proof techniques. Thus, at every step where design details are introduced we prove that the behaviour of the system is correct with respect to the model at the previous step.

### 8.5.3 Button Refinement Pattern

In this section the means of interaction between an operator and a controller is defined through buttons. We suggest defining *button events* to capture interactions between an operator and a set of physical buttons in a user interface. A button event modifies a *button variable*, which represents the state of a button, based on operator requests. Button events and variables are required in order to refine *command events* or *operator mode events*.

An example of a button event is a *Press* event which modifies a button variable from the state *not-press* to the *pressed* state. Note that our aim is not to provide a detailed model of a physical button, but we want to capture all possible states of a button through which an operator commands the controller to perform a specific functionality.

Table 8.3 represents a button event which modifies the set *BTN*. This set represents all buttons through which a system operator can interact with the controller. Modelling buttons will result in modifying the dependencies of the MCMC variable and event phenomena. The modifications are shown in Table 8.3 where the set *BTN* is added to the definition of an operator mode event and a command event.



As shown in Table 8.3 introducing buttons means that operator mode events (OMEs) and controller mode events (CMEs) can be distinguished, since the former can be affected by the set  $BTN$ , while  $BTN$  has no influence on the latter. As a reminder CMEs are mode events which are modified by the controller based on the environment and controlled quantities (i.e. MCMC variables), so an operator request has no effect on a CME.

Event Phenomenon	Depends on	Modifies
Monitor Event	$MNR_t$ CNT	$MNR_t$
Sense Event	$MNR_t$	SNS
Control Event	SNS CNT mode CMN	CNT
Button Event	BTN	BTN
Operator Mode Event	BTN SNS CNT mode CMN	mode
Controller Mode Event	SNS CNT mode CMN	mode
Command Event	BTN SNS mode CMN	CMN

Table 8.3: MCMC event and variable dependencies when introducing buttons.

When introducing a button to a model, we also suggest defining extra events to capture the following:

- What should happen when the state of the button is not modified, such as when a button is not pressed? It might be the case that the controller does nothing, or in some cases the controller might perform default actions. For instance, if a controller detects an error and it is not switched off manually, an automatic switch off action might take place.
- What should happen when the button state changes, but the conditions under which the controller should respond to the operator request do not hold? An

example is when a button is pressed, but the controller detects an error at the same time, thus it may decide to ignore the button press.

The points above show that it is important to model situations where either an event cannot occur because it has not been requested by the operator or a response to an operator request is not feasible. An example of this is shown in Section 8.6.3.2. Note that we use refinement in the Event-B language to model a button and its extra events. The refinement steps prove that the introduced buttons do not change the system behaviour. This is proven by showing that the events of Table 8.3 refine the events of Table 8.2.

#### 8.5.4 Actuator Refinement Pattern

In this section actuators are defined to update the environment quantity of controlled variable phenomena. To introduce an actuator to a model, we suggest to define an *actuator event* and an *actuator variable*. As shown in Figure 8.6, the role of an actuator event is to update a controlled variable based on its corresponding actuator variable. We suggest to define at least one actuator event for every controlled variable.

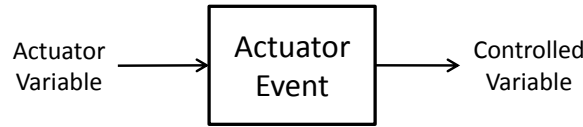


Figure 8.6: An actuator event.

An actuator event is represented in Table 8.4. Here,  $ACT$  represents the set of all actuator variables, and  $CNT$  captures the set of all controlled variables. Based on this definition, the dependencies of control and mode events can be modified as shown in Table 8.4. So control and mode events will depend on  $ACT$  rather than  $CNT$ . Note that  $ACT$  is a controller quantity, while  $CNT$  is an environment quantity. Also note that as will be discussed in Section 8.5.6, actuator events refine control events, while new control events which refine skip are defined.

One assumption that we made here is that the actuator event happens only after the control event has taken place. Therefore, an order between these two events should be defined. Also, note that when introducing actuators to a model in Event-B, an actuator event refines a control event and a new control event which is internal to the controller is defined. This is discussed further in Section 8.5.6.

#### 8.5.5 CAN Bus Refinement Pattern

As discussed the communication bus that is introduced in this work is a CAN bus. However, only an abstraction of this bus is formalised to allow a message with the

Event Phenomenon	Depends on	Modifies
Monitor Event	$MNR_t$ CNT	$MNR_t$
Sense Event	$MNR_t$	SNS
Control Event	SNS ACT mode CMN	ACT
actuator Event	ACT	CNT
Button Event	BTN	BTN
Operator Mode Event	BTN SNS ACT mode CMN	mode
Controller Mode Event	SNS ACT mode CMN	mode
Command Event	BTN SNS mode CMN	CMN

Table 8.4: Control event when introducing actuators.

minimum id among the messages which have requested transmission to be broadcast. Thus, details such as bit arbitration are not modelled in this work. A CAN bus is introduced to the model after formalising transmitter and receiver ECUs. To model the bus, we firstly define *intermediate events* which formalise the interactions between a controller and other ECUs (sensors, buttons and actuators). The second step is to introduce the *prioritisation of messages* based on the CAN protocol.

To explain the first step do this, consider the example of interactions between ECUs as shown in Figure 8.7. As illustrated, every ECU has local buffers whose values are used by the ECU task in order to perform its actions. In addition, an ECU can be either a transmitter, or a receiver, or both. For instance in Figure 8.7, ECU1 is a transmitter with a Tx buffer, while ECU2 is a receiver with an Rx buffer.

An example of the data flow in Figure 8.7 is that ECU1 generates a value which is stored in the ECU1 local buffer will be copied to the transmitter buffer. This buffer will

try to transmit the data as a message on the CAN bus. Based on the priority of the message, it may be transmitted immediately or with some delay. If there is a delay in the transmission, the message might be overwritten by the task of ECU1.

When the message of ECU1 is transmitted on the CAN bus, ECU2 will receive the message on its receiver (Rx) buffer. The value of the message will then be transmitted to the local buffer of the ECU2. It is based on the values of its local buffers that the ECU2 will perform its actions.

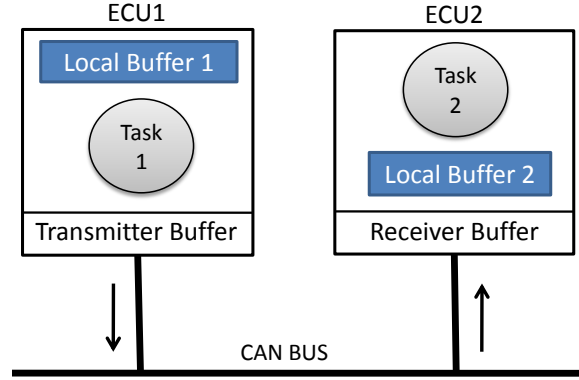


Figure 8.7: Transmission of a message between ECUs.

Based on the explained data flow, to model a message transmission from an ECU to another via CAN, the following intermediate steps are suggested:

1. **Buffering a message for transmission:** In this step a message which is prepared by an ECU task is buffered in the transmitter (Tx) buffer. In other words, the message of the ECU will be copied from its local buffer to Tx. This means the message is ready for transmission.
2. **Transmitting the message:** The transmission takes place by copying the message with the highest priority from a Tx buffer to receiver buffers (Rx) of all ECUs which should receive the message.
3. **Buffering the message locally:** The ECUs, which have received the message in their Rx buffers, will store a copy of the message in their local buffers. The receiver ECUs will then make their decisions based on the values of the messages which are locally stored.

After defining the above intermediate events the dependencies of MCMC events can be modified as shown in Table 8.5. This table represents the component to which the set of variable phenomena MNR, CNT, SNS and ACT belong to. For instance, a monitor event depends on the timed monitored variables and controlled variables which belong to the environment. These are shown by  $MNR_{t_{env}}$  and  $CNT_{env}$ . However, a sense event depends on the environment quantity of  $MNR_t$ , while it modifies the sensor variables which belong to the sensor itself ( $SNS_{sns}$ ).

As shown, controller quantities are represented with a *cnt* subscript. So,  $SNS_{cnt}$ ,  $BTN_{cnt}$  and  $ACT_{cnt}$  are respectively the sensed, buttons and actuated values internal to the controller. In other words, these values are used in the decision making process by the controller. The gap between these quantities and their counterparts in other components is bridged by the communication bus. For example, a CAN bus bridges the gap between  $SNS_{sns}$ , which belongs to sensor components, and  $SNS_{cnt}$ , which belongs to controller components, by transmitting the value of sensors to the controller. The same is true for  $ACT_{cnt}$  and  $ACT_{act}$  as well as  $BTN_{env}$  and  $BTN_{cnt}$ . Note that *mode* and *CMN* are always controller quantities.

Event Phenomenon	Depends on	Modifies
Monitor Event	$MNR_{t_{env}}$ $CNT_{env}$	$MNR_{t_{env}}$
Sense Event	$MNR_{t_{env}}$	$SNS_{sns}$
Control Event	$SNS_{cnt}$ $ACT_{cnt}$ mode CMN	$ACT_{cnt}$
actuator Event	$ACT_{act}$	$CNT_{env}$
Button Event	$BTN_{env}$	$BTN_{env}$
Operator Mode Event	$BTN_{cnt}$ $SNS_{cnt}$ $ACT_{cnt}$ mode CMN	mode
Controller Mode Event	$SNS_{cnt}$ $ACT_{cnt}$ mode CMN	mode
Command Event	$BTN_{cnt}$ $SNS_{cnt}$ mode CMN	CMN

Table 8.5: MCMC event and variable dependencies when introducing buttons.

When modelling a CAN bus, in addition to modelling message transmissions, it is necessary to define message priorities. As mention we model an abstraction of the CAN protocol. Thus, to model message priorities, we define a function from the id of the messages to a boolean value which represents whether or not a message needs to be

transmitted. An example of a function for a CAN bus which has up to three messages to transmit is  $1..3 \rightarrow \text{BOOL}$ .

### 8.5.6 Events Refinement and their Orders

This section provides an overview of the order of events after introducing vertical refinements of a model. In addition, we discuss refinement relations between the MCMC events and the *sense*, *button*, *actuator* and *CAN events* which are defined during vertical refinements. To do this, we use the diagrammatic notations suggested for decomposing atomicity of events [But09b, But09c].

In these diagrams, which are shown in Figure 8.8, 8.9 and 8.10, every rectangular node represents a single event in an Event-B model. The root nodes denote abstract events, while the leaves denote concrete events. Therefore, this diagram indicates how an abstract event is elaborated in refinement levels. The nodes should always be read from left to right which means this diagrammatic notation also represents the order in which the events are expected to perform.

A dashed line specifies that a leaf event refines *skip*, i.e. a new event is introduced in the refinement. However, a solid line shows that a leaf event refines the root node (abstract event). In these diagrams one execution of the root node corresponds to one execution of all the leaves from left to right.

Figure 8.8 shows the refinement steps and the events order when a sensor is introduced to a model. As illustrated, in a refinement level sense events which sample monitored variables are defined. After that, in another refinement level the CAN bus is introduced by defining *buffer*, *transmit* and *receive* events.

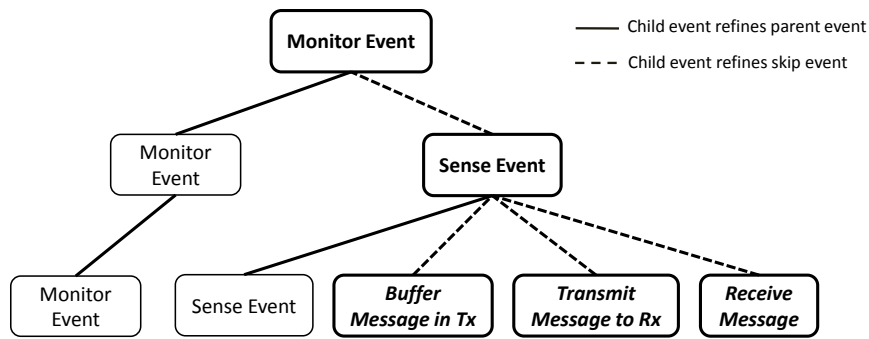


Figure 8.8: Vertical refinement of introducing sensors.

The refinement steps for introducing a button is shown in Figure 8.9. This figure represents the fact that a button event is defined in order for a command event to be informed of the interactions between the operator and the physical button. In another refinement level the value of the button is broadcast as a message via the CAN bus. Note that a command event and an operator mode event (OME) have similar refinement steps.

Thus, a diagram similar to Figure 8.9 which ends with an OME instead of a command event can be defined.

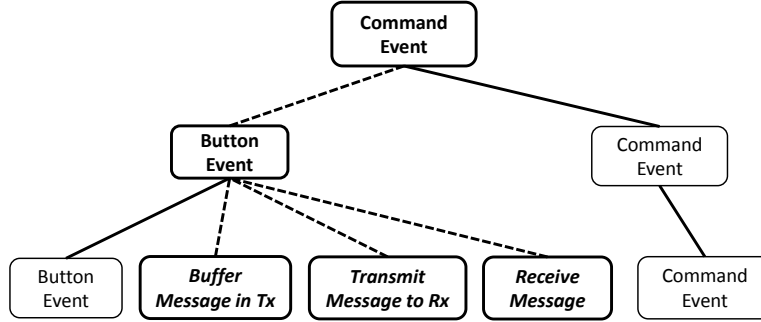


Figure 8.9: Vertical refinement of introducing buttons.

The order of events and the refinement steps for introducing an actuator is shown in Figure 8.10. The refinement step represents the fact that the abstract control event is refined to an actuator event and a control event which belongs to the controller. This refinement indicates that during horizontal refinements, where the abstract control event is defined, a controlled variable is modified by the control event. However, during vertical refinements, it is an actuator event which updates the controlled variable.

The second refinement steps shows that the decision of the control event, i.e. the value of the actuator variable, is transmitted to the actuator event via the CAN bus. Intermediate events namely, *buffer*, *transmit* and *receive* are defined in order to transmit this value.

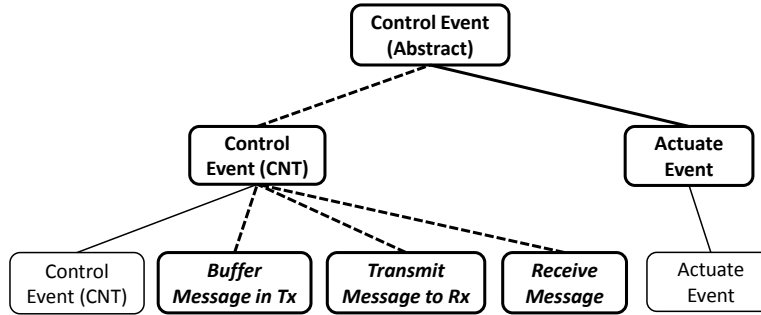


Figure 8.10: Vertical refinement of introducing actuators.

## 8.6 Case Study: Modelling Design Details of a Simplified CCS

In this section the vertical refinement patterns outlined in the previous sections are applied to a simplified cruise control system (SCCS). The overview of the SCCS is given in Section 8.6.1. In Section 8.6.2 the concrete model of the SCCS is explained. The

refinement steps which result in the concrete model are discussed in Section 8.6.3. The Event-b representation of this model can be found in Appendix B.

### 8.6.1 Overview of the SCCS

As a case study, we model a simplified version of the cruise control system requirements. This simplification involves modelling the main role of the cruise control which is to minimise the difference between the actual speed and the target speed. The architecture of the simplified cruise control system (SCCS) is shown in Figure 8.11. The SCCS consists of a *CAN bus* and four ECUs, namely a *speed sensor*, a *button*, an *actuator* and a *controller*. These ECUs and the CAN bus are explained further in the remainder.

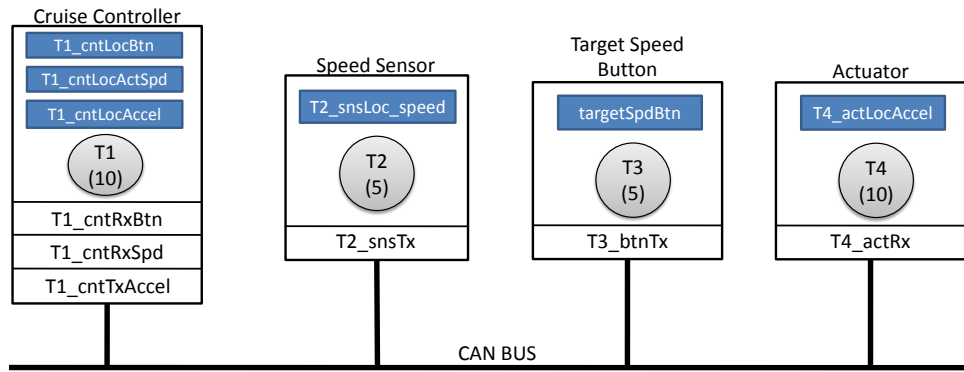


Figure 8.11: Overview of the connection in the simplified CCS.

- **Speed Sensor:** The task of the speed sensor ECU which is called T2 has periodicity 5. This means T2 samples the actual speed of the car and transmits the value every 5 time units. When the sampled value is prepared, it is initially stored in the local buffer of this ECU, called *T2\_snsLoc\_speed*. After that, this value is copied into the transmitter buffer which is shown as *T2\_snsTx*.
- **Button UI:** The button ECU is the means for setting the target speed. This ECU has the periodicity 5 which means it samples the physical set button every 5 time units. T3 is the task of the button ECU. This task updates the local buffer *targetSpdBtn* to indicate whether or not the button is pressed.
- **Controller ECU:** The controller ECU represents the SCCS. This ECU has a task called T1 which is responsible for determining the value of the acceleration based on the actual speed and the target speed. The actual speed will be received from the speed sensor via the CAN bus and it will be stored in the receiver buffer called *T1\_cntRxSpd*. Similarly, the receiver buffer *T1\_cntRxBtn* receives the state of the set button (pressed or not pressed) from the button ECU.

The values of the Rx buffers will be copied to local buffers, called *T1\_cntLocActSpd* and *T1\_cntLocBtn*, where T1 can access in order to make decisions. When T1 decides on the value of the acceleration, this value will be copied from the local buffer



$T1\_cntLocAccel$  to the Tx buffer  $T1\_cntTxAccel$ . This value will be transmitted from this Tx buffer to the actuator ECU.

- **Actuator ECU:** The value of the acceleration which is transmitted from the Tx buffer of the controller ECU will be received by the receiver buffer  $T4\_actRx$  in the actuator ECU. The received value will then be copied to the actuator's local buffer  $T4\_actLocAccel$ . The actuator task, T4, will set the acceleration which is an environment quantity based on the value of its local buffer. Note that we assume that T4 happens as part of the T1 cycle, meaning that T1 can be completed only after the completion of T4.
- **CAN Bus:** The communication bus is defined based on the protocol of a CAN bus as was discussed in Section 8.2. In the CAN the messages have an identifier which determines their priorities. The identifiers that we have chosen for the controller, button and sensor messages are 1, 2 and 3 respectively. This means the controller message which contains the value for actuating the controlled variable has the highest priority. Since it has the lowest identifier.

To model that a message needs to be transmitted, a variable called  $CAN\_msgFlg$  is defined. This variable represents a function from message identifiers to a boolean flag, i.e.,  $CAN\_msgFlg \in 1..3 \rightarrow BOOL$ . When this variable is TRUE for an identifier, it means the corresponding message is available to be transmitted and should be considered for the arbitration mechanism.

### 8.6.2 Concrete Model of the SCCS

The concrete model of the SCCS is explained in this section, while the refinement steps are discussed in Section 8.6.3. To model the SCCS we defined two cycle events which are described in Section 8.6.2.4, and five tasks, which are as follow:

- *an environment task* with a periodicity of 5, which is the GCD of all task cycles. This task is not explained here as it simply contains a monitor event which updates the actual car speed. However, it is discussed in the refinement levels in Section 8.6.3.3.
- *a speed sensor task*, with a periodicity of 5, that belongs to the speed sensor ECU and is explained in Section 8.6.2.1.
- *a set button Task* thorough which the target speed is updated. The formalisation of this task is not discussed as it is similar to the speed sensor task.
- *a controller task* which is the controller ECU task. The periodicity of this task is 10. A detailed discussion of this task is given in Section 8.6.2.2.

- *an actuator task* which belongs to the actuator ECU. Similarly to the controller task, this task has a periodicity of 10. This task is explained in Section 8.6.2.3.

### 8.6.2.1 Speed Sensor Task in SCCS

The speed sensor task (T2) consists of a sequence of three events. This is shown in Figure 8.12 as a state machine. This state machine contains a *start*, a *middle* and an *end event*.

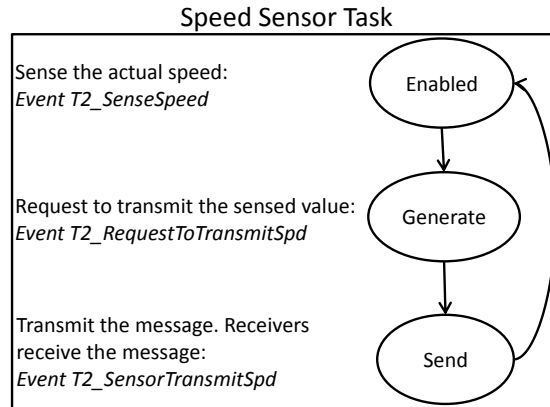


Figure 8.12: Sequence of events in speed sensor task in the concrete level.

The events corresponding to the above state machine are shown in Figure 8.13. Here, the state machine is modelled using the variable *T2\_state*. The states and their events are as follow:

- *Enable*: At this state the speed sensor senses the actual speed and stores the sensed value in its local buffer. This state consist of the start event which is called *T2\_SenseSpeed*. As guard *@grd5* represents a speed is chosen within a time range nondeterministically. Note that speed is a function of time. Then in *@act1* the local buffer of the sensor ECU which is *T2\_snsLoc\_speed* is set to the chosen value of *speed*.
- *Generate*: The sensor generates the sensed value which can be broadcast on the CAN. This state is realised by the event *T2\_RequestToTransmitSpd* in Figure 8.13. In this event, the Tx buffer *T2\_snsTx* is updated (*@act1*) and the *CAN\_msgFlg* flag is set to TRUE (*@act3*). So, this event declares that a message, which is the sensed actual speed, is ready to be transmitted via CAN.
- *Send*: The generated sensed speed will be transmitted and received by ECUs which require the car speed in order to make decisions. The end event corresponding to this state is shown as *T2\_SensorTransmitSpd* in Figure 8.13. As the guard *@grd5* in this event shows the message of the speed sensor can be transmitted only if this

message has the highest priority. In the case of the speed message this means no other messages are being sent on the CAN bus simultaneously.

Note that variable *msgCounter* has been defined to count the number of messages which will be broadcast on the CAN at every cycle. As will be discussed later, this allows us to define an invariants which imposes an upper bound on the number of messages on the CAN.

<pre> event T2_SenseSpeed refines ...   any spd c   where     @grd1 c ∈ ℕ     @grd2 c = cycle     @grd3 T2_senseEvaluated = FALSE     @grd4 spd ∈ 0..maxSpd     @grd5 spd ∈ speed[c-4..c]     @grd6 T2_state = Sns_EN   then     @act1 T2_snsLoc_speed := spd     @act2 T2_state := Sns_GENR   End </pre>	<pre> event T2_RequestToTransmitSpd   refines ...   any spd   where     @grd1 spd ∈ 0..maxSpd     @grd2 spd = T2_snsLoc_speed     @grd3 T2_state = Sns_GENR   then     @act1 T2_snsTX := spd     @act2 T2_state := Sns_SEND     @act3 CAN_msgFlg(3) := TRUE   end </pre>	<pre> event T2_SensorTransmitSpd refines ...   any canTrans   where     @grd1 canTrans ∈ 0..maxSpd     @grd2 canTrans = T2_snsTX     @grd3 T2_state = Sns_SEND     @grd4 dom(CAN_msgFlg &gt; {TRUE}) ≠ ∅     @grd5 min(dom(CAN_msgFlg &gt; {TRUE})) = 3   then     @act1 T1_cntRxSpd := canTrans     @act2 T2_senseEvaluated := TRUE     @act3 T2_state := Sns_EN     @act4 msgCounter := msgCounter + 1     @act4 CAN_msgFlg(3) := FALSE   end </pre>
---	--	--

Figure 8.13: The concrete model of the speed sensor task (T2).

Message transmissions and their reception by other ECUs happen synchronously. This is shown by the guard *@grd2* and the action *@act1* in the event *T2\_SensorTransmitSpd*. As shown in *@act1*, the value of the car speed is copied from the speed sensor Tx buffer to Rx buffers of any receiver ECU.

The action *@act2* in *T2\_SensorTransmitSpd* which sets the variable *T2\_senseEvaluated* to TRUE represents the end of the sequence. Also, the guard *@grd3* in *T2\_SenseSpeed* shows that the sequence can begin only when *T2\_senseEvaluated* = FALSE. This flag gets reset by the cycle events which update are discussed later on.

We modelled the set target speed button ECU similarly to the speed sensor ECU. The message of the button task (T3) represents whether or not the target speed set button is pressed. Thus, we defined this message as a boolean variable where TRUE (FALSE) denotes that the set button is (not) pressed. An event similar to *T2\_SensorTransmitSpd* which transmits the state of the button is also defined.

### 8.6.2.2 Controller Task in SCCS

Figure 8.14 represents the controller task (T1) as a state machine with a sequence of events. The state *enabled* contains the starting event *T1\_ReceiveEnvVal*, while the end event is called *T1\_TransmitAcceleration*, whose occurrence indicates the completion of T1. There are also four middle events which connect the start event to the end event. However, notice that in every task execution of T1, only one event from the *target speed set* state can occur.

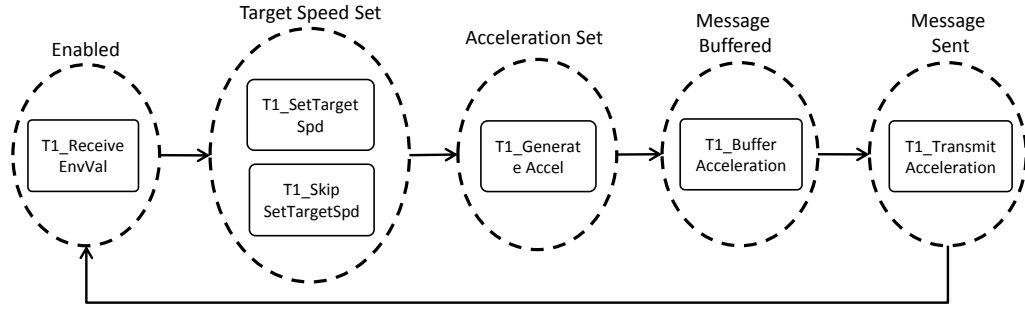


Figure 8.14: The order of events in the controller task (T1).

The Event-B representations of the above events are shown in Figures 8.15 and 8.16. The order between the events is modelled using the variable *T1\_state*.

The first event, *T1\_ReceiveEnvVal*, copies the messages of the speed sensor and the button ECUs from the controller Rx buffers to its local buffers. This is shown by the guards *@grd3* and *@grd4* as well as the actions *@act2* and *@act4*. Note that the variable *rcvFlg* is introduced to help with defining invariants and proving them automatically.

After receiving the environment values locally, the controller task will either set target speed or skip this step. This is modelled by the events *T1\_SetTargetSpd* and *T1\_SkipSetTargetSpd*. Only one of these events can take place in a single completion of the controller task. In other words, in the state *T1.TS* the controller task can perform different functionalities depending on its observation from the environment. If the set button is pressed the event *T1\_SetTargetSpd* will take place in order to update the target speed (*@grd2* and *@act1*), while if the button is not pressed the event *T1\_SkipSetTargetSpd* will occur.

The next event in the controller task is the control event *T1\_GenerateAccel* which generates the value of the *acceleration*. This value is then stored locally in the *T1\_cntLocAccel* buffer. The event *T1\_BufferAcceleration* copies this value from the local buffer to the Tx buffer. In the end event *T1\_TransmitAcceleration* the acceleration value is transmitted via the CAN bus in order for the actuator task (T4) to receive it. To show that the actuator task takes place immediately after T1 the flag *T4\_actEvaluated* is reset.

### 8.6.2.3 Actuator Task in SCCS

The role of the actuator task (T4) is to update the environment quantity, *acceleration*, based on the decision of the controller.

The Event-B representations of T4 is shown in Figure 8.17. This task consists of two events. The first which is called *T4\_ReceiveAcceleration* copies the value of the *acceleration* from the Rx buffer to the task local buffer. The second event called

<pre> event T1_ReceiveEnvVal refines ... any Rx2 Rx1 where   @grd1 Rx1 ∈ B00L   @grd2 Rx2 ∈ 0..maxSpd   @grd3 Rx1 = T1_cntRxBtn   @grd4 Rx2 = T1_cntRxSpd   @grd5 T1_cntEvaluated = FALSE   @grd6 T1_state = T1_EN then   @act1 T1_state := T1_TS   @act2 T1_cntLocActSpd := Rx2   @act3 rcvFlg := TRUE   @act4 T1_cntLocBtn := Rx1 end </pre>	<pre> event T1_SetTargetSpd refines ... any spd where   @grd1 spd ∈ lb..ub   @grd2 T1_state = T1_TS   @grd3 T1_cntLocBtn = TRUE then   @act1 cnt_targetSpd := spd   @act2 T1_state := T1_ACCEL end  event T1_SkipSetTargetSpd refines ... where   @grd1 T1_state = T1_TS   @grd2 T1_cntLocBtn = FALSE then   @act1 T1_state := T1_ACCEL end </pre>	<pre> event T1_GenerateAccel refines ... where   @grd1 T1_state = T1_ACCEL then   @act1 T1_cntLocAccel :=     Accel_Func(T1_cntLocActSpd ↦       cnt_targetSpd)   @act3 T1_state := T1_BFR end </pre>
--	--	---

Figure 8.15: The concrete model of the controller task (T1).

<pre> event T1_BufferAcceleration refines ... any accel where   @grd0 T1_state = T1_BFR   @grd2 accel ∈ N   @grd3 accel = T1_cntLocAccel then   @act1 T1_cntTxAccel := accel   @act3 CAN_msgFlg(1) := TRUE   @act4 T1_state := T1_SEND end </pre>	<pre> event T1_TransmitAcceleration refines ... any accel where   @grd0 T1_state = T1_SEND   @grd2 accel ∈ N   @grd3 accel = T1_cntTxAccel   @grd4 dom(CAN_msgFlg ▷ {TRUE}) ≠ ∅   @grd5 min(dom(CAN_msgFlg ▷ {TRUE})) = 1 then   @act1 T4_actRxAccel := accel   @act2 T4_actEvaluated := FALSE   @act4 CAN_msgFlg(1) := FALSE end </pre>
---	--

Figure 8.16: The final two events in the concrete model of T1.

*T4\_ActuatingAcceleration* updates the controlled variable *acceleration* based on the actuator local buffer.

In addition, the end event *T4\_ActuatingAcceleration* resets the flags *T1\_cntEvaluated* as well as *T4\_actEvaluated* to represent that T1 and T4 happen in one cycle. This event also increases the message counter (*msgCounter*) by 1 in order to represent that within this cycle one message may be transmitted.

Note that T4 happens after the completion of T1. This is modelled by using the flag *T4\_actEvaluated* which is set by T1.

<pre> event T4_ReceiveAcceleration refines ... any accel where   @grd1 T4_actEvaluated =     FALSE   @grd2 T4_state = T4_EN   @grd3 accel ∈ N   @grd4 accel = T4_actRxAccel then   @act1 T4_actLocAccel := accel   @act2 T4_state := T4_ACT end </pre>	<pre> event T4_ActuatingAcceleration refines ... any accel where   @grd1 T4_state = T4_ACT   @grd2 accel ∈ N   @grd3 accel = T4_actLocAccel then   @act1 acceleration := accel   @act2 T1_cntEvaluated := TRUE   @act3 T4_actEvaluated := TRUE   @act4 T1_state := T1_EN   @act5 T4_state := T4_EN   @act6 msgCounter := msgCounter+1 end </pre>
--	--

Figure 8.17: The concrete model of the actuator task (T4).

#### 8.6.2.4 Cycle Events in SCCS

In order to model the synchronisation between the tasks, *cycle events* which update the *cycle variable* are defined. Section 8.4.2 discussed guidelines and patterns for introducing a cycle variable and its corresponding event. The cycle variable defined for the SCCS is increased by 5, which is the GCD of all the tasks periodicities.

The guidelines of Section 8.4.2 also indicated that we need  $k$  number of cycle events where  $K \times GCD$  is the greatest periodicity. Therefore, in SCCS we model two cycle events which are shown in Figure 8.18.

The first cycle event which is called *UpdateCycle1* occurs every 5 time units. Thus, the tasks with periodicity 5, i.e. environment, T2 and T3, must have happened before this cycle event takes place. This is shown by the guards *@grd2*, *@grd3* and *@grd4* which define that the environment, T2 and T3 tasks have been completed. As shown in the actions of *UpdateCycle1*, the task flags (*env\_evaluated*, *T2\_sensEvaluated*, *T3\_btnEvaluated*) are reset to indicate that these tasks should take place in the next cycle.

The second cycle event which is called *UpdateCycle2* occurs every 10 time units. Therefore, in addition to the environment task, T2 and T3, the controller task (T1) should take place. The guard *@grd1* and the action *@act3* show that the controller task has been completed at every 10 time units. Note that the design of the system ensures that when the controller task completes the actuator task also completes. Therefore, it is not necessary to include *T4\_actEvaluated* flag.

To distinguish the completion of tasks a variable called *counter* is defined. This variable can be replaced by the *mod* operation, since  $counter = 1$  is equivalent to  $cycle \bmod 10 = 5$ , and  $counter = 0$  is equivalent to  $cycle \bmod 10 = 0$ . However, using *mod* can complicate the proof obligations. Thus, we use the variable *counter*.

Also note that the variable message counter, *msgCounter*, is reset to 0 in the cycle events. This is because there is an upper bound on the number of messages which can be sent within a cycle. At the start of a cycle this number is 0, and it will increase as messages are generated by tasks.

The cycle events schedule the tasks in a way that they are completed within their periodicity. For instance the controller and the actuator tasks (T1 and T4) take place once and only once every 10 time units, as their periodicity is 10. However, the periodicities of environment, sensor (T2) and button tasks (T3) are 5. Therefore, while within a 10 time unit T1 and T4 occur once, T2 and T3 happen twice.

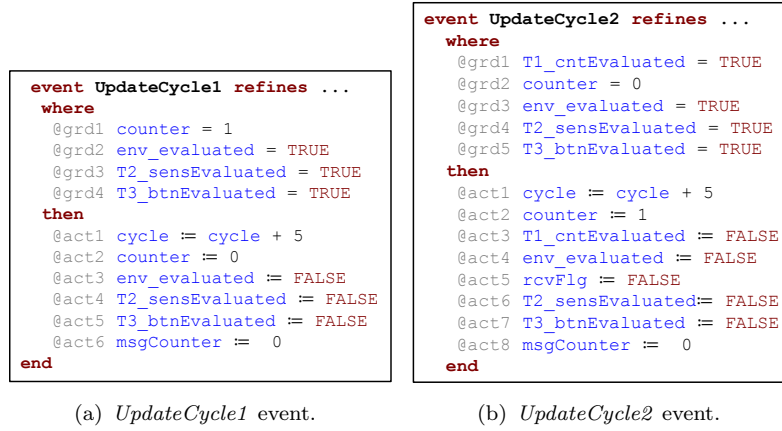


Figure 8.18: Updating cycle variable in the concrete level.

### 8.6.3 Refinement Steps of the SCCS

The refinement steps for introducing design details to the SCCS model are discussed in this section. These steps are ordered based on Figure 8.3. The most abstract level, which is discussed in Section 8.6.3.1, represents the requirements of the SCCS. After that Section 8.6.3.2 to 8.6.3.9 explain the design details, including the cycle variable, the timed monitored variable and tasks, which are introduced in refinement steps.

#### 8.6.3.1 Abstract Level

The abstract level includes the model of the requirements of the simplified cruise control system (SCCS). The most abstract level in the vertical refinement is the most concrete level of a horizontal refinement chain. The abstract model of the SCCS represents the property that the acceleration will be determined based on the actual speed and the target speed. So, this model contains a monitored variable called *actualSpd*, a commanded variable called *cnt.targetSpd* and a controlled variable called *acceleration*. The events which update these variables are shown in Figure 8.19.

The monitor event *UpdateSpd.Env* shows that the monitor variable *speed* is defined as a set. This is because the actual speed is a continuous value and this set represents a range of possible values for the actual speed. The command event *T1.SetTargetSpd* updates the target speed nondeterministically. To simplify the example, we assumed that when the driver requests the target speed to be set, the controller will update the target speed to a value between the lower band *lb* and the upper band *ub*.

The control event *GenerateAccel* updates the controlled variable *acceleration*. Similarly to the example of the cruise control system in Chapter 5, the function *Accel.Func* shows that the acceleration is a function of the actual speed and the target speed. At this level this function returns the value of the acceleration based on *cnt.targetSpd* and

an element, such as *actSpd*, from the set *actualSpd*. In later refinement steps the element *actSpd* will be replaced with the sensed value of speed.

Monitor Event	Command Event	Control Event
<pre> event UpdateSpd_Env any s where   @grd1 s ≤ 0..maxSpd then   @act1 actualSpd := s end </pre>	<pre> event SetTargetSpd any spd where   @grd1 spd ∈ lb..ub then   @act1 cnt_targetSpd = spd end </pre>	<pre> event GenerateAccel any actSpd where   @grd1 actSpd ∈ actualSpd then   @act1 acceleration :=     Accel_Func(actSpd ↦ cnt_targetSpd) end </pre>

Figure 8.19: Events of a simplified cruise control system (SCCS).

### 8.6.3.2 First Refinement: Introducing Cycle Variable

In the first refinement the variable *cycle* is defined. In addition, the cycle events which increase the cycle variable by 5, i.e., the GCD of tasks periodicities, are defined.

The events sequence in the controller task (T1) is also defined at this level. To do this, it is necessary to model the possibility that either a command event is requested and is feasible, or it is requested but is not feasible, or it is not requested.

The first case is modelled in the *T1\_SetTargetSpd* event which updates the value of the target speed. We model the latter two cases using an event called *T1\_SkipSetTargetSpd*. This event will not modify the target speed, because either updating the target speed is not feasible (e.g. there is an error), or the driver has not requested (e.g. the message received by the controller from the button task is ‘not pressed’).

### 8.6.3.3 Second Refinement: Introducing Timed Monitored Variable

In the second refinement the untimed monitored variable *actualSpd* is replaced by the variable *speed* which is a function of time, meaning that *speed* captures an estimate of the actual car speed for discrete moments of time. So, the *speed* variable defines a more elaborated structure for the monitored phenomenon car speed than the variable *actualSpd*.

The environment task consists of a single event. This event called *UpdateSpd\_Env* is shown in Figure 8.20. Every occurrence of the event *UpdateSpd\_Env* updates the variable speed with some nondeterministic values of the car speed for a cycle of 5 time units. This is shown in the guard *@grd2* and the action *@act1*.

Note that an environment task and its monitored events can be modelled in more detail. As an example the values of speed in a cycle can be represented as a relation which depends on the speed at the previous cycle. However, an abstract model of the environment is sufficient for our purpose which is to formalise the control system. In addition, an environment task can be elaborated in refinement steps.



```

event UpdateSpd_Env
any spd
where
  @grd1 env_evaluated = FALSE
  @grd2 spd ∈
    (cycle+1..cycle+5) → (0..maxSpd)
then
  @act1 speed := speed ∪ spd
  @act2 env_evaluated := TRUE
end

```

Figure 8.20: The variable *speed* models the monitored phenomenon car speed.

