EUROPEAN AVIATION SAFETY AGENCY
AGENCE EUROPÉENNE DE LA SÉCURITÉ AÉRIENNE
EUROPÄISCHE AGENTUR FÜR FLUGSICHERHEIT

Research Project EASA.2010/3

SOMCA - Safety implications in performing SOftware Model Coverage Analysis

easa.europa.eu

Disclaimer

This study has been carried out for the European Aviation Safety Agency by an external organization and expresses the opinion of the organization undertaking the study. It is provided for information purposes only and the views expressed in the study have not been adopted, endorsed or in any way approved by the European Aviation Safety Agency. Consequently it should not be relied upon as a statement, as any form of warranty, representation, undertaking, contractual, or other commitment binding in law upon the European Aviation Safety Agency.

FINAL STUDY REPORT

SOMCA

Prepared by: Amaya Atencia Yépez
Joaquín Autrán Cerqueira
Ramón Jurado Sagarminaga
Santiago Urueña Pascual

Approved by: Joaquín Autrán Cerqueira

Authorized by: Joaquín Autrán Cerqueira

Internal Code: GMVAD 21276/11 V5/11
Version: 1.E
Date: 14/10/2011

GMV
Isaac Newton 11, PTM Tres Cantos, 28760 Madrid
Tel. +34 918072100, Fax. +34 918072199
www.gmv.com

THIS PAGE IS INTENTIONALLY LEFT BLANK

DOCUMENT STATUS SHEET

Version	Date	Pages	Changes
1.A	29/07/2011	149	First issue for EASA comments.
1.B	01/09/2011	181	Several changes included as suggested by EASA and ESTEREL: ■ Minor typos and terms changed among the entire document. ■ Applicability of Criteria depending on the SW-DALs has been updated. ■ Removed some comments about further studies and Statement of Compliance. ■ Added missing references. ■ Added “Object Oriented Techniques” as a methodological solution in section 5. ■ Introduction of section 7 has been changed. ■ Changed SOMCA Prerequisite 9 in section 9.4. ■ Added clarification in SOMCA Prerequisite 20. ■ Minor modifications and some clarification on SOMCA Criteria 7, 11 and 13. ■ Added two new UF examples: 9.6.1.8: Water Heater Control and 13.4.3: Text Line Editor Modes. ■ Added ESTEREL Examples of Unintended Functions in section 13.6. ■ Added Annex C (Section 14) with the information gathered related with model formalism (Section 6 in IR1). ■ Changed objective of Task 5 in section 8.1.5. ■ Changed the term “Structural Coverage” by “Structural Code Coverage” to avoid misunderstandings. ■ Added clarification in SOMCA Prerequisite 20. ■ Corrected an error in the applicability of coverage to block formalisms in section 9.5.1.1 ■ Updated the Assessment results in section 9.6.2 with the new examples and corrected some minor errors. ■ Updated the justification for unused criteria in section 9.6.2.3. ■ Deleted al the Future Study introduction from section 9.7. ■ Added clarification in table 9-2. ■ Corrected minor errors in the Executive Summary (Section 4). ■ Corrected some errors in the summary tables in section 9.6.2.2. ■ Executive summary (section 4) updated according EASA Comments. ■ Conclusions (section 10) updated according to EASA Comments.
1.C	02/09/2011	181	Third issue implementing EASA comments. ■ Section 10.1, “SOMCA criteria definition” Items 4 and 5 were erroneous and should not be part of the FSR. ■ Section 10.1, “SOMCA assessment” numeration of items is not correct. ■ Section 10.1, point 1 under “Relation between Structural Code Coverage with Model Coverage” removed.
1.D	07/09/2011	180	Fourth issue adding EASA covers. Fixed a few minor typos.
1.E	14/10/2011	180	Fifth issue removing GMV copyright.

TABLE OF CONTENTS

1. INTRODUCTION 10
1.1. PURPOSE 10
1.2. SCOPE 10
1.3. ACRONYMS 10
2. REFERENCES 12
2.1. APPLICABLE DOCUMENTS 12
2.2. REFERENCE DOCUMENTS 12
3. ACKNOWLEDGEMENTS 14
4. EXECUTIVE SUMMARY 15
5. BACKGROUND AND MOTIVATION 19
5.1. AIRWORTHINESS CERTIFICATION 19
5.2. MBD FOR AIRBORNE SOFTWARE 19
5.3. MODEL COVERAGE ANALYSIS 20
6. OBJECTIVES 22
7. METHODOLOGY AND APPROACH 23
7.1. INTRODUCTION 23
7.2. DESCRIPTION OF THE APPROACH 23
7.2.1. ANALYSIS OF UNINTENDED FUNCTIONS 24
7.2.2. DEFINITION OF SOMCA CRITERIA 26
7.2.3. CHALLENGE SOMCA CRITERIA 26
7.2.4. ELABORATION OF CONCLUSIONS 27
8. IMPLEMENTATION 28
8.1. DETAILED DESCRIPTION OF TASKS 30
8.1.1. TASK 1: MODEL FORMALISM 30
8.1.2. TASK 2: MODEL COVERAGE CRITERIA 31
8.1.3. TASK 3: MCA METHODS AND TOOLS 31
8.1.4. TASK 4: STUDY CASE 32
8.1.5. TASK 5: CODE COVERAGE AND MODEL COVERAGE 33
9. RESULTS AND OUTCOMES 34
9.1. MODELLING FORMALISMS 34
9.1.1. BLOCK DIAGRAMS 35
9.1.2. STATE DIAGRAMS 35
9.1.3. CONCLUSIONS ABOUT MODELLING FORMALISMS 35
9.2. ANALYSIS OF UNINTENDED FUNCTIONS 36
9.2.1. WHAT IS AN UNINTENDED FUNCTION? 36
9.2.2. SOURCES OF UNINTENDED FUNCTIONS 37
9.2.3. TOWARDS A TAXONOMY FOR UNINTENDED FUNCTIONS 41
9.3. MODEL COVERAGE ANALYSIS 44
9.3.1. ANALYSIS OF COVERAGE CRITERIA APPLIED TO BLOCK DIAGRAMS 45
9.3.2. ANALYSIS OF COVERAGE CRITERIA APPLIED TO STATE DIAGRAMS 46
9.4. SOMCA MODEL COVERAGE 49

9.4.1. SOMCA PREREQUISITES 49
9.4.2. SOMCA RECOMMENDATIONS 53
9.4.3. SOMCA CRITERIA 54
9.4.4. CRITERION APPLICABILITY FOR EACH SOFTWARE ASSURANCE LEVEL 68
9.5. MCA SUPPORT IN COMERCIAL TOOLS 69
9.5.1. SCADE MTC 69
9.5.2. SIMULINK VALIDATION AND VERIFICATION TOOLBOX 70
9.5.3. SUPPORT OF SOMCA CRITERIA BY COMERCIAL TOOLS 73
9.6. ASSESSMENT OF SOMCA MCA CRITERIA 74
9.6.1. ILLUSTRATIVE UF EXAMPLES 74
9.6.2. ASSESSMENT RESULTS 86
9.7. MODEL COVERAGE VS. STRUCTURAL CODE COVERAGE 97
9.8. CERTIFICATION MEMORANDUM AMENDMENT 98
9.9. POSSIBLE CRITERIA EVOLUTIONS 102
9.9.1. FORMAL MODEL VERIFICATION 102
9.9.2. COVERAGE OF NUMERIC FLOWS 102
10. CONCLUSIONS 104
10.1. PARTIAL CONCLUSIONS 106
10.2. FINAL CONCLUSION 108
11. KEY TERMS 110
12. APPENDIX A: STRUCTURAL CODE COVERAGE 112
13. APPENDIX B: DETAILED UF EXAMPLES 114
13.1. SIMULINK SMOOTHING MODEL 114
13.1.1. APPROACH 114
13.1.2. SMOOTHING MODEL UNINTENDED FUNCTIONS 115
13.1.3. RESULTS OBTAINED IN A PSEUDO-OPERATIONAL ENVIRONMENT 123
13.2. SCADE SMOOTHING MODEL UF ANALYSIS 126
13.2.1. GAP NOT DETECTED 126
13.2.2. CYCLESLIP NOT DETECTED 127
13.3. CRUISE CONTROL STATE MACHINE UF ANALYSIS 127
13.3.1. INTRODUCTION AND APPROACH 127
13.3.2. UNINTENDED FUNCTIONS PRESENT IN THE MODEL 127
13.3.3. UNINTENDED FUNCTIONS TO BE ARTIFICIALLY INJECTED IN THE MODEL 128
13.4. STATEFLOW EXAMPLES 128
13.4.1. SERIAL PORT CONTROLLER 129
13.4.2. PETITION QUEUING IN TRANSACTION MANAGER 130
13.4.3. TEXT LINE EDITOR MODES 131
13.5. THEORETICAL UF EXAMPLES 132
13.5.1. UNSTABLE SYSTEM – TRANSFER FUNCTION 132
13.5.2. UNSYNCHRONIZED NAMING OF INPUT / OUTPUT PORTS 135
13.5.3. IMPLEMENTATION ERROR IN MAX BLOCK 136
13.5.4. FILTER BLOCK INPUTS 139
13.5.5. MEMORY-LATCH ERROR IN A LIMITER 140
13.5.6. COMMANDING FOR MULTIPLE BLOCKS 141
13.5.7. SYSTEM TIME ADJUST DELAY 142
13.5.8. CACHE CONFLICTS IN CONCURRENT BLOCKS 143
13.5.9. SATELLITE CLOCK STABILITY CHECK 144

13.5.10. DESCOPED FUNCTIONALITY INTERFERENCE: REUSE OF MESSAGE DECODER 144
13.5.11. DEBUG ELEMENTS PRESENT IN OPERATIONAL VERSIONS 145
13.5.12. HIGHLY COUPLED ALGORITHMIC ARCHITECTURE 147
13.5.13. EVENT LOG LENGTH 148
13.5.14. INTERNAL PRECISION LOSS IN ARITHMETIC BLOCK 149
13.5.15. PROTOCOL CONTROL COUPLING 150
13.5.16. ERROR IN MODEL INPUTS 151
13.6. ADDITIONAL UNINTENDED FUNCTIONS PROVIDED BY ESTEREL 152
13.6.1. DIVISION BY ZERO COVERAGE 152
13.6.2. MODEL INCONSISTENCY 155
13.6.3. UNINTENDED ACTIVATION 156
14. ANNEX C: ANALYSIS OF MODEL FORMALISMS 160
14.1. INTRODUCTION TO MODEL FORMALISMS 160
14.2. FORMALIZED REQUIREMENTS 161
14.2.1. CONCLUSIONS 163
14.3. FORMALIZED DESIGNS 163
14.3.1. BLOCK DIAGRAMS 164
14.3.2. STATE DIAGRAMS 170
14.4. CONCLUSIONS 175

LIST OF TABLES AND FIGURES

Table 1-1: List of Acronyms 11
Table 2-1: Applicable Documents 12
Table 2-2: Reference Documents 13
Table 8-1: Progress Meeting schedule. 29
Table 9-1: Activation example middle logic path. 60
Table 9-2: SOMCA MCA criteria 68
Table 9-3: Support of SOMCA Criteria by Simulink and SCADE 74
Table 9-4: SOMCA Results on Climate Control Example (1) 80
Table 9-5: SOMCA Results on Climate Control Example (2) 81
Table 9-6: SOMCA Results on Climate Control Example (3) 82
Table 9-7: Classification of Theoretical UF examples based on UF Type 87
Table 9-8: SOMCA Criteria and Theoretical Example UFs associated to them 89
Table 9-9: Summary of UF identification for Theoretical Examples 89
Table 9-10: Real Examples UFs ordered by type. 91
Table 9-11: SOMCA Criteria and Real Example UFs associated to them 92
Table 9-12: Summary of UF identification for real world examples 93
Table 9-13: SOMCA Prerequisites and Real Example UFs associated to them 94
Table 9-14: Summary of Real-world examples assessment 95
Table 9-15: Summary of SOMCA Criteria assessment 95
Table 9-16: Summary of SOMCA Criteria effectiveness 96
Table 9-17: List of Cert Memo changes to integrate the SOMCA criteria. 101
Table 10-1: SOMCA Criteria applicability depending of SW-DAL 105
Table 10-2: Summary of SOMCA Criteria assessment 107
Table 12-1: ED-12B Structural Code Coverage criteria 112
Table 13-1: SOMCA Results on Transaction Manager 131
Table 13-2: Inputs, outputs and error in the Moving Average Block 149
Table 14-1: Summary of model formalisms 178