#### 8.6.3.4 Third Refinement: Controller Task Receives Speed

In the third refinement the reception of the speed message by the controller task from the environment task is modelled. Since the speed sensor is not defined yet, the controller receives the car speed directly from the environment. A local buffer in the controller ECU which is called *T1\_cntLocActSpd* is defined to store the received value of speed.

The value of the *T1\_cntLocActSpd* buffer is always the speed from some time during the last 25 time units. This is captured as the invariant below to show that T1's decisions are based on the speed value which is not older than 25 time units prior to the current cycle.

$$T1\_cntLocActSpd \in speed[cycle - 25..cycle]$$

#### 8.6.3.5 Fourth, Fifth and Sixth Refinement: Modelling Speed Sensor Task

In the refinement steps four to six, we define the speed sensor task. As was shown in Figure 8.13, the speed sensor task consists of three events to sense the speed, to broadcast the sensed speed and to transmit it to the controller. These three events are each introduced at one level of refinement. However, the CAN bus is not defined here. This is because other ECUs which use the CAN bus should be formalised before the bus.

The following invariants can be defined at this level. The first invariant ensures that the value received by the controller Rx buffer is always from a recent time range. The second invariant ensures that the sensor always receives a fresh value from the environment.

$$T1\_cntRxSpd\_2 \in speed[cycle - 10..cycle]$$

$$T2\_state = Sns\_GENR \Rightarrow T2\_snsLoc\_speed \in speed[cycle - 4..cycle]$$

#### 8.6.3.6 Seventh Refinement: Modelling Set Button Task

At this refinement level the set target speed button task is defined. This task identifies whether the physical set button is pressed and then it transmits the driver request as a

message. We modelled this message as a boolean flag which is TRUE when the driver has pressed the button and FALSE when the button is not pressed. The controller task will receive this message in its local buffer called *T1\_cntLocBtn* and respond to it by performing either *T1\_SetTargetSpd* or *T1\_SkipSetTargetSpd*.

#### 8.6.3.7 Eighth Refinement: Modelling Actuator Task

In the eighth refinement level the actuator task is defined. To do this the controller task generates the value of the acceleration locally. This value is stored in a local buffer in the controller task, called *T1\_cntLocAccel*. The value of this buffer will then be transmitted as a message to the actuator task via the CAN bus. When the actuator receives the value of the acceleration, it will update the controlled variable *acceleration* which is an environment quantity.

#### 8.6.3.8 Ninth Refinement: Introducing Maximum Number of Messages on CAN

In this refinement level we introduce a message counter variable, called *msgCounter*. This variable represents the maximum number of messages which can be transmitted by the CAN bus in a single cycle. An invariant as below has been specified to ensure that the number of messages on the CAN are never above its upper bound.

$$msgCounter \leq MaxNumMSG$$

As shown in the concrete model, Section 8.6.2, the *msgCounter* variable is increased by one every time a task uses the CAN to broadcast a message.

#### 8.6.3.9 Tenth Refinement: Modelling the CAN Bus

At this level of refinement the CAN bus is introduced. Before introducing the CAN bus, the tasks, namely, the speed sensor, the set button and the actuator, transmitted their messages in an arbitrary order. Defining the CAN bus at this level will ensure that messages will be transmitted based on their priority.

Therefore, at this level we make the decision on the priority of the messages. In this example we decided that the message of the controller task (the value of acceleration) has the highest priority. So this message has the lowest identifier, which is 1. This will allow the controller to update the environment as soon as it decides on the value of the acceleration. The set button message has priority over the speed sensor messages. Therefore, the id 2 is given to the button message, while the sensor message has an id of 3.

A variable called *CAN\_msgFlg* which is a function from the id numbers to a boolean variable is defined to model that when a message is ready to be transmitted, this flag will be set to TRUE for the id of the message. Also, a message will be transmitted first if its id is the lowest. A guard such as  $\min(\text{dom}(\text{CAN\_msgFlg} \triangleright \{TRUE\})) = 3$  will ensure that the sensor message can be chosen message only if no other task is sending a message.

#### 8.6.4 Proof Obligations in SCCS

Refining SCCS vertically raised proof obligations (POs), all of which have been discharged either automatically or manually using the interactive prover in Rodin. Table 8.6 provides the statistics of the discharged POs. This table also represents the number of proofs which were discharged using the interactive prover.

As shown, only a small percentage of the POs were discharged manually. These POs were trivial and mostly related to proving a value belongs to a certain domain or range. As an example one of the POs which was not discharged automatically is:

$$0 \in (-25..0 \times \{0\})[0 - 5..0]$$

This is due to the limitations of the automatic prover, as it is unable to identify images of a relation. However, comparing the number of proofs discharged manually and the effort required to the number of automatic proofs shows the using Rodin and its prover can reduce the effort required in generating as well as discharging proof obligations.

Level of Refinement	Total POs	Automatically Discharged	Manually Discharged
Abstract level	7	7	0
First refinement	18	18	0
Second refinement	18	15	3
Third refinement	59	57	2
Fourth refinement	21	17	4
Fifth refinement	17	16	1
Sixth refinement	25	25	0
Seventh refinement	0	0	0
Eighth refinement	51	51	0
Ninth refinement	112	112	0
Tenth refinement	10	9	1

Table 8.6: Statistics of the proof obligations in SCCS.

Also, note that in some cases, additional invariants were introduced to facilitate the automatic proof. For instance, in the ninth refinement to prove that the number of messages in the CAN bus never exceeds the maximum, we defined additional invariants that show the maximum number of messages at different states of the system. Therefore, the number of automatically discharged POs at this level is higher than the remaining levels, as several additional invariants were introduced to assist the automatic proof.

## 8.7 Discussion on Vertical Refinements

In this section discussions on the vertical refinement patterns and assumptions are given. Section 8.7.1 explains that the timer we introduced in this thesis is coarse-grained. Section 8.7.2 represents assumptions that we made in vertical refinements. Finally, Section 8.7.3 shows that vertical refinement step for introducing buttons is different from sensors and actuators.

### 8.7.1 Coarse-Grained Timer and Invariants in Vertical Refinement

In this thesis, we avoid defining the time and timing properties in detail. This is because our main objective is to formalise design details. However, doing so requires defining a notion of timing. Therefore, we used an abstraction approach to model timing as cycle. This allows us to consider the behaviour of a task, its interactions and timing properties at every cycle, rather than at every single time unit. Others, such as [SB11], have developed approaches for defining a fine-grained time in Event-B.

In addition, in the SCCS case study we defined invariants such as following:

$$T1\_cntLocActSpd \in speed[cycle - 25..cycle]$$

$$T1\_cntRxSpd\_2 \in speed[cycle - 10..cycle]$$

These invariants ensure that values of messages received by the controller are fresh. For instance, the first invariant ensures the speed value in the controller local buffer  $T1\_cntLocActSpd$  is never older than 25 time units prior to the current time. Such invariants can be defined when using the vertical refinement guidelines and patterns.

### 8.7.2 Assumptions for Vertical Refinements Patterns

We have also assumptions in order to provide the guidelines and patterns of this chapter. These are:

- The system is error-free. So, failure in sensors, actuators and other components are not considered.

- Only one communication bus (i.e. the CAN), is discussed.
- There is an upper bound on the number of messages transmitted on the CAN at every cycle. This will ensure that tasks periodicities and message priorities are sufficiently long for all the messages to be broadcast.
- The duration of every task is less than its period, so at every cycle a task will complete. In other words, within a task cycle its start and end events will take place.
- The physical design of buttons will ensure that buttons are pressed long enough to allow the controller to register every button press. For instance in the example of the SCCS, the target speed set button should be pressed for two cycles (10 time unit) in order for the SCCS to register the button press.
- The actuator task takes place only after the controller task is completed. We modelled this assumption using the scheduling of the system.

The assumptions may restrict the application of the proposed guidelines and patterns. However, we have provided some basis for further development and expansion of the patterns.

### 8.7.3 Modelling of Buttons Differs from Actuators and Sensors

An overview of the vertical refinements of *actuators*, *sensors* and *buttons* is shown in Figure 8.21. The grey circles show the MCMC (monitored, controlled, mode and commanded) variable phenomena which were formalised during *horizontal refinements*, while the blue ovals represent the *vertical refinement* steps of introducing buttons, sensors and actuators. The box represents the decision making unit of the controller, i.e. control events and controller mode events.

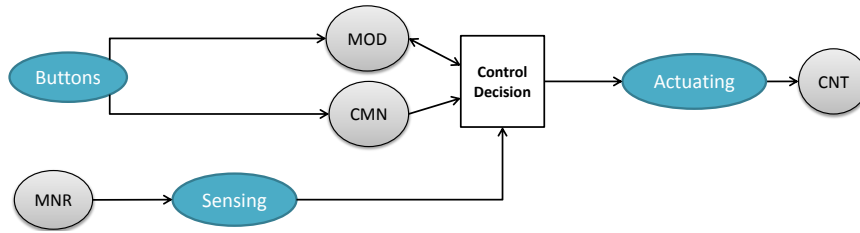


Figure 8.21: An overview of vertical refinement from phenomena point of view.

As Figure 8.21 illustrates, the vertical refinement of monitored and controlled phenomena means that a layer (sensors and actuators) is added in between the environment, where monitored and controlled phenomena reside, and the controller. So, this refinement is *inwards* since it is from the environment towards the controller. However, the vertical refinement of buttons means that the model is expanded to incorporate the buttons as

a new layer. So, this refinement is *outwards* since it is from the model of the controller towards the environment.

This shows that the introduction of sensors and actuators differs in nature from buttons. However, one might pose the question why we do not formalise buttons before introducing commanded or mode phenomena. There are two main reasons for this. Firstly, our experience as well as others, such as [Col12], has shown that defining requirements of a control system based on commanded and mode phenomena, rather than buttons, can facilitate the requirement gathering and structuring.

The second reason is that formalising buttons early can increase the complexity of the model. In our previous attempts we modelled buttons during horizontal refinements. This meant issues such as priorities between operator requests and the ordering among them (e.g. multiple button press) had to be resolved during the horizontal refinement. Therefore, events sequences and their priorities became the focus of the horizontal refinement.

However, we believe that such matters are design decisions which can be postponed to refinement levels after formalising system functionalities. Therefore, by modelling only the functional requirements which capture the main behaviour of the system during horizontal refinements we reduce the complexity of the modelling of the system.

## 8.8 Decomposition of a Control System

The decomposition of a model of a control system is discussed in this section. In Section 8.8.1 the system decomposition is shown based on the architecture of control systems. Section 8.8.2 explains the data flow in the decomposed components. In addition, this section presents a cycle component which acts as the global timer.

### 8.8.1 Decomposition based on Architecture

The vertical refinement patterns we provided in this chapter lead to a model which can be decomposed using the shared-event decomposition style.

A model of a control system may be decomposed into components of the following categories, *environment*, *button*, *sensor*, *actuator*, *controller*, *communication bus* and *timer*. There may be several components of each category in an architecture. For instance, an LCC has several sensors. Examples are a *speed sensor* and a *yaw rate sensor*. Therefore, it is possible to either decompose the LCC into a sensor component which consists of all the sensors, or decompose the LCC so that each sensor is captured by an individual component.

An overview of the decomposition of a control system model based on the system architecture is shown in Figure 8.22. In this figure the timing component is not discussed. We focus on the timing component in the next section.

The large boxes in Figure 8.22 denote the components, e.g., buttons and CAN. The events are shown by the shaded (red) rectangles. These events are defined based on the vertical refinement patterns. As shown, the events are either *local* to a component or *shared* amongst them.

For instance a *button event* which shows the interaction between an operator and a physical button is *local* to the button component. However, the event *button request transmit* which buffers the state of the button in the receiver buffer (Rx) of the CAN bus is *shared* amongst the button and the CAN machines. As shown, command and mode events are local to the controller component. This was discussed in Chapter 5 where we defined commanded and mode variables as controller quantities.

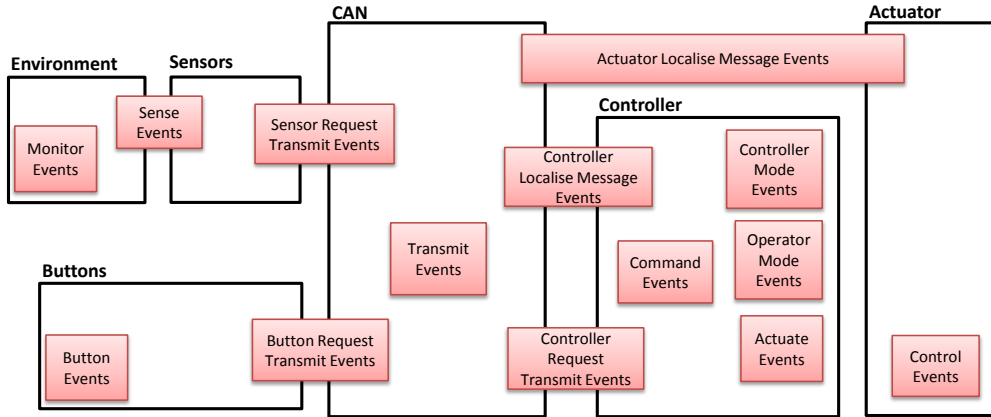


Figure 8.22: An overview of the decomposition based on the architecture.

### 8.8.2 Cycle Component in Decomposition

This section represents the decomposition of the timer which was modelled as a *global cycle*. It is through the cycle variable that components synchronise. The cycle variable and its corresponding events model the start and the end of every task. This is shown in Figure 8.23. For instance, a button task which belongs to a button component should take place once and only once within its periodicity. This is modelled by sharing the start and the end events of the button task amongst the button and the cycle machines. Note that a task can have several start or end events, all of which should be shared with the cycle component.

In addition to the cycle component, Figure 8.23 represents the data flow between the decomposed components. The values of the sensors and the buttons are transmitted from their corresponding local buffers to their transmitter buffers (Tx) in the CAN. Then these values are received by the controller receiver buffers (Rx). After that, the

controller copies the values to its local buffers based on which it will perform its actions. The controller then transmits its decision to its Tx buffer where the data will be copied to the actuator Rx buffer. The actuator uses the value of this buffer in order to actuate the environment quantity of the controlled variable.

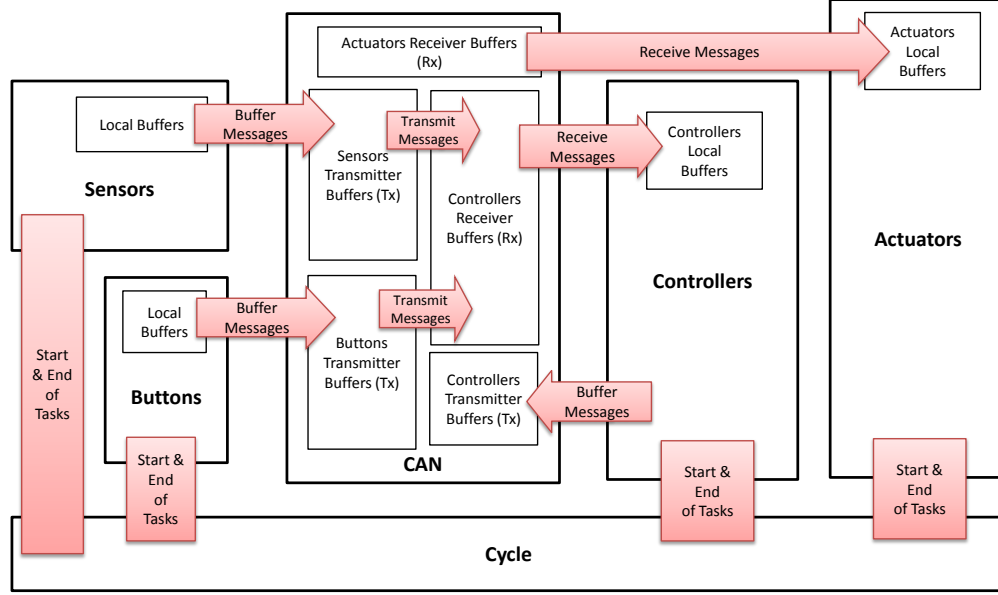


Figure 8.23: An overview of the data flow in the decomposition based on the cycle.

## 8.9 Case Study: Decomposition of a Simplified CCS

The concrete model of the SCCS is decomposed into *environment*, *sensor*, *button*, *controller*, *CAN bus* and *cycle* machines. The decomposition is similar to the approach discussed in the previous section.

Figure 8.24 represents a part of the SCCS decomposition which consists of the set target speed button, the CAN bus, the controller and the cycle machines. In order to do this decomposition we used the model decomposition plugin for Rodin [Dec10]. To decompose a model using the shared-event style, this plugin allows partitioning the variables between the components. The user of the plugin defines which component every variable belongs to and the plugin will decompose events in a way that the variables remain local to their components.

To illustrate how the model decomposition plugin was used, in Figure 8.24 we show the partition of the variables. The variables local to components are represented inside the components. Also, the shared events are shown in rectangles which are shared between the components. The Event-B representation of the decomposition of the SCCS can be found in Appendix B.





## 8.11 Conclusion

Broy and et al. have stated that “designing the architecture of an automotive IT system requires determination of the hardware architecture that consists of ECUs, bus systems, and communication devices, of sensors, actuators, and of the (hardware related) man-machine interface” [BKPS07]. They also propose an architecture-centric approach which contains different abstraction levels in order to model an automotive system.

In this chapter, we presented a model for an automotive control system that maps to its design architecture. To do this we used refinement and decomposition techniques. These techniques helped to reduce the complexity of modelling design details. However, the approach we proposed in this section is applicable to any control system, including automotive systems. We suggested decomposing a control system based on its architecture. Some of the advantage of this are:

- Decomposed components match an element (i.e., an ECU or a communication bus) of the system architecture. Thus, the implementable components can be identified.
- Every decomposed machine can be refined further in order to elaborate its specification with more requirements.
- Decomposition can develop team work, since each machine can be investigated as a separate model.
- Decomposition provides a means for distinguishing *software requirements* and its formal specification from *system* requirements and the overall specifications.
- Decomposition enforces an early realisation of the communication infrastructure (before implementation).
- Before decomposing, system assumptions, timing constraints and prioritisations of signals need to be modelled explicitly.



## Chapter 9

# Comparison, Future Work and Conclusion

In Section 9.1 an overview of the proposed approach for *horizontal refinement*, *vertical refinement* and *decomposition* of a control system is given. Also advantages of the MCMC approach are discussed in this section.

After that, Section 9.2 explains that the monitored, controlled, mode and commanded (MCMC) approach has evolved from the monitored, controlled and commanded (MCC) guidelines [But09a, But09d]. The weaknesses of the MCC guidelines are described in detail and we show how the MCMC approach overcomes them.

The MCMC approach is then compared to other requirement engineering approaches in Section 9.3. In Section 9.4, we discuss the usage of the MCMC approach in other state-based formal methods, namely, B, Z and VDM. Finally, Section 9.7 discusses our future work.

### 9.1 Overview of the Proposed Approach

The approach presented in Chapters 5, 6 and 8 of this thesis covers *structuring* a requirements document, *formalising* it as sub-models, *composing* the sub-models, introducing *design details* to the composed model and *decomposing* the formal model. An overview of the approach is shown in Figure 9.1.

The top part of Figure 9.1 represents the horizontal refinement steps where the structured requirements of a control system are formalised as MCMC sub-models. During this formalisation, refinement is used to construct the sub-models gradually. The most concrete levels of the sub-models are composed to form the overall system specification.

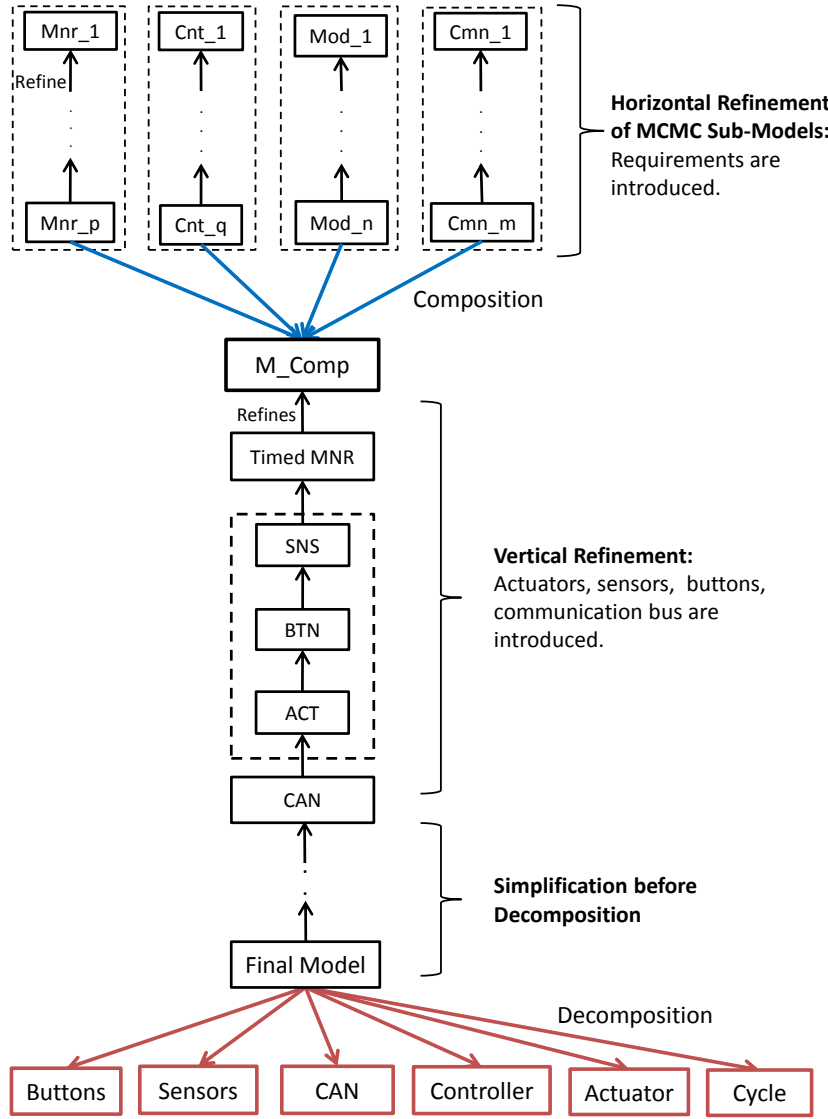


Figure 9.1: Overview of refinement steps and decomposition of a control system.

This is shown as *M.Comp* in Figure 9.1. The steps for horizontal refinement and composition of sub-models were discussed in Chapters 5 and 6 as the four-stage MCMC approach, which contains: *structuring* an RD, *formalising* as sub-models, *validating* RD against its formal model and revision steps, and *composing* sub-models.

The basis of this four-stage approach is *system phenomena* of monitored, controlled, commanded and mode. These phenomena were described and distinguish from one another in order to facilitate identifying them in a problem domain. The definition of the MCMC phenomena as well as the provided guidelines, patterns and case studies can assist a modeller in identifying phenomena of a control system. We also provided some criteria for constructing a refinement chain within every sub-model. In addition, guidelines, patterns and schemas for mapping informal requirements to a formal model have been provided.

The four-stage approach has been applied to three automotive case studies of a cruise control system (CCS), a lane departure warning system (LDWS) and a lane centering controller (LCC). Some of the advantages of this approach are:

- The RD of a control system can be organised and structured as a set of related sub-problems.
- We take a system-level view during the formalisation of the sub-problems. Focusing on system phenomena in horizontal refinement helps represent the requirements in terms of phenomena of the problem world (environment), which is more familiar to the modellers and stakeholders [JHJ07].
- Formalising sub-problems as sub-models can provide a means for separation of concerns and team work.
- The MCMC approach can help to reduce the effort required for modelling and the complexity of the formalisation process, since:
  - by formalising an RD as sub-models there are less requirements to model in every sub-model.
  - the refinement guidelines within every sub-model can help to order the refinement chain in a more deterministic manner.
  - the MCMC sub-models schemas and the MCMC events patterns, which respectively represent the overview of machines and their events, have been provided.
- The MCMC events patterns and the MCMC sub-models schemas can be used for validating the MCMC sub-models and their events. This gives a modeller a clear overview and perception of the final modelling outcome.
- By validating the model against its RD traceability links between the model and the requirements can be established.

Figure 9.1 also illustrates that the composed model is refined further in order to introduce the design elements. The design details of actuation, sensing, user interactions and communication bus were incorporated into the model in several *vertical refinement* steps. After this, the model is prepared for decomposition and it is decomposed based on the *system architecture*. Vertical refinement and decomposition mean that the *software specification* of a control system can be extracted from the *system specification*, which is the result of horizontal refinement and composition steps. Vertical refinement and decomposition were discussed in Chapter 8.

## 9.2 Comparing MCMC Approach and MCC Guidelines

The vertical refinement steps for introducing *actuators*, *sensors* and *buttons* are based on the MCC modelling guidelines [But09a, But09d], which were explained in Section 2.3.3. The focus of these guidelines is on formal modelling of control systems using *monitored*, *controlled* and *commanded* (MCC) phenomena. The MCC guidelines were followed in formal modelling of several case studies, such as a cruise control system [YBR10] and a FADEC system [Das10], which helped us to pinpoint the benefits and the drawbacks of these guidelines.

The application of the MCC modelling guidelines showed that they are insufficient and incomplete. In particular, they provided guidelines on introducing design detail (vertical refinement) to a model of a control system. However, no guidelines on formalising requirements (horizontal refinement) were given. In addition, the concept of commanded phenomena was defined vaguely, while precise definitions of the phenomena were required in order for the guidelines to be easy-to-follow.

The remainder of this section explains that in developing the MCMC approach we tried to consider the drawbacks of the MCC guidelines and overcome them.

### 9.2.1 Clarifying Commanded Phenomena

The main aim of providing patterns and guidelines for modellers is to reduce the barriers to formal modelling, which includes requiring experts for defining models. Therefore it is important that the guidelines are accurate and easy to understand. One drawback in the MCC guidelines was that the identification of phenomena was sometimes difficult. In particular, the concept of commanded phenomena could cause confusion. As explained in Section 5.2.3.3 a phenomenon such as an indicator in the LDWS can be treated as either a monitored or a commanded variable phenomenon.

In the MCMC approach, Section 5.2, we clarified the definitions of the phenomena. Also, commanded phenomena were explained in detail and through examples. We defined two types of UI, namely an existing interface and a specialised interface, which can help to differentiate between monitored and commanded phenomena [YB12].

### 9.2.2 Mode Phenomena

Applying the MCC guidelines to case studies drew our attentions to a special phenomenon, called *mode* which captures the states of a control system. As explained in Section 5.2 mode is a special phenomenon, because it can be updated by either the operator or the controller. This was the reason for categorising mode events into *operator mode events* and *controller mode events*. Capturing mode as a phenomenon distinct

from other system phenomena can facilitate the process of identifying the MCMC phenomena [YB12].

### 9.2.3 Structuring RD based on Phenomena

The very first step in following the MCMC approach is to identify the *monitored*, *controlled*, *mode* and *commanded* phenomena of the control system under consideration. This inspired us to reflect the MCMC phenomena in the structure of an RD [YB11, YB12].

Therefore, we proposed to structure requirements into the MCMC sub-problems. In this case every sub-problem has a degree of unity and can be formalised as a single sub-model. Structuring an RD will provide the means for separation of concerns.

### 9.2.4 Guidelines on Horizontal Refinement

As mentioned in the previous chapters we differentiate between refinements towards modelling the RD (horizontal refinement) and refinements towards implementation (vertical refinement). Applying the MCC guidelines to case studies demonstrated that they provide modelling patterns only for vertical refinement steps. In other words, it is after the formalisation of the entire RD of a control system that the MCC guidelines can be applied. The guidelines lacked patterns for horizontal refinement steps, while deciding on which requirement to model in the abstract level and which ones in refinement steps, as well as how to model them, can be challenging [Abr06].

In this thesis, we have developed a set of guidelines which covers both horizontal and vertical refinements. As discussed in Chapters 5 and 6, the guidelines for horizontal refinement steps use (de)composition techniques to simplify the modelling process. This also resulted in extending the structure of Event-B machines to simplify formalising sub-problems which share phenomena.

### 9.2.5 Guidelines on Vertical Refinement and Decomposition

The vertical refinement patterns of the MCC guidelines were insufficient, since the MCC guidelines support only one step of vertical refinement for every monitored, controlled and commanded (MCC) phenomena. The refinement steps covered in the MCC guidelines are refining: monitored variable by adding *sensors*; controlled variable by adding *actuators*; and commanded variable by adding *buttons*.

However, without introducing communication details between a controller and the environment it is not possible to decompose a vertically refined model. Therefore, in the MCMC guidelines we:



- provided detailed explanations on vertical refinement steps. Also, tables which show the dependencies between events and variables during vertical refinements were given in Section 8.5.
- introduced a communication bus.
- decomposed the model of a control system, so the software specification was extracted from the system specification.

### 9.3 Comparing the MCMC and Other RE Approaches

In this section the proposed MCMC approach is compared with the RE approaches discussed in Section 2.3. The comparison also provides the reader with a further insight into the MCMC approach.

#### 9.3.1 Four-variable model and MCMC

The presented MCMC approach is influenced by the four-variable model [PM95] which was discussed in Section 2.3.1. In particular, the concepts of *monitored* and *controlled* phenomena are taken from the four-variable model. Also, the *input* and *output* phenomena of the four-variable model can be mapped to the *sensed* and *actuator* phenomena in the MCMC approach. In this thesis, we proposed to capture monitored and controlled phenomena during horizontal refinement steps, while refinement patterns were provided in Chapter 8 to introduce sensed and actuator phenomena as part of vertical refinement.

Figure 9.2 illustrates the phenomena which are defined in the MCMC approach. As shown, in addition to the four-variable model phenomena the MCMC approach proposes the identification of the *commanded* and *mode* variables during the horizontal refinement steps, as well as *button* variables in the vertical refinement steps.

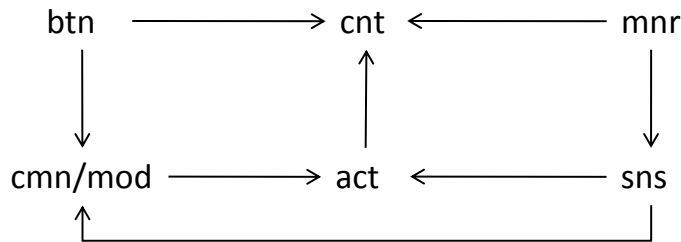


Figure 9.2: The MCMC approach has more phenomena than the four-variable model.

It can be argued that button phenomena can be captured as part of monitored phenomena. We believe that distinguishing between the monitored and button phenomena can simplify the process of formalisation, since the role of the operator can be clearly

identified. In addition, button phenomena are defined to modify commanded and mode phenomena which belong to the controller, while monitored phenomena belong to the environment.

The four-variable model also defines four relations, namely, NAT and REQ which represent a system-level view, and IN and OUT which relate the environment and the controller phenomena. In the MCMC approach, the NAT and REQ relations can be represented as *monitor events* and *control events*, while IN and OUT can be represented as *sense events* and *actuator events*.

### 9.3.2 Problem Frames and MCMC

Problem frames (PF) [Jac01], Section 2.3.2, is a general purpose requirement engineering approach which can be used for a wide range of problem domains. However, the MCMC approach deals with the specific domain of control systems. In addition, the main focus in the MCMC approach is to model requirements formally, while PF is a semi-formal approach.

As described in Section 2.3.2, PF categorises phenomena into 6 groups. The MCMC approach covered two types of phenomena, namely, *variable* and *event*, which correspond to the *entity* and *event* categories in PF. These two categories represent the dynamic behaviour of a system. It can be beneficial to consider other categories of phenomena in PF to extend the MCMC approach with guidelines which comprises of static system behaviours as well as dynamic. To do this the MCMC approach can be extended to include static phenomena. For instance *value phenomena* in PF, can be modelled as *numerical sets*, *carrier sets*, or *constants* in a context of an Event-B model.

Figure 9.3 shows a PF representation of the MCMC phenomena as well as the sensed (*sns*), actuator (*act*) and button (*btn*) phenomena which we introduce during vertical refinement. This figure is an extension of Figure 2.12 which represented the four-variable model in a PF diagram.

As illustrated, the domains (parts which can be relevant to a control system) in the diagram are: Environment; Operator; Sensor; Actuator; Button; Controller. Monitored (*mnr*) and controlled (*cnt*) phenomena are shared between the environment domain and the sensor and actuator domains respectively. They are also phenomena to which the requirements of the system refer. The phenomena shared between the controller domain and the sensor and actuator domains are *sns* and *act* variable phenomena respectively. The operator and the button domains in Figure 9.3 represent the role of an operator in the MCMC approach. As shown, they share button (*btn*) phenomena. Also, the controller and the button domains share *btn* phenomena.

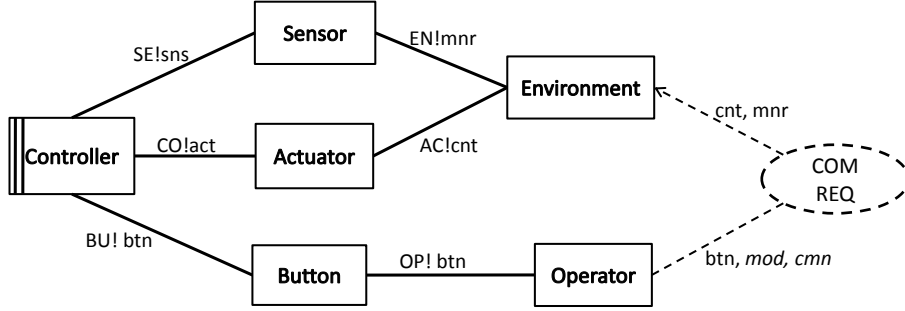


Figure 9.3: MCMC phenomena and their vertical refinement in PF.

Figure 9.3 shows that a difference between the MCMC approach and the PF is that phenomena internal to a domain are not captured explicitly in a PF diagram. This is because PF is not concerned with the internal mechanism of a domain. Therefore, commanded (*cmn*) and mode (*mod*) phenomena which are internal to the controller domain are not represented in the above figure. Our experience has shown that although *cmn* and *mod* phenomena can appear in the requirements. In fact, describing requirements of a control system in terms of these phenomena can simplify our understanding of the system.

Examining Figure 9.3 shows that the requirements may refer to *btn* phenomena. Since we considered buttons as design details, we suggest to capture requirements related to them in a *design requirements document* which can be constructed before the vertical refinements. Such a document may contain details about sensors, actuators, buttons and communication bus. In our future work we will look into structuring design requirements documents based on the phenomena introduced in the vertical refinement steps.

### 9.3.3 KAOS and MCMC

KAOS [vL09], which was discussed in Section 2.3.4, is a requirement engineering technique. In contrast, the MCMC approach is designed to suit structuring, formalising and verifying an RD. So, the MCMC approach pays less attention to requirements elicitation. In KAOS *goal models* are used mainly for requirement gathering. Therefore, we do not compare the MCMC approach to a goal model in KAOS.

However, our approach can be compared with the concept of agents and agent models in KAOS. In the MCMC approach, we focus on the *agents* of a control system and structure requirements according to its agents. The categories of agents in a commanded control system are: *operator* (or *button*), *sensor*, *controller* and *actuator*. Examples of these agents in an LCC is given in Table 9.1.

In KAOS, an agent can have *monitoring* and *control* links which represent agent capabilities. An agent can monitor/control an object's attribute if it can get/set the value

Table 9.1: Agents and their capabilities in the MCMC approach.

Agent	Example	Monitors	Controls
Operator	Driver	(Operator may monitor the environment)	Button Events; such as Offset Increase button
Sensor	Speed sensor	Monitored Phenomena	Sensed Phenomena
Controller	LCC	Sensed Phenomena; Button Phenomena	actuator Phenomena
Actuator	Steering actuator	actuator Phenomena	Controlled Phenomena

of the attribute. Considering the agent categories of the MCMC approach, phenomena which agents monitor or control are shown in Table 9.1.

Figure 9.4 shows a context diagram in KAOS which shows the relations between the MCMC agents and their capabilities. The nodes are active system components which communicate using shared phenomena labelled on the arrows. The communication between the agents is shown by arrows that connect two agents. The source agent of the arrow controls the shared phenomena, while the target agent monitors the phenomena. The text on the arrows represents values that the source agent can control and the target agent can monitor. Examples of these values are given in *italic*.

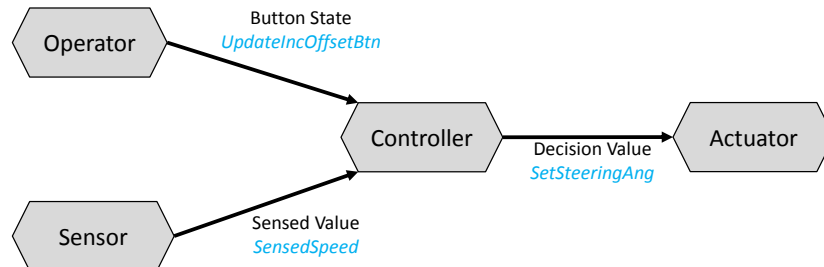


Figure 9.4: A representation of agents of the MCMC approach in a context diagram in KAOS.

Comparing the MCMC approach to Figure 9.4 shows that the agents in the context diagram can be mapped to a component in the decomposition step, e.g. an Event-B representation of the operator agent is a button machine which is derived from the formal model of a control system after decomposition. In addition, the links between the agents represent a means of communication among them is necessary.

### 9.3.4 Tabular Approach of SDV and MCMC

The tabular approach of SDV [LFM04, WL03] uses tabular notations for designing the requirements formally. Although understanding and using tabular notations can be easier than formal languages such as Event-B, they have limited expressiveness. For

instance, in a Software Design Description (SDD) document, pseudo-code might be used in addition to tables in order to document details of a system design.

Wassyng and Lawford state that “pseudo code is used when a sequence of operations is mandatory and cannot be described easily in tabular format” [WL03]. However, the MCMC approach is developed to be used in conjunction with state-based formal languages which are far more expressive than tabular notations. The MCMC approach in this thesis was used in conjunction with the Event-B language. In Section 9.4 we will discuss the possibility of applying the MCMC approach to other state-based formal languages, such as B, Z and VDM.

The tabular approach of SDV was applied to the Darlington shutdown system from the requirements to its design, implementation and verification. In comparison, the MCMC approach focuses on requirements, design and verification stages of a development life cycle. Although implementing a control system has not been considered in this thesis, it can be addressed in future work, since the gap between design and implementation in Event-B has been researched by developing automated code generation techniques which preserve the correctness of the code with respect to the formal design [MS11, ERB11].

The main aim of the tabular approach is to formally prove that the requirements are in compliance with the code. In the MCMC approach we aimed to develop an approach to facilitate the process of formalising a control system. Therefore, we provided modelling guidelines and patterns, while this is not the case for the tabular approach of SDV. In addition to this, the MCMC approach provides a systematic way of formally proving that requirements are in compliance with the decomposed formal model which can also be implemented.

We also proposed the identification of commanded phenomena which are not considered in the tabular approach of SDV. In addition, in the MCMC approach the consistency between the sub-models is proven by composing them based on the (de)composition techniques in Event-B. In the tabular approach of SDV the composition of tables provides the overall specification, however, the consistency between tables is not discussed.

### 9.3.5 SCR and MCMC

SCR (Software Cost Reduction) [HJL96] was discussed in Section 2.3.6. Like our approach, SCR is based on the four-variable model. In addition to the *monitored*, *controlled*, *input* and *output* variables, SCR uses *mode*, *terms*, *conditions* and *events*.

So, unlike the MCMC approach, *commanded* phenomena are not modelled explicitly in the SCR. Examining some of the examples of SCR [HJL96, JH03] showed that the interaction of an operator with a system are modelled as *monitored variables* and *terms*.