Figure 4-1 SOMCA Approach summary 16
Figure 5-1: Model-based development workflow 20
Figure 5-2: UF introduction at model level 21
Figure 7-1: SOMCA study approach 24
Figure 8-1: SOMCA Work breakdown structure 28
Figure 8-2: SOMCA project technical deliverables 29
Figure 9-1: SOMCA Taxonomy of Unintended Functions 42
Figure 9-2: Example model showing different Block Diagram elements 46
Figure 9-3: Example model showing different State Diagram elements 48
Figure 9-4: Range Coverage criterion example. 55
Figure 9-5: Functionality Coverage example. 57
Figure 9-6: Modified input coverage example. 57
Figure 9-7: Activation coverage example 58
Figure 9-8: Local Decision Coverage example. 59
Figure 9-9: Local Decision Coverage example (selector). 59

Figure 9-10: Logic Path Coverage criterion example 60
Figure 9-11: Parent state coverage example. 61
Figure 9-12: State History coverage example. 62
Figure 9-13: State machine with extra DebugMode transition. 63
Figure 9-14: State machine with extra DebugMode decision. 63
Figure 9-15: State machine with extra DebugMode condition. 64
Figure 9-16: Event coverage. 65
Figure 9-17: Activating event coverage. 66
Figure 9-18: Loop Criteria Example 67
Figure 9-19: Masking MC/DC in SCADE (courtesy of Esterel Technologies) 69
Figure 9-20: The three AND gates can be converted to a single AND block 71
Figure 9-21: The coverage analysis would be local to each block 72
Figure 9-22: 'Test objective' annotation to analyse the output of a relational block 72
Figure 9-23: Software Shutdown Example (1) 75
Figure 9-24: Software Shutdown Example (2) 76
Figure 9-25: Software Shutdown Example (3) 77
Figure 9-26: Gauss-Markov Filter 77
Figure 9-27: Climate System State Machine (1st Version) 80
Figure 9-28: Climate System State Machine (2nd version) 81
Figure 9-29: Climate System State Machine (3nd version) 82
Figure 9-30: Stopwatch in Stateflow (Hamon and Rushby, 2007) 84
Figure 9-31: Water Heater Control 86
Figure 9-32: Missing Abs block (previous to UF injection) 102
Figure 13-1: Gauss-Markov buffer length check injected UF 117
Figure 13-2: Cycleslip process injected UF 118
Figure 13-3: Missing Abs block (previous to UF injection) 119
Figure 13-4: Missing Abs block (after UF injection) 119
Figure 13-5: Simulink Coverage Error 122
Figure 13-6: Serial Controller State Machine 129
Figure 13-7: Transaction Manager State Machine 130
Figure 13-8: Example of text line editor 132
Figure 13-9: Definition of Transfer Function 133
Figure 13-10: Model containing a Function Transfer block 133
Figure 13-11: Output corresponding to the Function Transfer excited with a step signal. 133
Figure 13-12: Response of the system represented by Equation 1 to a square signal. 134
Figure 13-13: Root Locus diagram of the system represented in Equation 1. 134
Figure 13-14: Example of incoherent naming in Simulink 135
Figure 13-15: Implementation of uf_max block 136
Figure 13-16: Result of uf_max block when applied a sine function. 137
Figure 13-17: Correct implementation of uf_max block 139
Figure 13-18: Smooth filter block 139
Figure 13-19: Limiter block diagram 140
Figure 13-20: Dual Chain Processor 141
Figure 13-21: LogRotate inputs and outputs. Notice the DebugMode port. 146
Figure 13-22: LogRotate state machine with extra DebugMode transition. 146
Figure 13-23: Modified LogRotate state machine with extra DebugMode condition. 147
Figure 13-24: Example of Receivers Status Assessment 148

Figure 13-25: Example of protocol coupling 150
Figure 13-26: Robust Division 153
Figure 13-27: Uses of Robust Division 153
Figure 13-28:: Filter Gap Analysis and Reset 156
Figure 13-29:: Enclosing Branch Activation Logic 156
Figure 13-30: Actual versus Intended Inhibition Logic 157
Figure 14-1: SpecTRM model 162
Figure 14-2: Block Diagram Example 165
Figure 14-3: LabVIEW 8.2 167
Figure 14-4: Simulink 168
Figure 14-5: State diagram example 171
Figure 14-6: SCADE Safe State Machine 174

1. INTRODUCTION

1.1. PURPOSE

This document is the Final Study Report (FSR) of the SOMCA project; it constitutes, along with the two Interim Reports, the technical output of the SOMCA Study. The SOMCA project (Safety Implications in Performing Software Model Coverage Analysis) is funded by EASA and the main objective is to establish, in the context of the *Airborne Systems and Equipment Certification*, whether coverage analysis performed at model level can ensure an appropriate level of confidence for the detection of unspecified behaviours, with the aim of characterising the occurrence and impact of all Unintended Functions and to be able to repair them at specification level.

As explained later on in this document, the project was organised into a set of tasks whose intermediate outputs were compiled in the First Interim Report [RD.1] and Second Interim Report [RD.2]. The First Interim Report is focused on the preliminary analysis necessary before defining the SOMCA Model Coverage Criteria, while the Second Interim Report contains the definition of the SOMCA criteria itself and the assessment of the criteria. Finally, the FSR intends to provide a compilation of the main findings and conclusions of the project.

1.2. SCOPE

The FSR is aimed at providing a sound description of the SOMCA study and main results, covering the study background and motivation, organisation of the work and description of activities, the methodology used during the analysis, and finally, main conclusions and recommendations. To cover this scope the report has been organised according to the following structure:

- Introduction and References, describing the purpose and scope of the document, acronyms list and the list of referenced documents.
- Acknowledgements, acknowledgments of main contributors to this study.
- Executive summary, high-level description of the SOMCA study and main findings.
- Background and motivation, describing the context and the motivation of the study.
- Objectives, providing a list of the initial objectives to be addressed in the project.
- Methodology and Approach section describes the approach followed to perform the analysis.
- Implementation section gives a view of how the work has been structured and the methodology implemented.
- Results and outcomes, provides a description of the main technical outputs and conclusions of the different activities.
- Conclusions section is aimed at compiling main conclusions and recommendations of the project.
- There is a set of Annexes aimed at providing information that supports the main sections.