However, our experience is that distinguishing monitored and commanded phenomena facilitates system design and understanding of requirements, as they serve distinct roles.

The mode in SCR is different to the mode phenomenon defined in the MCMC. The SCR represents mode as a “state machine defined on the *monitored* variables” [HJL96]. So, in SCR, mode is derived from values of monitored phenomena. This definition is preserved for some of the SCR case studies. However, in the example of a CCS [JH03], mode represents the states of the controller. In the MCMC approach, we tried to avoid ambiguity in the definitions and the usage of phenomena.

In SCR, an engineer is required to identify the variables, mode, terms, conditions and events in order to specify a system. However, we break the complexity and the effort of identifying all phenomena upfront by using refinement and (de)composition techniques. We have a system-level view in which monitored, controlled, mode and commanded phenomena are introduced initially in a step-wise manner. It is only after the introduction of system phenomena that the design-related phenomena are specified. Although *terms* in SCR can be refined by introducing details of a specific term, there is no explicit support for refinement of the behaviour of a system, e.g., refinement of the tables.

The MCMC approach uses the Event-B formal language to specify the relation between phenomena, while SCR uses tables (the event, condition and mode transition tables). A brief consideration of these tables shows that they can be represented in Event-B. Furthermore, it might be possible to explicitly map SCR tables to the environment, control, mode and command events of the MCMC. For instance, a row of a *mode transition table* can be converted to an *environment event*. This is because a row of this table shows how the mode (which in SCR is the value of the monitored variable) changes on an event occurrence.

### 9.3.6 WRSPM and MCMC

Similarly to the MCMC guidelines, the WRSPM also looks at the phenomena (states and transitions) of a system and its environment. The WRSPM is compared to the four-variable model in [GGJZ00a]. We adjust this comparison to include the MCMC’s phenomena. Neither four-variable model nor MCMC guidelines discuss phenomena  $e_h$  which is presented in WRSPM as environment phenomena hidden from the system. This is because the objective of the MCMC approach is to model the system and its interactions with the environment and not the environment behaviour.

*Monitored* and *button* phenomena of the MCMC represent  $e_v$  which is the environment phenomena visible to the system. Also,  $s_v$ , system phenomena visible to the environment, is captured by the *controlled* phenomena. *Commanded*, *mode*, *sensed* and *actuator* phenomena of the MCMC can be mapped to  $s_h$  (system phenomena hidden from the environment) as they represent internal phenomena of the system.

As mentioned, [JHLR10] introduces an approach based on the WRSPM which helps to establish traceability between requirements and an Event-B model. Both our approach and the traceability approach in [JHLR10] are concerned with formal modelling of informal requirements. However, the primary focus in [JHLR10] is on traceability between an RD and the formal model, while the MCMC approach focuses on structuring and formal modelling. Also, we provided guidelines for refinement-based modelling, while this is not specified in the traceability approach [JHLR10].

### 9.3.7 HJJ and MCMC

Our approach has similarities to HJJ [HJJ03] which was discussed in Section 2.3.8. Similarly to HJJ a specification in the MCMC approach starts with modelling *system* phenomena, rather than *software* phenomena. This means both approaches have a system level view of requirements.

The focus of the HJJ approach is on providing a method for identifying and formally documenting assumptions of physical components (rely conditions). However, our focus is on structuring and formalising requirements on the control system. In addition, we proposed categories for phenomena which can help modellers with identifying the system phenomena.

The MCMC approach discussed commanded control systems (controllers with operators) in addition to autonomous control systems (controllers without operators). However, the HJJ approach focuses on the latter control systems. Commanded control systems can provide challenges, since minimal (or no) assumptions should be made on operators behaviours. This is mainly because their behaviours are unpredictable. Also, in HJJ all requirements are dealt with in one flat step of specification, while our approach uses refinement and (de)composition techniques to introduce requirements in horizontal refinement steps.

The HJJ also deals with faults. This is done by defining subproblems which treat faulty and fault-free specification separately. Although in this thesis we have not considered faults and fault tolerance, others have investigated an approach based on the MCMC to identify and specify faults [Col12].

## 9.4 MCMC Approach in other Formal Methods

Structuring requirements as sub-problems which can be formalised individually has also been suggested in other research, (e.g. [Jac01], [Jac95] and [ZJ93]). In this thesis, we proposed an approach for identifying sub-models of a control system and formalising them in the Event-B language. However, we believe the MCMC approach can be used

to formalise requirements of a control system using state-based formal methods other than Event-B. This section discusses the possibility of specifying requirements in the B method, Z and VDM using the MCMC approach. To do this we consider whether the composition techniques in B, Z and VDM allow for composing sub-models to obtain the overall specification.

#### 9.4.1 MCMC Sub-Models in B-Method

The classical B method is a refinement-based formal language. However, it does not allow introduction of new operations (transactions) in refinement steps. Thus, it is not possible to introduce requirements which require refining a dummy skip event as was the case when Event-B was used. However, it is possible to formalise requirements of a sub-problem abstractly and then refine the abstract model to introduce more details. This results in the MCMC sub-models which may have read-only access to variables of other sub-models.

In order to obtain the overall specification, the sub-models will need to be composed. Some keywords for composing machines (or sub-models) in the B method are as follow:

- Includes: An abstract machine can be linked to another abstract machine by including that machine. When a machine such as M2 includes M1, it means states and operations of M1 become part of M2. However, M2 has read access to the state of M1. The restricted access to M1's variables means that the internal consistency of M1 is guaranteed by its invariants. In addition, operations can be combined by conjoining their preconditions and combining their postconditions in parallel.
- Uses: The Uses clause introduces a form of sharing between abstract machines. M2 USES M1 means that M2 has read access to sets, constants and variables of M1. Since M2 may refer to variables of M1 in its invariants, there will be a third machines, such as an M3 which includes M1 and M2 in order to prove that the cross-cutting invariants (invariants referring to variables of M1 as well as M2) are preserved.
- Sees: This clause allows a machine to have read access to sets, constants and variables in another machine. However, M2 SEES M1 means that M2 cannot refer to M1's variables in its invariants.

The above keywords enable a modeller to compose the MCMC sub-models. Therefore, it is possible to apply the MCMC approach to the formalisation of a control system in the B method.



### 9.4.2 MCMC Sub-Models in Z

Daniel Jackson describes structuring a Z specification as *views* [Jac95] where every view is a partial specification of the system under consideration. Views consist of a state space and a set of operations. They are defined as Z schemas which are constructed independently of one another. In order to obtain the overall specification views can be combined.

In Z composition can be done by combining schemas. To do this the declaration lists of schemas (the top part in schemas) are merged and their predicates (the bottom part in schemas) are conjunct. It is possible to combine schemas with common variables, if these variables are type compatible. Also, schemas can be combined by including the name of a schema within the definition of another.

Daniel Jackson suggests combining views of a system by either introducing invariants which connect the states of different views or composing operations of views. In most cases however, it is necessary to use both techniques in order to combine views.

The concept of views in [Jac95] is similar to defining sub-models. However, views are structured based on “natural view separations in the problem domain”, while we proposed guidelines for determining the MCMC sub-problems from which sub-models can be constructed. In later research, Jackson et al. suggest defining views of a system in a Z specification based on the frames in the PF approach [JJ96].

The possibility of combining schemas means that the MCMC approach can be used to specify a control system as composable schemas. Such schemas may share variables, but the shared variables should be compatible. However, one main difference between using Z and Event-B for combining specification is that in Z conjunction does not preserve refinement. In other words, a refinement of  $schema1 \wedge schema2$  is not generally a refinement of both  $schema1$  and  $schema2$ . However, in the shared-event composition in Event-B, composed machines are monotonic which allows further refinements of sub-components while refinement of the composition is preserved [SB10].

### 9.4.3 MCMC Sub-Models in VDM

Refinement techniques in VDM are described as *data reification* and *operation decomposition*. The former involves the transition from abstract to concrete data types and the justification of the transition. The data types at the end of this process are those of the implementation language, however, the transformations are defined implicitly by the pre/post conditions. The process of operation decomposition develops implementations for abstract and implicit operations specifications. The implementations are developed in terms of the available language and software.

One difference between refinement in VDM and Event-B is that in Event-B an abstract event can be refined to multiple concrete events. This is done by refining an abstract dummy event called *skip*. However, in VDM (similarly in B and Z) operations are refined on a one-to-one basis, meaning that for every concrete operation there exists an abstract representation of that operation.

VDM also provides a mechanism which allows entities defined in modules to be used by other modules. To do this a module can *export* its entities to other parts of the specification. This is done using the keyword *exports* as follow: `exports entitiesDescription`. Also a module can *import* entities. This is done using the keyword *imports* as follow: `imports from ModuleName entitiesDescription`. Note that only exported entities can be imported.

Using refinement techniques as well as import and export keywords in VDM, we can potentially develop the MCMC sub-models in VDM. However, VDM does not allow 'cyclic' imports; i.e., two modules cannot import from each other. Therefore, in order to use VDM in developing the MCMC sub-models, their representations may require changing.

## 9.5 Role of Rodin and its Plugins

To develop formal models of the case studies described in this thesis, we used the Rodin tool and some of its plugins. The tool and plugins were explained in Sections 4.6 and 4.7 respectively. In this section we aim to give an overview of how Rodin and the plugins were used in this thesis. In addition their benefits and limitations in developing case studies of this thesis are discussed.

The Rodin tool is easy to use. Since it is developed on eclipse, the user can intuitively navigate around the tool. It facilitates the process of modelling and generating as well as discharging proof obligations. Other functionalities, such as decomposing a model, can be added to the Rodin tool by installing plugins. One drawback of plugin installation is their compatibility with Rodin. This means, because compatible versions of plugins are not usually available at the same time as a new Release of Rodin, a modeller might use an older version for some time before being able to upgrade to the new release.

The plugins that were used in this thesis are:

- **ProB** [Pro11, LB03]: ProB was mainly used as an animator. This plugin enabled us to execute models and identify errors and design flaws. In particular, ProB was used to ensure that the order in which events become enabled is right. In other words, this plugin helped identifying whether guards in events were too strong.

In addition to animation, ProB was used in a number of cases to check for deadlock freeness. The plugin allows the modeller to tick a box and run the model checker to identify deadlocks in the model. This approach is very useful in abstract levels, or the first few steps of refinements. However, after multiple refinement levels, the ProB model checking can take some time. Therefore, it becomes a matter of judgement to consider the benefits of running the ProB model checker.

- **Model Decomposition** [Dec10, SPHB10, SPHB11]: This plugin was very helpful in decomposing models and it saves us a considerable amount of time. The manual decomposition involves creating sub-machines and updating each sub-machine with their corresponding variables and events manually. This manual process can take a few hours, depending on the size of the model. However, the plugin can create the decomposed sub-machines and update them with the relevant events and variables within a few minutes. This plugin also shows the user if any of the actions or guards are too complex to be decomposed. This can be used to ensure that the right variables are partitioned amongst the sub-machines.
- **Shared-Event Composition** [Com11]: This plugin was used to generate the model of a system from its sub-models. This plugin does not support the composition of machines with extended structure, as suggested in this thesis. Therefore, in order to compose the sub-models, we defined the sub-models using witnesses as described in Section 6.5.5.
- **Camille** [Cam11, BFJ<sup>+</sup>11]: This plugin was used to specify and design the models. The main advantage of this plugin is that it provides a syntax-aware text editor. This enables copying/pasting a part of the model into another machine, or another text editor. It also helps with readability of models. One limitation of this plugin is that occasionally the updates files are not saved and thus a modeller may lose a part of their work. We managed this limitation by saving the files regularly.

Overall, our experience in using Rodin and the plugins showed that the benefits of using the tool outweigh its limitations. In particular, the tool helped us to model the case studies in Event-B considerably quicker than the manual modelling would have taken.

## 9.6 Generalisability and Scalability of the MCMC Approach

Sections 9.6.1 and 9.6.2 discuss the generalisability and scalability of the MCMC approach respectively.

### 9.6.1 Generalisability of the Approach

To consider the generalisability of the MCMC approach and the formalisation process that has been proposed in this thesis, we divided the approach into the following parts:

- **Structuring Requirements:** This can be used in various types of control system. This is because requirements of a control system can be structured regardless of its architecture. In addition, no specific formal language or tool is required to structure requirements.
- **Formalising Structured Requirements:** To formalise requirements, a modeller is required to identify MCMC events and variables and model them based on the syntax and semantic of the chosen formal language. Any state-based formal language that requires states (variables) and transactions (events) of a system to be modelled, can be used to formalise structured requirements.
- **Refining a Model Horizontally:** The guidelines, patterns and schemas that have been provided for horizontal refinement are language-specific. The patterns of this thesis are based on the syntax and semantics of Event-B. While the exact patterns or schemas cannot be used for modelling a structured RD in other formal languages, they can be adjusted to the syntax and semantics of other formal languages.
- **Refining a Model Vertically:** The vertical refinement guidelines and patterns presented in this thesis are language and architecture-specific. This is because the patterns are based on the Event-B language. In addition, the patterns are provided for a control system that consists of sensors, buttons, actuators, controllers and the CAN bus. For instance, if a control systems uses a different communication bus to CAN, some parts of the patterns may require adjustments.

Although not all the presented patterns and guidelines are generalisable, they provide a basis for developing further patterns for other formal languages as well as various types of control systems.

### 9.6.2 Scalability of the Approach

Using the MCMC approach in various case studies has shown that this approach is scalable. In this section we discuss scalability of the approach in terms of requirements analysis as well as formal modelling.

The scalability in analysing the requirements of a system is addressed by the idea of identifying sub-problems and using categories to assist with the identification of sub-problems. As an example, in addition to the MCMC sub-problems, a number of domain-specific sub-problems can be identified for a control system. Furthermore, the MCMC

approach can be used in conjunction with other requirements structuring approaches. For instance, one can structure a set of requirements into the MCMC sub-problems, and then structure each sub-problem based on other criteria. Also, other RE approaches, such as Problem Frames, can be used to define the MCMC sub-problems as frames which then can be formalised. Combining the MCMC approach and other RE approaches can be investigated further as part of future work.

Another reason that we believe the MCMC approach is scalable is that it is based on the four-variable model. Other approaches that are developed from the four-variable model, such as SCR [Hei07, HJL96], have been used in modelling large-scale systems. As an example, the tabular approach of SDV was applied to the Darlington Nuclear Generation Station Shutdown System as part of the software development process [LFM04, WL03].

In terms of scalability of the formal modelling, as shown in Figure 9.1, we used refinement and composition techniques to model the requirements of a system in horizontal refinement steps. These techniques would allow for scalability, since requirements can be formalised as small chunks and also new functionalities can be modelled and incorporated into the existing models by using refinement and composition.

However, in our approach, a system can be vertically refinement as a single chain of refinements. This means design requirements can be incorporated to a model only in refinement steps. Therefore, vertical refinement remains a bottleneck that could hinder scalability. As part of future work, flexible composition and decomposition techniques for vertical refinement of control systems should be investigated.

## 9.7 Conclusion and Future Work

This thesis provided an approach from structuring and modelling informal requirements of a control system to introducing design details and decomposing the model. The approach included patterns, guidelines and schemas for *horizontal refinements* where requirements are formalised, *vertical refinements* where design details were introduced to the model and *decomposition* where the model is partitioned into components based on the system architecture.

The patterns and guidelines defined in this thesis are based on the notation of the Event-B language. The approach and the provided patterns and guidelines can be adjusted to suit other state-based formal language. As part of our future work we will apply our approach to case studies specified in other formal methods, such as B and Z.

Other future work related to this thesis are:

- Expanding the MCMC phenomena to include derived phenomena which were discussed in Section 7.9.2.

- Using other communication buses, such as LIN bus.
- Refining the MCMC sub-models (or schemas) independently in order to introduce design details. This means sub-models can be refined vertically while the effort and complexity of the model is managed.
- Expanding the MCMC approach to include non-functional requirements in more detail.
- Considering lossiness and faultiness of components, such as sensors and actuators.
- Considering misbehaviours of operators, e.g. pressing a wrong button at a wrong time, in more detail.
- Exploring other types of buttons such as press/release buttons where pressing a button triggers an action and releasing it stops the or multifunctional buttons where responses of the controller depend on the status of the system at that moment in time.
- Improving traceability in the MCMC approach. One possible way of doing this is to consider the traceability approach of [JHLR10] and the ProR plugin for Rodin tool [Pro12].
- Expanding the MCMC approach by including a design requirements document which contains the requirements related to user interfaces, sensors, actuators and communication buses. This document can be structured into four sections which represent requirements of user interfaces, sensors, actuators and communication buses. In addition, the possibility of establishing traceability and validation links between a vertically-refined model and its design requirements can be explored.
- Developing plugins for the Rodin tool to support the extended structure of machines (machines with import and export variables clauses) and generate their compositions.
- Considering a form of composition that composing machines with extended structure results in a machine which contains imported and exported variables. In other words, composition of machines results in an extended structure machine, rather than a normal Event-B machine.

The approach in this thesis provides a basis for simplifying and facilitating the process of designing a formal model from informal requirements. Furthermore, the proposed future work helps explore other aspects of designing a control system formally, such as introducing fault tolerance. Developing plugins to support the proposed approach will also provide support to the users of the approach.



# Appendix A

## Sub-Models of LCC

This chapter presents the case study of the LCC which is modelled as the monitored, controlled, mode and commanded sub-models. This case study is modelled based on the original machine structure in Event-b. The usage of the extended machine structure in the example of the LCC was discussed in Chapter 7.

### A.1 Contexts of LCC

#### A.1.1 Context C0\_MNR

**CONTEXT** C0\_MNR

**SETS**

$LP$

$RC$

$YA$

$YR$

**CONSTANTS**

$maxSpd$

**AXIOMS**

$axm1 : maxSpd \in \mathbb{N}$

**END**

#### A.1.2 Context C0\_CNT

**CONTEXT** C0\_CNT

**EXTENDS** C0\_MNR



**SETS***TargetPath***CONSTANTS***TargPathFunc***AXIOMS***axm1* :  $TargPathFunc \in (LP \times RC \times \mathbb{Z}) \rightarrow TargetPath$ **END****A.1.3 Context C0\_MOD****CONTEXT** C0\_MOD**SETS***STATUS***CONSTANTS***Off**Standby**Active**Overridden**Unavailable***AXIOMS***axm1* :  $partition(STATUS, \{Off\}, \{Standby\}, \{Active\}, \{Overridden\}, \{Unavailable\})$ **END****A.1.4 Context C0\_CMN****CONTEXT** C0\_CMN**CONSTANTS***offsetLB**offsetUB***AXIOMS***axm1* :  $offsetLB \in \mathbb{Z}$ *axm2* :  $offsetUB \in \mathbb{N}$ *axm3* :  $offsetLB \leq 0$ *axm4* :  $offsetUB > offsetLB$ **END**

**A.1.5 Context C1\_MNR****CONTEXT** C1\_MNR**EXTENDS** C0\_MNR**SETS***ACC\_STATUS***CONSTANTS***ACC\_Active**ACC\_Deactive***AXIOMS***axm1* : *partition*(*ACC\_STATUS*, {*ACC\_Active*}, {*ACC\_Deactive*})**END****A.1.6 Context C1\_CNT****CONTEXT** C1\_CNT**EXTENDS** C0\_CNT**CONSTANTS***SMTargUB**SMTargFunc***AXIOMS***axm1* : *SMTargUB* ∈ ℕ*axm2* : *SMTargFunc* ∈ ℕ → 0 .. *SMTargUB***END****A.1.7 Context C2\_CNT****CONTEXT** C2\_CNT**EXTENDS** C1\_CNT**SETS***PredictPath***CONSTANTS***PredPathFunc**SMPredUB**SMPredFunc***AXIOMS***axm1* : *SMPredUB* ∈ ℕ

$\text{axm2} : SMPredFunc \in \mathbb{N} \rightarrow 0 .. SMPredUB$   
 $\text{axm3} : PredPathFunc \in (YA \times YR \times \mathbb{N}) \rightarrow PredictPath$   
**END**

#### A.1.8 Context C3\_CNT

**CONTEXT** C3\_CNT

**EXTENDS** C2\_CNT

**SETS**

*SteerAngle*

**CONSTANTS**

*SteerAnglFunc*

**AXIOMS**

$\text{axm1} : SteerAnglFunc \in (TargetPath \times PredictPath) \rightarrow SteerAngle$   
**END**

#### A.1.9 Context C4\_CNT

**CONTEXT** C4\_CNT

**EXTENDS** C3\_CNT

**CONSTANTS**

*SMTargThreshold*

*SMPredThreshold*

**AXIOMS**

$\text{axm1} : SMTargThreshold \in \mathbb{N}$   
 $\text{axm2} : SMPredThreshold \in \mathbb{N}$   
 $\text{axm3} : SMTargThreshold < SMTargUB$   
 $\text{axm4} : SMPredThreshold < SMPredUB$   
**END**

#### A.1.10 Context C5\_CNT

**CONTEXT** C5\_CNT

**EXTENDS** C4\_CNT

**SETS**

*Light\_Colours*

**CONSTANTS**

```

    GREEN
    YELLOW
    RED
    OFF
AXIOMS

    axm1 : partition(Light_Colours, {GREEN}, {YELLOW}, {RED}, {OFF})
END

```

## A.2 Monitored Sub-Model of LCC

### A.2.1 Monitored Sub-Model: Abstract Level

```

MACHINE  Mntr_0
SEES    C0_MNR
VARIABLES

    lateralPos
    roadCurv
    yawAngl
    yawRate
    speed
INVARIANTS

    inv1 : lateralPos  $\subseteq$  LP
    inv2 : roadCurv  $\subseteq$  RC
    inv3 : yawAngl  $\subseteq$  YA
    inv4 : yawRate  $\subseteq$  YR
    inv5 : speed  $\subseteq$  0 .. maxSpd
EVENTS

Initialisation

    begin

        act1 : lateralPos :=  $\emptyset$ 
        act2 : roadCurv :=  $\emptyset$ 
        act3 : yawAngl :=  $\emptyset$ 
        act4 : yawRate :=  $\emptyset$ 
        act5 : speed :=  $\emptyset$ 

    end

Event    UpdateLateralPos  $\hat{=}$ 

    any

```

```

       $lp$ 
    where
       $grd1 : lp \subseteq LP$ 
    then
       $act1 : lateralPos := lp$ 
    end
  Event UpdateRoadCurve  $\hat{=}$ 
    any
       $rc$ 
    where
       $grd1 : rc \subseteq RC$ 
    then
       $act1 : roadCurv := rc$ 
    end
  Event UpdateYawAngle  $\hat{=}$ 
    any
       $ya$ 
    where
       $grd1 : ya \subseteq YA$ 
    then
       $act1 : yawAngl := ya$ 
    end
  Event UpdateYawRate  $\hat{=}$ 
    any
       $yr$ 
    where
       $grd1 : yr \subseteq YR$ 
    then
       $act1 : yawRate := yr$ 
    end
  Event UpdateLateralSpeed  $\hat{=}$ 
    any
       $sp$ 
    where
       $grd1 : sp \subseteq 0 .. maxSpd$ 
    then
       $act1 : speed := sp$ 
    end
  END

```

### A.2.2 Monitored Sub-Model: First Refinement

**MACHINE** Mntr\_1

**REFINES** Mntr\_0

**SEES** C1\_MNR

**VARIABLES**

*lateralPos*

*roadCurv*

*yawAngl*

*yawRate*

*speed*

*ACC\_status*

*indicator*

*steering*

**INVARIANTS**

*inv1* : *ACC\_status*  $\in$  *ACC\_STATUS*

*inv2* : *indicator*  $\in$  *BOOL*

*inv3* : *steering*  $\in$  *BOOL*

**EVENTS**

**Initialisation**

*extended*

**begin**

*act1* : *lateralPos* :=  $\emptyset$

*act2* : *roadCurv* :=  $\emptyset$

*act3* : *yawAngl* :=  $\emptyset$

*act4* : *yawRate* :=  $\emptyset$

*act5* : *speed* :=  $\emptyset$

*act6* : *ACC\_status* := *ACC\_Deactive*

*act7* : *indicator* := *FALSE*

*act8* : *steering* := *FALSE*

**end**

**Event** *UpdateLateralPos*  $\hat{=}$

**extends** *UpdateLateralPos*

**any**

*lp*

**where**

*grd1* : *lp*  $\subseteq$  *LP*

**then**

```

        act1 : lateralPos := lp
    end
Event   UpdateRoadCurve  $\hat{=}$ 
extends UpdateRoadCurve
    any
        rc
    where
        grd1 :  $rc \subseteq RC$ 
    then
        act1 : roadCurv := rc
    end
Event   UpdateYawAngle  $\hat{=}$ 
extends UpdateYawAngle
    any
        ya
    where
        grd1 :  $ya \subseteq YA$ 
    then
        act1 : yawAngl := ya
    end
Event   UpdateYawRate  $\hat{=}$ 
extends UpdateYawRate
    any
        yr
    where
        grd1 :  $yr \subseteq YR$ 
    then
        act1 : yawRate := yr
    end
Event   UpdatelateralSpeed  $\hat{=}$ 
extends UpdatelateralSpeed
    any
        sp
    where
        grd1 :  $sp \subseteq 0 .. maxSpd$ 
    then
        act1 : speed := sp
    end
Event   UpdateACC_Status  $\hat{=}$ 

```

```

    any
      st
    where

    then grd1 : st ∈ ACC_STATUS

    end   act1 : ACC_status := st
Event  UpdateIndicator ≡
    any
      indicate
    where

    then grd1 : indicate ∈ BOOL

    end   act1 : indicator := indicate
Event  UpdateSteering ≡
    any
      steer
    where

    then grd1 : steer ∈ BOOL

    end   act1 : steering := steer
END

```

### A.2.3 Monitored Sub-Model: Second Refinement

**MACHINE** *Mntr\_2*

**SEES** *C1\_MNR*

**VARIABLES**

*lateralPos*  
*roadCurv*  
*yawAngl*  
*yawRate*  
*speed*  
*ACC\_status*  
*indicator*  
*steering*

**INVARIANTS**



$\text{inv11} : \text{lateralPos} \subseteq LP$   
 $\text{inv12} : \text{roadCurv} \subseteq RC$   
 $\text{inv13} : \text{yawAngl} \subseteq YA$   
 $\text{inv14} : \text{yawRate} \subseteq YR$   
 $\text{inv15} : \text{speed} \subseteq 0 \dots \text{maxSpd}$   
 $\text{inv21} : \text{ACC\_status} \in \text{ACC\_STATUS}$   
 $\text{inv22} : \text{indicator} \in \text{BOOL}$   
 $\text{inv23} : \text{steering} \in \text{BOOL}$

## EVENTS

### Initialisation

**begin**

$\text{act1} : \text{lateralPos} := \emptyset$   
 $\text{act2} : \text{roadCurv} := \emptyset$   
 $\text{act3} : \text{yawAngl} := \emptyset$   
 $\text{act4} : \text{yawRate} := \emptyset$   
 $\text{act5} : \text{speed} := \emptyset$   
 $\text{act6} : \text{ACC\_status} := \text{ACC\_Deactive}$   
 $\text{act7} : \text{indicator} := \text{FALSE}$   
 $\text{act8} : \text{steering} := \text{FALSE}$

**end**

**Event**  $\text{UpdateLateralPos} \hat{=}$

**any**

$lp$   
**where**

$\text{grd1} : lp \subseteq LP$   
**then**

$\text{act1} : \text{lateralPos} := lp$   
**end**

**Event**  $\text{UpdateRoadCurve} \hat{=}$

**any**

$rc$   
**where**

$\text{grd1} : rc \subseteq RC$   
**then**

$\text{act1} : \text{roadCurv} := rc$   
**end**

**Event**  $\text{UpdateYawAngle} \hat{=}$

**any**

$ya$

```

    where

    then    grd1 : ya  $\subseteq$  YA

    end    act1 : yawAngl := ya

Event UpdateYawRate  $\hat{=}$ 
    any

    where    yr

    then    grd1 : yr  $\subseteq$  YR

    end    act1 : yawRate := yr

Event UpdatelateralSpeed  $\hat{=}$ 
    any

    where    sp

    then    grd1 : sp  $\subseteq$  0 .. maxSpd

    end    act1 : speed := sp

Event UpdateACC_Status  $\hat{=}$ 
    any

    where    st

    then    grd1 : st  $\in$  ACC_STATUS

    end    act1 : ACC_status := st

Event UpdateIndicator  $\hat{=}$ 
    any

    where    indicate

    then    grd1 : indicate  $\in$  BOOL

    end    act1 : indicator := indicate

Event UpdateSteering  $\hat{=}$ 
    any

```

```

    steer
  where
    grd1 : steer ∈ BOOL
  then
    act1 : steering := steer
  end
END

```

### A.3 Controlled Sub-Model of LCC

#### A.3.1 Controlled Sub-Model: Abstract Level

**MACHINE** Cntrl\_0

**SEES** C0\_CNT

**VARIABLES**

*targetPath*  
**INVARIANTS**

*inv1* : *targetPath* ∈ *TargetPath*  
**EVENTS**

**Initialisation**

```

  begin
    act1 : targetPath := TargetPath
  end
Event UpdateTargetPath ≡
  any
    latPos
    rdCurv
    offst
  where
    grd1 : latPos ∈ LP
    grd2 : rdCurv ∈ RC
    grd3 : offst ∈  $\mathbb{Z}$ 
  then
    act1 : targetPath := TargPathFunc(latPos ↦ rdCurv ↦ offst)
  end
END

```

### A.3.2 Controlled Sub-Model: First Refinement

**MACHINE** Cntrl\_1

**REFINES** Cntrl\_0

**SEES** C1\_CNT

**VARIABLES**

*targetPath*

*safeMrgTarg*

**INVARIANTS**

*inv1* : *safeMrgTarg*  $\in 0 \dots SMTargUB$

**EVENTS**

**Initialisation**

*extended*

**begin**

*act1* : *targetPath*  $\in TargetPath$

*act2* : *safeMrgTarg* := 0

**end**

**Event** *UpdateTargetPath*  $\hat{=}$

**extends** *UpdateTargetPath*

**any**

*latPos*

*rdCurv*

*offst*

*spd*

**where**

*grd1* : *latPos*  $\in LP$

*grd2* : *rdCurv*  $\in RC$

*grd3* : *offst*  $\in \mathbb{Z}$

*grd4* : *spd*  $\in \mathbb{N}$

**then**

*act1* : *targetPath* := *TargPathFunc*(*latPos*  $\mapsto$  *rdCurv*  $\mapsto$  *offst*)

*act2* : *safeMrgTarg* := *SMTargFunc*(*spd*)

**end**

**END**

### A.3.3 Controlled Sub-Model: Second Refinement

**MACHINE** Cntrl\_2

**REFINES** Cntrl\_1

**SEES** C2\_CNT

**VARIABLES**

*targetPath*  
*safeMrgTarg*  
*predictPath*  
*safeMrgPred*

**INVARIANTS**

*inv1* : *predictPath*  $\in$  *PredictPath*  
*inv2* : *safeMrgPred*  $\in$  0 .. *SMPredUB*

**EVENTS**

**Initialisation**

*extended*

**begin**

*act1* : *targetPath*  $\in$  *TargetPath*  
*act2* : *safeMrgTarg* := 0  
*act3* : *predictPath*  $\in$  *PredictPath*  
*act4* : *safeMrgPred* := 0

**end**

**Event** *UpdateTargetPath*  $\hat{=}$

**extends** *UpdateTargetPath*

**any**

*latPos*  
*rdCurv*  
*offst*  
*spd*

**where**

*grd1* : *latPos*  $\in$  *LP*  
*grd2* : *rdCurv*  $\in$  *RC*  
*grd3* : *offst*  $\in$   $\mathbb{Z}$   
*grd4* : *spd*  $\in$   $\mathbb{N}$

**then**

*act1* : *targetPath* := *TargPathFunc*(*latPos*  $\mapsto$  *rdCurv*  $\mapsto$  *offst*)  
*act2* : *safeMrgTarg* := *SMTargFunc*(*spd*)

**end**

**Event** *UpdatePredictPath*  $\hat{=}$

**any**

*yawAng*  
*yawR*

```

      spd
where

      grd1 : yawAng ∈ YA
      grd2 : yawR ∈ YR
      grd3 : spd ∈ ℕ
then

      act1 : predictPath := PredPathFunc(yawAng ↦ yawR ↦ spd)
      act2 : safeMrgPred := SMPredFunc(spd)
end
END

```

#### A.3.4 Controlled Sub-Model: Third Refinement

**MACHINE** Cntrl\_3

**REFINES** Cntrl\_2

**SEES** C3\_CNT, C0\_MOD

**VARIABLES**

*targetPath*  
*safeMrgTarg*  
*predictPath*  
*safeMrgPred*  
*steeringAngle*

**INVARIANTS**

*inv1* : *steeringAngle* ∈ *SteerAngle*

**EVENTS**

**Initialisation**

*extended*

**begin**

*act1* : *targetPath* :∈ *TargetPath*  
*act2* : *safeMrgTarg* := 0  
*act3* : *predictPath* :∈ *PredictPath*  
*act4* : *safeMrgPred* := 0  
*act5* : *steeringAngle* :∈ *SteerAngle*

**end**

**Event** *UpdateTargetPath* ≐

**extends** *UpdateTargetPath*

**any**

*latPos*

```

    rdCurv
    offst
    spd
where
    grd1 : latPos ∈ LP
    grd2 : rdCurv ∈ RC
    grd3 : offst ∈ ℤ
    grd4 : spd ∈ ℕ
then
    act1 : targetPath := TargPathFunc(latPos ↦ rdCurv ↦ offst)
    act2 : safeMrgTarg := SMTargFunc(spd)
end
Event UpdatePredictPath ≐
extends UpdatePredictPath
any
    yawAng
    yawR
    spd
where
    grd1 : yawAng ∈ YA
    grd2 : yawR ∈ YR
    grd3 : spd ∈ ℕ
then
    act1 : predictPath := PredPathFunc(yawAng ↦ yawR ↦ spd)
    act2 : safeMrgPred := SMPredFunc(spd)
end
Event UpdateSteeringAngle ≐
any
    mode
where
    grd1 : mode ∈ STATUS
    grd2 : mode = Active
    grd3 : safeMrgTarg > 0
    grd4 : safeMrgPred > 0
then
    act1 : steeringAngle := SteerAnglFunc(targetPath ↦ predictPath)
end
END

```

### A.3.5 Controlled Sub-Model: Fourth Refinement

**MACHINE** Cntrl\_4

**REFINES** Cntrl\_3

**SEES** C4\_CNT, C0\_MOD

**VARIABLES**

*targetPath*

*safeMrgTarg*

*predictPath*

*safeMrgPred*

*steeringAngle*

*warning*

**INVARIANTS**

*inv1* : *warning*  $\in$  *BOOL*

**EVENTS**

**Initialisation**

*extended*

**begin**

*act1* : *targetPath*  $\in$  *TargetPath*

*act2* : *safeMrgTarg* := 0

*act3* : *predictPath*  $\in$  *PredictPath*

*act4* : *safeMrgPred* := 0

*act5* : *steeringAngle*  $\in$  *SteerAngle*

*act6* : *warning* := *FALSE*

**end**

**Event** *UpdateTargetPath*  $\hat{=}$

**extends** *UpdateTargetPath*

**any**

*latPos*

*rdCurv*

*offst*

*spd*

**where**

*grd1* : *latPos*  $\in$  *LP*

*grd2* : *rdCurv*  $\in$  *RC*

*grd3* : *offst*  $\in$   $\mathbb{Z}$

*grd4* : *spd*  $\in$   $\mathbb{N}$

**then**

*act1* : *targetPath* := *TargPathFunc*(*latPos*  $\mapsto$  *rdCurv*  $\mapsto$  *offst*)



```

        act2 : safeMrgTarg := SMTargFunc(spdl)
    end
Event UpdatePredictPath  $\hat{=}$ 
extends UpdatePredictPath
    any
        yawAng
        yawR
        spdl
    where
        grd1 : yawAng  $\in$  YA
        grd2 : yawR  $\in$  YR
        grd3 : spdl  $\in$  N
    then
        act1 : predictPath := PredPathFunc(yawAng  $\mapsto$  yawR  $\mapsto$  spdl)
        act2 : safeMrgPred := SMPredFunc(spdl)
    end
Event UpdateSteeringAngle  $\hat{=}$ 
extends UpdateSteeringAngle
    any
        mode
    where
        grd1 : mode  $\in$  STATUS
        grd2 : mode = Active
        grd3 : safeMrgTarg > 0
        grd4 : safeMrgPred > 0
    then
        act1 : steeringAngle := SteerAnglFunc(targetPath  $\mapsto$  predictPath)
    end
Event SetWarning_NoTargetPath  $\hat{=}$ 
    any
        mode
    where
        grd1 : mode  $\in$  STATUS
        grd2 : mode = Active
        grd3 : warning = FALSE
        grd4 : safeMrgTarg < SMTargThreshold
        grd5 : safeMrgTarg > 0
    then
        act1 : warning := TRUE

```

```

end

Event SetWarning_NoPredictPath  $\hat{=}$ 
  any

    mode
  where

    grd1 : mode  $\in$  STATUS
    grd2 : mode = Active
    grd3 : warning = FALSE
    grd4 : safeMrgPred < SPredThreshold
    grd5 : safeMrgPred > 0
  then

    act1 : warning := TRUE
  end

Event StopWarning  $\hat{=}$ 
  when

    grd1 : warning = TRUE
  then

    act1 : warning := FALSE
  end

END

```

### A.3.6 Controlled Sub-Model: Fifth Refinement

```

MACHINE Cntrl_5
REFINES Cntrl_4
SEES C5_CNT, C0_MOD
VARIABLES

```

```

  targetPath
  safeMrgTarg
  predictPath
  safeMrgPred
  steeringAngle
  warning
  displayClr

```

#### **INVARIANTS**

```

inv1 : displayClr  $\in$  Light_Colours
inv2 : warning = FALSE  $\Rightarrow$  displayClr  $\in$  {GREEN, OFF}
inv3 : warning = TRUE  $\Rightarrow$  displayClr = YELLOW

```

**EVENTS****Initialisation***extended***begin**

*act1* : *targetPath* :∈ *TargetPath*  
*act2* : *safeMrgTarg* := 0  
*act3* : *predictPath* :∈ *PredictPath*  
*act4* : *safeMrgPred* := 0  
*act5* : *steeringAngle* :∈ *SteerAngle*  
*act6* : *warning* := *FALSE*  
*act7* : *displayClr* := *OFF*

**end****Event** *UpdateTargetPath* ≐**extends** *UpdateTargetPath***any**

*latPos*  
*rdCurv*  
*offst*  
*spd*

**where**

*grd1* : *latPos* ∈ *LP*  
*grd2* : *rdCurv* ∈ *RC*  
*grd3* : *offst* ∈  $\mathbb{Z}$   
*grd4* : *spd* ∈  $\mathbb{N}$

**then**

*act1* : *targetPath* := *TargPathFunc*(*latPos* ↦ *rdCurv* ↦ *offst*)  
*act2* : *safeMrgTarg* := *SMTargFunc*(*spd*)

**end****Event** *UpdatePredictPath* ≐**extends** *UpdatePredictPath***any**

*yawAng*  
*yawR*  
*spd*

**where**

*grd1* : *yawAng* ∈ *YA*  
*grd2* : *yawR* ∈ *YR*  
*grd3* : *spd* ∈  $\mathbb{N}$

**then**

*act1* : *predictPath* := *PredPathFunc*(*yawAng* ↦ *yawR* ↦ *spd*)

```

        act2 : safeMrgPred := SMPredFunc(spd)
    end
Event UpdateSteeringAngle  $\hat{=}$ 
extends UpdateSteeringAngle
    any
        mode
    where
        grd1 : mode  $\in$  STATUS
        grd2 : mode = Active
        grd3 : safeMrgTarg > 0
        grd4 : safeMrgPred > 0
    then
        act1 : steeringAngle := SteerAnglFunc(targetPath  $\mapsto$  predictPath)
    end
Event SetDisplayGreen  $\hat{=}$ 
    when
        grd1 : warning = FALSE
    then
        act1 : displayClr := GREEN
    end
Event SetWarning_NoTargetPath_SetDisplay  $\hat{=}$ 
extends SetWarning_NoTargetPath
    any
        mode
    where
        grd1 : mode  $\in$  STATUS
        grd2 : mode = Active
        grd3 : warning = FALSE
        grd4 : safeMrgTarg < SMTargThreshold
        grd5 : safeMrgTarg > 0
    then
        act1 : warning := TRUE
        act2 : displayClr := YELLOW
    end
Event SetWarning_NoPredictPath_SetDisplay  $\hat{=}$ 
extends SetWarning_NoPredictPath
    any
        mode
    where

```

```

    grd1 : mode ∈ STATUS
    grd2 : mode = Active
    grd3 : warning = FALSE
    grd4 : safeMrgPred < SMPredThreshold
    grd5 : safeMrgPred > 0
  then
    act1 : warning := TRUE
    act2 : displayClr := YELLOW
  end
Event StopWarning_ActiveMode ≐
extends StopWarning
  any
  where mode
  then
    grd1 : warning = TRUE
    grd2 : mode ∈ STATUS
    grd3 : mode = Active
  then
    act1 : warning := FALSE
    act2 : displayClr := GREEN
  end
Event StopWarningInactivateMode ≐
extends StopWarning
  when
  then
    grd1 : warning = TRUE
  then
    act1 : warning := FALSE
    act2 : displayClr := OFF
  end
Event DisplayOff ≐
  when
  then
    grd1 : warning = FALSE
  then
    act1 : displayClr := OFF
  end
END

```

### A.3.7 Controlled Sub-Model: Sixth Refinement

**MACHINE** Cntrl\_6

**SEES** C5\_CNT, C0\_MOD

## VARIABLES

*targetPath*  
*safeMrgTarg*  
*predictPath*  
*safeMrgPred*  
*steeringAngle*  
*warning*  
*displayClr*

## INVARIANTS

*inv01* :  $targetPath \in TargetPath$   
*inv11* :  $safeMrgTarg \in 0 .. SMTargUB$   
*inv21* :  $predictPath \in PredictPath$   
*inv22* :  $safeMrgPred \in 0 .. SMPredUB$   
*inv31* :  $steeringAngle \in SteerAngle$   
*inv41* :  $warning \in BOOL$   
*inv51* :  $displayClr \in Light\_Colours$   
*inv52* :  $warning = FALSE \Rightarrow displayClr \in \{GREEN, OFF\}$   
*inv53* :  $warning = TRUE \Rightarrow displayClr = YELLOW$

## EVENTS

### Initialisation

**begin**

*act1* :  $targetPath : \in TargetPath$   
*act2* :  $safeMrgTarg := 0$   
*act3* :  $predictPath : \in PredictPath$   
*act4* :  $safeMrgPred := 0$   
*act5* :  $steeringAngle : \in SteerAngle$   
*act6* :  $warning := FALSE$   
*act7* :  $displayClr := OFF$

**end**

**Event** *UpdateTargetPath*  $\hat{=}$

**any**

*latPos*  
*rdCurv*  
*offst*  
*spd*

**where**

*grd1* :  $latPos \in LP$

```

    grd2 : rdCurv ∈ RC
    grd3 : offst ∈ ℤ
    grd4 : spd ∈ ℕ
  then
    act1 : targetPath := TargPathFunc(latPos ↦ rdCurv ↦ offst)
    act2 : safeMrgTarg := SMTargFunc(spd)
  end
Event UpdatePredictPath ≐
  any
    yawAng
    yawR
    spd
  where
    grd1 : yawAng ∈ YA
    grd2 : yawR ∈ YR
    grd3 : spd ∈ ℕ
  then
    act1 : predictPath := PredPathFunc(yawAng ↦ yawR ↦ spd)
    act2 : safeMrgPred := SMPredFunc(spd)
  end
Event UpdateSteeringAngle ≐
  any
    mode
  where
    grd1 : mode ∈ STATUS
    grd2 : mode = Active
    grd3 : safeMrgTarg > 0
    grd4 : safeMrgPred > 0
  then
    act1 : steeringAngle := SteerAnglFunc(targetPath ↦ predictPath)
  end
Event SetDisplayGreen ≐
  when
    grd1 : warning = FALSE
  then
    act1 : displayClr := GREEN
  end
Event SetWarning_NoTargetPath_SetDisplay ≐
  any

```

```

    mode
where

    grd1 : mode ∈ STATUS
    grd2 : mode = Active
    grd3 : warning = FALSE
    grd4 : safeMrgTarg < SMTargThreshold
    grd5 : safeMrgTarg > 0
then

    act1 : warning := TRUE
    act2 : displayClr := YELLOW
end

Event SetWarning_NoPredictPath_SetDisplay ≐
any

    mode
where

    grd1 : mode ∈ STATUS
    grd2 : mode = Active
    grd3 : warning = FALSE
    grd4 : safeMrgPred < SPredThreshold
    grd5 : safeMrgPred > 0
then

    act1 : warning := TRUE
    act2 : displayClr := YELLOW
end

Event StopWarning_ActiveMode_SetDisplay ≐
any

    mode
where

    grd1 : warning = TRUE
    grd2 : mode ∈ STATUS
    grd3 : mode = Active
then

    act1 : warning := FALSE
    act2 : displayClr := GREEN
end

Event StopWarningInactivateMode_DisplayOff ≐
when

    grd1 : warning = TRUE
then

    act1 : warning := FALSE

```



```

        act2 : displayClr := OFF
    end
Event DisplayOff  $\hat{=}$ 
    when

        grd1 : warning = FALSE
    then

        act1 : displayClr := OFF
    end
END

```

## A.4 Mode Sub-Model of LCC

### A.4.1 Mode Sub-Model: Abstract Level

**MACHINE** *Mode\_0*

**SEES** *C0\_MOD*

**VARIABLES**

*status*

**INVARIANTS**

*inv1* : *status*  $\in$  *STATUS*

**EVENTS**

**Initialisation**

**begin**

*act1* : *status* := *Unavailable*

**end**

**Event** *SwitchOn*  $\hat{=}$

**when**

*grd1* : *status* = *Off*

**then**

*act1* : *status* := *Standby*

**end**

**Event** *Activate*  $\hat{=}$

**when**

*grd1* : *status*  $\in$  {*Standby*, *Overridden*}

**then**

*act1* : *status* := *Active*

**end**

**Event** *Override*  $\hat{=}$

```

    when

    then      grd1 : status = Active

    end      act1 : status := Overridden
Event SwitchOff  $\hat{=}$ 
    when

    then      grd1 : status  $\in \{Active, Standby, Overridden, Unavailable\}$ 

    end      act1 : status := Off
Event LCC_Unavailable  $\hat{=}$ 
    when

    then      grd1 : status  $\in \{Active, Standby, Overridden, Off\}$ 

    end      act1 : status := Unavailable
END

```

#### A.4.2 Mode Sub-Model: First Refinement

```

MACHINE Mode_1
REFINES Mode_0
SEES C0_MOD, C1_MNR
VARIABLES

    status
EVENTS
Initialisation
    extended
    begin

    end      act1 : status := Unavailable
Event LCC_Available  $\hat{=}$ 
refines SwitchOff
    any

    where      ACC_st

    then      grd1 : ACC_st  $\in ACC\_STATUS$ 

```

```

        grd2 : ACC_st = ACC_Active
        grd3 : status = Unavailable
    then
        act1 : status := Off
    end
Event SwitchOn  $\hat{=}$ 
extends SwitchOn
    any
    where ACC_st
        grd1 : status = Off
        grd2 : ACC_st  $\in$  ACC_STATUS
        grd3 : ACC_st = ACC_Active
    then
        act1 : status := Standby
    end
Event Activate  $\hat{=}$ 
extends Activate
    any
    where ACC_st
        grd1 : status  $\in$  {Standby, Overridden}
        grd2 : ACC_st  $\in$  ACC_STATUS
        grd3 : ACC_st = ACC_Active
    then
        act1 : status := Active
    end
Event Override  $\hat{=}$ 
extends Override
    any
    where ACC_st
        grd1 : status = Active
        grd2 : ACC_st  $\in$  ACC_STATUS
        grd3 : ACC_st = ACC_Active
    then
        act1 : status := Overridden
    end
Event SwitchOff  $\hat{=}$ 
refines SwitchOff

```

```

    any
      ACC_st
    where
      grd1 : ACC_st ∈ ACC_STATUS
      grd2 : ACC_st = ACC_Active
      grd3 : status ∈ {Active, Standby, Overridden}
    then
      act1 : status := Off
    end
  Event LCC_Unavailable ≡
  extends LCC_Unavailable
    any
      ACC_st
    where
      grd1 : status ∈ {Active, Standby, Overridden, Off}
      grd2 : ACC_st ∈ ACC_STATUS
      grd3 : ACC_st = ACC_Deactive
    then
      act1 : status := Unavailable
    end
  END

```

#### A.4.3 Mode Sub-Model: Second Refinement

```

MACHINE Mode_2
REFINES Mode_1
SEES C0_MOD, C1_MNR
VARIABLES
  status
EVENTS
  Initialisation
    extended
    begin
      act1 : status := Unavailable
    end
  Event LCC_Available ≡
  extends LCC_Available
    any
      ACC_st

```

```

where

    grd1 :  $ACC\_st \in ACC\_STATUS$ 
    grd2 :  $ACC\_st = ACC\_Active$ 
    grd3 :  $status = Unavailable$ 
then

    act1 :  $status := Off$ 
end

Event SwitchOn  $\hat{=}$ 
extends SwitchOn

    any

         $ACC\_st$ 
        where

            grd1 :  $status = Off$ 
            grd2 :  $ACC\_st \in ACC\_STATUS$ 
            grd3 :  $ACC\_st = ACC\_Active$ 
        then

            act1 :  $status := Standby$ 
        end

Event Activate  $\hat{=}$ 
refines Activate

    any

         $ACC\_st$ 
         $indicate$ 
        where

            grd1 :  $status = Standby$ 
            grd2 :  $ACC\_st \in ACC\_STATUS$ 
            grd3 :  $ACC\_st = ACC\_Active$ 
            grd4 :  $indicate \in BOOL$ 
            grd5 :  $indicate = FALSE$ 
        then

            act1 :  $status := Active$ 
        end

Event Resume  $\hat{=}$ 
refines Activate

    any

         $ACC\_st$ 
         $indicate$ 
        where

            grd1 :  $status = Overridden$ 
            grd2 :  $ACC\_st \in ACC\_STATUS$ 

```

```

    grd3 : ACC_st = ACC_Active
    grd4 : indicate ∈ BOOL
    grd5 : indicate = FALSE
  then

    act1 : status := Active
  end

Event Override_Indicator ≐
extends Override
  any
    ACC_st
    indicate
  where

    grd1 : status = Active
    grd2 : ACC_st ∈ ACC_STATUS
    grd3 : ACC_st = ACC_Active
    grd4 : indicate ∈ BOOL
    grd5 : indicate = FALSE
  then

    act1 : status := Overridden
  end

Event Override ≐
extends Override
  any
    ACC_st
  where

    grd1 : status = Active
    grd2 : ACC_st ∈ ACC_STATUS
    grd3 : ACC_st = ACC_Active
  then

    act1 : status := Overridden
  end

Event SwitchOff ≐
extends SwitchOff
  any
    ACC_st
  where

    grd1 : ACC_st ∈ ACC_STATUS
    grd2 : ACC_st = ACC_Active
    grd3 : status ∈ {Active, Standby, Overridden}
  then

```

```

        act1 : status := Off
    end
Event LCC_Unavailable  $\hat{=}$ 
extends LCC_Unavailable
    any
        ACC_st
    where
        grd1 : status  $\in$  {Active, Standby, Overridden, Off}
        grd2 : ACC_st  $\in$  ACC_STATUS
        grd3 : ACC_st = ACC_Deactive
    then
        act1 : status := Unavailable
    end
END

```

#### A.4.4 Mode Sub-Model: Third Refinement

```

MACHINE Mode_3
REFINES Mode_2
SEES C0_MOD, C1_MNR
VARIABLES
    status
EVENTS
Initialisation
    extended
    begin
        act1 : status := Unavailable
    end
Event LCC_Available  $\hat{=}$ 
extends LCC_Available
    any
        ACC_st
    where
        grd1 : ACC_st  $\in$  ACC_STATUS
        grd2 : ACC_st = ACC_Active
        grd3 : status = Unavailable
    then
        act1 : status := Off
    end

```

**Event** *SwitchOn*  $\hat{=}$

**extends** *SwitchOn*

**any**

*ACC\_st*

**where**

*grd1* : *status* = *Off*

*grd2* : *ACC\_st*  $\in$  *ACC\_STATUS*

*grd3* : *ACC\_st* = *ACC\_Active*

**then**

*act1* : *status* := *Standby*

**end**

**Event** *Activate*  $\hat{=}$

**extends** *Activate*

**any**

*ACC\_st*

*indicate*

*steer*

**where**

*grd1* : *status* = *Standby*

*grd2* : *ACC\_st*  $\in$  *ACC\_STATUS*

*grd3* : *ACC\_st* = *ACC\_Active*

*grd4* : *indicate*  $\in$  *BOOL*

*grd5* : *indicate* = *FALSE*

*grd6* : *steer*  $\in$  *BOOL*

*grd7* : *steer* = *FALSE*

**then**

*act1* : *status* := *Active*

**end**

**Event** *Resume*  $\hat{=}$

**extends** *Resume*

**any**

*ACC\_st*

*indicate*

*steer*

**where**

*grd1* : *status* = *Overridden*

*grd2* : *ACC\_st*  $\in$  *ACC\_STATUS*

*grd3* : *ACC\_st* = *ACC\_Active*

*grd4* : *indicate*  $\in$  *BOOL*

*grd5* : *indicate* = *FALSE*



```

        grd6 : steer ∈ BOOL
        grd7 : steer = FALSE
    then

        act1 : status := Active
    end

Event Override_Indicator ≐
extends Override_Indicator
    any
        ACC_st
        indicate
    where
        grd1 : status = Active
        grd2 : ACC_st ∈ ACC_STATUS
        grd3 : ACC_st = ACC_Active
        grd4 : indicate ∈ BOOL
        grd5 : indicate = FALSE
    then

        act1 : status := Overridden
    end

Event Override_Steering ≐
extends Override
    any
        ACC_st
        steer
    where
        grd1 : status = Active
        grd2 : ACC_st ∈ ACC_STATUS
        grd3 : ACC_st = ACC_Active
        grd4 : steer ∈ BOOL
        grd5 : steer = TRUE
    then

        act1 : status := Overridden
    end

Event SwitchOff_FromActive ≐
refines SwitchOff
    any
        ACC_st
    where
        grd1 : ACC_st ∈ ACC_STATUS
        grd2 : ACC_st = ACC_Active

```

```

    then    grd3 : status = Active
  then
    act1 : status := Off
  end
Event SwitchOff_FromInactive  $\hat{=}$ 
refines SwitchOff
  any
    ACC_st
  where
    grd1 : ACC_st  $\in$  ACC_STATUS
    grd2 : ACC_st = ACC_Active
    grd3 : status  $\in$  {Standby, Overridden}
  then
    act1 : status := Off
  end
Event LCC_Unavailable_FromActive  $\hat{=}$ 
refines LCC_Unavailable
  any
    ACC_st
  where
    grd1 : status = Active
    grd2 : ACC_st  $\in$  ACC_STATUS
    grd3 : ACC_st = ACC_Deactive
  then
    act1 : status := Unavailable
  end
Event LCC_Unavailable_FromInactive  $\hat{=}$ 
refines LCC_Unavailable
  any
    ACC_st
  where
    grd1 : status  $\in$  {Standby, Overridden, Off}
    grd2 : ACC_st  $\in$  ACC_STATUS
    grd3 : ACC_st = ACC_Deactive
  then
    act1 : status := Unavailable
  end
END

```

#### A.4.5 Mode Sub-Model: Fourth Refinement

**MACHINE** Mode\_4

**SEES** C0\_MOD, C1\_MNR

**VARIABLES**

*status*

**INVARIANTS**

*inv1* : *status* ∈ *STATUS*

**EVENTS**

**Initialisation**

**begin**

**end**     *act1* : *status* := *Unavailable*

**Event** *LCC\_Available*  $\hat{=}$

**any**

**where**     *ACC\_st*

*grd1* : *ACC\_st* ∈ *ACC\_STATUS*

*grd2* : *ACC\_st* = *ACC\_Active*

*grd3* : *status* = *Unavailable*

**then**

**end**     *act1* : *status* := *Off*

**Event** *SwitchOn*  $\hat{=}$

**any**

**where**     *ACC\_st*

*grd1* : *status* = *Off*

*grd2* : *ACC\_st* ∈ *ACC\_STATUS*

*grd3* : *ACC\_st* = *ACC\_Active*

**then**

**end**     *act1* : *status* := *Standby*

**Event** *Activate*  $\hat{=}$

**any**

*ACC\_st*

*indicate*

*steer*

**where**

```

    grd1 : status = Standby
    grd2 : ACC_st ∈ ACC_STATUS
    grd3 : ACC_st = ACC_Active
    grd4 : indicate ∈ BOOL
    grd5 : indicate = FALSE
    grd6 : steer ∈ BOOL
    grd7 : steer = FALSE
  then
    act1 : status := Active
  end
Event Resume ≐
  any
    ACC_st
    indicate
    steer
  where
    grd1 : status = Overridden
    grd2 : ACC_st ∈ ACC_STATUS
    grd3 : ACC_st = ACC_Active
    grd4 : indicate ∈ BOOL
    grd5 : indicate = FALSE
    grd6 : steer ∈ BOOL
    grd7 : steer = FALSE
  then
    act1 : status := Active
  end
Event Override_Indicator ≐
  any
    ACC_st
    indicate
  where
    grd1 : status = Active
    grd2 : ACC_st ∈ ACC_STATUS
    grd3 : ACC_st = ACC_Active
    grd4 : indicate ∈ BOOL
    grd5 : indicate = FALSE
  then
    act1 : status := Overridden
  end
Event Override_Steering ≐
  any

```

```

    ACC_st
    steer
  where
    grd1 : status = Active
    grd2 : ACC_st ∈ ACC_STATUS
    grd3 : ACC_st = ACC_Active
    grd4 : steer ∈ BOOL
    grd5 : steer = TRUE
  then
    act1 : status := Overridden
  end
Event SwitchOff_FromActive ≐
  any
    ACC_st
  where
    grd1 : ACC_st ∈ ACC_STATUS
    grd2 : ACC_st = ACC_Active
    then
      grd3 : status = Active
    then
      act1 : status := Off
    end
  end
Event SwitchOff_FromInactive ≐
  any
    ACC_st
  where
    grd1 : ACC_st ∈ ACC_STATUS
    grd2 : ACC_st = ACC_Active
    grd3 : status ∈ {Standby, Overridden}
  then
    act1 : status := Off
  end
Event LCC_Unavailable_FromActive ≐
  any
    ACC_st
  where
    grd1 : status = Active
    grd2 : ACC_st ∈ ACC_STATUS
    grd3 : ACC_st = ACC_Deactive
  then
    act1 : status := Unavailable

```

```

    end
Event LCC_Unavailable_FromInactive  $\hat{=}$ 
    any
        ACC_st
    where
        grd1 : status  $\in$  {Standby, Overridden, Off}
        grd2 : ACC_st  $\in$  ACC_STATUS
        grd3 : ACC_st = ACC_Deactive
    then
        act1 : status := Unavailable
    end
END

```

## A.5 Commanded Sub-Model of LCC

### A.5.1 Commanded Sub-Model: Abstract Level

```

MACHINE Cmnd_0
SEES C0_CMN
VARIABLES
    offset
INVARIANTS
    inv1 : offset  $\in$  offsetLB .. offsetUB
EVENTS
Initialisation
    begin
        act1 : offset := 0
    end
Event IncreaseOffset  $\hat{=}$ 
    when
        grd1 : offset < offsetUB
    then
        act1 : offset := offset + 1
    end
Event DecreaseOffset  $\hat{=}$ 
    when
        grd1 : offset > offsetLB
    then

```

```

        act1 : offset := offset - 1
    end
END

```

### A.5.2 Commanded Sub-Model: First Refinement

**MACHINE** Cmnd\_1

**REFINES** Cmnd\_0

**SEES** C0\_CMN, C0\_MOD

**VARIABLES**

*offset*

**EVENTS**

**Initialisation**

*extended*

**begin**

```

        act1 : offset := 0
    end

```

**Event** *IncreaseOffset*  $\hat{=}$

**extends** *IncreaseOffset*

**any**

**where** *mode*

*grd1* : *offset* < *offsetUB*

*grd2* : *mode*  $\in$  *STATUS*

*grd3* : *mode* = *Active*

**then**

```

        act1 : offset := offset + 1
    end

```

**Event** *DecreaseOffset*  $\hat{=}$

**extends** *DecreaseOffset*

**any**

**where** *mode*

*grd1* : *offset* > *offsetLB*

*grd2* : *mode*  $\in$  *STATUS*

*grd3* : *mode* = *Active*

**then**

```

        act1 : offset := offset - 1
    end

```

**END**

### A.5.3 Commanded Sub-Model: Second Refinement

**MACHINE** Cmnd\_2

**SEES** C0\_CMN, C0\_MOD

**VARIABLES**

*offset*

**INVARIANTS**

*inv1* :  $offset \in offsetLB .. offsetUB$

**EVENTS**

**Initialisation**

**begin**

*act1* :  $offset := 0$

**end**

**Event** *IncreaseOffset*  $\hat{=}$

**any**

**where** *mode*

*grd1* :  $offset < offsetUB$

*grd2* :  $mode \in STATUS$

*grd3* :  $mode = Active$

**then**

*act1* :  $offset := offset + 1$

**end**

**Event** *DecreaseOffset*  $\hat{=}$

**any**

**where** *mode*

*grd1* :  $offset > offsetLB$

*grd2* :  $mode \in STATUS$

*grd3* :  $mode = Active$

**then**

*act1* :  $offset := offset - 1$

**end**

**END**

## A.6 Composition of the MCMC Sub-Models of LCC

**MACHINE** LCC\_M0



**SEES** C1\_MNR, C5\_CNT, C0\_MOD, C0\_CMN

## VARIABLES

*lateralPos*  
*roadCurv*  
*yawAngl*  
*yawRate*  
*speed*  
*ACC\_status*  
*indicator*  
*steering*  
*targetPath*  
*safeMrgTarg*  
*predictPath*  
*safeMrgPred*  
*steeringAngle*  
*warning*  
*displayClr*  
*status*  
*offset*

## INVARIANTS

LCC\_Mntr\_2\_inv11 : *lateralPos*  $\subseteq LP$   
 LCC\_Mntr\_2\_inv12 : *roadCurv*  $\subseteq RC$   
 LCC\_Mntr\_2\_inv13 : *yawAngl*  $\subseteq YA$   
 LCC\_Mntr\_2\_inv14 : *yawRate*  $\subseteq YR$   
 LCC\_Mntr\_2\_inv15 : *speed*  $\subseteq 0 \dots maxSpd$   
 LCC\_Mntr\_2\_inv21 : *ACC\_status*  $\in ACC\_STATUS$   
 LCC\_Mntr\_2\_inv22 : *indicator*  $\in BOOL$   
 LCC\_Mntr\_2\_inv23 : *steering*  $\in BOOL$   
 LCC\_Cntrl\_6\_inv01 : *targetPath*  $\in TargetPath$   
 LCC\_Cntrl\_6\_inv11 : *safeMrgTarg*  $\in 0 \dots SMTargUB$   
 LCC\_Cntrl\_6\_inv21 : *predictPath*  $\in PredictPath$   
 LCC\_Cntrl\_6\_inv22 : *safeMrgPred*  $\in 0 \dots SMPredUB$   
 LCC\_Cntrl\_6\_inv31 : *steeringAngle*  $\in SteerAngle$   
 LCC\_Cntrl\_6\_inv41 : *warning*  $\in BOOL$   
 LCC\_Cntrl\_6\_inv51 : *displayClr*  $\in Light\_Colours$   
 LCC\_Cntrl\_6\_inv52 : *warning* = FALSE  $\Rightarrow displayClr \in \{GREEN, OFF\}$   
 LCC\_Cntrl\_6\_inv53 : *warning* = TRUE  $\Rightarrow displayClr = YELLOW$

`LCC_Mode_4_inv1` :  $status \in STATUS$

`LCC_Cmnd_2_inv1` :  $offset \in offsetLB .. offsetUB$

## EVENTS

### Initialisation

**begin**

`LCC_Mntr_2_act1` :  $lateralPos := \emptyset$

`LCC_Mntr_2_act2` :  $roadCurv := \emptyset$

`LCC_Mntr_2_act3` :  $yawAngl := \emptyset$

`LCC_Mntr_2_act4` :  $yawRate := \emptyset$

`LCC_Mntr_2_act5` :  $speed := \emptyset$

`LCC_Mntr_2_act6` :  $ACC\_status := ACC\_Deactive$

`LCC_Mntr_2_act7` :  $indicator := FALSE$

`LCC_Mntr_2_act8` :  $steering := FALSE$

`LCC_Cntrl_6_act1` :  $targetPath \in TargetPath$

`LCC_Cntrl_6_act2` :  $safeMrgTarg := 0$

`LCC_Cntrl_6_act3` :  $predictPath \in PredictPath$

`LCC_Cntrl_6_act4` :  $safeMrgPred := 0$

`LCC_Cntrl_6_act5` :  $steeringAngle \in SteerAngle$

`LCC_Cntrl_6_act6` :  $warning := FALSE$

`LCC_Cntrl_6_act7` :  $displayClr := OFF$

`LCC_Mode_4_act1` :  $status := Unavailable$

`LCC_Cmnd_2_act1` :  $offset := 0$

**end**

**Event**  $UpdateLateralPos \hat{=}$

**any**

$lp$

**where**

`LCC_Mntr_2_grd1` :  $lp \subseteq LP$

**then**

`LCC_Mntr_2_act1` :  $lateralPos := lp$

**end**

**Event**  $UpdateRoadCurve \hat{=}$

**any**

$rc$

**where**

`LCC_Mntr_2_grd1` :  $rc \subseteq RC$

**then**

`LCC_Mntr_2_act1` :  $roadCurv := rc$

**end**

**Event**  $UpdateYawAngle \hat{=}$

```

    any
      ya
    where

      LCC_Mntr_2_grd1 :  $ya \subseteq YA$ 
    then

      LCC_Mntr_2_act1 :  $yawAngl := ya$ 
    end
Event UpdateYawRate  $\hat{=}$ 
  any
    yr
  where

    LCC_Mntr_2_grd1 :  $yr \subseteq YR$ 
  then

    LCC_Mntr_2_act1 :  $yawRate := yr$ 
  end
Event UpdatelateralSpeed  $\hat{=}$ 
  any
    sp
  where

    LCC_Mntr_2_grd1 :  $sp \subseteq 0 .. maxSpd$ 
  then

    LCC_Mntr_2_act1 :  $speed := sp$ 
  end
Event UpdateACC_Status  $\hat{=}$ 
  any
    st
  where

    LCC_Mntr_2_grd1 :  $st \in ACC\_STATUS$ 
  then

    LCC_Mntr_2_act1 :  $ACC\_status := st$ 
  end
Event UpdateIndicator  $\hat{=}$ 
  any
    indicate
  where

    LCC_Mntr_2_grd1 :  $indicate \in BOOL$ 
  then

    LCC_Mntr_2_act1 :  $indicator := indicate$ 
  end
Event UpdateSteering  $\hat{=}$ 

```

```

any

    steer
where

    LCC_Mntr_2_grd1 : steer ∈ BOOL
then

    LCC_Mntr_2_act1 : steering := steer
end
Event UpdateTargetPath ≐
any

    latPos
    rdCurv
    offst
    spd
where

    LCC_Cntrl_6_grd1 : latPos ∈ LP
    LCC_Cntrl_6_grd2 : rdCurv ∈ RC
    LCC_Cntrl_6_grd3 : offst ∈  $\mathbb{Z}$ 
    LCC_Cntrl_6_grd4 : spd ∈  $\mathbb{N}$ 
then

    LCC_Cntrl_6_act1 : targetPath := TargPathFunc
    (latPos ↦ rdCurv ↦ offst)
    LCC_Cntrl_6_act2 : safeMrgTarg := SMTargFunc(spd)
end
Event UpdatePredictPath ≐
any

    yawAng
    yawR
    spd
where

    LCC_Cntrl_6_grd1 : yawAng ∈ YA
    LCC_Cntrl_6_grd2 : yawR ∈ YR
    LCC_Cntrl_6_grd3 : spd ∈  $\mathbb{N}$ 
then

    LCC_Cntrl_6_act1 : predictPath := PredPathFunc
    (yawAng ↦ yawR ↦ spd)
    LCC_Cntrl_6_act2 : safeMrgPred := SMPredFunc(spd)
end
Event UpdateSteeringAngle ≐
any

    mode

```

```

where

    LCC_Cntrl_6_grd1 : mode ∈ STATUS
    LCC_Cntrl_6_grd2 : mode = Active
    LCC_Cntrl_6_grd3 : safeMrgTarg > 0
    LCC_Cntrl_6_grd4 : safeMrgPred > 0
then

    LCC_Cntrl_6_act1 : steeringAngle := SteerAnglFunc
    (targetPath ↦ predictPath)
end

Event Activate_SetDisplayGreen ≐

any

    ACC_st
    indicate
    steer
where

    LCC_Cntrl_6_grd1 : warning = FALSE
    LCC_Mode_4_grd1 : status = Standby
    LCC_Mode_4_grd2 : ACC_st ∈ ACC_STATUS
    LCC_Mode_4_grd3 : ACC_st = ACC_Active
    LCC_Mode_4_grd4 : indicate ∈ BOOL
    LCC_Mode_4_grd5 : indicate = FALSE
    LCC_Mode_4_grd6 : steer ∈ BOOL
    LCC_Mode_4_grd7 : steer = FALSE
then

    LCC_Cntrl_6_act1 : displayClr := GREEN
    LCC_Mode_4_act1 : status := Active
end

Event Resume_SetDisplayGreen ≐

any

    ACC_st
    indicate
    steer
where

    LCC_Cntrl_6_grd1 : warning = FALSE
    LCC_Mode_4_grd1 : status = Overridden
    LCC_Mode_4_grd2 : ACC_st ∈ ACC_STATUS
    LCC_Mode_4_grd3 : ACC_st = ACC_Active
    LCC_Mode_4_grd4 : indicate ∈ BOOL
    LCC_Mode_4_grd5 : indicate = FALSE
    LCC_Mode_4_grd6 : steer ∈ BOOL
    LCC_Mode_4_grd7 : steer = FALSE

```

```

    then

        LCC_Cntrl_6_act1 : displayClr := GREEN
        LCC_Mode_4_act1 : status := Active
    end

Event SetWarning_NoTargetPath_SetDisplay  $\hat{=}$ 
    any

        mode
    where

        LCC_Cntrl_6_grd1 : mode  $\in$  STATUS
        LCC_Cntrl_6_grd2 : mode = Active
        LCC_Cntrl_6_grd3 : warning = FALSE
        LCC_Cntrl_6_grd4 : safeMrgTarg < SMTargThreshold
        LCC_Cntrl_6_grd5 : safeMrgTarg > 0
    then

        LCC_Cntrl_6_act1 : warning := TRUE
        LCC_Cntrl_6_act2 : displayClr := YELLOW
    end

Event SetWarning_NoPredictPath_SetDisplay  $\hat{=}$ 
    any

        mode
    where

        LCC_Cntrl_6_grd1 : mode  $\in$  STATUS
        LCC_Cntrl_6_grd2 : mode = Active
        LCC_Cntrl_6_grd3 : warning = FALSE
        LCC_Cntrl_6_grd4 : safeMrgPred < SMPredThreshold
        LCC_Cntrl_6_grd5 : safeMrgPred > 0
    then

        LCC_Cntrl_6_act1 : warning := TRUE
        LCC_Cntrl_6_act2 : displayClr := YELLOW
    end

Event StopWarning_ActiveMode  $\hat{=}$ 
    any

        mode
    where

        LCC_Cntrl_6_grd1 : warning = TRUE
        LCC_Cntrl_6_grd2 : mode  $\in$  STATUS
        LCC_Cntrl_6_grd3 : mode = Active
    then

        LCC_Cntrl_6_act1 : warning := FALSE
        LCC_Cntrl_6_act2 : displayClr := GREEN

```

**end**

**Event** *StopWarning\_OverrideIndicator\_DisplayOff*  $\hat{=}$

**any**

*ACC\_st*

*indicate*

**where**

*LCC\_Cntrl1\_6\_grd1* : *warning* = *TRUE*  
*LCC\_Mode\_4\_grd1* : *status* = *Active*  
*LCC\_Mode\_4\_grd2* : *ACC\_st*  $\in$  *ACC\_STATUS*  
*LCC\_Mode\_4\_grd3* : *ACC\_st* = *ACC\_Active*  
*LCC\_Mode\_4\_grd4* : *indicate*  $\in$  *BOOL*  
*LCC\_Mode\_4\_grd5* : *indicate* = *FALSE*

**then**

*LCC\_Cntrl1\_6\_act1* : *warning* := *FALSE*  
*LCC\_Cntrl1\_6\_act2* : *displayClr* := *OFF*  
*LCC\_Mode\_4\_act1* : *status* := *Overridden*

**end**

**Event** *StopWarning\_OverrideSteering\_DisplayOff*  $\hat{=}$

**any**

*ACC\_st*

*steer*

**where**

*LCC\_Cntrl1\_6\_grd1* : *warning* = *TRUE*  
*LCC\_Mode\_4\_grd1* : *status* = *Active*  
*LCC\_Mode\_4\_grd2* : *ACC\_st*  $\in$  *ACC\_STATUS*  
*LCC\_Mode\_4\_grd3* : *ACC\_st* = *ACC\_Active*  
*LCC\_Mode\_4\_grd4* : *steer*  $\in$  *BOOL*  
*LCC\_Mode\_4\_grd5* : *steer* = *TRUE*

**then**

*LCC\_Cntrl1\_6\_act1* : *warning* := *FALSE*  
*LCC\_Cntrl1\_6\_act2* : *displayClr* := *OFF*  
*LCC\_Mode\_4\_act1* : *status* := *Overridden*

**end**

**Event** *StopWarning\_SwitchOff\_DisplayOff*  $\hat{=}$

**any**

*ACC\_st*

**where**

*LCC\_Cntrl1\_6\_grd1* : *warning* = *TRUE*  
*LCC\_Mode\_4\_grd1* : *ACC\_st*  $\in$  *ACC\_STATUS*  
*LCC\_Mode\_4\_grd2* : *ACC\_st* = *ACC\_Active*

```

    LCC_Mode_4_grd3 : status = Active
  then
    LCC_Cntrl_6_act1 : warning := FALSE
    LCC_Cntrl_6_act2 : displayClr := OFF
    LCC_Mode_4_act1 : status := Off
  end
Event StopWarning_Unavailable_DisplayOff ≐
  any
    ACC_st
  where
    LCC_Cntrl_6_grd1 : warning = TRUE
    LCC_Mode_4_grd1 : status = Active
    LCC_Mode_4_grd2 : ACC_st ∈ ACC_STATUS
    LCC_Mode_4_grd3 : ACC_st = ACC_Deactive
  then
    LCC_Cntrl_6_act1 : warning := FALSE
    LCC_Cntrl_6_act2 : displayClr := OFF
    LCC_Mode_4_act1 : status := Unavailable
  end
Event OverrideIndicator_DisplayOff ≐
  any
    ACC_st
    indicate
  where
    LCC_Cntrl_6_grd1 : warning = FALSE
    LCC_Mode_4_grd1 : status = Active
    LCC_Mode_4_grd2 : ACC_st ∈ ACC_STATUS
    LCC_Mode_4_grd3 : ACC_st = ACC_Active
    LCC_Mode_4_grd4 : indicate ∈ BOOL
    LCC_Mode_4_grd5 : indicate = FALSE
  then
    LCC_Cntrl_6_act1 : displayClr := OFF
    LCC_Mode_4_act1 : status := Overridden
  end
Event OverrideSteering_DisplayOff ≐
  any
    ACC_st
    steer
  where
    LCC_Cntrl_6_grd1 : warning = FALSE

```



```

    LCC_Mode_4_grd1 : status = Active
    LCC_Mode_4_grd2 : ACC_st ∈ ACC_STATUS
    LCC_Mode_4_grd3 : ACC_st = ACC_Active
    LCC_Mode_4_grd4 : steer ∈ BOOL
    LCC_Mode_4_grd5 : steer = TRUE
  then
    LCC_Cntrl_6_act1 : displayClr := OFF
    LCC_Mode_4_act1 : status := Overridden
  end
Event SwitchOff_FromActive_DisplayOff ≐
  any
    where
      ACC_st
    then
      LCC_Cntrl_6_grd1 : warning = FALSE
      LCC_Mode_4_grd1 : ACC_st ∈ ACC_STATUS
      LCC_Mode_4_grd2 : ACC_st = ACC_Active
      LCC_Mode_4_grd3 : status = Active
    then
      LCC_Cntrl_6_act1 : displayClr := OFF
      LCC_Mode_4_act1 : status := Off
    end
  end
Event Unavailable_FromActive_DisplayOff ≐
  any
    where
      ACC_st
    then
      LCC_Cntrl_6_grd1 : warning = FALSE
      LCC_Mode_4_grd1 : status = Active
      LCC_Mode_4_grd2 : ACC_st ∈ ACC_STATUS
      LCC_Mode_4_grd3 : ACC_st = ACC_Deactive
    then
      LCC_Cntrl_6_act1 : displayClr := OFF
      LCC_Mode_4_act1 : status := Unavailable
    end
  end
Event LCC_Available ≐
  any
    where
      ACC_st
    then
      LCC_Mode_4_grd1 : ACC_st ∈ ACC_STATUS
      LCC_Mode_4_grd2 : ACC_st = ACC_Active
      LCC_Mode_4_grd3 : status = Unavailable

```

```

    then
        LCC_Mode_4_act1 : status := Off
    end
Event SwitchOn  $\hat{=}$ 
    any
        ACC_st
    where
        LCC_Mode_4_grd1 : status = Off
        LCC_Mode_4_grd2 : ACC_st  $\in$  ACC_STATUS
        LCC_Mode_4_grd3 : ACC_st = ACC_Active
    then
        LCC_Mode_4_act1 : status := Standby
    end
Event SwitchOff_FromInactive  $\hat{=}$ 
    any
        ACC_st
    where
        LCC_Mode_4_grd1 : ACC_st  $\in$  ACC_STATUS
        LCC_Mode_4_grd2 : ACC_st = ACC_Active
        LCC_Mode_4_grd3 : status  $\in$  {Standby, Overridden}
    then
        LCC_Mode_4_act1 : status := Off
    end
Event LCC_Unavailable_FromInactive  $\hat{=}$ 
    any
        ACC_st
    where
        LCC_Mode_4_grd1 : status  $\in$  {Standby, Overridden, Off}
        LCC_Mode_4_grd2 : ACC_st  $\in$  ACC_STATUS
        LCC_Mode_4_grd3 : ACC_st = ACC_Deactive
    then
        LCC_Mode_4_act1 : status := Unavailable
    end
Event IncreaseOffset  $\hat{=}$ 
    any
        mode
    where
        LCC_Cmnd_2_grd1 : offset < offsetUB
        LCC_Cmnd_2_grd2 : mode  $\in$  STATUS
        LCC_Cmnd_2_grd3 : mode = Active

```

---

```

    then
        LCC_Cmnd_2_act1 : offset := offset + 1
    end
Event DecreaseOffset  $\hat{=}$ 
    any
        mode
    where
        LCC_Cmnd_2_grd1 : offset > offsetLB
        LCC_Cmnd_2_grd2 : mode  $\in$  STATUS
        LCC_Cmnd_2_grd3 : mode = Active
    then
        LCC_Cmnd_2_act1 : offset := offset - 1
    end
END

```

## Appendix B

# Vertical Refinement and Decomposition of SCCS

This chapter presents the vertical refinement steps and the decomposition of the simplified cruise control system (SCCS). The patterns which are used in the SCCS case study were described in Chapter 8.