1.3. ACRONYMS

Table 1-1 contains the acronyms used through the document. A list of key terms used in the project is provided in section 11.

Acronym / Term	Definition
AD	Applicable Document (see section 2.1)
AMC	Acceptable Means of Compliance (EASA)
BS	British Standards
CC	Condition Coverage
CRI	Certification Review Item (EASA)
CS	Certification Specifications
DAL	Design Assurance Level

Acronym / Term	Definition
DC	Decision Coverage
DO	Defense Order
DOI	Digital Object Identifier
DOT/FAA	Department of Transportation Federal Aviation Administration
EASA	European Aviation Safety Agency
ED	(EASA) Executive Director
FAA	(U.S.) Federal Aviation Administration
FSR	Final Study Report
HiLR	High-Level Requirements
HW	Hardware
IR	Interim Report
LLR	Low-Level Requirements
MBD	Model-Based Development
MC	Model Coverage
MCA	Model Coverage Analysis
MC/DC	Modified Condition / Decision Coverage
NASA/TM	(U.S.) National Aeronautics and Space Administration Technical Memorandum
OBC	Object Branch Coverage
OIC	Object Instruction Coverage
RD	Reference Document (see section 2.2)
RSML	Requirements State Machine Language
RSML^{-e}	RSML without Events
SAO	*Spécification Assistée par Ordinateur* (Computer-Assisted Specification)
SAL	Symbolic Analysis Laboratory
SCR	Software Cost Reduction
SOMCA	Safety Implications in Performing Software Model Coverage Analysis
SW	Software
SWCEH	Software & Complex Electronic Hardware
SysML	Systems Modelling Language
UF	Unintended Function
U.S.	United States
V&V	Validation & Verification
WP	Work Package

Table 1-1: List of Acronyms

2. REFERENCES

2.1. APPLICABLE DOCUMENTS

Applicable documents are those referenced in the Contract or approved by the Approval Authority. They are referenced in this document in the form [AD.X]:

Ref.	Title	Code	Ver.	Date
[AD.1]	Contract: SOMCA — Safety implications in performing SOftware Model Coverage Analysis	EASA.2010.C11	-	23/12/2010
[AD.2]	Specifications attached to the Invitation to Tender	EASA.2010.OP.18	-	24/09/2010
[AD.3]	Technical and Management Proposal "SAFETY IMPLICATIONS IN PERFORMING SOFTWARE MODEL COVERAGE ANALYSIS"	GMVAD 10543/10 V1/10	-	29/10/2010

Table 2-1: Applicable Documents

2.2. REFERENCE DOCUMENTS

Reference documents are those not applicable and referenced within this document. They are referenced in this document in the form [RD.X]:

Ref.	Title	Code	Ver.	Date
[RD.1]	Interim Report 1: Model Formalisms	SOMCA-GMV-IR-001	1.B	01/04/2011
[RD.2]	Second Interim Report: Model Coverage Analysis	SOMCA-GMV-IR-002	1.C	29/07/2011
[RD.3]	EASA (2010). *EASA Certification Specifications for Large Aeroplanes,* ED Decision 2003/02/RM CS-25, Amendment 11.	EASA CS-25	11	05/08/2010
[RD.4]	EASA (2003). Documents for "Recognition of EUROCAE ED-12B / RTCA DO-178B", *General Acceptable Means of Compliance for Airworthiness of Products, Parts and Appliances — AMC-20*.	EASA AMC 20-115B	-	2003
[RD.5]	EUROCAE (1999). *Software Considerations in Airborne Systems and Equipment Certification*	ED-12B / DO-178B	-	1999
[RD.6]	RTCA (2001). *Final report for clarification of DO-178B: Software Considerations in Airborne Systems and Equipment Certification*	DO-248B	-	2001
[RD.7]	EUROCAE (1996). *Guidelines for Development of Civil Aircraft and Systems*.	ED-79 / ARP 4754	-	1996
[RD.8]	EASA (2011). *EASA Certification Memorandum*.	SWCEH–002 Issue: 01	01	09/02/2011
[RD.9]	David L. Lempia, Steven P. Miller (2009). *Requirements Engineering Management Findings Report*, U.S. Department of Transportation Federal Aviation Administration.	DOT/FAA/AR-08/34	-	2009
[RD.10]	Daniel Buck and Andreas Rau (2001). "On modelling guidelines: Flowchart patterns for Stateflow", *Softwaretechnik-Trends*, vol.21 no.2.	-	-	2001
[RD.11]	Pontus Boström and Lionel Morel (2007). *Formal Definition of a Mode-Automata Like Architecture in Simulink/Stateflow*. Turku Centre for Computer Science technical report.	TUCS Technical Report 830	-	2007
[RD.12]	MathWorks Automotive Advisory Board (MAAB). *Control Algorithm Modeling Guidelines Using Matlab, Simulink, and Stateflow*.	-	Ver.2	2007
[RD.13]	Nancy G. Leveson (2000), "System safety in computer-controlled automotive systems", *SAE transactions*, vol.109, no 7.	DOI:10.4271/2000-01-1048	-	2000
[RD.14]	Grégoire Hamon, John Rushby (2007), "An Operational Semantics for Stateflow". In *International Journal on Software Tools for Technology Transfer* (STTT), vol.9, n.5-6, pp. 447-456.	-	-	2007
[RD.15]	Mike Anthony, Jon Friedman (2008). *Model-Based Design for Large Safety-Critical Systems: A Discussion Regarding Model Architecture*, The MathWorks technical report.	-	-	2008
[RD.16]	André Baresel, Mirko Conrad, Sadegh Sadeghipour, Joachim Wegener (2003). "The interplay between model coverage and code coverage", *Proceedings of EuroCAST 2003*.	-	-	2003
[RD.17]	Raimund Kirner (2009). "Towards Preserving Model Coverage and Structural Code Coverage", *EURASIP Journal on Embedded Systems*,	DOI:10.1155/2009/127945	-	2009

Ref.	Title	Code	Ver.	Date
[RD.18]	Abdesselam Lakehal, Ioannis Parissis (2005). "Structural Test Coverage Criteria for Lustre Programs", *Proceedings of the 10th international workshop on Formal methods for industrial critical systems* (FMICS '05)	-	-	2005
[RD.19]	U.S. Federal Aviation Administration (2001). *An Investigation of Three Forms of the Modified Condition Decision Coverage (MCDC) Criterion*. Final Report	DOT/FAA/AR-01/18	Final	April 2001
[RD.20]	Kelly J. Hayhurst, Dan S. Veerhusen, John J. Chilenski, and Leanna K. Rierson (2001). *A Practical Tutorial on Modified Condition/Decision Coverage*, NASA technical report.	NASA/TM-2001-210876	-	May 2001
[RD.21]	William Aldrich (2001). *Coverage Analysis for Model Based Design Tools*, The MathWorks technical report.	-	-	2001
[RD.22]	British Computer Society Specialist Interest Group in Software Testing — BCS SIGIST (2001), *Standard for Software Component Testing*, Working Draft	BS 7925-2	3.4	27/04/2001
[RD.23]	Stuart C. Reid (1997), "An Empirical Analysis of Equivalence Partitioning, Boundary Value Analysis and Random Testing", *Proceedings of the Fourth International Software Metrics Symposium (METRICS'97)*.	DOI:10.1109/METRIC.1997.637166	-	05/11/1997
[RD.24]	Matteo Bordin et al. (2009), "Couverture: an Innovative Open Framework for Coverage Analysis of Safety Critical Applications", *Ada User Journal*, vol. 30, no. 4, pp.248–255. ISSN 1381-6551.	-	-	December 2009
[RD.25]	Constance L. Heitmeyer, Ralph D. Jeffords, and Bruce G. Labaw (1996). "Automated consistency checking of requirements specifications", *ACM Trans. Softw. Eng. Methodol.*, vol.5, issue 3.	DOI: 10.1145/234426.234431		1996
[RD.26]	David Harel (1987). "Statecharts: a visual formalism for complex systems", *Science of Computer Programming*, vol.8, no.3, pp.231-274.	DOI: 10.1016/0167-6423(87)90035-9		1987
[RD.27]	Nancy G. Leveson, Jon D. Reese, and Mats P.E. Heimdahl (1998). "SpecTRM: A CAD System for Digital Automation", in *Proceedings of the 17th Digital Avionics Systems Conference*.	-	-	1998
[RD.28]	The MathWorks (2010). *Simulink 7 User's Guide*. Online only. Revised for Simulink 7.6 (Release 2010b)	-	-	2010
[RD.29]	Michael von der Beeck (1994). A comparison of Statecharts variants, in *Formal Techniques in Real-Time and Fault-Tolerant Systems*, LNCS 863, p.128-148	DOI: 10.1007/3-540-58468-4_163	-	1994
[RD.30]	Michelle Crane and Juergen Dingel (2007). "UML vs. classical vs. rhapsody statecharts: not all models are created equal", *Software and Systems Modeling*, vol.6, issue 4, pp.415-435.	DOI: 10.1109/32.54292	-	1990
[RD.31]	David Harel and Hillel Kugler (2004). "The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML)", *Integration of Software Specification Techniques for Applications in Engineering*, pp.325-354, LNCS 3147.	DOI: 10.1007/978-3-540-27863-4_19	-	2004
[RD.32]	Gérard Berry and Laurent Cosserat (1984). *The Esterel synchronous programming language and its mathematical semantics*, INRIA technical report.	RR-327	-	1994
[RD.33]	Charles André (1996). *SyncCharts: a Visual Representation of Reactive Behaviors*, INRIA technical report.	RR 95-52	-	1996

Table 2-2: Reference Documents

3. ACKNOWLEDGEMENTS

The SOMCA project, developed by GMV, has successfully achieved the objectives initially defined thanks to the support received from different organisations and companies. The main contributor, of course, is EASA as project enabler. On the one hand, the project has been fully founded by EASA and EASA has established the contractual and financial framework for the project. On the other hand, EASA contributed to increase the quality of the results of the study by two means: firstly, providing very valuable technical and programmatic inputs to the different tasks, and secondly, with the feedback received during of the review of the different reports.

GMV also acknowledges the support received from the tool vendors Esterel Technologies and MathWorks. They have contributed to the success of the SOMCA project by ensuring the availability of the tools during the study with free licences. Both Esterel and MathWorks, have provided technical support on the use of the tools SCADE and Simulink / Stateflow respectively. Additionally they have contributed to the study with their deep knowledge of the Model Coverage concept and the problematic of the Unintended Functions.

GMV also appreciates the work of all the authors of previous investigations that, in one way or the other, have addressed the problem of the Model Coverage Analysis paving the way to the SOMCA Model Criteria.

Finally, the authors would like to acknowledge the support received from GMV colleagues with a large experience in Model Based Design tools and techniques.

4. EXECUTIVE SUMMARY

Modern economy, where shorter time-to-market has a great impact on success, inflicts major restrictions to the development of systems in terms of schedule and cost. Aeronautical industry relies on the quality of the service provided by the technology (understanding quality as a synonymous for customer's trust that can be measured in terms of compliance with functional, reliability, safety, robustness or security expectations). Embedded software is playing a more important role every day since the number of functionalities supported by software is increasing on new aircrafts. To reach this demanding level of quality and productivity in the development of embedded software, new methodological solutions are increasingly being used, like Model-Based Design (MBD) Software Engineering approaches.

In a European context, airborne systems have to be compliant with EASA regulatory framework in order to get airworthiness type certification. And specifically to be compliant with Book 1 Certification Specification CS 25.1309 [RD.3] which is applicable to any equipment or system as installed in the aeroplane. Among all requirements included in CS 25.1309 we are especially interested in the following one: *"The aeroplane equipment and systems must be designed and installed so that those required for type certification or by operating rules, or whose improper functioning would reduce safety, perform as intended under the aeroplane operating and environmental conditions.*"

Book 2 of CS 25 provides Acceptable Means of Compliance with requirements introduced in Book 1. Specifically, AMC 25.1309 "System Design and Analysis" recognises EUROCAE ED-12B / RTCA DO-178B Software Consideration in Airborne Systems and Equipment Certification, as a means, but not the only means, to secure approval of an applicant for EASA certification for any software-based equipment or system which uses the considerations outlined in ED-12B.