### B.1 Contexts of SCCS

#### B.1.1 Abstract Context: C0

**CONTEXT** C0

**CONSTANTS**

*Accel\_Func*

*maxSpd*

*lb*

*ub*

**AXIOMS**

**axm1** :  $maxSpd \in \mathbb{N}$

**axm2** :  $maxSpd = 7$

**axm3** :  $Accel\_Func \in 0 .. maxSpd \times 0 .. maxSpd \rightarrow \mathbb{N}$

**axm4** :  $lb \in 0 .. maxSpd$

**axm5** :  $ub \in 0 .. maxSpd$

**axm6** :  $ub > lb$

**END**

**B.1.2 First Extended Context: C1****CONTEXT** C1**EXTENDS** C0**SETS***CntOrder*  
**CONSTANTS***TargSpd*  
**AXIOMS***axm1* : *partition*(*CntOrder*, {*TargSpd*})  
**END****B.1.3 Second Extended Context: C2****CONTEXT** C2**EXTENDS** C1**SETS***T3State*  
*T1State*  
*T2State*  
**CONSTANTS***T3\_EN*  
*T3\_SEND*  
*T3\_GENR*  
*T1\_EN*  
*T1\_TS*  
*T1\_ACCEL*  
*T1\_BFR*  
*T1\_SEND*  
*Sns\_EN*  
*Sns\_SEND*  
*Sns\_GENR*  
**AXIOMS***axm1* : *partition*(*T3State*, {*T3\_EN*}, {*T3\_SEND*}, {*T3\_GENR*})  
*axm3* : *partition*(*T1State*, {*T1\_EN*}, {*T1\_ACCEL*}, {*T1\_TS*},  
{*T1\_BFR*}, {*T1\_SEND*})  
*axm4* : *partition*(*T2State*, {*Sns\_EN*}, {*Sns\_GENR*}, {*Sns\_SEND*})  
**END**

**B.1.4 Third Extended Context: C3****CONTEXT** C3**EXTENDS** C2**SETS***T4State***CONSTANTS***T4\_EN**T4\_ACT***AXIOMS***axm1* : *partition*(*T4State*, {*T4\_EN*}, {*T4\_ACT*})**END****B.1.5 Fourth Extended Context: C4****CONTEXT** C4**EXTENDS** C3**CONSTANTS***MaxNumMSG***AXIOMS***axm1* : *MaxNumMSG* ∈ ℕ*axm2* : *MaxNumMSG* = 3**END****B.2 Vertical Refinement of SCCS****B.2.1 Abstract Machine: CC0****MACHINE** CC0**SEES** C0**VARIABLES***acceleration**actualSpd**cnt\_targetSpd***INVARIANTS***inv1* : *acceleration* ∈ ℕ*inv2* : *actualSpd* ⊆ 0 .. *maxSpd*

```

    inv3 : cnt_targetSpd ∈ 0 .. maxSpd
EVENTS
Initialisation
    begin
        act1 : acceleration := 0
        act2 : actualSpd := {0}
        act3 : cnt_targetSpd := 0
    end
Event SetTargetSpd ≐
    any
        spd
    where
        grd1 : spd ∈ lb .. ub
    then
        act1 : cnt_targetSpd := spd
    end
Event GenerateAccel ≐
    any
        actSpd
    where
        grd1 : actSpd ∈ actualSpd
    then
        act1 : acceleration := Accel_Func(actSpd ↦ cnt_targetSpd)
    end
Event Env_UpdateSpd ≐
    any
        s
    where
        grd1 : s ⊆ 0 .. maxSpd
    then
        act1 : actualSpd := s
    end
END

```

### B.2.2 Refinement 1: Machine CC1

**MACHINE** CC1

Cycle and skip events are defined.

**REFINES** CC0

**SEES** C1**VARIABLES**

*acceleration* CC0 - Functional Req  
*actualSpd* CC0 - Functional Req  
*cnt\_targetSpd* CC0 - Functional Req  
*cycle* CC1 - cycle of tasks  
*T1\_cntEvaluated* CC1 - controller task is evaluated  
*cnt\_proc\_enbl* CC1 - enabling flags  
*cnt\_order* CC1 - to define order between events  
*counter* CC1 - to avoid using mod in invariants

**INVARIANTS**

*inv1* :  $cycle \in \mathbb{N}$   
*inv2* :  $T1\_cntEvaluated \in \text{BOOL}$   
*inv3* :  $cnt\_proc\_enbl \in \text{BOOL}$   
*inv4* :  $cnt\_order \subseteq \text{CntOrder}$   
*inv5* :  $counter \in \{0, 1\}$   
*inv6* :  $cnt\_proc\_enbl = \text{TRUE} \Rightarrow T1\_cntEvaluated = \text{FALSE}$   
*inv7* :  $TargSpd \in cnt\_order \Rightarrow cnt\_proc\_enbl = \text{TRUE}$   
*inv8* :  $TargSpd \in cnt\_order \Rightarrow T1\_cntEvaluated = \text{FALSE}$

**EVENTS****Initialisation**

*extended*  
**begin**  
     *act1* :  $acceleration := 0$   
     *act2* :  $actualSpd := \{0\}$   
     *act3* :  $cnt\_targetSpd := 0$   
     *act4* :  $cycle := 0$   
     *act5* :  $T1\_cntEvaluated := \text{TRUE}$   
     *act6* :  $cnt\_proc\_enbl := \text{FALSE}$   
     *act7* :  $cnt\_order := \emptyset$   
     *act8* :  $counter := 0$   
**end**

**Event**  $UpdateCycle \hat{=}$

**extends**  $Env\_UpdateSpd$

**any**

*s*

*b*

*c*

**where**



```

    grd1 :  $s \subseteq 0 .. maxSpd$ 
    grd2 :  $b \in BOOL$ 
    grd3 :  $c \in \{0, 1\}$ 
    grd4 :  $counter \in \{1\} \vee (T1\_cntEvaluated = TRUE)$ 
    grd5 :  $counter = 1 \Rightarrow b = T1\_cntEvaluated \wedge c = 0$ 
    grd6 :  $counter = 0 \Rightarrow b = FALSE \wedge c = 1$ 
  then
    act1 :  $actualSpd := s$ 
    act2 :  $cycle := cycle + 5$ 
    act3 :  $counter := c$ 
    act4 :  $T1\_cntEvaluated := b$ 
  end
Event T1_EnableACCEL  $\hat{=}$ 
  when
    grd1 :  $cnt\_proc\_enbl = FALSE$ 
    grd2 :  $T1\_cntEvaluated = FALSE$ 
  then
    act1 :  $cnt\_proc\_enbl := TRUE$ 
  end
Event T1_SetTargetSpd  $\hat{=}$ 
extends SetTargetSpd
  any
     $spd$ 
  where
    grd1 :  $spd \in lb .. ub$ 
    grd2 :  $cnt\_proc\_enbl = TRUE$ 
    grd3 :  $cnt\_order = \emptyset$ 
  then
    act1 :  $cnt\_targetSpd := spd$ 
    act2 :  $cnt\_order := \{TargSpd\}$ 
  end
Event T1_SkipSetTargetSpd  $\hat{=}$ 
  when
    grd1 :  $cnt\_proc\_enbl = TRUE$ 
    grd2 :  $cnt\_order = \emptyset$ 
  then
    act1 :  $cnt\_order := \{TargSpd\}$ 
  end
Event GenerateAccel  $\hat{=}$ 
refines GenerateAccel

```

```

any

    actSpd
where

    grd1 : actSpd  $\in$  actualSpd
    grd2 : TargSpd  $\in$  cnt_order
then

    act1 : acceleration := Accel_Func(actSpd  $\mapsto$  cnt_targetSpd)
    act2 : T1_cntEvaluated := TRUE
    act3 : cnt_proc_enbl := FALSE
    act4 : cnt_order :=  $\emptyset$ 
end
END

```

### B.2.3 Refinement 2: Machine CC2

**MACHINE** CC2

Environment event which sets speed for every value is introduced

**REFINES** CC1

**SEES** C1

**VARIABLES**

*acceleration* CC0 - Functional Req  
*cnt\_targetSpd* CC0 - Functional Req  
*cycle* CC1 - cycle of tasks  
*T1\_cntEvaluated* CC1 - controller task is evaluated  
*cnt\_proc\_enbl* CC1 - enabling flags  
*cnt\_order* CC1 - to define order between events  
*counter* CC1 - to avoid using mod in invariants  
*env\_evaluated* CC2 - environment should be evaluated at every cycle  
*speed* CC2 - value of speed at each moment of time

**INVARIANTS**

*inv1* : *env\_evaluated*  $\in$  *BOOL*  
*inv2* : *env\_evaluated* = *FALSE*  $\Rightarrow$  *speed*  $\in$   $-25 \dots (cycle) \rightarrow 0 \dots maxSpd$   
*inv3* : *env\_evaluated* = *TRUE*  $\Rightarrow$  *speed*  $\in$   $-25 \dots (cycle + 5) \rightarrow 0 \dots maxSpd$   
*inv4* : *speed* $[-25 \dots 0]$  =  $\{0\}$   
*inv5* : *ran*(*speed*)  $\subseteq 0 \dots maxSpd$   
*inv6* : *actualSpd* = *speed* $[cycle - 25 \dots cycle]$

**EVENTS**

**Initialisation**

**begin**

```

    act1 : acceleration := 0
    act2 : cnt_targetSpd := 0
    act3 : cycle := 0
    act4 : T1_cntEvaluated := TRUE
    act5 : cnt_proc_enbl := FALSE
    act6 : cnt_order := ∅
    act7 : counter := 0
    act8 : env_evaluated := FALSE
    act9 : speed := -25 .. 0 × {0}
end

Event UpdateCycle ≐
refines UpdateCycle
  any
    b
    c
  where
    grd1 : b ∈ BOOL
    grd2 : c ∈ {0, 1}
    grd3 : counter = 1 ∨ (T1_cntEvaluated = TRUE)
    grd4 : (counter = 1) ⇒ b = T1_cntEvaluated ∧ c = 0
    grd5 : (counter = 0) ⇒ b = FALSE ∧ c = 1
    grd6 : env_evaluated = TRUE
  with
  then
    s : s = speed[cycle - 20 .. cycle + 5]
  then
    act1 : cycle := cycle + 5
    act2 : counter := c
    act3 : T1_cntEvaluated := b
    act4 : env_evaluated := FALSE
  end

Event UpdateSpd_Env ≐
  any
    spd
  where
    grd1 : env_evaluated = FALSE
    grd2 : spd ∈ (cycle + 1 .. cycle + 5) → (0 .. maxSpd)
  then
    act1 : speed := speed ∪ spd
    act2 : env_evaluated := TRUE
  end
end

```

**Event**  $T1\_EnableACCEL \hat{=}$

**extends**  $T1\_EnableACCEL$

**when**

$grd1 : cnt\_proc\_enbl = FALSE$

$grd2 : T1\_cntEvaluated = FALSE$

**then**

$act1 : cnt\_proc\_enbl := TRUE$

**end**

**Event**  $T1\_SetTargetSpd \hat{=}$

**extends**  $T1\_SetTargetSpd$

**any**

$spd$

**where**

$grd1 : spd \in lb \dots ub$

$grd2 : cnt\_proc\_enbl = TRUE$

$grd3 : cnt\_order = \emptyset$

**then**

$act1 : cnt\_targetSpd := spd$

$act2 : cnt\_order := \{TargSpd\}$

**end**

**Event**  $T1\_SkipSetTargetSpd \hat{=}$

**extends**  $T1\_SkipSetTargetSpd$

**when**

$grd1 : cnt\_proc\_enbl = TRUE$

$grd2 : cnt\_order = \emptyset$

**then**

$act1 : cnt\_order := \{TargSpd\}$

**end**

**Event**  $GenerateAccel \hat{=}$

**refines**  $GenerateAccel$

**any**

$actSpd$

**where**

$grd1 : actSpd \in speed[cycle - 25 \dots cycle]$

$grd2 : TargSpd \in cnt\_order$

**then**

$act1 : acceleration := Accel\_Func(actSpd \mapsto cnt\_targetSpd)$

$act2 : T1\_cntEvaluated := TRUE$

$act3 : cnt\_proc\_enbl := FALSE$

$act4 : cnt\_order := \emptyset$

**end**

**END**

### B.2.4 Refinement 3: Machine CC3

**MACHINE** CC3

Local buffer for CNT task (T1) is introduced

**REFINES** CC2

**SEES** C1

**VARIABLES**

*acceleration* CC0 - Functional Req

*cnt\_targetSpd* CC0 - Functional Req

*cycle* CC1 - cycle of tasks

*T1\_cntEvaluated* CC1 - controller task is evaluated

*cnt\_proc\_enbl* CC1 - enabling flags

*cnt\_order* CC1 - to define order between events

*counter* CC1 - to avoid using mod in invariants

*env\_evaluated* CC2 - environment should be evaluated at every cycle

*speed* CC2 - value of speed at each moment of time

*T1\_cntLocActSpd* CC3 - Local buffer of T1 (CNT Task)

*rcvFlg* CC3 - TRUE when the CNT receive events happens. Reset at the end of every 10ms (by updateCycle2)

**INVARIANTS**

*inv1* :  $T1\_cntLocActSpd \in 0 \dots maxSpd$

*inv2* :  $T1\_cntLocActSpd \in speed[cycle - 25 \dots cycle]$

*inv3* :  $rcvFlg \in BOOL$

*inv4* :  $T1\_cntEvaluated = TRUE \Rightarrow rcvFlg = TRUE$

*inv5* :  $cnt\_proc\_enbl = TRUE \vee TargSpd \in cnt\_order \Rightarrow rcvFlg = TRUE$

*inv6* :  $counter = 0 \wedge rcvFlg = TRUE \Rightarrow$

$T1\_cntLocActSpd \in speed[cycle - 15 \dots cycle]$

*inv7* :  $counter = 1 \wedge rcvFlg = TRUE \Rightarrow$

$T1\_cntLocActSpd \in speed[cycle - 10 \dots cycle]$

*inv8* :  $counter = 1 \wedge rcvFlg = FALSE \Rightarrow$

$T1\_cntLocActSpd \in speed[cycle - 20 \dots cycle]$

**EVENTS**

**Initialisation**

*extended*

**begin**

*act1* :  $acceleration := 0$

```

    act2 : cnt_targetSpd := 0
    act3 : cycle := 0
    act4 : T1_cntEvaluated := TRUE
    act5 : cnt_proc_enbl := FALSE
    act6 : cnt_order :=  $\emptyset$ 
    act7 : counter := 0
    act8 : env_evaluated := FALSE
    act9 : speed :=  $-25 \dots 0 \times \{0\}$ 
    act10 : T1_cntLocActSpd := 0
    act11 : rcvFlg := TRUE
end
Event UpdateCycle1_RcvT  $\hat{=}$ 
refines UpdateCycle
    when
        grd1 : counter = 1
        grd2 : env_evaluated = TRUE
        grd3 : rcvFlg = TRUE
    with
        c : c = 0
    then
        b : b = T1_cntEvaluated

        act1 : cycle := cycle + 5
        act2 : counter := 0
        act3 : env_evaluated := FALSE
    end
Event UpdateCycle1_RcvF  $\hat{=}$ 
refines UpdateCycle
    when
        grd1 : counter = 1
        grd2 : env_evaluated = TRUE
        grd3 : rcvFlg = FALSE
    with
        c : c = 0
    then
        b : b = T1_cntEvaluated

        act1 : cycle := cycle + 5
        act2 : counter := 0
        act3 : env_evaluated := FALSE
    end
Event UpdateCycle2  $\hat{=}$ 
refines UpdateCycle

```

```

when

    grd1 : T1_cntEvaluated = TRUE
    grd2 : counter = 0
    grd3 : env_evaluated = TRUE
with

    c : c = 1
    b : b = FALSE
then

    act1 : cycle := cycle + 5
    act2 : counter := 1
    act3 : T1_cntEvaluated := FALSE
    act4 : env_evaluated := FALSE
    act5 : rcvFlg := FALSE
end

Event   UpdateSpd_Env  $\hat{=}$ 
extends UpdateSpd_Env

    any

        spd
    where

        grd1 : env_evaluated = FALSE
        grd2 : spd  $\in$  (cycle + 1 .. cycle + 5)  $\rightarrow$  (0 .. maxSpd)
    then

        act1 : speed := speed  $\cup$  spd
        act2 : env_evaluated := TRUE
    end

Event   T1_ReceiveEnvVal  $\hat{=}$ 
refines T1_EnableACCEL

    any

        Rx2
    where

        grd1 : Rx2  $\in$  speed[cycle - 10 .. cycle]
        grd2 : cnt_proc_enbl = FALSE
        grd3 : T1_cntEvaluated = FALSE
    then

        act1 : cnt_proc_enbl := TRUE
        act2 : T1_cntLocActSpd := Rx2
        act3 : rcvFlg := TRUE
    end

Event   T1_SetTargetSpd  $\hat{=}$ 
extends T1_SetTargetSpd

    any

```

```

      spd
where
      grd1 : spd ∈ lb .. ub
      grd2 : cnt_proc_enbl = TRUE
      grd3 : cnt_order = ∅
then
      act1 : cnt_targetSpd := spd
      act2 : cnt_order := {TargSpd}
end
Event T1_SkipSetTargetSpd ≐
extends T1_SkipSetTargetSpd
when
      grd1 : cnt_proc_enbl = TRUE
      grd2 : cnt_order = ∅
then
      act1 : cnt_order := {TargSpd}
end
Event GenerateAccel ≐
refines GenerateAccel
when
      grd1 : TargSpd ∈ cnt_order
with
      actSpd : actSpd = T1_cntLocActSpd
then
      act1 : acceleration := Accel_Func(T1_cntLocActSpd ↦ cnt_targetSpd)
      act2 : T1_cntEvaluated := TRUE
      act3 : cnt_proc_enbl := FALSE
      act4 : cnt_order := ∅
end
END

```

### B.2.5 Refinement 4: Machine CC4

**MACHINE** CC4

Sensor transmitting to T1\_RX is defined

**REFINES** CC3

**SEES** C1

**VARIABLES**

*acceleration* CC0 - Functional Req

*cnt\_targetSpd* CC0 - Functional Req



*cycle* CC1 - cycle of tasks  
*T1\_cntEvaluated* CC1 - controller task is evaluated  
*cnt\_proc\_enbl* CC1 - enabling flags  
*cnt\_order* CC1 - to define order between events  
*counter* CC1 - to avoid using mod in invariants  
*env\_evaluated* CC2 - environment should be evaluated at every cycle  
*speed* CC2 - value of speed at each moment of time  
*T1\_cntLocActSpd* CC3 - Local buffer of T1 (CNT Task)  
*rcvFlg* CC3 - TRUE when the CNT receive events happens. Reset at the end of every 10ms (by updateCycle2)  
*T1\_cntRxSpd* CC4 - RX buffer in task 1 (CNT)  
*T2\_sensEvaluated* CC4 - sensor must be evaluated before the cycle can progress

## INVARIANTS

*inv1* :  $T1\_cntRxSpd \in 0 .. maxSpd$   
*inv2* :  $T2\_sensEvaluated \in BOOL$   
*inv3* :  $T1\_cntRxSpd \in speed[cycle - 10 .. cycle]$   
*inv3-1* :  $T2\_sensEvaluated = TRUE \Rightarrow$   
 $T1\_cntRxSpd \in speed[cycle - 5 .. cycle]$   
*inv3-2* :  $T2\_sensEvaluated = FALSE \Rightarrow$   
 $T1\_cntRxSpd \in speed[cycle - 10 .. cycle]$

## EVENTS

### Initialisation

*extended*

**begin**

*act1* :  $acceleration := 0$   
*act2* :  $cnt\_targetSpd := 0$   
*act3* :  $cycle := 0$   
*act4* :  $T1\_cntEvaluated := TRUE$   
*act5* :  $cnt\_proc\_enbl := FALSE$   
*act6* :  $cnt\_order := \emptyset$   
*act7* :  $counter := 0$   
*act8* :  $env\_evaluated := FALSE$   
*act9* :  $speed := -25 .. 0 \times \{0\}$   
*act10* :  $T1\_cntLocActSpd := 0$   
*act11* :  $rcvFlg := TRUE$   
*act12* :  $T1\_cntRxSpd := 0$   
*act13* :  $T2\_sensEvaluated := TRUE$

**end**

**Event**  $UpdateCycle1\_RcvT \hat{=}$

**extends** *UpdateCycle1\_RcvT*

**when**

*grd1* : *counter* = 1  
*grd2* : *env\_evaluated* = *TRUE*  
*grd3* : *rcvFlg* = *TRUE*  
*grd4* : *T2\_sensEvaluated* = *TRUE*

**then**

*act1* : *cycle* := *cycle* + 5  
*act2* : *counter* := 0  
*act3* : *env\_evaluated* := *FALSE*  
*act4* : *T2\_sensEvaluated* := *FALSE*

**end**

**Event** *UpdateCycle1\_RcvF*  $\hat{=}$

**extends** *UpdateCycle1\_RcvF*

**when**

*grd1* : *counter* = 1  
*grd2* : *env\_evaluated* = *TRUE*  
*grd3* : *rcvFlg* = *FALSE*  
*grd4* : *T2\_sensEvaluated* = *TRUE*

**then**

*act1* : *cycle* := *cycle* + 5  
*act2* : *counter* := 0  
*act3* : *env\_evaluated* := *FALSE*  
*act4* : *T2\_sensEvaluated* := *FALSE*

**end**

**Event** *UpdateCycle2*  $\hat{=}$

**extends** *UpdateCycle2*

**when**

*grd1* : *T1\_cntEvaluated* = *TRUE*  
*grd2* : *counter* = 0  
*grd3* : *env\_evaluated* = *TRUE*  
*grd4* : *T2\_sensEvaluated* = *TRUE*

**then**

*act1* : *cycle* := *cycle* + 5  
*act2* : *counter* := 1  
*act3* : *T1\_cntEvaluated* := *FALSE*  
*act4* : *env\_evaluated* := *FALSE*  
*act5* : *rcvFlg* := *FALSE*  
*act6* : *T2\_sensEvaluated* := *FALSE*

**end**

**Event** *UpdateSpd\_Env*  $\hat{=}$

**extends** *UpdateSpd\_Env*

**any**

*spd*  
**where**

*grd1* : *env\_evaluated* = *FALSE*

*grd2* : *spd* ∈ (*cycle* + 1 .. *cycle* + 5) → (0 .. *maxSpd*)

**then**

*act1* : *speed* := *speed* ∪ *spd*

*act2* : *env\_evaluated* := *TRUE*

**end**

**Event** *T2\_SensorTransmitSpd* ≐

**any**

*canTrans*  
**where**

*grd1* : *canTrans* ∈ *speed*[*cycle* − 5 .. *cycle*]

*grd2* : *T2\_sensEvaluated* = *FALSE*

**then**

*act1* : *T1\_cntRxSpd* := *canTrans*

*act2* : *T2\_sensEvaluated* := *TRUE*

**end**

**Event** *T1\_ReceiveEnvVal* ≐

**refines** *T1\_ReceiveEnvVal*

**any**

*Rx2*  
**where**

*grd1* : *Rx2* = *T1\_cntRxSpd*

*grd2* : *cnt\_proc\_enbl* = *FALSE*

*grd3* : *T1\_cntEvaluated* = *FALSE*

**then**

*act1* : *cnt\_proc\_enbl* := *TRUE*

*act2* : *T1\_cntLocActSpd* := *Rx2*

*act3* : *rcvFlg* := *TRUE*

**end**

**Event** *T1\_SetTargetSpd* ≐

**extends** *T1\_SetTargetSpd*

**any**

*spd*  
**where**

*grd1* : *spd* ∈ *lb* .. *ub*

*grd2* : *cnt\_proc\_enbl* = *TRUE*

```

        grd3 : cnt_order =  $\emptyset$ 
    then
        act1 : cnt_targetSpd := spd
        act2 : cnt_order := {TargSpd}
    end
Event T1_SkipSetTargetSpd  $\hat{=}$ 
extends T1_SkipSetTargetSpd
    when
        grd1 : cnt_proc_enbl = TRUE
    then
        grd2 : cnt_order =  $\emptyset$ 
        act1 : cnt_order := {TargSpd}
    end
Event GenerateAccel  $\hat{=}$ 
extends GenerateAccel
    when
        grd1 : TargSpd  $\in$  cnt_order
    then
        act1 : acceleration := Accel_Func(T1_cntLocActSpd  $\mapsto$  cnt_targetSpd)
        act2 : T1_cntEvaluated := TRUE
        act3 : cnt_proc_enbl := FALSE
        act4 : cnt_order :=  $\emptyset$ 
    end
END

```

### B.2.6 Refinement 5: Machine CC5

**MACHINE** CC5

Sensor requests to transmit the sensed value is added

**REFINES** CC4

**SEES** C1

**VARIABLES**

*acceleration* CC0 - Functional Req  
*cnt\_targetSpd* CC0 - Functional Req  
*cycle* CC1 - cycle of tasks  
*T1\_cntEvaluated* CC1 - controller task is evaluated  
*cnt\_proc\_enbl* CC1 - enabling flags  
*cnt\_order* CC1 - to define order between events  
*counter* CC1 - to avoid using mod in invariants

*env\_evaluated* CC2 - environment should be evaluated at every cycle  
*speed* CC2 - value of speed at each moment of time  
*T1\_cntLocActSpd* CC3 - Local buffer of T1 (CNT Task)  
*rcvFlg* CC3 - TRUE when the CNT receive events happens. Reset at the end of every 10ms (by updateCycle2)  
*T1\_cntRxSpd* CC4 - RX buffer in task 1 (CNT)  
*T2\_sensEvaluated* CC4 - sensor must be evaluated before the cycle can progress  
*T2\_snsTX* CC5 - Transmitter buffer of sensor  
*bfrdFlg* CC5 - This is used to show the ordering

## INVARIANTS

*inv1* : *bfrdFlg*  $\in$  *BOOL*  
*inv2* : *T2\_snsTX*  $\in$   $0 \dots \text{maxSpd}$   
*inv3* : *bfrdFlg* = *TRUE*  $\Rightarrow$  *T2\_sensEvaluated* = *FALSE*  
*inv4* : *bfrdFlg* = *TRUE*  $\Rightarrow$  *T2\_snsTX*  $\in$  *speed*[*cycle* - 5 .. *cycle*]

## EVENTS

### Initialisation

*extended*

**begin**

*act1* : *acceleration* := 0  
*act2* : *cnt\_targetSpd* := 0  
*act3* : *cycle* := 0  
*act4* : *T1\_cntEvaluated* := *TRUE*  
*act5* : *cnt\_proc\_enbl* := *FALSE*  
*act6* : *cnt\_order* :=  $\emptyset$   
*act7* : *counter* := 0  
*act8* : *env\_evaluated* := *FALSE*  
*act9* : *speed* :=  $-25 \dots 0 \times \{0\}$   
*act10* : *T1\_cntLocActSpd* := 0  
*act11* : *rcvFlg* := *TRUE*  
*act12* : *T1\_cntRxSpd* := 0  
*act13* : *T2\_sensEvaluated* := *TRUE*  
*act14* : *bfrdFlg* := *FALSE*  
*act15* : *T2\_snsTX* := 0

**end**

**Event** *UpdateCycle1\_RcvT*  $\hat{=}$

**extends** *UpdateCycle1\_RcvT*

**when**

*grd1* : *counter* = 1  
*grd2* : *env\_evaluated* = *TRUE*

```

    grd3 : rcvFlg = TRUE
    grd4 : T2_sensEvaluated = TRUE
  then

    act1 : cycle := cycle + 5
    act2 : counter := 0
    act3 : env_evaluated := FALSE
    act4 : T2_sensEvaluated := FALSE
  end

```

**Event** *UpdateCycle1\_RcvF*  $\hat{=}$

**extends** *UpdateCycle1\_RcvF*

**when**

```

    grd1 : counter = 1
    grd2 : env_evaluated = TRUE
    grd3 : rcvFlg = FALSE
    grd4 : T2_sensEvaluated = TRUE
  then

```

**then**

```

    act1 : cycle := cycle + 5
    act2 : counter := 0
    act3 : env_evaluated := FALSE
    act4 : T2_sensEvaluated := FALSE
  end

```

**Event** *UpdateCycle2*  $\hat{=}$

**extends** *UpdateCycle2*

**when**

```

    grd1 : T1_cntEvaluated = TRUE
    grd2 : counter = 0
    grd3 : env_evaluated = TRUE
    grd4 : T2_sensEvaluated = TRUE
  then

```

**then**

```

    act1 : cycle := cycle + 5
    act2 : counter := 1
    act3 : T1_cntEvaluated := FALSE
    act4 : env_evaluated := FALSE
    act5 : rcvFlg := FALSE
    act6 : T2_sensEvaluated := FALSE
  end

```

**end**

**Event** *UpdateSpd\_Env*  $\hat{=}$

**extends** *UpdateSpd\_Env*

**any**

**where** *spd*

```

    grd1 : env_evaluated = FALSE
    grd2 : spd ∈ (cycle + 1 .. cycle + 5) → (0 .. maxSpd)
  then
    act1 : speed := speed ∪ spd
    act2 : env_evaluated := TRUE
  end
Event T2_RequestToTransmitSpd ≐
  any
    spd
  where
    grd1 : spd ∈ speed[cycle - 4 .. cycle]
    grd2 : T2_sensEvaluated = FALSE
  then
    act1 : T2_snsTX := spd
    act2 : bfrdFlg := TRUE
  end
Event T2_SensorTransmitSpd ≐
refines T2_SensorTransmitSpd
  any
    canTrans
  where
    grd1 : canTrans = T2_snsTX
    grd2 : bfrdFlg = TRUE
  then
    act1 : T1_cntRxSpd := canTrans
    act2 : T2_sensEvaluated := TRUE
    act3 : bfrdFlg := FALSE
  end
Event T1_ReceiveEnvVal ≐
extends T1_ReceiveEnvVal
  any
    Rx2
  where
    grd1 : Rx2 = T1_cntRxSpd
    grd2 : cnt_proc_enbl = FALSE
    grd3 : T1_cntEvaluated = FALSE
  then
    act1 : cnt_proc_enbl := TRUE
    act2 : T1_cntLocActSpd := Rx2
    act3 : rcvFlg := TRUE

```

```

    end
Event T1_SetTargetSpd  $\hat{=}$ 
extends T1_SetTargetSpd
    any
        spd
    where
        grd1 : spd  $\in lb .. ub$ 
        grd2 : cnt_proc_enbl = TRUE
        grd3 : cnt_order =  $\emptyset$ 
    then
        act1 : cnt_targetSpd := spd
        act2 : cnt_order := {TargSpd}
    end
Event T1_SkipSetTargetSpd  $\hat{=}$ 
extends T1_SkipSetTargetSpd
    when
        grd1 : cnt_proc_enbl = TRUE
        grd2 : cnt_order =  $\emptyset$ 
    then
        act1 : cnt_order := {TargSpd}
    end
Event GenerateAccel  $\hat{=}$ 
extends GenerateAccel
    when
        grd1 : TargSpd  $\in cnt\_order$ 
    then
        act1 : acceleration := Accel_Func(T1_cntLocActSpd  $\mapsto cnt\_targetSpd$ )
        act2 : T1_cntEvaluated := TRUE
        act3 : cnt_proc_enbl := FALSE
        act4 : cnt_order :=  $\emptyset$ 
    end
END

```

### B.2.7 Refinement 6: Machine CC6

**MACHINE** CC6

**REFINES** CC5

**SEES** C2

**VARIABLES**



*acceleration* CC0 - Functional Req  
*cnt\_targetSpd* CC0 - Functional Req  
*cycle* CC1 - cycle of tasks  
*T1\_cntEvaluated* CC1 - controller task is evaluated  
*cnt\_proc\_enbl* CC1 - enabling flags  
*cnt\_order* CC1 - to define order between events  
*counter* CC1 - to avoid using mod in invariants  
*env\_evaluated* CC2 - environment should be evaluated at every cycle  
*speed* CC2 - value of speed at each moment of time  
*T1\_cntLocActSpd* CC3 - Local buffer of T1 (CNT Task)  
*rcvFlg* CC3 - TRUE when the CNT receive events happens. Reset at the end of every 10ms (by updateCycle2)  
*T1\_cntRxSpd* CC4 - RX buffer in task 1 (CNT)  
*T2\_sensEvaluated* CC4 - sensor must be evaluated before the cycle can progress  
*T2\_snsTX* CC5 - Transmitter buffer of sensor  
*T2\_snsLoc\_speed* CC6 - local buffer of the sensor which generates the sensed value  
*T2\_state* CC6 - To model the order between events within the task T2.

## INVARIANTS

*inv1* :  $T2\_snsLoc\_speed \in 0 .. maxSpd$   
*inv2* :  $T2\_state \in T2State$   
*inv3* :  $T2\_state = Sns\_SEND \Rightarrow bfrdFlg = TRUE$   
*inv4* :  $T2\_state = Sns\_GENR \Rightarrow T2\_sensEvaluated = FALSE$   
*inv5* :  $T2\_state = Sns\_GENR \Rightarrow T2\_snsLoc\_speed \in speed[cycle - 4 .. cycle]$

## EVENTS

### Initialisation

**begin**

*act1* :  $acceleration := 0$   
*act2* :  $cnt\_targetSpd := 0$   
*act3* :  $cycle := 0$   
*act4* :  $T1\_cntEvaluated := TRUE$   
*act5* :  $cnt\_proc\_enbl := FALSE$   
*act6* :  $cnt\_order := \emptyset$   
*act7* :  $counter := 0$   
*act8* :  $env\_evaluated := FALSE$   
*act9* :  $speed := -25 .. 0 \times \{0\}$   
*act10* :  $T1\_cntLocActSpd := 0$   
*act11* :  $rcvFlg := TRUE$

```

    act12 : T1_cntRxSpd := 0
    act13 : T2_sensEvaluated := TRUE
    act14 : T2_snsTX := 0
    act15 : T2_snsLoc_speed := 0
    act16 : T2_state := Sns_EN
  end
Event UpdateCycle1_RcvT  $\hat{=}$ 
extends UpdateCycle1_RcvT
  when

    grd1 : counter = 1
    grd2 : env_evaluated = TRUE
    grd3 : rcvFlg = TRUE
    grd4 : T2_sensEvaluated = TRUE
  then

    act1 : cycle := cycle + 5
    act2 : counter := 0
    act3 : env_evaluated := FALSE
    act4 : T2_sensEvaluated := FALSE
  end
Event UpdateCycle1_RcvF  $\hat{=}$ 
extends UpdateCycle1_RcvF
  when

    grd1 : counter = 1
    grd2 : env_evaluated = TRUE
    grd3 : rcvFlg = FALSE
    grd4 : T2_sensEvaluated = TRUE
  then

    act1 : cycle := cycle + 5
    act2 : counter := 0
    act3 : env_evaluated := FALSE
    act4 : T2_sensEvaluated := FALSE
  end
Event UpdateCycle2  $\hat{=}$ 
extends UpdateCycle2
  when

    grd1 : T1_cntEvaluated = TRUE
    grd2 : counter = 0
    grd3 : env_evaluated = TRUE
    grd4 : T2_sensEvaluated = TRUE
  then

```

```

    act1 : cycle := cycle + 5
    act2 : counter := 1
    act3 : T1_cntEvaluated := FALSE
    act4 : env_evaluated := FALSE
    act5 : rcvFlg := FALSE
    act6 : T2_sensEvaluated := FALSE
end
Event   UpdateSpd_Env  $\hat{=}$ 
extends UpdateSpd_Env
    any
        spd
    where
        grd1 : env_evaluated = FALSE
        grd2 : spd  $\in$  (cycle + 1 .. cycle + 5)  $\rightarrow$  (0 .. maxSpd)
    then
        act1 : speed := speed  $\cup$  spd
        act2 : env_evaluated := TRUE
    end
Event   T2_SenseSpeed  $\hat{=}$ 
    any
        spd
    where
        grd1 : T2_sensEvaluated = FALSE
        grd2 : spd  $\in$  0 .. maxSpd
        grd3 : spd  $\in$  speed[cycle - 4 .. cycle]
        grd4 : T2_state = Sns_EN
    then
        act1 : T2_snsLoc_speed := spd
        act2 : T2_state := Sns_GENR
    end
Event   T2_RequestToTransmitSpd  $\hat{=}$ 
refines T2_RequestToTransmitSpd
    any
        spd
    where
        grd1 : spd = T2_snsLoc_speed
        grd2 : T2_state = Sns_GENR
    then
        act1 : T2_snsTX := spd
        act2 : T2_state := Sns_SEND

```

```

    end
Event T2_SensorTransmitSpd  $\hat{=}$ 
refines T2_Spd
    any
        canTrans
        where
            grd1 : canTrans  $\in$  0 .. maxSpd
            grd2 : canTrans = T2_snsTX
            grd3 : T2_state = Sns_SEND
        then
            act1 : T1_cntRxSpd := canTrans
            act2 : T2_sensEvaluated := TRUE
            act3 : T2_state := Sns_EN
        end
Event T1_ReceiveEnvVal  $\hat{=}$ 
extends T1_ReceiveEnvVal
    any
        Rx2
        where
            grd1 : Rx2 = T1_cntRxSpd
            grd2 : cnt_proc_enbl = FALSE
            grd3 : T1_cntEvaluated = FALSE
        then
            act1 : cnt_proc_enbl := TRUE
            act2 : T1_cntLocActSpd := Rx2
            act3 : rcvFlg := TRUE
        end
Event T1_SetTargetSpd  $\hat{=}$ 
extends T1_SetTargetSpd
    any
        spd
        where
            grd1 : spd  $\in$  lb .. ub
            grd2 : cnt_proc_enbl = TRUE
            grd3 : cnt_order =  $\emptyset$ 
        then
            act1 : cnt_targetSpd := spd
            act2 : cnt_order := {TargSpd}
        end
Event T1_SkipSetTargetSpd  $\hat{=}$ 

```

```

extends T1_SkipSetTargetSpd
  when
    grd1 : cnt_proc_enbl = TRUE
    grd2 : cnt_order =  $\emptyset$ 
  then
    act1 : cnt_order :=  $\{TargSpd\}$ 
  end
Event GenerateAccel  $\hat{=}$ 
extends GenerateAccel
  when
    grd1 : TargSpd  $\in$  cnt_order
  then
    act1 : acceleration := Accel_Func(T1_cntLocActSpd  $\mapsto$  cnt_targetSpd)
    act2 : T1_cntEvaluated := TRUE
    act3 : cnt_proc_enbl := FALSE
    act4 : cnt_order :=  $\emptyset$ 
  end
END

```

### B.2.8 Refinement 7: Machine CC7

**MACHINE** CC7

Button ECU is introduced completely

**REFINES** CC6

**SEES** C2

**VARIABLES**

*acceleration* CC0 - Functional Req  
*cnt\_targetSpd* CC0 - Functional Req  
*cycle* CC1 - cycle of tasks  
*T1\_cntEvaluated* CC1 - controller task is evaluated  
*cnt\_proc\_enbl* CC1 - enabling flags  
*cnt\_order* CC1 - to define order between events  
*counter* CC1 - to avoid using mod in invariants  
*env\_evaluated* CC2 - environment should be evaluated at every cycle  
*speed* CC2 - value of speed at each moment of time  
*T1\_cntLocActSpd* CC3 - Local buffer of T1 (CNT Task)  
*rcvFlg* CC3 - TRUE when the CNT receive events happens. Reset at the end  
of every 10ms (by updateCycle2)  
*T1\_cntRxSpd* CC4 - RX buffer in task 1 (CNT)

*T2\_sensEvaluated* CC4 - sensor must be evaluated before the cycle can progress  
*T2\_snsTX* CC5 - Transmitter buffer of sensor  
*T2\_snsLoc\_speed* CC6 - local buffer of the sensor which generates the sensed value  
*T2\_state* CC6 - To model the order between events within the task T2.  
*T3\_btnTX* CC7 - The transmitter buffer of button ECU  
*T3\_state* CC7 - To define the order between the events in the BTN task (T3)  
*T3\_btnEvaluated* CC7 - T3 should be evaluated before cycle can progress  
*targetSpdBtn* CC7 - The button in the environment  
*T1\_cntRxBtn* CC7 - The Receiver buffer in T1 (CNT task)  
*T1\_cntLocBtn* CC7 - Controller's local interpretation of the button value.