However, neither the current standard applicable for airborne software (ED-12B / DO-178B) nor the current standard for system development (ED-79 / ARP4754) directly addresses the verification or validation activities that should be carried out when model-based specifications are used.

Although there is not an standardized definition for Unintended Functions, in the context of the SOMCA study ***Unintended Function*** *is considered as "any unspecified —not defined in the higher-level requirements— and uncontrolled behaviour of the software under the aeroplane operating and environmental conditions, not detected after the validation and verification has been performed*."

The *Safety Implications in Performing* ***SO****ftware* ***M****odel* ***C****overage* ***A****nalysis* (SOMCA) study, funded by European Aviation Safety Agency (EASA), is aimed at establishing whether coverage analysis performed at model level can ensure an appropriate level of confidence for the detection of Unintended Functions, with the aim of characterising the occurrence and impact of all Unintended Functions and to be able to repair them at specification and modelling level. Hence, at the end, the aim of this 6-month study is to provide a sound scientific basis for preparing a certification policy update taking into account the Model-Based Development (MBD) CRI and the Certification Memo, the forthcoming release of EUROCAE ED-12C, and to provide recommendations on the acceptance criteria for Model Coverage Analysis (MCA).

The study is organised starting from a sound theoretical assessment of the types of formalism and model paradigms mostly used within the realm of the aerospace industry along with the tools supporting these paradigms. The outputs of this assessment, compiled in two Interim Reports (IR) [RD.1] and [RD.2] and the present Final Study Report (FSR), is reinforced by the analysis of the UF concept from different perspectives trying to produce a taxonomy of the UFs and investigating the SW lifecycle to establish a list of the possible sources of UF or activities that could lead to misdetections.

In subsequent phases, the theoretical analysis with its high level of abstraction was challenged against actual situations in which the presence of UF have been identified, by using literature references, GMV's own experience, EASA inputs, as well as an investigation of basic modelling blocks. Based on this view, conclusions were derived in the first Interim Report about under which conditions the occurrence of such situations may appear, and hence support the derivation of a set of Model Coverage Criteria and associated recomendations (preliminary) in the second Interim Report in order to implement a model-based coverage technique that allows responding to the strict safety criteria that could be applicable on a certification process.

At this point, the previously mentioned recommendations was applied over an existing piece of SW, developed and fully verified against DO-178B DAL-C equivalent standards based on conventional software development procedures. The same piece of software have been designed and verified with a model-based approach following the identified criteria, assuming a DAL-A assurance level. The outcomes of this process can be then

compared with the conventional development process outcomes, in order to assess how the proposed criteria behave in terms of: effectiveness in power detection of UFs, requirements coverage achieved, effective functional and performance behaviour achieved, and safety implications or oversights which may have been detected along the process (based on either of those methodologies).

A detailed assessment of the SOMCA criteria has been also produced in the second Interim Report and Final Study Report, indicating for each criterion the types of UFs allocated to it and the concrete examples of UFs detected by each criterion.

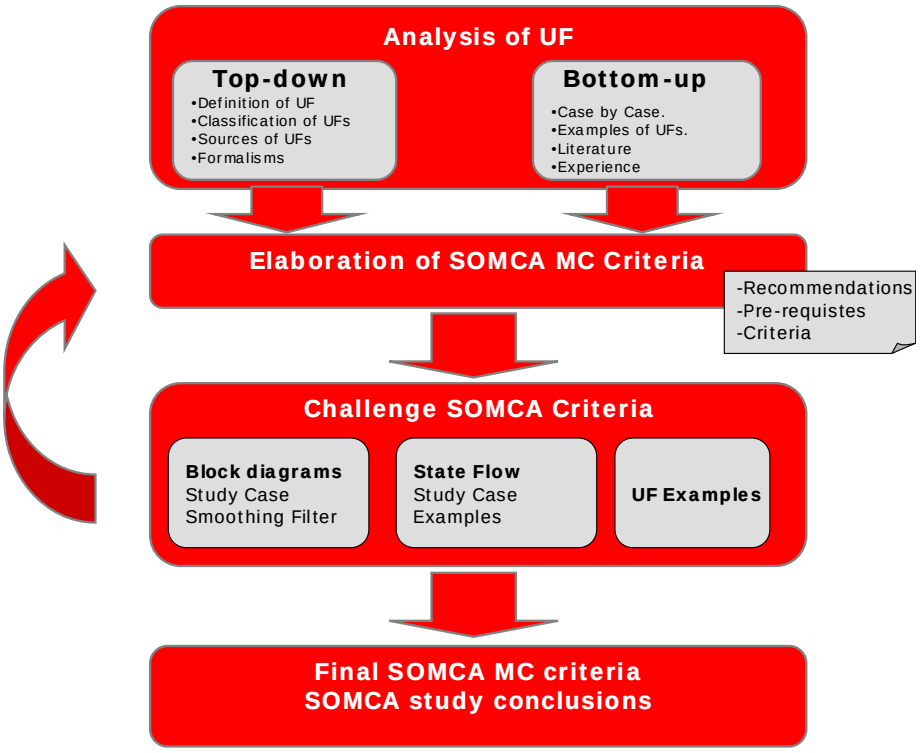


Figure 4-1 SOMCA Approach summary

The outputs of the different phases of the projects have been compiled into what is called the **SOMCA Model Coverage Criteria**. These outputs are grouped into three types of items:

- Criterion: an objective activity vital for detecting some UF types, and that must be enforced and analysed for all model elements. Part of the MCA criteria.
- Prerequisite: an objective activity that should be applied to the Formalized Design before performing the Model Coverage Analysis, that is not enforced per model element but for the whole design or group of model elements (e.g. a component). Intended to be part of an additional verification activity related to MCA.
- Recommendation: a recommendation that could be useful to avoid introducing further some UF types, or to ease detecting them, but too prescriptive to be added as an MCA criterion or prerequisite. Cannot be considered a verification activity, and can refer also to recommendations related to the source code.