## INVARIANTS

*inv1* :  $T3\_btnTX \in \text{BOOL}$   
*inv2* :  $T3\_state \in T3State$   
*inv3* :  $T3\_btnEvaluated \in \text{BOOL}$   
*inv4* :  $targetSpdBtn \in \text{BOOL}$   
*inv5* :  $T1\_cntRxBtn \in \text{BOOL}$   
*inv6* :  $T1\_cntLocBtn \in \text{BOOL}$

## EVENTS

### Initialisation

*extended*

**begin**

*act1* :  $acceleration := 0$   
*act2* :  $cnt\_targetSpd := 0$   
*act3* :  $cycle := 0$   
*act4* :  $T1\_cntEvaluated := \text{TRUE}$   
*act5* :  $cnt\_proc\_enbl := \text{FALSE}$   
*act6* :  $cnt\_order := \emptyset$   
*act7* :  $counter := 0$   
*act8* :  $env\_evaluated := \text{FALSE}$   
*act9* :  $speed := -25 .. 0 \times \{0\}$   
*act10* :  $T1\_cntLocActSpd := 0$   
*act11* :  $rcvFlg := \text{TRUE}$   
*act12* :  $T1\_cntRxSpd := 0$   
*act13* :  $T2\_sensEvaluated := \text{TRUE}$   
*act14* :  $T2\_snsTX := 0$   
*act15* :  $T2\_snsLoc\_speed := 0$   
*act16* :  $T2\_state := Sns\_EN$   
*act17* :  $T3\_state := T3\_EN$

```

    act18 : T3_btnTX := FALSE
    act19 : T3_btnEvaluated := TRUE
    act20 : targetSpdBtn := FALSE
    act21 : T1_cntRxBtn := FALSE
    act22 : T1_cntLocBtn := FALSE
  end
Event UpdateCycle1_RcvT  $\hat{=}$ 
extends UpdateCycle1_RcvT
  when

    grd1 : counter = 1
    grd2 : env_evaluated = TRUE
    grd3 : rcvFlg = TRUE
    grd4 : T2_sensEvaluated = TRUE
    then
      grd5 : T3_btnEvaluated = TRUE

      act1 : cycle := cycle + 5
      act2 : counter := 0
      act3 : env_evaluated := FALSE
      act4 : T2_sensEvaluated := FALSE
      act5 : T3_btnEvaluated := FALSE
    end
Event UpdateCycle1_RcvF  $\hat{=}$ 
extends UpdateCycle1_RcvF
  when

    grd1 : counter = 1
    grd2 : env_evaluated = TRUE
    grd3 : rcvFlg = FALSE
    grd4 : T2_sensEvaluated = TRUE
    then
      grd5 : T3_btnEvaluated = TRUE

      act1 : cycle := cycle + 5
      act2 : counter := 0
      act3 : env_evaluated := FALSE
      act4 : T2_sensEvaluated := FALSE
      act5 : T3_btnEvaluated := FALSE
    end
Event UpdateCycle2  $\hat{=}$ 
extends UpdateCycle2
  when

    grd1 : T1_cntEvaluated = TRUE
    grd2 : counter = 0

```

```

    grd3 : env_evaluated = TRUE
    grd4 : T2_sensEvaluated = TRUE
    grd5 : T3_btnEvaluated = TRUE
  then

    act1 : cycle := cycle + 5
    act2 : counter := 1
    act3 : T1_cntEvaluated := FALSE
    act4 : env_evaluated := FALSE
    act5 : rcvFlg := FALSE
    act6 : T2_sensEvaluated := FALSE
    act7 : T3_btnEvaluated := FALSE
  end

Event   UpdateSpd_Env  $\hat{=}$ 
extends UpdateSpd_Env

  any

    spd
  where

    grd1 : env_evaluated = FALSE
    grd2 : spd  $\in$  (cycle + 1 .. cycle + 5)  $\rightarrow$  (0 .. maxSpd)
  then

    act1 : speed := speed  $\cup$  spd
    act2 : env_evaluated := TRUE
  end

Event   T2_SenseSpeed  $\hat{=}$ 
extends T2_SenseSpeed

  any

    spd
  where

    grd1 : T2_sensEvaluated = FALSE
    grd2 : spd  $\in$  0 .. maxSpd
    grd3 : spd  $\in$  speed[cycle - 4 .. cycle]
    grd4 : T2_state = Sns_EN
  then

    act1 : T2_snsLoc_speed := spd
    act2 : T2_state := Sns_GENR
  end

Event   T2_RequestToTransmitSpd  $\hat{=}$ 
extends T2_RequestToTransmitSpd

  any

    spd

```



```

where

    grd1 : spd = T2_snsLoc_speed
    grd2 : T2_state = Sns_GENR
then

    act1 : T2_snsTX := spd
    act2 : T2_state := Sns_SEND
end

Event T2_SensorTransmitSpd  $\hat{=}$ 
extends T2_SensorTransmitSpd

    any

        canTrans
        where

            grd1 : canTrans  $\in$  0 .. maxSpd
            grd2 : canTrans = T2_snsTX
            grd3 : T2_state = Sns_SEND
        then

            act1 : T1_cntRxSpd := canTrans
            act2 : T2_snsEvaluated := TRUE
            act3 : T2_state := Sns_EN
        end

Event T3_GenerateTargetSpdBtn  $\hat{=}$ 

    any

        btn
        where

            grd1 : T3_btnEvaluated = FALSE
            grd2 : btn  $\in$  BOOL
            grd3 : T3_state = T3_EN
        then

            act1 : targetSpdBtn := btn
            act2 : T3_state := T3_GENR
        end

Event T3_BufferTargetSpdBtn  $\hat{=}$ 

    any

        btn
        where

            grd1 : btn = targetSpdBtn
            grd2 : T3_state = T3_GENR
        then

            act1 : T3_btnTX := btn
            act2 : T3_state := T3_SEND

```

```

    end
Event T3_TransmitTargetSpdBtn  $\hat{=}$ 
    any
        canTrans
    where
        grd1 : canTrans  $\in$  BOOL
        grd2 : canTrans = T3_btnTX
        grd3 : T3_state = T3_SEND
    then
        act1 : T1_cntRxBtn := canTrans
        act2 : T3_state := T3_EN
        act3 : T3_btnEvaluated := TRUE
    end
Event T1_ReceiveEnvVal  $\hat{=}$ 
extends T1_ReceiveEnvVal
    any
        Rx2
    where
        Rx1
    where
        grd1 : Rx2 = T1_cntRxSpd
        grd2 : cnt_proc_enbl = FALSE
        grd3 : T1_cntEvaluated = FALSE
        grd4 : Rx1 = T1_cntRxBtn
    then
        act1 : cnt_proc_enbl := TRUE
        act2 : T1_cntLocActSpd := Rx2
        act3 : rcvFlg := TRUE
        act4 : T1_cntLocBtn := Rx1
    end
Event T1_SetTargetSpd  $\hat{=}$ 
extends T1_SetTargetSpd
    any
        spd
    where
        grd1 : spd  $\in$  lb .. ub
        grd2 : cnt_proc_enbl = TRUE
        grd3 : cnt_order =  $\emptyset$ 
        grd4 : T1_cntLocBtn = TRUE
    then
        act1 : cnt_targetSpd := spd

```

```

        act2 : cnt_order := {TargSpd}
    end
Event T1_SkipSetTargetSpd  $\hat{=}$ 
extends T1_SkipSetTargetSpd
    when

        grd1 : cnt_proc_enbl = TRUE
        grd2 : cnt_order =  $\emptyset$ 
        grd3 : T1_cntLocBtn = FALSE
    then

        act1 : cnt_order := {TargSpd}
    end
Event GenerateAccel  $\hat{=}$ 
extends GenerateAccel
    when

        grd1 : TargSpd  $\in$  cnt_order
    then

        act1 : acceleration := Accel_Func(T1_cntLocActSpd  $\mapsto$  cnt_targetSpd)
        act2 : T1_cntEvaluated := TRUE
        act3 : cnt_proc_enbl := FALSE
        act4 : cnt_order :=  $\emptyset$ 
    end
END

```

### B.2.9 Refinement 8: Machine CC8

**MACHINE** CC8

Introducing actuator

**REFINES** CC7

**SEES** C3

**VARIABLES**

*acceleration* CC0 - Functional Req

*cnt\_targetSpd* CC0 - Functional Req

*cycle* CC1 - cycle of tasks

*T1\_cntEvaluated* CC1 - controller task is evaluated

*cnt\_proc\_enbl* CC1 - enabling flags

*cnt\_order* CC1 - to define order between events

*counter* CC1 - to avoid using mod in invariants

*env\_evaluated* CC2 - environment should be evaluated at every cycle

*speed* CC2 - value of speed at each moment of time

*T1\_cntLocActSpd* CC3 - Local buffer of T1 (CNT Task)  
*rcvFlg* CC3 - TRUE when the CNT receive events happens. Reset at the end of every 10ms (by *updateCycle2*)  
*T1\_cntRxSpd* CC4 - RX buffer in task 1 (CNT)  
*T2\_sensEvaluated* CC4 - sensor must be evaluated before the cycle can progress  
*T2\_snsTX* CC5 - Transmitter buffer of sensor  
*T2\_snsLoc\_speed* CC6 - local buffer of the sensor which generates the sensed value  
*T2\_state* CC6 - To model the order between events within the task T2.  
*T3\_btnTX* CC7 - The transmitter buffer of button ECU  
*T3\_state* CC7 - To define the order between the events in the BTN task (T3)  
*T3\_btnEvaluated* CC7 - T3 should be evaluated before cycle can progress.  
*targetSpdBtn* CC7 - The button in the environment  
*T1\_cntRxBtn* CC7 - The Receiver buffer in T1 (CNT task)  
*T1\_cntLocBtn* CC7 - Controller's local interpretation of the button value.  
*T1\_cntLocAccel* CC8 - Local buffer in T1 which stores the decision of the CCS.  
*T1\_cntTxAccel* CC8 - Transmitter buffer in the CAN which stores the value of acceleration.  
*T4\_actRxAccel* CC8 - Receiver buffer in T4 (ACT) which gets the value of acceleration from the CAN.  
*T4\_actLocAccel* CC8 - Local buffer in T4 which stores the received acceleration.  
*T4\_actEvaluated* CC8 - T4 should be evaluated before cycle can progress.  
*cnt\_accel* CC8 - Ordering Flag  
*brfAccelFlg* CC8 - Ordering Flag  
*T4\_state* CC8 - The state of T4.

## INVARIANTS

*inv1* :  $T1\_cntLocAccel \in \mathbb{N}$   
*inv2* :  $T1\_cntTxAccel \in \mathbb{N}$   
*inv3* :  $T4\_actEvaluated \in \text{BOOL}$   
*inv4* :  $T4\_actEvaluated = \text{FALSE} \Rightarrow TargSpd \in cnt\_order$   
*inv5* :  $brfAccelFlg = \text{TRUE} \Rightarrow cnt\_accel = \text{TRUE}$   
*inv6* :  $brfAccelFlg = \text{TRUE} \Rightarrow$   
 $T1\_cntTxAccel = \text{Accel\_Func}(T1\_cntLocActSpd \mapsto cnt\_targetSpd)$   
*inv7* :  $cnt\_accel \in \text{BOOL}$   
*inv8* :  $cnt\_accel = \text{TRUE} \Rightarrow TargSpd \in cnt\_order$   
*inv9* :  $cnt\_accel = \text{TRUE} \Rightarrow$   
 $T1\_cntLocAccel = \text{Accel\_Func}(T1\_cntLocActSpd \mapsto cnt\_targetSpd)$   
*inv10* :  $T4\_actRxAccel \in \mathbb{N}$   
*inv11* :  $T4\_state \in T4State$

$\text{inv12} : T4\_state = T4\_ACT \Rightarrow T4\_actEvaluated = FALSE$   
 $\text{inv13} : T4\_actEvaluated = FALSE \Rightarrow$   
 $T4\_actRxAccel = Accel\_Func(T1\_cntLocActSpd \mapsto cnt\_targetSpd)$   
 $\text{inv14} : T4\_actLocAccel \in \mathbb{N}$   
 $\text{inv15} : brfAccelFlg \in BOOL$   
 $\text{inv16} : T4\_state = T4\_ACT \Rightarrow T4\_actLocAccel = T4\_actRxAccel$

## EVENTS

### Initialisation

*extended*

**begin**

$\text{act1} : acceleration := 0$   
 $\text{act2} : cnt\_targetSpd := 0$   
 $\text{act3} : cycle := 0$   
 $\text{act4} : T1\_cntEvaluated := TRUE$   
 $\text{act5} : cnt\_proc\_enbl := FALSE$   
 $\text{act6} : cnt\_order := \emptyset$   
 $\text{act7} : counter := 0$   
 $\text{act8} : env\_evaluated := FALSE$   
 $\text{act9} : speed := -25 .. 0 \times \{0\}$   
 $\text{act10} : T1\_cntLocActSpd := 0$   
 $\text{act11} : rcvFlg := TRUE$   
 $\text{act12} : T1\_cntRxSpd := 0$   
 $\text{act13} : T2\_sensEvaluated := TRUE$   
 $\text{act14} : T2\_snsTX := 0$   
 $\text{act15} : T2\_snsLoc\_speed := 0$   
 $\text{act16} : T2\_state := Sns\_EN$   
 $\text{act17} : T3\_state := T3\_EN$   
 $\text{act18} : T3\_btnTX := FALSE$   
 $\text{act19} : T3\_btnEvaluated := TRUE$   
 $\text{act20} : targetSpdBtn := FALSE$   
 $\text{act21} : T1\_cntRxBtn := FALSE$   
 $\text{act22} : T1\_cntLocBtn := FALSE$   
 $\text{act23} : T4\_actEvaluated := TRUE$   
 $\text{act24} : T1\_cntLocAccel := 0$   
 $\text{act25} : cnt\_accel := FALSE$   
 $\text{act26} : T1\_cntTxAccel := 0$   
 $\text{act27} : T4\_actRxAccel := 0$   
 $\text{act28} : T4\_state := T4\_EN$   
 $\text{act29} : T4\_actLocAccel := 0$   
 $\text{act30} : brfAccelFlg := FALSE$

**end**

**Event**  $UpdateCycle1\_RcvT \hat{=}$

**extends**  $UpdateCycle1\_RcvT$

**when**

$grd1 : counter = 1$

$grd2 : env\_evaluated = TRUE$

$grd3 : rcvFlg = TRUE$

$grd4 : T2\_sensEvaluated = TRUE$

$grd5 : T3\_btnEvaluated = TRUE$

**then**

$act1 : cycle := cycle + 5$

$act2 : counter := 0$

$act3 : env\_evaluated := FALSE$

$act4 : T2\_sensEvaluated := FALSE$

$act5 : T3\_btnEvaluated := FALSE$

**end**

**Event**  $UpdateCycle1\_RcvF \hat{=}$

**extends**  $UpdateCycle1\_RcvF$

**when**

$grd1 : counter = 1$

$grd2 : env\_evaluated = TRUE$

$grd3 : rcvFlg = FALSE$

$grd4 : T2\_sensEvaluated = TRUE$

$grd5 : T3\_btnEvaluated = TRUE$

**then**

$act1 : cycle := cycle + 5$

$act2 : counter := 0$

$act3 : env\_evaluated := FALSE$

$act4 : T2\_sensEvaluated := FALSE$

$act5 : T3\_btnEvaluated := FALSE$

**end**

**Event**  $UpdateCycle2 \hat{=}$

**extends**  $UpdateCycle2$

**when**

$grd1 : T1\_cntEvaluated = TRUE$

$grd2 : counter = 0$

$grd3 : env\_evaluated = TRUE$

$grd4 : T2\_sensEvaluated = TRUE$

$grd5 : T3\_btnEvaluated = TRUE$

**then**

$act1 : cycle := cycle + 5$

$act2 : counter := 1$

```

    act3 : T1_cntEvaluated := FALSE
    act4 : env_evaluated := FALSE
    act5 : rcvFlg := FALSE
    act6 : T2_sensEvaluated := FALSE
    act7 : T3_btnEvaluated := FALSE
end
Event   UpdateSpd_Env  $\hat{=}$ 
extends UpdateSpd_Env
    any
        spd
    where
        grd1 : env_evaluated = FALSE
        grd2 : spd  $\in$  (cycle + 1 .. cycle + 5)  $\rightarrow$  (0 .. maxSpd)
    then
        act1 : speed := speed  $\cup$  spd
        act2 : env_evaluated := TRUE
    end
Event   T2_SenseSpeed  $\hat{=}$ 
extends T2_SenseSpeed
    any
        spd
    where
        grd1 : T2_sensEvaluated = FALSE
        grd2 : spd  $\in$  0 .. maxSpd
        grd3 : spd  $\in$  speed[cycle - 4 .. cycle]
        grd4 : T2_state = Sns_EN
    then
        act1 : T2_snsLoc_speed := spd
        act2 : T2_state := Sns_GENR
    end
Event   T2_RequestToTransmitSpd  $\hat{=}$ 
extends T2_RequestToTransmitSpd
    any
        spd
    where
        grd1 : spd = T2_snsLoc_speed
        grd2 : T2_state = Sns_GENR
    then
        act1 : T2_snsTX := spd
        act2 : T2_state := Sns_SEND

```

```

    end
Event T2_SensorTransmitSpd  $\hat{=}$ 
extends T2_SensorTransmitSpd
    any
        canTrans
        where
            grd1 : canTrans  $\in$  0 .. maxSpd
            grd2 : canTrans = T2_snsTX
            grd3 : T2_state = Sns_SEND
        then
            act1 : T1_cntRxSpd := canTrans
            act2 : T2_sensEvaluated := TRUE
            act3 : T2_state := Sns_EN
        end
Event T3_GenerateTargetSpdBtn  $\hat{=}$ 
extends T3_GenerateTargetSpdBtn
    any
        btn
        where
            grd1 : T3_btnEvaluated = FALSE
            grd2 : btn  $\in$  BOOL
            grd3 : T3_state = T3_EN
        then
            act1 : targetSpdBtn := btn
            act2 : T3_state := T3_GENR
        end
Event T3_BufferTargetSpdBtn  $\hat{=}$ 
extends T3_BufferTargetSpdBtn
    any
        btn
        where
            grd1 : btn = targetSpdBtn
            grd2 : T3_state = T3_GENR
        then
            act1 : T3_btnTX := btn
            act2 : T3_state := T3_SEND
        end
Event T3_TransmitTargetSpdBtn  $\hat{=}$ 
extends T3_TransmitTargetSpdBtn
    any
        canTrans

```



```

where

    grd1 : canTrans ∈ BOOL
    grd2 : canTrans = T3_btnTX
    grd3 : T3_state = T3_SEND
then

    act1 : T1_cntRxBtn := canTrans
    act2 : T3_state := T3_EN
    act3 : T3_btnEvaluated := TRUE
end

Event T1_ReceiveEnvVal ≐
extends T1_ReceiveEnvVal

    any

        Rx2
        Rx1
    where

        grd1 : Rx2 = T1_cntRxSpd
        grd2 : cnt_proc_enbl = FALSE
        grd3 : T1_cntEvaluated = FALSE
        grd4 : Rx1 = T1_cntRxBtn
    then

        act1 : cnt_proc_enbl := TRUE
        act2 : T1_cntLocActSpd := Rx2
        act3 : rcvFlg := TRUE
        act4 : T1_cntLocBtn := Rx1
    end

Event T1_SetTargetSpd ≐
extends T1_SetTargetSpd

    any

        spd
    where

        grd1 : spd ∈ lb .. ub
        grd2 : cnt_proc_enbl = TRUE
        grd3 : cnt_order = ∅
        grd4 : T1_cntLocBtn = TRUE
    then

        act1 : cnt_targetSpd := spd
        act2 : cnt_order := {TargSpd}
    end

Event T1_SkipSetTargetSpd ≐
extends T1_SkipSetTargetSpd

    when

```

```

    grd1 : cnt_proc_enbl = TRUE
    grd2 : cnt_order = ∅
    grd3 : T1_cntLocBtn = FALSE
  then

    act1 : cnt_order := {TargSpd}
  end

Event T1_GenerateAccel ≐
  when

    grd1 : TargSpd ∈ cnt_order
    grd2 : cnt_accel = FALSE
  then

    act1 : T1_cntLocAccel := Accel_Func(T1_cntLocActSpd ↦ cnt_targetSpd)
    act2 : cnt_accel := TRUE
  end

Event T1_BufferAcceleration ≐
  any

    accel
  where

    grd1 : cnt_accel = TRUE
    grd2 : accel ∈ ℕ
    grd3 : accel = T1_cntLocAccel
  then

    act1 : T1_cntTxAccel := accel
    act2 : brfAccelFlg := TRUE
  end

Event T1_TransmitAcceleration ≐
  any

    accel
  where

    grd1 : brfAccelFlg = TRUE
    grd2 : accel ∈ ℕ
    grd3 : accel = T1_cntTxAccel
  then

    act1 : T4_actRxAccel := accel
    act2 : T4_actEvaluated := FALSE
  end

Event T4_ReceiveAcceleration ≐
  any

    accel
  where

```

```

    grd1 : T4_actEvaluated = FALSE
    grd2 : T4_state = T4_EN
    grd3 : accel ∈ ℕ
    grd4 : accel = T4_actRxAccel
  then

    act1 : T4_actLocAccel := accel
    act2 : T4_state := T4_ACT
  end

Event   T4_ActuatingAcceleration ≐
refines GenerateAccel

  any

    accel
  where

    grd1 : T4_state = T4_ACT
    grd2 : accel ∈ ℕ
    grd3 : accel = T4_actLocAccel
  then

    act1 : acceleration := accel
    act2 : T1_cntEvaluated := TRUE
    act3 : cnt_proc_enbl := FALSE
    act4 : cnt_order := ∅
    act5 : T4_actEvaluated := TRUE
    act6 : cnt_accel := FALSE
    act7 : T4_state := T4_EN
    act8 : brfAccelFlg := FALSE
  end
end
END

```

### B.2.10 Refinement 9: Machine CC9

**MACHINE** CC9

Upper bound on the number of messages which can be sent and received  
within every cycle are defined.

**REFINES** CC8

**SEES** C4

**VARIABLES**

*acceleration* CC0 - Functional Req  
*cnt\_targetSpd* CC0 - Functional Req  
*cycle* CC1 - cycle of tasks  
*T1\_cntEvaluated* CC1 - controller task is evaluated

*cnt\_proc\_enbl* CC1 - enabling flags  
*cnt\_order* CC1 - to define order between events  
*counter* CC1 - to avoid using mod in invariants  
*env\_evaluated* CC2 - environment should be evaluated at every cycle  
*speed* CC2 - value of speed at each moment of time  
*T1\_cntLocActSpd* CC3 - Local buffer of T1 (CNT Task)  
*rcvFlg* CC3 - TRUE when the CNT receive events happens. Reset at the end of every 10ms (by updateCycle2)  
*T1\_cntRxSpd* CC4 - RX buffer in task 1 (CNT)  
*T2\_sensEvaluated* CC4 - sensor must be evaluated before the cycle can progress  
*T2\_snsTX* CC5 - Transmitter buffer of sensor  
*T2\_snsLoc\_speed* CC6 - local buffer of the sensor which generates the sensed value  
*T2\_state* CC6 - To model the order between events in the task T2.  
*T3\_btnTX* CC7 - The transmitter buffer of button ECU  
*T3\_state* CC7 - To define the order between the events in the BTN task (T3)  
*T3\_btnEvaluated* CC7 - T3 should be evaluated before cycle can progress.  
*targetSpdBtn* CC7 - The button in the environment  
*T1\_cntRxBtn* CC7 - The Receiver buffer in T1 (CNT task)  
*T1\_cntLocBtn* CC7 - Controller's local interpretation of the button value.  
*T1\_cntLocAccel* CC8 - Local buffer in T1 which stores the decision of the CCS.  
*T1\_cntTxAccel* CC8 - Transmitter buffer in the CAN which stores the value of acceleration.  
*T4\_actRxAccel* CC8 - Receiver buffer in T4 (ACT) which gets the value of acceleration from the CAN.  
*T4\_actLocAccel* CC8 - Local buffer in T4 which stores the received acceleration.  
*T4\_actEvaluated* CC8 - T4 should be evaluated before cycle can progress.  
*cnt\_accel* CC8 - Ordering Flag  
*brfAccelFlg* CC8 - Ordering Flag  
*T4\_state* CC8 - The state of T4.  
*msgCounter* CC9 - Counting the number of messages that are sent within a cycle.

## INVARIANTS

$inv1 : msgCounter \in \mathbb{N}$   
 $inv2 : msgCounter \leq MaxNumMSG$   
 $inv3 : T2\_sensEvaluated \neq TRUE \wedge T1\_cntEvaluated \neq TRUE \wedge T3\_btnEvaluated \neq TRUE \Rightarrow msgCounter = 0$   
 $inv4 : T2\_sensEvaluated \neq TRUE \wedge T1\_cntEvaluated = TRUE \wedge T3\_btnEvaluated \neq TRUE \Rightarrow msgCounter \leq 1$

$\text{inv5} : T2\_sensEvaluated \neq TRUE \wedge T1\_cntEvaluated \neq TRUE \wedge$   
 $T3\_btnEvaluated = TRUE \Rightarrow msgCounter = 1$   
 $\text{inv6} : T2\_sensEvaluated = TRUE \wedge T1\_cntEvaluated \neq TRUE \wedge$   
 $T3\_btnEvaluated \neq TRUE \Rightarrow msgCounter = 1$   
 $\text{inv7} : T2\_sensEvaluated = TRUE \wedge T1\_cntEvaluated = TRUE \wedge$   
 $T3\_btnEvaluated \neq TRUE \Rightarrow msgCounter \leq 2$   
 $\text{inv8} : T2\_sensEvaluated = TRUE \wedge T1\_cntEvaluated \neq TRUE \wedge$   
 $T3\_btnEvaluated = TRUE \Rightarrow msgCounter = 2$   
 $\text{inv9} : T2\_sensEvaluated \neq TRUE \wedge T1\_cntEvaluated = TRUE \wedge$   
 $T3\_btnEvaluated = TRUE \Rightarrow msgCounter \leq 2$   
 $\text{inv10} : T2\_sensEvaluated = TRUE \wedge T1\_cntEvaluated = TRUE \wedge$   
 $T3\_btnEvaluated = TRUE \Rightarrow msgCounter \leq 3$   
 $\text{inv11} : T2\_state = Sns\_SEND \Rightarrow T2\_sensEvaluated \neq TRUE$   
 $\text{inv12} : T2\_sensEvaluated \neq TRUE \Rightarrow msgCounter \leq 2$   
 $\text{inv13} : T1\_cntEvaluated \neq TRUE \Rightarrow msgCounter \leq 2$   
 $\text{inv14} : T3\_btnEvaluated \neq TRUE \Rightarrow msgCounter \leq 2$   
 $\text{inv15} : T3\_state = T3\_SEND \Rightarrow T3\_btnEvaluated \neq TRUE$   
 $\text{inv16} : T3\_state = T3\_GENR \Rightarrow T3\_btnEvaluated \neq TRUE$

## EVENTS

### Initialisation

*extended*

**begin**

$\text{act1} : acceleration := 0$   
 $\text{act2} : cnt\_targetSpd := 0$   
 $\text{act3} : cycle := 0$   
 $\text{act4} : T1\_cntEvaluated := TRUE$   
 $\text{act5} : cnt\_proc\_enbl := FALSE$   
 $\text{act6} : cnt\_order := \emptyset$   
 $\text{act7} : counter := 0$   
 $\text{act8} : env\_evaluated := FALSE$   
 $\text{act9} : speed := -25 .. 0 \times \{0\}$   
 $\text{act10} : T1\_cntLocActSpd := 0$   
 $\text{act11} : rcvFlg := TRUE$   
 $\text{act12} : T1\_cntRxSpd := 0$   
 $\text{act13} : T2\_sensEvaluated := TRUE$   
 $\text{act14} : T2\_snsTX := 0$   
 $\text{act15} : T2\_snsLoc\_speed := 0$   
 $\text{act16} : T2\_state := Sns\_EN$   
 $\text{act17} : T3\_state := T3\_EN$   
 $\text{act18} : T3\_btnTX := FALSE$   
 $\text{act19} : T3\_btnEvaluated := TRUE$   
 $\text{act20} : targetSpdBtn := FALSE$

```

    act21 : T1_cntRxBtn := FALSE
    act22 : T1_cntLocBtn := FALSE
    act23 : T4_actEvaluated := TRUE
    act24 : T1_cntLocAccel := 0
    act25 : cnt_accel := FALSE
    act26 : T1_cntTxAccel := 0
    act27 : T4_actRxAccel := 0
    act28 : T4_state := T4_EN
    act29 : T4_actLocAccel := 0
    act30 : brfAccelFlg := FALSE
    act31 : msgCounter := 3
  end
Event UpdateCycle1_RcvT  $\hat{=}$ 
extends UpdateCycle1_RcvT
  when

    grd1 : counter = 1
    grd2 : env_evaluated = TRUE
    grd3 : rcvFlg = TRUE
    grd4 : T2_sensEvaluated = TRUE
    grd5 : T3_btnEvaluated = TRUE
  then

    act1 : cycle := cycle + 5
    act2 : counter := 0
    act3 : env_evaluated := FALSE
    act4 : T2_sensEvaluated := FALSE
    act5 : T3_btnEvaluated := FALSE
    act6 : msgCounter := 0
  end
Event UpdateCycle1_RcvF  $\hat{=}$ 
extends UpdateCycle1_RcvF
  when

    grd1 : counter = 1
    grd2 : env_evaluated = TRUE
    grd3 : rcvFlg = FALSE
    grd4 : T2_sensEvaluated = TRUE
    grd5 : T3_btnEvaluated = TRUE
  then

    act1 : cycle := cycle + 5
    act2 : counter := 0
    act3 : env_evaluated := FALSE
    act4 : T2_sensEvaluated := FALSE

```

```

    act5 : T3_btnEvaluated := FALSE
    act6 : msgCounter := 0
end
Event UpdateCycle2  $\hat{=}$ 
extends UpdateCycle2
when
    grd1 : T1_cntEvaluated = TRUE
    grd2 : counter = 0
    grd3 : env_evaluated = TRUE
    grd4 : T2_sensEvaluated = TRUE
    grd5 : T3_btnEvaluated = TRUE
then
    act1 : cycle := cycle + 5
    act2 : counter := 1
    act3 : T1_cntEvaluated := FALSE
    act4 : env_evaluated := FALSE
    act5 : rcvFlg := FALSE
    act6 : T2_sensEvaluated := FALSE
    act7 : T3_btnEvaluated := FALSE
    act8 : msgCounter := 0
end
Event UpdateSpd_Env  $\hat{=}$ 
extends UpdateSpd_Env
any
    spd
where
    grd1 : env_evaluated = FALSE
    grd2 : spd  $\in$  (cycle + 1 .. cycle + 5)  $\rightarrow$  (0 .. maxSpd)
then
    act1 : speed := speed  $\cup$  spd
    act2 : env_evaluated := TRUE
end
Event T2_SenseSpeed  $\hat{=}$ 
extends T2_SenseSpeed
any
    spd
where
    grd1 : T2_sensEvaluated = FALSE
    grd2 : spd  $\in$  0 .. maxSpd
    grd3 : spd  $\in$  speed[cycle - 4 .. cycle]

```

```

    then    grd4 : T2_state = Sns_EN
  then
    act1 : T2_snsLoc_speed := spd
    act2 : T2_state := Sns_GENR
  end
Event T2_RequestToTransmitSpd  $\hat{=}$ 
extends T2_RequestToTransmitSpd
  any
    spd
  where
    grd1 : spd = T2_snsLoc_speed
    grd2 : T2_state = Sns_GENR
  then
    act1 : T2_snsTX := spd
    act2 : T2_state := Sns_SEND
  end
Event T2_SensorTransmitSpd  $\hat{=}$ 
extends T2_SensorTransmitSpd
  any
    canTrans
  where
    grd1 : canTrans  $\in 0 \dots \text{maxSpd}$ 
    grd2 : canTrans = T2_snsTX
    grd3 : T2_state = Sns_SEND
  then
    act1 : T1_cntRxSpd := canTrans
    act2 : T2_sensEvaluated := TRUE
    act3 : T2_state := Sns_EN
    act4 : msgCounter := msgCounter + 1
  end
Event T3_GenerateTargetSpdBtn  $\hat{=}$ 
extends T3_GenerateTargetSpdBtn
  any
    btn
  where
    grd1 : T3_btnEvaluated = FALSE
    grd2 : btn  $\in \text{BOOL}$ 
    grd3 : T3_state = T3_EN
  then
    act1 : targetSpdBtn := btn

```



```

    end    act2 : T3_state := T3_GENR
Event T3_BufferTargetSpdBtn  $\hat{=}$ 
extends T3_BufferTargetSpdBtn
    any
        btn
    where
        grd1 : btn = targetSpdBtn
        grd2 : T3_state = T3_GENR
    then
        act1 : T3_btnTX := btn
        act2 : T3_state := T3_SEND
    end
Event T3_TransmitTargetSpdBtn  $\hat{=}$ 
extends T3_TransmitTargetSpdBtn
    any
        canTrans
    where
        grd1 : canTrans  $\in$  BOOL
        grd2 : canTrans = T3_btnTX
        grd3 : T3_state = T3_SEND
    then
        act1 : T1_cntRxBtn := canTrans
        act2 : T3_state := T3_EN
        act3 : T3_btnEvaluated := TRUE
        act4 : msgCounter := msgCounter + 1
    end
Event T1_ReceiveEnvVal  $\hat{=}$ 
extends T1_ReceiveEnvVal
    any
        Rx2
        Rx1
    where
        grd1 : Rx2 = T1_cntRxSpd
        grd2 : cnt_proc_enbl = FALSE
        grd3 : T1_cntEvaluated = FALSE
        grd4 : Rx1 = T1_cntRxBtn
    then
        act1 : cnt_proc_enbl := TRUE
        act2 : T1_cntLocActSpd := Rx2
        act3 : rcvFlg := TRUE

```

```

    end      act4 : T1_cntLocBtn := Rx1
Event  T1_SetTargetSpd  $\hat{=}$ 
extends T1_SetTargetSpd
    any
        spd
    where
        grd1 : spd  $\in lb \dots ub$ 
        grd2 : cnt_proc_enbl = TRUE
        grd3 : cnt_order =  $\emptyset$ 
        grd4 : T1_cntLocBtn = TRUE
    then
        act1 : cnt_targetSpd := spd
        act2 : cnt_order := {TargSpd}
    end
Event  T1_SkipSetTargetSpd  $\hat{=}$ 
extends T1_SkipSetTargetSpd
    when
        grd1 : cnt_proc_enbl = TRUE
        grd2 : cnt_order =  $\emptyset$ 
        grd3 : T1_cntLocBtn = FALSE
    then
        act1 : cnt_order := {TargSpd}
    end
Event  T1_GenerateAccel  $\hat{=}$ 
extends T1_GenerateAccel
    when
        grd1 : TargSpd  $\in cnt\_order$ 
        grd2 : cnt_accel = FALSE
    then
        act1 : T1_cntLocAccel := Accel_Func(T1_cntLocActSpd  $\mapsto$  cnt_targetSpd)
        act2 : cnt_accel := TRUE
    end
Event  T1_BufferAcceleration  $\hat{=}$ 
extends T1_BufferAcceleration
    any
        accel
    where
        grd1 : cnt_accel = TRUE
        grd2 : accel  $\in \mathbb{N}$ 

```

```

    grd3 : accel = T1_cntLocAccel
  then
    act1 : T1_cntTxAccel := accel
    act2 : brfAccelFlg := TRUE
  end
Event T1_TransmitAcceleration  $\hat{=}$ 
extends T1_TransmitAcceleration
  any
    accel
  where
    grd1 : brfAccelFlg = TRUE
    grd2 : accel  $\in \mathbb{N}$ 
    grd3 : accel = T1_cntTxAccel
  then
    act1 : T4_actRxAccel := accel
    act2 : T4_actEvaluated := FALSE
  end
Event T4_ReceiveAcceleration  $\hat{=}$ 
extends T4_ReceiveAcceleration
  any
    accel
  where
    grd1 : T4_actEvaluated = FALSE
    grd2 : T4_state = T4_EN
    grd3 : accel  $\in \mathbb{N}$ 
    grd4 : accel = T4_actRxAccel
  then
    act1 : T4_actLocAccel := accel
    act2 : T4_state := T4_ACT
  end
Event T4_ActuatingAcceleration  $\hat{=}$ 
extends T4_ActuatingAcceleration
  any
    accel
  where
    grd1 : T4_state = T4_ACT
    grd2 : accel  $\in \mathbb{N}$ 
    grd3 : accel = T4_actLocAccel
  then
    act1 : acceleration := accel
    act2 : T1_cntEvaluated := TRUE

```

```

    act3 : cnt_proc_enbl := FALSE
    act4 : cnt_order := ∅
    act5 : T4_actEvaluated := TRUE
    act6 : cnt_accel := FALSE
    act7 : T4_state := T4_EN
    act8 : brfAccelFlg := FALSE
    act9 : msgCounter := msgCounter + 1
end
END

```

### B.2.11 Refinement 10: Machine CC10

**MACHINE** CC10

CAN is introduced to prioritise the transmission between

BTN, SNS and ACT

**REFINES** CC9

**SEES** C4

**VARIABLES**

*acceleration* CC0 - Functional Req

*cnt\_targetSpd* CC0 - Functional Req

*cycle* CC1 - cycle of tasks

*T1\_cntEvaluated* CC1 - controller task is evaluated

*cnt\_proc\_enbl* CC1 - enabling flags

*cnt\_order* CC1 - to define order between events

*counter* CC1 - to avoid using mod in invariants

*env\_evaluated* CC2 - environment should be evaluated at every cycle

*speed* CC2 - value of speed at each moment of time

*T1\_cntLocActSpd* CC3 - Local buffer of T1 (CNT Task)

*rcvFlg* CC3 - TRUE when the CNT receive events happens. Reset at the end of every 10ms (by updateCycle2)

*T1\_cntRxSpd* CC4 - RX buffer in task 1 (CNT)

*T2\_sensEvaluated* CC4 - sensor must be evaluated before the cycle can progress

*T2\_snsTX* CC5 - Transmitter buffer of sensor

*T2\_snsLoc\_speed* CC6 - local buffer of the sensor which generates the sensed value

*T2\_state* CC6 - To model the order between events within the task T2.