These three outputs: criteria, pre-requisites and recommendations, are the set of recommendations to be used to amend the certification material and guidelines. As such, section 1.1 lists the changes needed in the Certification Memo to include the SOMCA criteria, which is therefore an essential input to the elaboration of the future ED-12C.

The study provides a set of recommendations for the update of the EASA Certification Memo, that is based on concrete evaluation of theoretical and real-world examples involving UF.

The following table lists the SOMCA Model Coverage Analysis criteria:

Criterion Name	Block Diagrams DAL					State Diagrams DAL				
	A	B	C	D	E	A	B	C	D	E
SOMCA Criterion 1: Range coverage	√	√	√			√	√	√		
SOMCA Criterion 2: Functionality coverage	√	√	√			√	√	√		
SOMCA Criterion 3: Modified input coverage	√	√	√			√	√	√		
SOMCA Criterion 4: Activation coverage	√	√	√			√	√	√		
SOMCA Criterion 5: Local Decision Coverage	√	√				NA	NA	NA	NA	NA
SOMCA Criterion 6: Logic Path coverage	√					NA	NA	NA	NA	NA
SOMCA Criterion 7: Parent State coverage	NA	NA	NA	NA	NA	√	√			
SOMCA Criterion 8: State History coverage	NA	NA	NA	NA	NA	√				
SOMCA Criterion 9: Transition coverage	NA	NA	NA	NA	NA	√	√	√		
SOMCA Criterion 10: Transition Decision Coverage	NA	NA	NA	NA	NA	√	√			
SOMCA Criterion 11: Transition MC/DC	NA	NA	NA	NA	NA	√				
SOMCA Criterion 12: Event coverage	NA	NA	NA	NA	NA	√	√	√		
SOMCA Criterion 13: Activating event coverage	NA	NA	NA	NA	NA	√	√			
SOMCA Criterion 14: Level-N Loop coverage	NA	NA	NA	NA	NA	√ (3)*	√ (2)*	√ (1)*		

(* The number in brackets means the N level used in the coverage criterion)

The main conclusion of this study is that Model Coverage is an efficient way of detecting Unintended functions introduced at the level of a Formalized Design. The Model Coverage Provided by EASA, as part of the Certification Memo [RD.8] section 23.2.9, was necessary although the study has revealed that they were not sufficient to address all the types of Unintended Functions. Therefore, it was necessary to complement these criteria to extend the power of detection of Unintended Functions in a Formalised Design.

The results of the assessment of this SOMCA criteria provided in the second Interim Report and this Final Study Report show that the Coverage Criteria from the Certification Memo have been enhanced, becoming much more robust and covering a wider concept and scope of Unintended Functions. Additionally, it has been found that the SOMCA criteria provide an important added value to assess and enhance the quality of the V&V activities performed at model level, and hence of the final product, advancing the detection of UFs and verification flaws to early stages of the project.

Additionally, since nothing has been detected that prevents this objective, the criteria defined is in the good way to be used as a solely Acceptable Means of Compliance within EASA regulatory framework in order to get the airworthiness type certification for airborne equipment and systems developed using MBD techniques.

This study could not establish whether the Structural Coverage Analysis and Model Coverage Analysis could be equivalent under given conditions. This will require additional research and, due to the inherent difference between those two analyses, it is not trivial to establish such an equivalency.

With respect to the objective of finding UFs, the most difficult UFs to be detected are those related with flaws in the specification and those related with the arithmetic operations. For the first group, it would be worth to analyse in the future the use of Formal methods, not initially covered by this study. These methods have

demonstrated that would reduce significantly the probability of injecting or misdetecting this type of Unintended Functions. The second group of UF would be addressed by the "Arithmetic path" concept and associate coverage criterion, only a draft is provided in this study and also further investigation would be needed to consolidate it.

Finally, during the different phases of the project, the criteria has evolved and changed substantially. This seems to suggest that the provided criteria are not yet definitive and further studies will improve them with new inputs obtained case by case from the experience (bottom up approach) and also from analysis (top down).

5. BACKGROUND AND MOTIVATION

5.1. AIRWORTHINESS CERTIFICATION

Airborne equipment and systems need to be compliant with EASA regulatory framework in order to get the airworthiness Type Certificate (TC) for the European Union. For example, a Large Aeroplane needs specifically to be compliant with the Certification Specification CS 25.1309 [RD.3] related to Equipment, Systems and Installations, which is applicable to any equipment or system as installed in the aircraft [RD.3]. Among all requirements included in CS 25.1309, the driver for this study is the requirement (a) (1):

> "*The aeroplane equipment and systems must be designed and installed so that those required for type certification or by operating rules, or whose improper functioning would reduce safety, perform as intended under the aeroplane operating and environmental conditions.*"

Book 2 of CS 25 provides Acceptable Means of Compliance with requirements introduced in Book 1. Specifically, AMC 25.1309 "System Design and Analysis" describes acceptable means for showing compliance with the requirements of CS 25.1309 and particularly provides means to comply with CS 25.1309 (a)(1). But also refers to AMC 20-115B which recognises EUROCAE ED-12B / RTCA DO-178B *Software Consideration in Airborne Systems and Equipment Certification*, issued December 1992 [RD.5], as a means, but not the only means, to secure approval of an applicant for EASA certification for any software-based equipment or system which uses the considerations outlined in ED-12B [RD.4].

5.2. MBD FOR AIRBORNE SOFTWARE

AMC 25.1309 emphasises a concern regarding the efficiency and coverage of the techniques used for assessing safety aspects of highly-integrated systems that perform complex and interrelated functions, particularly through the use of electronic technology and software-based techniques. The concern is that design and analysis techniques traditionally applied to deterministic risks or to conventional, non-complex systems may not provide adequate safety coverage for more complex systems. AMC 25.1309 addresses this concern providing the application of other assurance techniques to these more complex systems, such as development assurance utilising a combination of process assurance and verification coverage criteria, or structured analysis or assessment techniques applied at the aeroplane level, if necessary, or at least across integrated or interacting systems. Their systematic use increases confidence that errors in requirements or design, and integration or interaction effects have been adequately identified and corrected.