*T3\_btnTX* CC7 - The transmitter buffer of button ECU

*T3\_state* CC7 - To define the order between the events in the BTN task (T3)

*T3\_btnEvaluated* CC7 - T3 should be evaluated before cycle can progress.

*targetSpdBtn* CC7 - The button in the environment  
*T1\_cntRxBtn* CC7 - The Receiver buffer in T1 (CNT task)  
*T1\_cntLocBtn* CC7 - Controller's local interpretation of the button value.  
*T1\_cntLocAccel* CC8 - Local buffer in T1 which stores the decision of  
 the CCS.  
*T1\_cntTxAccel* CC8 - Transmitter buffer in the CAN which stores the value  
 of acceleration.  
*T4\_actRxAccel* CC8 - Receiver buffer in T4 (ACT) which gets the value  
 of acceleration from the CAN.  
*T4\_actLocAccel* CC8 - Local buffer in T4 which stores the received  
 acceleration.  
*T4\_actEvaluated* CC8 - T4 should be evaluated before cycle can progress.  
*cnt\_accel* CC8 - Ordering Flag  
*brfAccelFlg* CC8 - Ordering Flag  
*T4\_state* CC8 - The state of T4.  
*msgCounter* CC9 - Counting the number of messages that are sent within a cycle.  
*CAN\_msgFlg* CC10 - CAN flag is set by each ECU to request sending a message

## INVARIANTS

*inv1* :  $CAN\_msgFlg \in 1..3 \rightarrow BOOL$

## EVENTS

### Initialisation

*extended*

**begin**

*act1* : *acceleration* := 0  
*act2* : *cnt\_targetSpd* := 0  
*act3* : *cycle* := 0  
*act4* : *T1\_cntEvaluated* := *TRUE*  
*act5* : *cnt\_proc\_enbl* := *FALSE*  
*act6* : *cnt\_order* :=  $\emptyset$   
*act7* : *counter* := 0  
*act8* : *env\_evaluated* := *FALSE*  
*act9* : *speed* :=  $-25..0 \times \{0\}$   
*act10* : *T1\_cntLocActSpd* := 0  
*act11* : *rcvFlg* := *TRUE*  
*act12* : *T1\_cntRxSpd* := 0  
*act13* : *T2\_sensEvaluated* := *TRUE*  
*act14* : *T2\_snsTX* := 0  
*act15* : *T2\_snsLoc\_speed* := 0  
*act16* : *T2\_state* := *Sns\_EN*  
*act17* : *T3\_state* := *T3\_EN*  
*act18* : *T3\_btnTX* := *FALSE*

```

    act19 : T3_btnEvaluated := TRUE
    act20 : targetSpdBtn := FALSE
    act21 : T1_cntRxBtn := FALSE
    act22 : T1_cntLocBtn := FALSE
    act23 : T4_actEvaluated := TRUE
    act24 : T1_cntLocAccel := 0
    act25 : cnt_accel := FALSE
    act26 : T1_cntTxAccel := 0
    act27 : T4_actRxAccel := 0
    act28 : T4_state := T4_EN
    act29 : T4_actLocAccel := 0
    act30 : brfAccelFlg := FALSE
    act31 : msgCounter := 3
    act32 : CAN_msgFlg := {1 ↦ FALSE, 2 ↦ FALSE, 3 ↦ FALSE}
  end

Event UpdateCycle1_RcvT ≐
extends UpdateCycle1_RcvT
  when

    grd1 : counter = 1
    grd2 : env_evaluated = TRUE
    grd3 : rcvFlg = TRUE
    grd4 : T2_sensEvaluated = TRUE
    grd5 : T3_btnEvaluated = TRUE
  then

    act1 : cycle := cycle + 5
    act2 : counter := 0
    act3 : env_evaluated := FALSE
    act4 : T2_sensEvaluated := FALSE
    act5 : T3_btnEvaluated := FALSE
    act6 : msgCounter := 0
  end

Event UpdateCycle1_RcvF ≐
extends UpdateCycle1_RcvF
  when

    grd1 : counter = 1
    grd2 : env_evaluated = TRUE
    grd3 : rcvFlg = FALSE
    grd4 : T2_sensEvaluated = TRUE
    grd5 : T3_btnEvaluated = TRUE
  then

    act1 : cycle := cycle + 5

```

```

    act2 : counter := 0
    act3 : env_evaluated := FALSE
    act4 : T2_sensEvaluated := FALSE
    act5 : T3_btnEvaluated := FALSE
    act6 : msgCounter := 0
  end
Event UpdateCycle2  $\hat{=}$ 
extends UpdateCycle2
  when
    grd1 : T1_cntEvaluated = TRUE
    grd2 : counter = 0
    grd3 : env_evaluated = TRUE
    grd4 : T2_sensEvaluated = TRUE
    grd5 : T3_btnEvaluated = TRUE
  then
    act1 : cycle := cycle + 5
    act2 : counter := 1
    act3 : T1_cntEvaluated := FALSE
    act4 : env_evaluated := FALSE
    act5 : rcvFlg := FALSE
    act6 : T2_sensEvaluated := FALSE
    act7 : T3_btnEvaluated := FALSE
    act8 : msgCounter := 0
  end
Event UpdateSpd_Env  $\hat{=}$ 
extends UpdateSpd_Env
  any
    spd
  where
    grd1 : env_evaluated = FALSE
    grd2 : spd  $\in$  (cycle + 1 .. cycle + 5)  $\rightarrow$  (0 .. maxSpd)
  then
    act1 : speed := speed  $\cup$  spd
    act2 : env_evaluated := TRUE
  end
Event T2_SenseSpeed  $\hat{=}$ 
extends T2_SenseSpeed
  any
    spd
  where

```

```

    grd1 : T2_sensEvaluated = FALSE
    grd2 : spd ∈ 0 .. maxSpd
    grd3 : spd ∈ speed[cycle - 4 .. cycle]
    grd4 : T2_state = Sns_EN
  then

    act1 : T2_snsLoc_speed := spd
    act2 : T2_state := Sns_GENR
  end

Event T2_RequestToTransmitSpd ≐
extends T2_RequestToTransmitSpd
  any

    spd
  where

    grd1 : spd = T2_snsLoc_speed
    grd2 : T2_state = Sns_GENR
  then

    act1 : T2_snsTX := spd
    act2 : T2_state := Sns_SEND
    act3 : CAN_msgFlg(3) := TRUE
  end

Event T2_SensorTransmitSpd ≐
extends T2_SensorTransmitSpd
  any

    canTrans
  where

    grd1 : canTrans ∈ 0 .. maxSpd
    grd2 : canTrans = T2_snsTX
    grd3 : T2_state = Sns_SEND
    grd4 : dom(CAN_msgFlg ▷ {TRUE}) ≠ ∅
    grd5 : min(dom(CAN_msgFlg ▷ {TRUE})) = 3
  then

    act1 : T1_cntRxSpd := canTrans
    act2 : T2_sensEvaluated := TRUE
    act3 : T2_state := Sns_EN
    act4 : msgCounter := msgCounter + 1
    act5 : CAN_msgFlg(3) := FALSE
  end

Event T3_GenerateTargetSpdBtn ≐
extends T3_GenerateTargetSpdBtn
  any

```



```

    where  $btn$ 
    grd1 :  $T3\_btnEvaluated = FALSE$ 
    grd2 :  $btn \in BOOL$ 
    grd3 :  $T3\_state = T3\_EN$ 
  then
    act1 :  $targetSpdBtn := btn$ 
    act2 :  $T3\_state := T3\_GENR$ 
  end

Event  $T3\_BufferTargetSpdBtn \hat{=}$ 
extends  $T3\_BufferTargetSpdBtn$ 
  any
    where  $btn$ 
    grd1 :  $btn = targetSpdBtn$ 
    grd2 :  $T3\_state = T3\_GENR$ 
  then
    act1 :  $T3\_btnTX := btn$ 
    act2 :  $T3\_state := T3\_SEND$ 
    act3 :  $CAN\_msgFlg(2) := TRUE$ 
  end

Event  $T3\_TransmitTargetSpdBtn \hat{=}$ 
transmission is part of the cycle!!
extends  $T3\_TransmitTargetSpdBtn$ 
  any
    where  $canTrans$ 
    grd1 :  $canTrans \in BOOL$ 
    grd2 :  $canTrans = T3\_btnTX$ 

    @grd3  $can\_RcvdFlg = FALSE$ 

    grd3 :  $T3\_state = T3\_SEND$ 
    grd4 :  $dom(CAN\_msgFlg \triangleright \{TRUE\}) \neq \emptyset$ 
    grd5 :  $min(dom(CAN\_msgFlg \triangleright \{TRUE\})) = 2$ 
  then
    act1 :  $T1\_cntRxBtn := canTrans$ 
    act2 :  $T3\_state := T3\_EN$ 
    act3 :  $T3\_btnEvaluated := TRUE$ 
    act4 :  $msgCounter := msgCounter + 1$ 
    act5 :  $CAN\_msgFlg(2) := FALSE$ 

```

```

    end
Event  $T1\_ReceiveEnvVal \hat{=}$ 
extends  $T1\_ReceiveEnvVal$ 
    any
         $Rx2$ 
         $Rx1$ 
    where
         $grd1 : Rx2 = T1\_cntRxSpd$ 
         $grd2 : cnt\_proc\_enbl = FALSE$ 
         $grd3 : T1\_cntEvaluated = FALSE$ 
         $grd4 : Rx1 = T1\_cntRxBtn$ 
    then
         $act1 : cnt\_proc\_enbl := TRUE$ 
         $act2 : T1\_cntLocActSpd := Rx2$ 
         $act3 : rcvFlg := TRUE$ 
         $act4 : T1\_cntLocBtn := Rx1$ 
    end
Event  $T1\_SetTargetSpd \hat{=}$ 
extends  $T1\_SetTargetSpd$ 
    any
         $spd$ 
    where
         $grd1 : spd \in lb \dots ub$ 
         $grd2 : cnt\_proc\_enbl = TRUE$ 
         $grd3 : cnt\_order = \emptyset$ 
         $grd4 : T1\_cntLocBtn = TRUE$ 
    then
         $act1 : cnt\_targetSpd := spd$ 
         $act2 : cnt\_order := \{TargSpd\}$ 
    end
Event  $T1\_SkipSetTargetSpd \hat{=}$ 
extends  $T1\_SkipSetTargetSpd$ 
    when
         $grd1 : cnt\_proc\_enbl = TRUE$ 
         $grd2 : cnt\_order = \emptyset$ 
         $grd3 : T1\_cntLocBtn = FALSE$ 
    then
         $act1 : cnt\_order := \{TargSpd\}$ 
    end
Event  $T1\_GenerateAccel \hat{=}$ 

```

**extends** *T1\_GenerateAccel*

**when**

*grd1* : *TargSpd*  $\in$  *cnt\_order*

*grd2* : *cnt\_accel* = *FALSE*

**then**

*act1* : *T1\_cntLocAccel* := *Accel\_Func*(*T1\_cntLocActSpd*  $\mapsto$  *cnt\_targetSpd*)

*act2* : *cnt\_accel* := *TRUE*

**end**

**Event** *T1\_BufferAcceleration*  $\hat{=}$

**extends** *T1\_BufferAcceleration*

**any**

*accel*  
**where**

*grd1* : *cnt\_accel* = *TRUE*

*grd2* : *accel*  $\in$   $\mathbb{N}$

*grd3* : *accel* = *T1\_cntLocAccel*

**then**

*act1* : *T1\_cntTxAccel* := *accel*

*act2* : *brfAccelFlg* := *TRUE*

*act3* : *CAN\_msgFlg*(1) := *TRUE*

**end**

**Event** *T1\_TransmitAcceleration*  $\hat{=}$

**extends** *T1\_TransmitAcceleration*

**any**

*accel*  
**where**

*grd1* : *brfAccelFlg* = *TRUE*

*grd2* : *accel*  $\in$   $\mathbb{N}$

*grd3* : *accel* = *T1\_cntTxAccel*

*grd4* : *dom*(*CAN\_msgFlg*  $\triangleright$  {*TRUE*})  $\neq \emptyset$

*grd5* : *min*(*dom*(*CAN\_msgFlg*  $\triangleright$  {*TRUE*})) = 1

**then**

*act1* : *T4\_actRxAccel* := *accel*

*act2* : *T4\_actEvaluated* := *FALSE*

*act3* : *CAN\_msgFlg*(1) := *FALSE*

**end**

**Event** *T4\_ReceiveAcceleration*  $\hat{=}$

**extends** *T4\_ReceiveAcceleration*

**any**

*accel*

```

where

    grd1 :  $T4\_actEvaluated = FALSE$ 
    grd2 :  $T4\_state = T4\_EN$ 
    grd3 :  $accel \in \mathbb{N}$ 
    grd4 :  $accel = T4\_actRxAccel$ 
then

    act1 :  $T4\_actLocAccel := accel$ 
    act2 :  $T4\_state := T4\_ACT$ 
end

Event  $T4\_ActuatingAcceleration \hat{=}$ 
extends  $T4\_ActuatingAcceleration$ 
any

     $accel$ 
where

    grd1 :  $T4\_state = T4\_ACT$ 
    grd2 :  $accel \in \mathbb{N}$ 
    grd3 :  $accel = T4\_actLocAccel$ 
then

    act1 :  $acceleration := accel$ 

    Accel.Func( $T1\_cntLocActSpd \mapsto cnt\_targetSpd$ )

    act2 :  $T1\_cntEvaluated := TRUE$ 
    act3 :  $cnt\_proc\_enbl := FALSE$ 
    act4 :  $cnt\_order := \emptyset$ 
    act5 :  $T4\_actEvaluated := TRUE$ 
    act6 :  $cnt\_accel := FALSE$ 
    act7 :  $T4\_state := T4\_EN$ 
    act8 :  $brfAccelFlg := FALSE$ 
    act9 :  $msgCounter := msgCounter + 1$ 
end
END

```

### B.2.12 Refinement 11: Machine CC11

**MACHINE** CC11

Clarifying ordering variables of T1 and merging 2 of the cycle events.

**REFINES** CC10

**SEES** C4

**VARIABLES**

$acceleration$  CC0 - Functional Req

*cnt\_targetSpd* CC0 - Functional Req  
*cycle* CC1 - cycle of tasks  
*T1\_cntEvaluated* CC1 - controller task is evaluated  
*counter* CC1 - to avoid using mod in invariants  
*env\_evaluated* CC2 - environment should be evaluated at every cycle  
*speed* CC2 - value of speed at each moment of time  
*T1\_cntLocActSpd* CC3 - Local buffer of T1 (CNT Task)  
*rcvFlg* CC3 - TRUE when the CNT receive events happens. Reset at the end of every 10ms (by updateCycle2)  
*T1\_cntRxSpd* CC4 - RX buffer in task 1 (CNT)  
*T2\_sensEvaluated* CC4 - sensor must be evaluated before the cycle can progress  
*T2\_snsTX* CC5 - Transmitter buffer of sensor  
*T2\_snsLoc\_speed* CC6 - local buffer of the sensor which generates the sensed value  
*T2\_state* CC6 - To model the order between events within the task T2.  
*T3.btnTX* CC7 - The transmitter buffer of button ECU  
*T3.state* CC7 - To define the order between the events in the BTN task (T3)  
*T3.btnEvaluated* CC7 - T3 should be evaluated before cycle can progress.  
*targetSpdBtn* CC7 - The button in the environment  
*T1\_cntRxBtn* CC7 - The Receiver buffer in T1 (CNT task)  
*T1\_cntLocBtn* CC7 - Controller's local interpretation of the button value.  
*T1\_cntLocAccel* CC8 - Local buffer in T1 which stores the decision of the CCS.  
*T1\_cntTxAccel* CC8 - Transmitter buffer in the CAN which stores the value of acceleration.  
*T4\_actRxAccel* CC8 - Receiver buffer in T4 (ACT) which gets the value of acceleration from the CAN.  
*T4\_actLocAccel* CC8 - Local buffer in T4 which stores the received acceleration.  
*T4\_actEvaluated* CC8 - T4 should be evaluated before cycle can progress.  
*T4.state* CC8 - The state of T4.  
*msgCounter* CC9 - Counting the number of messages that are sent within a cycle.  
*CAN\_msgFlg* CC10 - CAN flag is set by each ECU to request sending a message.  
*T1.state* CC11 - Replacing T1 ordering flags with a single variable.

## INVARIANTS

*inv1* :  $T1\_state \in T1State$   
*inv2* :  $T1\_state = T1\_TS \Rightarrow cnt\_proc\_enbl = TRUE \wedge cnt\_order = \emptyset$   
*inv3* :  $T1\_state = T1\_ACCEL \Rightarrow TargSpd \in cnt\_order \wedge cnt\_accel = FALSE$   
*inv4* :  $T1\_state = T1\_BFR \Rightarrow cnt\_accel = TRUE$

*inv5* :  $T1\_state = T1\_SEND \Rightarrow bnfAccelFlg = TRUE$

*inv6* :  $T1\_state = T1\_EN \Rightarrow cnt\_proc\_enbl = FALSE \wedge cnt\_order = \emptyset$

## EVENTS

### Initialisation

**begin**

*act1* :  $acceleration := 0$

*act2* :  $cnt\_targetSpd := 0$

*act3* :  $cycle := 0$

*act4* :  $T1\_cntEvaluated := TRUE$

*act5* :  $counter := 0$

*act6* :  $env\_evaluated := FALSE$

*act7* :  $speed := -25 \dots 0 \times \{0\}$

*act8* :  $T1\_cntLocActSpd := 0$

*act9* :  $rcvFlg := TRUE$

*act10* :  $T1\_cntRxSpd := 0$

*act11* :  $T2\_sensEvaluated := TRUE$

*act12* :  $T2\_snsTX := 0$

*act13* :  $T2\_snsLoc\_speed := 0$

*act14* :  $T2\_state := Sns\_EN$

*act15* :  $T3\_state := T3\_EN$

*act16* :  $T3\_btnTX := FALSE$

*act17* :  $T3\_btnEvaluated := TRUE$

*act18* :  $targetSpdBtn := FALSE$

*act19* :  $T1\_cntRxBtn := FALSE$

*act20* :  $T1\_cntLocBtn := FALSE$

*act21* :  $T4\_actEvaluated := TRUE$

*act22* :  $T1\_cntLocAccel := 0$

*act23* :  $T1\_cntTxAccel := 0$

*act24* :  $T4\_actRxAccel := 0$

*act25* :  $T4\_state := T4\_EN$

*act26* :  $T4\_actLocAccel := 0$

*act27* :  $msgCounter := 3$

*act28* :  $CAN\_msgFlg := \{1 \mapsto FALSE, 2 \mapsto FALSE, 3 \mapsto FALSE\}$

*act29* :  $T1\_state := T1\_EN$

**end**

**Event**  $UpdateCycle1 \hat{=}$

**refines**  $UpdateCycle1\_RcvT, UpdateCycle1\_RcvF$

**when**

*grd1* :  $counter = 1$

*grd2* :  $env\_evaluated = TRUE$

*grd3* :  $T2\_sensEvaluated = TRUE$

```

    grd4 : T3_btnEvaluated = TRUE
  then
    act1 : cycle := cycle + 5
    act2 : counter := 0
    act3 : env_evaluated := FALSE
    act4 : T2_sensEvaluated := FALSE
    act5 : T3_btnEvaluated := FALSE
    act6 : msgCounter := 0
  end
Event UpdateCycle2  $\hat{=}$ 
extends UpdateCycle2
  when
    grd1 : T1_cntEvaluated = TRUE
    grd2 : counter = 0
    grd3 : env_evaluated = TRUE
    grd4 : T2_sensEvaluated = TRUE
    grd5 : T3_btnEvaluated = TRUE
  then
    act1 : cycle := cycle + 5
    act2 : counter := 1
    act3 : T1_cntEvaluated := FALSE
    act4 : env_evaluated := FALSE
    act5 : rcvFlg := FALSE
    act6 : T2_sensEvaluated := FALSE
    act7 : T3_btnEvaluated := FALSE
    act8 : msgCounter := 0
  end
Event UpdateSpd_Env  $\hat{=}$ 
refines UpdateSpd_Env
  any
    spd
    c
  where
    grd1 : env_evaluated = FALSE
    grd2 : c  $\in \mathbb{N}$ 
    grd3 : c = cycle
    grd4 : spd  $\in (c + 1 .. c + 5) \rightarrow (0 .. maxSpd)$ 
  then
    act1 : speed := speed  $\cup$  spd
    act2 : env_evaluated := TRUE
  end
end

```

```

Event  $T2\_SenseSpeed \hat{=}$ 
refines  $T2\_SenseSpeed$ 

  any

     $spd$ 
     $c$ 
  where

     $grd1 : T2\_sensEvaluated = FALSE$ 
     $grd2 : c \in \mathbb{N}$ 
     $grd3 : c = cycle$ 
     $grd4 : spd \in \mathbb{N}$ 
     $grd5 : spd \in 0 .. maxSpd$ 
     $grd6 : spd \in speed[c - 4 .. c]$ 
     $grd7 : T2\_state = Sns\_EN$ 
  then

     $act1 : T2\_snsLoc\_speed := spd$ 
     $act2 : T2\_state := Sns\_GENR$ 
  end

Event  $T2\_RequestToTransmitSpd \hat{=}$ 
extends  $T2\_RequestToTransmitSpd$ 

  any

     $spd$ 
  where

     $grd1 : spd = T2\_snsLoc\_speed$ 
     $grd2 : T2\_state = Sns\_GENR$ 
  then

     $act1 : T2\_snsTX := spd$ 
     $act2 : T2\_state := Sns\_SEND$ 
     $act3 : CAN\_msgFlg(3) := TRUE$ 
  end

Event  $T2\_SensorTransmitSpd \hat{=}$ 
extends  $T2\_SensorTransmitSpd$ 

  any

     $canTrans$ 
  where

     $grd1 : canTrans \in 0 .. maxSpd$ 
     $grd2 : canTrans = T2\_snsTX$ 
     $grd3 : T2\_state = Sns\_SEND$ 
     $grd4 : dom(CAN\_msgFlg \triangleright \{TRUE\}) \neq \emptyset$ 
     $grd5 : min(dom(CAN\_msgFlg \triangleright \{TRUE\})) = 3$ 
  then

     $act1 : T1\_cntRxSpd := canTrans$ 

```



```

    act2 : T2_sensEvaluated := TRUE
    act3 : T2_state := Sns_EN
    act4 : msgCounter := msgCounter + 1
    act5 : CAN_msgFlg(3) := FALSE
end

Event T3_GenerateTargetSpdBtn  $\hat{=}$ 
extends T3_GenerateTargetSpdBtn
any
    btn
where
    grd1 : T3_btnEvaluated = FALSE
    grd2 : btn  $\in$  BOOL
    grd3 : T3_state = T3_EN
then
    act1 : targetSpdBtn := btn
    act2 : T3_state := T3_GENR
end

Event T3_BufferTargetSpdBtn  $\hat{=}$ 
extends T3_BufferTargetSpdBtn
any
    btn
where
    grd1 : btn = targetSpdBtn
    grd2 : T3_state = T3_GENR
then
    act1 : T3_btnTX := btn
    act2 : T3_state := T3_SEND
    act3 : CAN_msgFlg(2) := TRUE
end

Event T3_TransmitTargetSpdBtn  $\hat{=}$ 
extends T3_TransmitTargetSpdBtn
any
    canTrans
where
    grd1 : canTrans  $\in$  BOOL
    grd2 : canTrans = T3_btnTX
    grd3 : T3_state = T3_SEND
    grd4 : dom(CAN_msgFlg  $\triangleright$  {TRUE})  $\neq \emptyset$ 
    grd5 : min(dom(CAN_msgFlg  $\triangleright$  {TRUE})) = 2
then

```

```

    act1 : T1_cntRxBtn := canTrans
    act2 : T3_state := T3_EN
    act3 : T3_btnEvaluated := TRUE
    act4 : msgCounter := msgCounter + 1
    act5 : CAN_msgFlg(2) := FALSE
  end

Event T1_ReceiveEnvVal  $\hat{=}$ 
refines T1_ReceiveEnvVal
  any
    Rx2
  where
    Rx1
  where
    grd1 : Rx2 = T1_cntRxSpd
    grd2 : T1_state = T1_EN
    grd3 : T1_cntEvaluated = FALSE
    grd4 : Rx1 = T1_cntRxBtn
  then
    act1 : T1_state := T1_TS
    act2 : T1_cntLocActSpd := Rx2
    act3 : rcvFlg := TRUE
    act4 : T1_cntLocBtn := Rx1
  end

Event T1_SetTargetSpd  $\hat{=}$ 
refines T1_SetTargetSpd
  any
    spd
  where
    grd1 : spd  $\in lb \dots ub$ 
    grd2 : T1_state = T1_TS
    grd3 : T1_cntLocBtn = TRUE
  then
    act1 : cnt_targetSpd := spd
    act2 : T1_state := T1_ACCEL
  end

Event T1_SkipSetTargetSpd  $\hat{=}$ 
refines T1_SkipSetTargetSpd
  when
    grd1 : T1_state = T1_TS
    grd2 : T1_cntLocBtn = FALSE
  then
    act1 : T1_state := T1_ACCEL

```

```

    end
Event T1_GenerateAccel  $\hat{=}$ 
refines T1_GenerateAccel
    when
        then
            grd1 : T1_state = T1_ACCEL

            act1 : T1_cntLocAccel := Accel_Func(T1_cntLocActSpd  $\mapsto$  cnt_targetSpd)
            act2 : T1_state := T1_BFR
        end
Event T1_BufferAcceleration  $\hat{=}$ 
refines T1_BufferAcceleration
    any
        where
            accel
        where
            grd1 : T1_state = T1_BFR
            grd2 : accel  $\in \mathbb{N}$ 
            grd3 : accel = T1_cntLocAccel
        then
            act1 : T1_cntTxAccel := accel
            act2 : CAN_msgFlg(1) := TRUE
            act3 : T1_state := T1_SEND
        end
Event T1_TransmitAcceleration  $\hat{=}$ 
refines T1_TransmitAcceleration
    any
        where
            accel
        where
            grd1 : T1_state = T1_SEND
            grd2 : accel  $\in \mathbb{N}$ 
            grd3 : accel = T1_cntTxAccel
            grd4 : dom(CAN_msgFlg  $\triangleright$  {TRUE})  $\neq \emptyset$ 
            grd5 : min(dom(CAN_msgFlg  $\triangleright$  {TRUE})) = 1
        then
            act1 : T4_actRxAccel := accel
            act2 : T4_actEvaluated := FALSE
            act3 : CAN_msgFlg(1) := FALSE
        end
Event T4_ReceiveAcceleration  $\hat{=}$ 
extends T4_ReceiveAcceleration
    any

```

```

      accel
where

      grd1 : T4_actEvaluated = FALSE
      grd2 : T4_state = T4_EN
      grd3 : accel ∈ ℕ
      grd4 : accel = T4_actRxAccel
then

      act1 : T4_actLocAccel := accel
      act2 : T4_state := T4_ACT
end

Event T4_ActuatingAcceleration ≐
refines T4_ActuatingAcceleration

any

      accel
where

      grd1 : T4_state = T4_ACT
      grd2 : accel ∈ ℕ
      grd3 : accel = T4_actLocAccel
then

      act1 : acceleration := accel
      act2 : T1_cntEvaluated := TRUE
      act3 : T4_actEvaluated := TRUE
      act4 : T4_state := T4_EN
      act5 : T1_state := T1_EN
      act6 : msgCounter := msgCounter + 1
end

END

```

### B.2.13 Refinement 12: Machine CC12

**MACHINE** CC12

**REFINES** CC11

**SEES** C4

**VARIABLES**

*acceleration* CC0 - Functional Req  
*cnt\_targetSpd* CC0 - Functional Req  
*cycle* CC1 - cycle of tasks  
*T1\_cntEvaluated* CC1 - controller task is evaluated  
*counter* CC1 - to avoid using mod in invariants

*env\_evaluated* CC2 - environment should be evaluated at every cycle  
*speed* CC2 - value of speed at each moment of time  
*T1\_cntLocActSpd* CC3 - Local buffer of T1 (CNT Task)  
*rcvFlg* CC3 - TRUE when the CNT receive events happens. Reset at the end of every 10ms (by updateCycle2)  
*T1\_cntRxSpd* CC4 - RX buffer in task 1 (CNT)  
*T2\_sensEvaluated* CC4 - sensor must be evaluated before the cycle can progress  
*T2\_snsTX* CC5 - Transmitter buffer of sensor  
*T2\_snsLoc\_speed* CC6 - local buffer of the sensor which generates the sensed value  
*T2\_state* CC6 - To model the order between events within the task T2.  
*T3.btnTX* CC7 - The transmitter buffer of button ECU  
*T3\_state* CC7 - To define the order between the events in the BTN task (T3)  
*T3.btnEvaluated* CC7 - T3 should be evaluated before cycle can progress.  
*targetSpdBtn* CC7 - The button in the environment  
*T1\_cntRxBtn* CC7 - The Receiver buffer in T1 (CNT task)  
*T1\_cntLocBtn* CC7 - Controller's local interpretation of the button value.  
*T1\_cntLocAccel* CC8 - Local buffer in T1 which stores the decision of the CCS.  
*T1\_cntTxAccel* CC8 - Transmitter buffer in the CAN which stores the value of acceleration.  
*T4.actRxAccel* CC8 - Receiver buffer in T4 (ACT) which gets the value of acceleration from the CAN.  
*T4.actLocAccel* CC8 - Local buffer in T4 which stores the received acceleration.  
*T4.actEvaluated* CC8 - T4 should be evaluated before cycle can progress.  
*T4\_state* CC8 - The state of T4.  
*msgCounter* CC9 - Counting the number of messages that are sent within a cycle.  
*CAN\_msgFlg* CC10 - CAN flag is set by each ECU to request sending a message  
*T1\_state* CC11 - Replacing T1 ordering flags with a single variable.

## EVENTS

### Initialisation

#### begin

```

act1 : acceleration := 0
act2 : cnt_targetSpd := 0
act3 : cycle := 0
act4 : T1_cntEvaluated := TRUE
act5 : counter := 0
act6 : env_evaluated := FALSE
  
```

```

act7 : speed := -25 .. 0 × {0}
act8 : T1_cntLocActSpd := 0
act9 : rcvFlg := TRUE
act10 : T1_cntRxSpd := 0
act11 : T2_sensEvaluated := TRUE
act12 : T2_snsTX := 0
act13 : T2_snsLoc_speed := 0
act14 : T2_state := Sns_EN
act15 : T3_state := T3_EN
act16 : T3_btnTX := FALSE
act17 : T3_btnEvaluated := TRUE
act18 : targetSpdBtn := FALSE
act19 : T1_cntRxBtn := FALSE
act20 : T1_cntLocBtn := FALSE
act21 : T4_actEvaluated := TRUE
act22 : T1_cntLocAccel := 0
act23 : T1_cntTxAccel := 0
act24 : T4_actRxAccel := 0
act25 : T4_state := T4_EN
act26 : T4_actLocAccel := 0
act27 : msgCounter := 3
act28 : CAN_msgFlg := {1 ↦ FALSE, 2 ↦ FALSE, 3 ↦ FALSE}
act29 : T1_state := T1_EN
end

Event UpdateCycle1 ≐
refines UpdateCycle1
when
    grd1 : counter = 1
    grd2 : env_evaluated = TRUE
    grd3 : T2_sensEvaluated = TRUE
    then
        grd4 : T3_btnEvaluated = TRUE

        act1 : cycle := cycle + 5
        act2 : counter := 0
        act3 : env_evaluated := FALSE
        act4 : T2_sensEvaluated := FALSE
        act5 : T3_btnEvaluated := FALSE
        act6 : msgCounter := 0
    end

Event UpdateCycle2 ≐
refines UpdateCycle2

```

```

when

    grd1 :  $T1\_cntEvaluated = TRUE$ 
    grd2 :  $counter = 0$ 
    grd3 :  $env\_evaluated = TRUE$ 
    grd4 :  $T2\_sensEvaluated = TRUE$ 
    grd5 :  $T3\_btnEvaluated = TRUE$ 
then

    act1 :  $cycle := cycle + 5$ 
    act2 :  $counter := 1$ 
    act3 :  $T1\_cntEvaluated := FALSE$ 
    act4 :  $env\_evaluated := FALSE$ 
    act5 :  $rcvFlg := FALSE$ 
    act6 :  $T2\_sensEvaluated := FALSE$ 
    act7 :  $T3\_btnEvaluated := FALSE$ 
    act8 :  $msgCounter := 0$ 
end

Event  $UpdateSpd\_Env \hat{=}$ 
refines  $UpdateSpd\_Env$ 
any

     $spd$ 
     $c$ 
where

    grd1 :  $env\_evaluated = FALSE$ 
    grd2 :  $c \in \mathbb{N}$ 
    grd3 :  $c = cycle$ 
    grd4 :  $spd \in (c + 1 .. c + 5) \rightarrow (0 .. maxSpd)$ 
then

    act1 :  $speed := speed \cup spd$ 
    act2 :  $env\_evaluated := TRUE$ 
end

Event  $T2\_SenseSpeed \hat{=}$ 
refines  $T2\_SenseSpeed$ 
any

     $spd$ 
     $c$ 
where

    grd1 :  $T2\_sensEvaluated = FALSE$ 
    grd2 :  $c \in \mathbb{N}$ 
    grd3 :  $c = cycle$ 
    grd4 :  $spd \in \mathbb{N}$ 

```

```

    grd5 :  $spd \in 0 .. maxSpd$ 
    grd6 :  $spd \in speed[c - 4 .. c]$ 
    grd7 :  $T2\_state = Sns\_EN$ 
  then

    act1 :  $T2\_snsLoc\_speed := spd$ 
    act2 :  $T2\_state := Sns\_GENR$ 
  end

Event  $T2\_RequestToTransmitSpd \hat{=}$ 
refines  $T2\_RequestToTransmitSpd$ 
  any

     $spd$ 
  where

    grd1 :  $spd = T2\_snsLoc\_speed$ 
    grd2 :  $T2\_state = Sns\_GENR$ 
  then

    act1 :  $T2\_snsTX := spd$ 
    act2 :  $T2\_state := Sns\_SEND$ 
    act3 :  $CAN\_msgFlg(3) := TRUE$ 
  end

Event  $T2\_SensorTransmitSpd \hat{=}$ 
refines  $T2\_SensorTransmitSpd$ 
  any

     $canTrans$ 
  where

    grd1 :  $canTrans \in 0 .. maxSpd$ 
    grd2 :  $canTrans = T2\_snsTX$ 
    grd3 :  $T2\_state = Sns\_SEND$ 
    grd4 :  $dom(CAN\_msgFlg \triangleright \{TRUE\}) \neq \emptyset$ 
    grd5 :  $min(dom(CAN\_msgFlg \triangleright \{TRUE\})) = 3$ 
  then

    act1 :  $T1\_cntRxSpd := canTrans$ 
    act2 :  $T2\_sensEvaluated := TRUE$ 
    act3 :  $T2\_state := Sns\_EN$ 
    act4 :  $msgCounter := msgCounter + 1$ 
    act5 :  $CAN\_msgFlg(3) := FALSE$ 
  end

Event  $T3\_GenerateTargetSpdBtn \hat{=}$ 
refines  $T3\_GenerateTargetSpdBtn$ 
  any

```



```

    where  $btn$ 
    grd1 :  $T3\_btnEvaluated = FALSE$ 
    grd2 :  $btn \in BOOL$ 
    grd3 :  $T3\_state = T3\_EN$ 
  then
    act1 :  $targetSpdBtn := btn$ 
    act2 :  $T3\_state := T3\_GENR$ 
  end
Event  $T3\_BufferTargetSpdBtn \hat{=}$ 
refines  $T3\_BufferTargetSpdBtn$ 
  any
    where  $btn$ 
    grd1 :  $btn = targetSpdBtn$ 
    grd2 :  $T3\_state = T3\_GENR$ 
  then
    act1 :  $T3\_btnTX := btn$ 
    act2 :  $T3\_state := T3\_SEND$ 
    act3 :  $CAN\_msgFlg(2) := TRUE$ 
  end
Event  $T3\_TransmitTargetSpdBtn \hat{=}$ 
refines  $T3\_TransmitTargetSpdBtn$ 
  any
    where  $canTrans$ 
    grd1 :  $canTrans \in BOOL$ 
    grd2 :  $canTrans = T3\_btnTX$ 
    grd3 :  $T3\_state = T3\_SEND$ 
    grd4 :  $dom(CAN\_msgFlg \triangleright \{TRUE\}) \neq \emptyset$ 
    grd5 :  $min(dom(CAN\_msgFlg \triangleright \{TRUE\})) = 2$ 
  then
    act1 :  $T1\_cntRxBtn := canTrans$ 
    act2 :  $T3\_state := T3\_EN$ 
    act3 :  $T3\_btnEvaluated := TRUE$ 
    act4 :  $msgCounter := msgCounter + 1$ 
    act5 :  $CAN\_msgFlg(2) := FALSE$ 
  end
Event  $T1\_ReceiveEnvVal \hat{=}$ 
refines  $T1\_ReceiveEnvVal$ 

```

```

any
     $Rx2$ 
     $Rx1$ 
where
     $grd1 : Rx2 = T1\_cntRxSpd$ 
     $grd2 : T1\_state = T1\_EN$ 
     $grd3 : T1\_cntEvaluated = FALSE$ 
     $grd4 : Rx1 = T1\_cntRxBtn$ 
then
     $act1 : T1\_state := T1\_TS$ 
     $act2 : T1\_cntLocActSpd := Rx2$ 
     $act3 : rcvFlg := TRUE$ 
     $act4 : T1\_cntLocBtn := Rx1$ 
end

Event  $T1\_SetTargetSpd \hat{=}$ 
refines  $T1\_SetTargetSpd$ 
    any
         $spd$ 
    where
         $grd1 : spd \in lb \dots ub$ 
         $grd2 : T1\_state = T1\_TS$ 
         $grd3 : T1\_cntLocBtn = TRUE$ 
    then
         $act1 : cnt\_targetSpd := spd$ 
         $act2 : T1\_state := T1\_ACCEL$ 
    end

Event  $T1\_SkipSetTargetSpd \hat{=}$ 
refines  $T1\_SkipSetTargetSpd$ 
    when
         $grd1 : T1\_state = T1\_TS$ 
         $grd2 : T1\_cntLocBtn = FALSE$ 
    then
         $act1 : T1\_state := T1\_ACCEL$ 
    end

Event  $T1\_GenerateAccel \hat{=}$ 
refines  $T1\_GenerateAccel$ 
    when
         $grd1 : T1\_state = T1\_ACCEL$ 
    then
         $act1 : T1\_cntLocAccel := Accel\_Func(T1\_cntLocActSpd \mapsto cnt\_targetSpd)$ 

```

```

    end    act2 : T1_state := T1_BFR
Event T1_BufferAcceleration ≐
refines T1_BufferAcceleration
  any
    where accel
      grd1 : T1_state = T1_BFR
      grd2 : accel ∈ ℕ
      grd3 : accel = T1_cntLocAccel
    then
      act1 : T1_cntTxAccel := accel
      act2 : CAN_msgFlg(1) := TRUE
      act3 : T1_state := T1_SEND
    end
Event T1_TransmitAcceleration ≐
refines T1_TransmitAcceleration
  any
    where accel
      grd1 : T1_state = T1_SEND
      grd2 : accel ∈ ℕ
      grd3 : accel = T1_cntTxAccel
      grd4 : dom(CAN_msgFlg ▷ {TRUE}) ≠ ∅
      grd5 : min(dom(CAN_msgFlg ▷ {TRUE})) = 1
    then
      act1 : T4_actRxAccel := accel
      act2 : T4_actEvaluated := FALSE
      act3 : CAN_msgFlg(1) := FALSE
    end
Event T4_ReceiveAcceleration ≐
refines T4_ReceiveAcceleration
  any
    where accel
      grd1 : T4_actEvaluated = FALSE
      grd2 : T4_state = T4_EN
      grd3 : accel ∈ ℕ
      grd4 : accel = T4_actRxAccel
    then

```

```

        act1 : T4_actLocAccel := accel
        act2 : T4_state := T4_ACT
    end
Event   T4_ActuatingAcceleration  $\hat{=}$ 
refines T4_ActuatingAcceleration
    any
        accel
    where
        grd1 : T4_state = T4_ACT
        grd2 : accel  $\in \mathbb{N}$ 
        grd3 : accel = T4_actLocAccel
    then
        act1 : acceleration := accel
        act2 : T1_cntEvaluated := TRUE
        act3 : T4_actEvaluated := TRUE
        act4 : T4_state := T4_EN
        act5 : T1_state := T1_EN
        act6 : msgCounter := msgCounter + 1
    end
END

```

## B.3 Decomposition of SCCS

### B.3.1 Button Machine

**MACHINE** BTN

**SEES** C4

**VARIABLES**

*T3\_state* CC7 - To define the order between the events within the  
 BTN task (T3)  
*targetSpdBtn* CC7 - The button in the environment

**INVARIANTS**

CC7inv2 : *T3\_state*  $\in T3State$

CC7inv4 : *targetSpdBtn*  $\in BOOL$

**EVENTS**

**Initialisation**

**begin**

act15 : *T3\_state* := *T3\_EN*

```

        act18 : targetSpdBtn := FALSE
    end
Event T3_GenerateTargetSpdBtn  $\hat{=}$ 
    any
        btn
    where
        grd2 : btn  $\in$  BOOL
        grd3 : T3_state = T3_EN
    then
        act1 : targetSpdBtn := btn
        act2 : T3_state := T3_GENR
    end
Event T3_BufferTargetSpdBtn  $\hat{=}$ 
    any
        btn
    where
        grd1 : btn = targetSpdBtn
        grd2 : T3_state = T3_GENR
    then
        act2 : T3_state := T3_SEND
    end
Event T3_TransmitTargetSpdBtn  $\hat{=}$ 
    when
        grd3 : T3_state = T3_SEND
    then
        act2 : T3_state := T3_EN
    end
END

```

### B.3.2 Environment Machine

**MACHINE** ENV

**SEES** C4

**VARIABLES**

*speed* CC2 - value of speed at each moment of time

**INVARIANTS**

typing\_speed :  $speed \in \mathbb{P}(\mathbb{Z} \times \mathbb{Z})$

CC2inv4 :  $speed[-25..0] = \{0\}$

$CC2inv5 : ran(speed) \subseteq 0 .. maxSpd$   
**EVENTS**  
**Initialisation**  
     **begin**  
          $act7 : speed := -25 .. 0 \times \{0\}$   
     **end**  
**Event**  $UpdateSpd\_Env \hat{=}$   
     **any**  
          $spd$   
          $c$   
     **where**  
          $grd2 : c \in \mathbb{N}$   
          $grd4 : spd \in (c + 1 .. c + 5) \rightarrow (0 .. maxSpd)$   
     **then**  
          $act1 : speed := speed \cup spd$   
     **end**  
**Event**  $T2\_SenseSpeed \hat{=}$   
     **any**  
          $spd$   
          $c$   
     **where**  
          $grd2 : c \in \mathbb{N}$   
          $grd4 : spd \in \mathbb{N}$   
          $grd5 : spd \in 0 .. maxSpd$   
          $grd6 : spd \in speed[c - 4 .. c]$   
     **then**  
          $skip$   
     **end**  
**END**

### B.3.3 Sensor Machine

**MACHINE** SNS  
**SEES** C4  
**VARIABLES**

$T2\_snsLoc\_speed$  CC6 - local buffer of the sensor which generates  
     the sensed value  
 $T2\_state$  CC6 - To model the order between events within the task T2  
**INVARIANTS**

```

    CC6inv1 :  $T2\_snsLoc\_speed \in 0 \dots maxSpd$ 
    CC6inv2 :  $T2\_state \in T2State$ 
EVENTS
Initialisation
    begin
        act13 :  $T2\_snsLoc\_speed := 0$ 
        act14 :  $T2\_state := Sns\_EN$ 
    end
Event  $T2\_SenseSpeed \hat{=}$ 
    any
         $spd$ 
    where
        grd4 :  $spd \in \mathbb{N}$ 
        grd5 :  $spd \in 0 \dots maxSpd$ 
        grd7 :  $T2\_state = Sns\_EN$ 
    then
        act1 :  $T2\_snsLoc\_speed := spd$ 
        act2 :  $T2\_state := Sns\_GENR$ 
    end
Event  $T2\_RequestToTransmitSpd \hat{=}$ 
    any
         $spd$ 
    where
        grd1 :  $spd = T2\_snsLoc\_speed$ 
        grd2 :  $T2\_state = Sns\_GENR$ 
    then
        act2 :  $T2\_state := Sns\_SEND$ 
    end
Event  $T2\_SensorTransmitSpd \hat{=}$ 
    when
        grd3 :  $T2\_state = Sns\_SEND$ 
    then
        act3 :  $T2\_state := Sns\_EN$ 
    end
END

```

### B.3.4 Controller Machine

**MACHINE** CNT

**SEES** C4**VARIABLES**

*acceleration* CC0 - Functional Req  
*cnt\_targetSpd* CC0 - Functional Req  
*T1\_cntLocActSpd* CC3 - Local buffer of T1 (CNT Task)  
*T1\_cntLocBtn* CC7 - Controller's local interpretation of the button value.  
*T1\_cntLocAccel* CC8 - Local buffer in T1 which stores the decision of the CCS  
*T1\_state* CC11 - Replacing T1 ordering flags with a single variable.

**INVARIANTS**

CC0inv1 : *acceleration*  $\in \mathbb{N}$   
 CC0inv3 : *cnt\_targetSpd*  $\in 0 \dots \text{maxSpd}$   
 CC3inv1 : *T1\_cntLocActSpd*  $\in 0 \dots \text{maxSpd}$   
 CC7inv6 : *T1\_cntLocBtn*  $\in \text{BOOL}$   
 CC8inv1 : *T1\_cntLocAccel*  $\in \mathbb{N}$   
 CC11inv1 : *T1\_state*  $\in \text{T1State}$

**EVENTS****Initialisation****begin**

*act1* : *acceleration* := 0  
*act2* : *cnt\_targetSpd* := 0  
*act8* : *T1\_cntLocActSpd* := 0  
*act20* : *T1\_cntLocBtn* := FALSE  
*act22* : *T1\_cntLocAccel* := 0  
*act29* : *T1\_state* := T1\_EN

**end****Event** *T1\_ReceiveEnvVal*  $\hat{=}$ **any***Rx2**Rx1***where**

*grd1* : *Rx1*  $\in \text{BOOL}$   
*grd2* : *Rx2*  $\in 0 \dots \text{maxSpd}$   
*grd3* : *T1\_state* = T1\_EN

**then**

*act1* : *T1\_state* := T1\_TS  
*act2* : *T1\_cntLocActSpd* := *Rx2*  
*act4* : *T1\_cntLocBtn* := *Rx1*

**end**



**Event**  $T1\_SetTargetSpd \hat{=}$

**any**

*spd*

**where**

*grd1* :  $spd \in lb \dots ub$

*grd2* :  $T1\_state = T1\_TS$

*grd3* :  $T1\_cntLocBtn = TRUE$

**then**

*act1* :  $cnt\_targetSpd := spd$

*act2* :  $T1\_state := T1\_ACCEL$

**end**

**Event**  $T1\_SkipSetTargetSpd \hat{=}$

**when**

*grd1* :  $T1\_state = T1\_TS$

*grd2* :  $T1\_cntLocBtn = FALSE$

**then**

*act1* :  $T1\_state := T1\_ACCEL$

**end**

**Event**  $T1\_GenerateAccel \hat{=}$

**when**

*grd1* :  $T1\_state = T1\_ACCEL$

**then**

*act1* :  $T1\_cntLocAccel := Accel\_Func(T1\_cntLocActSpd \mapsto cnt\_targetSpd)$

*act2* :  $T1\_state := T1\_BFR$

**end**

**Event**  $T1\_BufferAcceleration \hat{=}$

**any**

*accel*

**where**

*grd1* :  $T1\_state = T1\_BFR$

*grd2* :  $accel \in \mathbb{N}$

*grd3* :  $accel = T1\_cntLocAccel$

**then**

*act3* :  $T1\_state := T1\_SEND$

**end**

**Event**  $T1\_TransmitAcceleration \hat{=}$

**when**

*grd1* :  $T1\_state = T1\_SEND$

**then**

*skip*

```

    end
Event T4_ActuatingAcceleration  $\hat{=}$ 
    any
        accel
        where
            grd2 : accel  $\in \mathbb{N}$ 
        then
            act1 : acceleration := accel
            act5 : T1_state := T1_EN
        end
    end
END

```

### B.3.5 Actuator Machine

**MACHINE** ACT

**SEES** C4

**VARIABLES**

*T4\_actLocAccel* CC8 - Local buffer in T4 which stores the received  
acceleration  
*T4\_state* CC8 - The state of T4.

**INVARIANTS**

*CC8inv11* : *T4\_state*  $\in T4State$

*CC8inv14* : *T4\_actLocAccel*  $\in \mathbb{N}$

**EVENTS**

**Initialisation**

```

    begin
        act25 : T4_state := T4_EN
        act26 : T4_actLocAccel := 0
    end
Event T4_ReceiveAcceleration  $\hat{=}$ 
    any
        accel
        where
            grd2 : T4_state = T4_EN
            grd3 : accel  $\in \mathbb{N}$ 
        then
            act1 : T4_actLocAccel := accel
            act2 : T4_state := T4_ACT
        end
    end

```

**Event**  $T4\_ActuatingAcceleration \triangleq$

**any**

*accel*  
**where**

*grd1* :  $T4\_state = T4\_ACT$

*grd2* :  $accel \in \mathbb{N}$

*grd3* :  $accel = T4\_actLocAccel$

**then**

*act4* :  $T4\_state := T4\_EN$   
**end**

**END**

### B.3.6 Cycle Machine

**MACHINE** CYCLE

**SEES** C4

**VARIABLES**

*cycle* CC1 - cycle of tasks

*T1\_cntEvaluated* CC1 - controller task is evaluated

*counter* CC1 - to avoid using mod in invariants

*env\_evaluated* CC2 - environment should be evaluated at every cycle

*rcvFlg* CC3 - TRUE when the CNT receive events happens. Reset at the end of every 10ms (by updateCycle2)

*T2\_sensEvaluated* CC4 - sensor must be evaluated before the cycle can

progress

*T3\_btnEvaluated* CC7 - T3 should be evaluated before cycle can progress

*T4\_actEvaluated* CC8 - T4 should be evaluated before cycle can progress

*msgCounter* CC9 - Counting the number of messages that are sent within a cycle

**INVARIANTS**

*CC1inv1* :  $cycle \in \mathbb{N}$

*CC1inv2* :  $T1\_cntEvaluated \in \text{BOOL}$

*CC1inv5* :  $counter \in \{0, 1\}$

*CC2inv1* :  $env\_evaluated \in \text{BOOL}$

*CC3inv3* :  $rcvFlg \in \text{BOOL}$

*CC4inv2* :  $T2\_sensEvaluated \in \text{BOOL}$

*CC7inv3* :  $T3\_btnEvaluated \in \text{BOOL}$

*CC8inv3* :  $T4\_actEvaluated \in \text{BOOL}$

$CC9inv1 : msgCounter \in \mathbb{N}$

## EVENTS

### Initialisation

**begin**

$act3 : cycle := 0$   
 $act4 : T1\_cntEvaluated := TRUE$   
 $act5 : counter := 0$   
 $act6 : env\_evaluated := FALSE$   
 $act9 : rcvFlg := TRUE$   
 $act11 : T2\_sensEvaluated := TRUE$   
 $act17 : T3\_btnEvaluated := TRUE$   
 $act21 : T4\_actEvaluated := TRUE$   
 $act27 : msgCounter := 3$

**end**

**Event**  $UpdateCycle1 \hat{=}$

**when**

$grd1 : counter = 1$   
 $grd2 : env\_evaluated = TRUE$   
 $grd3 : T2\_sensEvaluated = TRUE$   
 $grd4 : T3\_btnEvaluated = TRUE$

**then**

$act1 : cycle := cycle + 5$   
 $act2 : counter := 0$   
 $act3 : env\_evaluated := FALSE$   
 $act4 : T2\_sensEvaluated := FALSE$   
 $act5 : T3\_btnEvaluated := FALSE$   
 $act6 : msgCounter := 0$

**end**

**Event**  $UpdateCycle2 \hat{=}$

**when**

$grd1 : T1\_cntEvaluated = TRUE$   
 $grd2 : counter = 0$   
 $grd3 : env\_evaluated = TRUE$   
 $grd4 : T2\_sensEvaluated = TRUE$   
 $grd5 : T3\_btnEvaluated = TRUE$

**then**

$act1 : cycle := cycle + 5$   
 $act2 : counter := 1$   
 $act3 : T1\_cntEvaluated := FALSE$   
 $act4 : env\_evaluated := FALSE$

```

    act5 : rcvFlg := FALSE
    act6 : T2_sensEvaluated := FALSE
    act7 : T3_btnEvaluated := FALSE
    act8 : msgCounter := 0
end
Event UpdateSpd_Env  $\hat{=}$ 
  any
     $c$ 
  where
    grd1 : env_evaluated = FALSE
    grd2 :  $c \in \mathbb{N}$ 
    grd3 :  $c = cycle$ 
  then
    act2 : env_evaluated := TRUE
  end
Event T2_SenseSpeed  $\hat{=}$ 
  any
     $c$ 
  where
    grd1 : T2_sensEvaluated = FALSE
    grd2 :  $c \in \mathbb{N}$ 
    grd3 :  $c = cycle$ 
  then
    skip
  end
Event T2_SensorTransmitSpd  $\hat{=}$ 
  begin
    act2 : T2_sensEvaluated := TRUE
    act4 : msgCounter := msgCounter + 1
  end
Event T3_GenerateTargetSpdBtn  $\hat{=}$ 
  when
    grd1 : T3_btnEvaluated = FALSE
  then
    skip
  end
Event T3_TransmitTargetSpdBtn  $\hat{=}$ 
  begin
    act3 : T3_btnEvaluated := TRUE

```

```

        act4 : msgCounter := msgCounter + 1
    end
Event  T1_ReceiveEnvVal  $\hat{=}$ 
    when
        then
            grd3 : T1_cntEvaluated = FALSE
        then
            act3 : rcvFlg := TRUE
        end
Event  T1_TransmitAcceleration  $\hat{=}$ 
    begin
        end
        act2 : T4_actEvaluated := FALSE
Event  T4_ReceiveAcceleration  $\hat{=}$ 
    when
        then
            grd1 : T4_actEvaluated = FALSE
        then
            skip
        end
Event  T4_ActuatingAcceleration  $\hat{=}$ 
    begin
        act2 : T1_cntEvaluated := TRUE
        act3 : T4_actEvaluated := TRUE
        act6 : msgCounter := msgCounter + 1
    end
END

```

### B.3.7 CAN Machine

**MACHINE** CAN

**SEES** C4

**VARIABLES**

*T1\_cntRxSpd* CC4 - RX buffer in task 1 (CNT)  
*T2\_snsTX* CC5 - Transmitter buffer of sensor  
*T3\_btnTX* CC7 - The transmitter buffer of button ECU  
*T1\_cntRxBtn* CC7 - The Receiver buffer in T1 (CNT task)  
*T1\_cntTxAccel* CC8 - Transmitter buffer in the CAN which stores the  
 value of acceleration.  
*T4\_actRxAccel* CC8 - Receiver buffer in T4 (ACT) which gets the  
 value of acceleration from the CAN.

*CAN\_msgFlg* CC10 - CAN flag is set by each ECU to request sending a message

## INVARIANTS

$CC4_{inv1} : T1\_cntRxSpd \in 0 .. maxSpd$   
 $CC5_{inv2} : T2\_snsTX \in 0 .. maxSpd$   
 $CC7_{inv1} : T3\_btnTX \in BOOL$   
 $CC7_{inv5} : T1\_cntRxBtn \in BOOL$   
 $CC8_{inv2} : T1\_cntTxAccel \in \mathbb{N}$   
 $CC8_{inv10} : T4\_actRxAccel \in \mathbb{N}$   
 $CC10_{inv1} : CAN\_msgFlg \in 1 .. 3 \rightarrow BOOL$

## EVENTS

### Initialisation

**begin**

$act10 : T1\_cntRxSpd := 0$   
 $act12 : T2\_snsTX := 0$   
 $act16 : T3\_btnTX := FALSE$   
 $act19 : T1\_cntRxBtn := FALSE$   
 $act23 : T1\_cntTxAccel := 0$   
 $act24 : T4\_actRxAccel := 0$   
 $act28 : CAN\_msgFlg := \{1 \mapsto FALSE, 2 \mapsto FALSE, 3 \mapsto FALSE\}$

**end**

**Event**  $T2\_RequestToTransmitSpd \hat{=}$

**any**

**where**  $spd$

**then**  $grd1 : spd \in 0 .. maxSpd$

$act1 : T2\_snsTX := spd$   
 $act3 : CAN\_msgFlg(3) := TRUE$

**end**

**Event**  $T2\_SensorTransmitSpd \hat{=}$

**any**

**where**  $canTrans$

**then**  $grd1 : canTrans \in 0 .. maxSpd$   
 $grd2 : canTrans = T2\_snsTX$   
 $grd4 : dom(CAN\_msgFlg \triangleright \{TRUE\}) \neq \emptyset$   
 $grd5 : min(dom(CAN\_msgFlg \triangleright \{TRUE\})) = 3$

```

        act1 : T1_cntRxSpd := canTrans
        act5 : CAN_msgFlg(3) := FALSE
    end
Event T3_BufferTargetSpdBtn  $\hat{=}$ 
    any
        btn
    where
        grd1 : btn  $\in$  BOOL
    then
        act1 : T3_btnTX := btn
        act3 : CAN_msgFlg(2) := TRUE
    end
Event T3_TransmitTargetSpdBtn  $\hat{=}$ 
    any
        canTrans
    where
        grd1 : canTrans  $\in$  BOOL
        grd2 : canTrans = T3_btnTX
        grd4 : dom(CAN_msgFlg  $\triangleright$  {TRUE})  $\neq$   $\emptyset$ 
        grd5 : min(dom(CAN_msgFlg  $\triangleright$  {TRUE})) = 2
    then
        act1 : T1_cntRxBtn := canTrans
        act5 : CAN_msgFlg(2) := FALSE
    end
Event T1_ReceiveEnvVal  $\hat{=}$ 
    any
        Rx2
        Rx1
    where
        grd1 : Rx2 = T1_cntRxSpd
        grd4 : Rx1 = T1_cntRxBtn
    then
        skip
    end
Event T1_BufferAcceleration  $\hat{=}$ 
    any
        accel
    where
        grd2 : accel  $\in$   $\mathbb{N}$ 

```



```

    then
        act1 : T1_cntTxAccel := accel
        act2 : CAN_msgFlg(1) := TRUE
    end
Event T1_TransmitAcceleration  $\hat{=}$ 
    any
        accel
    where
        grd2 : accel  $\in \mathbb{N}$ 
        grd3 : accel = T1_cntTxAccel
        grd4 : dom(CAN_msgFlg  $\triangleright \{TRUE\}$ )  $\neq \emptyset$ 
        grd5 : min(dom(CAN_msgFlg  $\triangleright \{TRUE\}$ )) = 1
    then
        act1 : T4_actRxAccel := accel
        act3 : CAN_msgFlg(1) := FALSE
    end
Event T4_ReceiveAcceleration  $\hat{=}$ 
    any
        accel
    where
        grd3 : accel  $\in \mathbb{N}$ 
        grd4 : accel = T4_actRxAccel
    then
        skip
    end
END

```

# References

- [ABH<sup>+</sup>10] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12:447–466, 2010.
- [ABHV06] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for Event-B. In *ICFEM*, pages 588–605, 2006.
- [Abr96] Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 1996.
- [Abr06] Jean-Raymond Abrial. Formal methods in industry: achievements, problems, future. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 761–768, 2006.
- [Abr09] Jean-Raymond Abrial. Cruise control requirement document. Technical report, Internal report of the Deploy project, 2009.
- [Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
- [ACM05] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. Refinement and reachability in Event-B. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B*, volume 3455 of *Lecture Notes in Computer Science*, pages 222–241. Springer Berlin / Heidelberg, 2005.
- [AH07] Jean-Raymond Abrial and Stefan Hallerstede. Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.
- [ALN<sup>+</sup>91] J. Abrial, M. Lee, D. Neilson, P. Scharbach, and I. Srensen. The B-method. In Sren Prehn and Hans Toetenel, editors, *VDM '91 Formal Software Development Methods*, volume 552 of *Lecture Notes in Computer Science*, pages 398–405. Springer Berlin / Heidelberg, 1991.

- [Bac90] R. J. R. Back. Refinement calculus, part II: parallel and reactive programs. In *Proceedings on Stepwise refinement of distributed systems: models, formalisms, correctness*, REX workshop, pages 67–93, New York, NY, USA, 1990. Springer-Verlag.
- [BBFM99] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Mtor: A successful application of B in a large project. In Jeannette Wing, Jim Woodcock, and Jim Davies, editors, *FM99 Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 712–712. Springer Berlin / Heidelberg, 1999.
- [BD09] A. T. Bahill and Frank F. Dean. *Handbook of systems engineering and management*, chapter Discovering System Requirements, pages 205–266. Wiley, New York, 2009.
- [BFJ<sup>+</sup>11] Jens Bendisposto, Fabian Fritz, Michael Jastram, Michael Leuschel, and Ingo Weigelt. Developing Camille, a text editor for Rodin. *Software: Practice and Experience*, 41(2):189–198, 2011.
- [BH95] Jonathan P. Bowen and Michael G Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.
- [BH05] A. T. Bahill and S. J. Henderson. Requirements development, verification, and validation exhibited in famous failures. *Systems Engineering*, 8(1):1–14, 2005.
- [BH07] Michael Butler and Stefan Hallerstede. The Rodin formal modelling tool. *BCS-FACS Christmas 2007 Meeting-Formal Methods In Industry, London*, December 2007.
- [Bje05] Per Bjesse. What is formal verification? *SIGDA Newsl.*, 35, December 2005.
- [BKPS07] Manfred Broy, Ingolf H. Kruger, Alexander Pretschner, and Christian Salzmann. Engineering Automotive Software. *Proceedings of the IEEE*, 95(2):356–373, 2007.
- [BL05] Michael Butler and Michael Leuschel. Combining CSP and B for specification and property verification. In *Formal Methods 2005*, number LNCS 3, pages 221–236. Springer, January 2005.
- [Boe84] B.W. Boehm. Verifying and validating software requirements and design specifications. *Software, IEEE*, 1(1):75–88, January 1984.
- [Bow03] Jonathan Bowen. *Formal Specification and Documentation using Z: A Case Study Approach*. International Thomson Computer Press, 2003.

- [BP88] Barry W. Boehm and Philip N. Papaccio. Understanding and controlling software costs. *IEEE Transactions on Software Engineering*, 14(10):1462–1477, October 1988.
- [BR95] J. Bicarregui and B. Ritchie. Invariants, frames and postconditions: a comparison of the VDM and B notations. *IEEE Transactions on Software Engineering*, 21(2):79–89, February 1995.
- [But97] Michael Butler. An approach to the design of distributed systems with B AMN. In J. Bowen, M. Hinchey, and D. Till, editors, *Proc. 10th Int. Conf. of Z Users: The Z Formal Specification Notation (ZUM)*, LNCS 1212, pages 223–241. Springer-Verlag, Berlin, April 1997.
- [But06] Michael Butler. Synchronisation-based decomposition for Event-B, Deploy deliverable D19, intermediate report on methodology. <http://rodin.cs.ncl.ac.uk/deliverables/D19.pdf>, 2006. Cited 2011 January.
- [But09a] Michael Butler. Chapter 8 modelling guidelines for discrete control systems, Deploy deliverable D15, D6.1 advances in methods public document. <http://www.deploy-project.eu/pdf/D15-D6.1-Advances-in-Methodological-WPs..pdf>, 2009. Cited 2011 January.
- [But09b] Michael Butler. Decomposition structures for Event-B. In *Integrated Formal Methods iFM2009*, Springer, LNCS 5423, volume LNCS. Springer, February 2009.
- [But09c] Michael Butler. Incremental design of distributed systems with Event-B. *Engineering Methods and Tools for Software Safety and Security-Marktoberdorf Summer School 2008*, pages 131–160, 2009.
- [But09d] Michael Butler. Towards a cookbook for modelling and refinement of control problems. in Working Paper. ECS, University of Southampton, 2009.
- [Cam11] Camille editor. [http://wiki.event-b.org/index.php/Text\\_Editor](http://wiki.event-b.org/index.php/Text_Editor), 2011. Cited 2011 March.
- [CAN13] CAN in Automotive. <http://www.can-cia.org/>, 2013. Cited February 2013.
- [CGP99] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, Cambridge, MA, USA, 1999.
- [CJ05] Joey W. Coleman and Cliff B. Jones. Examples of how to determine the specifications of control systems. In *Proceedings of the Workshop on Rigorous Engineering of Fault-Tolerant Systems (REFT 2005)*, number CS-TR-915 in Technical Report Series, pages 65–73. University of Newcastle Upon Tyne, June 2005.

- [CM03] Dominique Cansell and Dominique Méry. Foundations of the B method. *Computing and Informatics*, 22, 2003.
- [Col12] John Colley. D.5.1-advance process integration i, chapter 4: Safety analysis. [http://www.advance-ict.eu/sites/www.advance-ict.eu/files/AdvanceD5.1\\_0.pdf](http://www.advance-ict.eu/sites/www.advance-ict.eu/files/AdvanceD5.1_0.pdf), September 2012. Cited 2013 March.
- [Com11] Shared event composition plug-in. [http://wiki.event-b.org/index.php/Parallel\\_Composition\\_using\\_Event-B#Shared\\_Event\\_Composition\\_Plug-in](http://wiki.event-b.org/index.php/Parallel_Composition_using_Event-B#Shared_Event_Composition_Plug-in), 2011. cited 2012 April.
- [CW96] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, 1996.
- [Das10] Sambrita Das. Formal modelling of a FADEC system using Event-B. MSc Project Dissertation, 2010.
- [DB09] Kriangsak Damchoom and Michael Butler. Applying event and machine decomposition to a flash-based filestore in Event-B. In *SBMF*, pages 134–152, 2009.
- [DBBL07] RobertI. Davis, Alan Burns, ReinderJ. Bril, and JohanJ. Lukkien. Controller area network (can) schedulability analysis: Refuted, revisited and revised. *Real-Time Systems*, 35(3):239–272, 2007.
- [Dec10] Decomposition plug-in user guide. [http://wiki.event-b.org/index.php/Decomposition\\_Plug-in\\_User\\_Guide](http://wiki.event-b.org/index.php/Decomposition_Plug-in_User_Guide), 2010. Cited 2011 March.
- [Dep09] Deploy deliverable D7 D11.1 measurement methodology guide public document. <http://www.deploy-project.eu/pdf/D7-revised-final.pdf>, 2009. Cited 2012 June.
- [Dep12] What is the position of standards regarding formal methods in my industry segment? <http://www.fm4industry.org/index.php/ExFac-HM-1>, 2012. Cited June 2012.
- [EH83] E. Allen Emerson and Joseph Y. Halpern. “sometimes” and “not never” revisited: on branching versus linear time (preliminary report). In *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL ’83, pages 127–140, New York, NY, USA, 1983. ACM.
- [ERB11] Andrew Edmunds, Abdolbaghi Rezazadeh, and Michael Butler. From event-b models to code: sensing, actuating, and the environment. In *SBMF2011*, September 2011.
- [Eve11] Event-B and the Rodin platform. <http://www.event-b.org/>, 2011. Cited 2011 March.

- [Fed05] Federal Motor Carrier Safety Administration. Concept of operations and voluntary operational requirements for lane departure warning systems (LDWS) on-board commercial motor vehicles. <http://www.fmcsa.dot.gov/facts-research/research-technology/report/lane-departure-warning-systems.htm>, 2005. Cited 2011 January.
- [FLV07] John S. Fitzgerald, Peter Gorm Larsen, and Marcel Verhoef. *Vienna Development Method*. John Wiley & Sons, Inc., 2007.
- [For10] Ford Focus lane keeping aid. [http://www.youtube.com/watch?v=1\\_mUyQmxJQY](http://www.youtube.com/watch?v=1_mUyQmxJQY), 2010. Video; Cited January 2012.
- [GGJZ00a] C.A. Gunter, E.L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications. *Software, IEEE*, 17(3):37–43, 2000.
- [GGJZ00b] C.A. Gunter, E.L. Gunter, M. Jackson, and P. Zave. A reference model for requirements and specifications-extended abstract. In *Requirements Engineering, 2000. Proceedings. 4th International Conference on*, page 189, 2000.
- [GH93] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag, New York, NY, USA, 1993.
- [Gmb91] ROBERT BOSCH GmbH. CAN Specification Version 2.0, 1991.
- [Hal90] Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7:11–19, 1990.
- [Hal06] Stefan Hallerstede. Justifications for the Event-B modelling notation. *Lecture Notes in Computer Science*, 4355:49–63, 2006.
- [Hei07] C.L. Heitmeyer. Formal methods for specifying, validating, and verifying requirements. *Journal of Universal Computer Science*, 13(5):607–618, 2007.
- [Hen80] K.L. Heninger. Specifying software requirements for complex systems: New techniques and their application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, January 1980.
- [HJJ03] Ian Hayes, Michael Jackson, and Cliff Jones. Determining the specification of a control system from that of its environment. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 154–169. Springer Verlag, 2003.
- [HJL96] Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5:231–261, July 1996.

- [HKL98] Constance Heitmeyer, James Kirby, Bruce Labaw, and Ramesh Bharadwaj. SCR\*: A toolset for specifying and analyzing software requirements. In *Computer Aided Verification*, volume 1427 of *Lecture Notes in Computer Science*, pages 526–531. Springer Berlin / Heidelberg, 1998.
- [Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12:576–580, October 1969.
- [Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice Hall International Series in Computer Science, 1985.
- [Hol04] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2004.
- [IEE90] IEEE. IEEE standard glossary of software engineering terminology, 1990.
- [Jac95] Daniel Jackson. Structuring Z specifications with views. *ACM Transactions on Software Engineering and Methodology*, 4(4):365–389, 1995.
- [Jac01] Michael Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
- [Jac04] M. Jackson. Problems, subproblems and concerns. In *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*, pages 23–26, 2004.
- [Jac05] Michael Jackson. Problem frames and software engineering. *Information and Software Technology*, 47(14):903 – 912, 2005.
- [JH03] Ralph D. Jeffords and Constance L. Heitmeyer. A strategy for efficiently verifying requirements specifications using composition and invariants. *SIGSOFT Software Engineering Notes*, 28:28–37, September 2003.
- [JHJ07] Cliff B. Jones, Ian J. Hayes, and Michael A. Jackson. Deriving specifications for systems that are connected to the physical world. In *Formal Methods and Hybrid Real-Time Systems*, pages 364–390, 2007.
- [JHLR10] Michael Jastram, Stefan Hallerstede, Michael Leuschel, and Aryllo Russo. An approach of requirements tracing in formal refinement. In *Verified Software: Theories, Tools, Experiments*, volume 6217 of *Lecture Notes in Computer Science*, pages 97–111. Springer Berlin / Heidelberg, 2010.
- [Jin10] Jin-Woo Lee. Model based predictive control for automated lane centering/changing control systems. <http://www.faqs.org/patents/app/20100228420>, September 2010. US patent number: 20100228420; Cited 2012 January.

- [JJ96] Daniel Jackson and Michael Jackson. Problem decomposition for reuse. *Software Engineering Journal*, 11(1):19–30, 1996.
- [Jon90] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.
- [JTN05] Karl Henrik Johansson, Martin Trngren, and Lars Nielsen. Vehicle applications of controller area network. In *Handbook of Networked and Embedded Control Systems*, Control Engineering, pages 741–765. 2005.
- [Lam80] Leslie Lamport. “sometime” is sometimes “not never”: on the temporal logic of programs. In *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL ’80, pages 174–185, New York, NY, USA, 1980. ACM.
- [Lam83] Leslie Lamport. What good is temporal logic? In *IFIP Congress’83*, pages 657–668, 1983.
- [LB03] Michael Leuschel and Michael Butler. ProB: A model checker for B. In *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer Berlin / Heidelberg, 2003.
- [LCY+97] X. Liu, Z. Chen, H. Yang, H. Zedan, and William C. Chu. A design framework for system re-engineering. In *Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference*, APSEC, Washington, DC, USA, 1997. IEEE Computer Society.
- [LFM04] M Lawford, P Froebel, and G Moum. Application of tabular methods to the specification and verification of a nuclear reactor shutdown system. *Formal Methods in System Design*, 2004.
- [LH95] K Lano and H Haughton. Formal development in B abstract machine notation. *Information and Software Technology*, 37(5-6):303–316, 1995.
- [LIN13] Local Interconnect Network. <http://www.lin-subbus.org/>, 2013. Cited February 2013.
- [LL09] Jin-Woo Lee and Bakhtiar Litkouhi. Control and validation of automated lane centering and lane changing maneuver. *ASME Conference Proceedings*, pages 411–417, 2009.
- [LYZ97] X. Liu, H. Yang, and H. Zedan. Formal methods for the re-engineering of computing systems: a comparison. In *Computer Software and Applications Conference, 1997. COMPSAC ’97. Proceedings, The Twenty-First Annual International*, pages 409–414, August 1997.
- [Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.



- [Min97] Ministry of Defence. Defence Standard 00-55: Requirements for Safety related Software in Defence Equipment, Part2/Issue2. [http://www.software-supportability.org/Docs/00-55\\_Part\\_2.pdf](http://www.software-supportability.org/Docs/00-55_Part_2.pdf), August 1997. Cited June 2012.
- [MP92] Zohar Manna and Amir Pnueli. *The temporal logic of reactive and concurrent systems*. Springer-Verlag, New York, NY, USA, 1992.
- [MS11] Dominique Méry and Neeraj Kumar Singh. Automatic code generation from Event-B models. In *SoICT 2011*, Hanoi, Viet Nam, October 2011. Hanoi University, ACM ICPS.
- [MT01] S.P. Miller and A.C. Tribble. Extending the four-variable model to bridge the system-software gap. In *Digital Avionics Systems, 2001. DASC. The 20th Conference*, volume 1, 2001.
- [PD09] C. Ponsard and M. Delehay. Towards a model-driven approach for mapping requirements on aadl architectures. In *14th IEEE International Conference on Engineering of Complex Computer Systems*, pages 353–358, June 2009.
- [PD11] Christophe Ponsard and Xavier Devroey. Generating high-level event-b system models from kaos requirements models. 2011.
- [PM95] David Lorge Parnas and Jan Madey. Functional documents for computer systems. *Science of Computer Programming*, 25(1):41–61, 1995.
- [PMGB05] A. Polychronopoulos, N. Mhler, S. Ghosh, and A. Beutner. System design of a situation adaptive lane keeping support system, the safelane system. In *Advanced Microsystems for Automotive Applications 2005*, pages 169–183. Springer Berlin / Heidelberg, 2005.
- [PP95] A. Palnitkar and D. Parham. Cycle simulation techniques. In *Verilog HDL Conference, 1995. Proceedings., 1995 IEEE International*, pages 2–8, 1995.
- [Pre10] Roger Pressman. *Software Engineering: A Practitioner’s Approach*. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010.
- [Pro11] The ProB animator and model checker. [http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main\\_Page](http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page), 2011. Cited 2011 March.
- [Pro12] ProR editor. <http://wiki.event-b.org/index.php/ProR>, 2012. Cited 2012 June.
- [PS12] Michael Poppleton and Gintautas Sulskus. Patterns for modelling fault tolerant systems in Event-B. Rodin User and Developer Workshop, 27-29 February 2012, 2012. Cited February 2012.

- [PTG97] S. Patwardhan, Han-Shue Tan, and J. Guldner. A general framework for automatic steering control: system analysis. In *American Control Conference, 1997. Proceedings of the 1997*, volume 3, pages 1598–1602 vol.3, June 1997.
- [Rei85] Wolfgang Reisig. *Petri Nets: an introduction*. Springer-Verlag, New York, NY, USA, 1985.
- [RME00] R. Risack, N. Mohler, and W. Enkelmann. A video-based lane keeping assistant. In *Intelligent Vehicles Symposium, 2000. IV 2000. Proceedings of the IEEE*, pages 356–361, 2000.
- [SB06] Colin Snook and Michael Butler. UML-B: Formal modeling and design aided by UML. *ACM Transactions on Software Engineering and Methodology*, 15:92–122, January 2006.
- [SB10] Renato Silva and Michael Butler. Shared event composition/decomposition in Event-B. In *FMCO Formal Methods for Components and Objects*, November 2010.
- [SB11] Mohammad Reza Sarshogh and Michael Butler. Specification and refinement of discrete timing properties in Event-B. In *AVoCS 2011*, September 2011.
- [Sch02] Steve Schneider. *The B-Method: An Introduction*. Palgrave, 2002.
- [Som07] Ian Sommerville. *Software engineering (8. ed.)*. Addison-Wesley, 2007.
- [SPHB10] Renato Silva, Carine Pascal, T. Son Hoang, and Michael Butler. Decomposition tool for Event-B. In *Workshop on Tool Building in Formal Methods-ABZ Conference*, January 2010.
- [SPHB11] Renato Silva, Carine Pascal, Thai Son Hoang, and Michael Butler. Decomposition tool for Event-B. *Software: Practice and Experience*, 41(2):199–208, February 2011.
- [Spi92] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
- [Val98] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the volumes are based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, 1998. Springer-Verlag.
- [vL08] Axel van Lamsweerde. Requirements engineering: from craft to discipline. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT ’08/FSE-16, pages 238–249, 2008.

- [vL09] Axel van Lamsweerde. *Requirements Engineering-From System Goals to UML Models to Software Specifications*. Wiley, 2009.
- [WC02] Jim Woodcock and Ana Cavalcanti. The semantics of Circus. In *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184–203. Springer Berlin / Heidelberg, 2002.
- [Win90] Jeannette M. Wing. A specifier’s introduction to formal methods. *IEEE Computer*, 23(9):8–24, 1990.
- [WL03] Alan Wassyng and Mark Lawford. Lessons learned from a successful implementation of formal methods in an industrial project. *FME 2003: Formal Methods*, pages 133–153, 2003.
- [YB11] Sanaz Yeganehfar and Michael Butler. Structuring functional requirements of control systems to facilitate refinement-based formalisation. In *Proceedings of the 11th Workshop on Automated Verification of Critical Systems*, volume 46, 2011.
- [YB12] Sanaz Yeganehfar and Michael Butler. Control systems: phenomena and structuring functional requirement documents. In *17th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2012)*., pages 39–48, April 2012.
- [YB13] Sanaz Yeganehfar and Michael Butler. Problem decomposition and sub-model reconciliation of control systems in Event-B. In *IEEE International Workshop on Formal Methods Integration*, August 2013.
- [YBR10] Sanaz Yeganehfar, Michael Butler, and Abdolbaghi Rezazadeh. Evaluation of a guideline by formal modelling of cruise control system in Event-B. In *Proceedings of the Second NASA Formal Methods Symposium (NFM 2010)*, NASA/CP-2010-216215, pages 182–191. NASA, April 2010.
- [ZJ93] Pamela Zave and Michael Jackson. Conjunction as composition. *ACM Transactions on Software Engineering and Methodology*, 2(4):379–411, 1993.