In order to reach the adequate level of quality and integrity, the development of complex and high-integrated software systems may use several methodological solutions:

- The conventional paradigm of rigorous software development (following ED-12B);
- Object Oriented Techniques;
- Model-Based Development (MBD) notations and methods for modelling software requirements;
- Formal methods based on techniques from logics and mathematics.

In recent years, Model-Based Development has become one of the most important advances for building complex and highly-integrated safety-critical software. Instead of using natural language or static diagrams for the requirements specification or system design, model-based development employs a formal notation to unambiguously describe the system behaviour and/or its architecture. Then, with the aid of tools, the model can be used to predict the behaviour of the system before its construction —like in other engineering disciplines, e.g. modelling of the wings of an aircraft— and to support other development phases like the requirements traceability or the generation of source code. Thus MBD has the potential to detect system problems early in the development process.

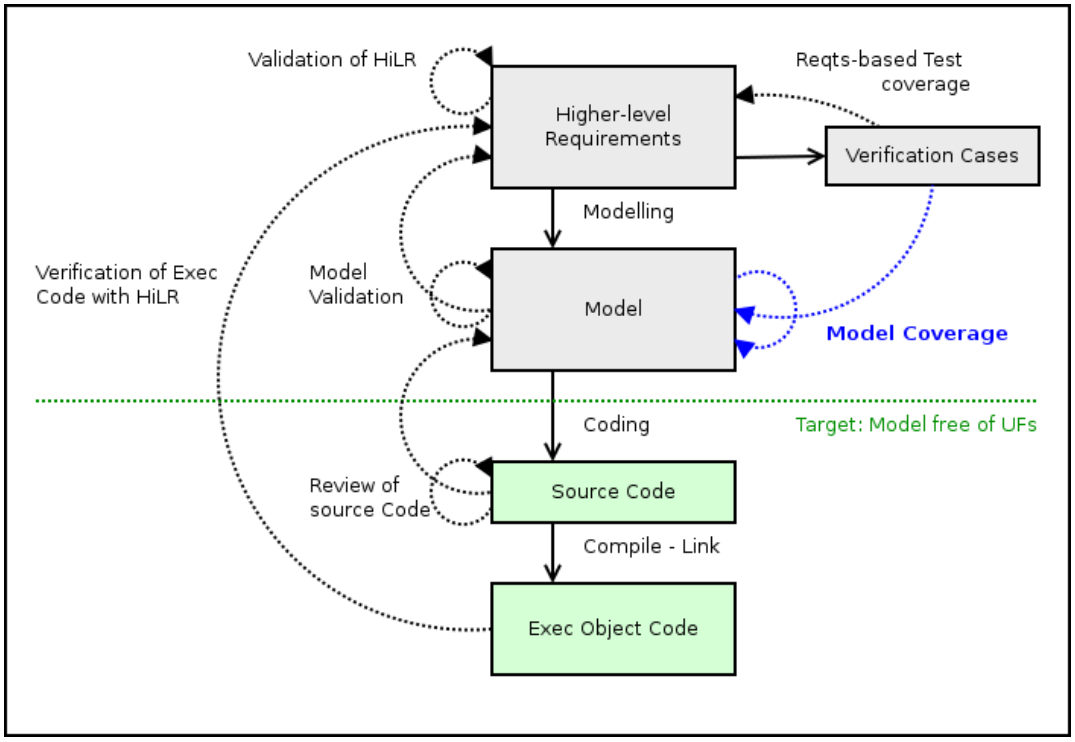


Figure 5-1: Model-based development workflow

5.3. MODEL COVERAGE ANALYSIS

In the civil aviation context, historical evidences show that about 10% of the total serious accidents due to operational and airframe-related causes were attributed to *Failure Conditions* caused by the aeroplane's systems [RD.3]. Accidents and incidents caused by software are usually related to problems with the specification [RD.13];

- It may happen that the software correctly implements the specification, but the requirements mandate an unsafe behaviour within the system.
- The software follows the specification, but the requirements are incomplete and do not consider some functionality needed to avoid accidents.
- That software deviates from the specification, containing **unintended behaviours** under the operating and environmental conditions.

This study is focused on the last problem, on the validation and verification activities in the context of Model-Based Development approaches with the aim of ensuring compliance with CS 25.1309 [RD.3] (a)(1) to guarantee the **intended** performance of the software running on airborne equipment and systems required for type certification or by operating rules, or whose improper functioning would reduce safety, under the aeroplane operating and environmental conditions. Specifically, this project has been based on the usage of Model Coverage Analysis techniques to ensure a design model has been properly verified to obtain a good level of confidence that potential **unintended functions** are identified and managed at specification level.

When MBD is employed for development of critical software, Unintended Functions can also be introduced at model level, and then injected to the source code and finally to the Executable Object Code. Requirements traceability analysis may not be able to detect the Unintended Functionality at model level due to the granularity of the requirements. In addition, traceability between the model (Low-Level Requirements) and the source code or Structural Code Coverage would not be able to catch this problem because the UF within the model is by definition a Low-level requirement. Therefore, Coverage Analysis at model level is required to identify UF types not easily detected by other means, and earlier in the development process than when applied to the source code.

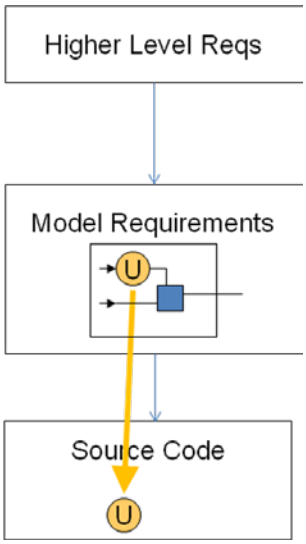


Figure 5-2: UF introduction at model level

Neither the current standard applicable for airborne software (ED-12B / DO-178B) nor the current standard for system development (ED-79 / ARP4754) [RD.7] directly address the verification or validation activities that should be carried out when formalized specifications or model-based specifications are used. However, previously to this study, EASA has collected some criteria regarding coverage of Formalized Design in order to ensure that it contains no Unintended Function, as well as a set of general principles and activities applicable to the cases where either Formalized Requirements or Design has been used [RD.8]. According to the EASA Certification Memo the following criteria may be used to assess the coverage of the Formalized Design:

- all the conditions of the logic components
- all equivalence classes (valid/in-range and invalid/out-of-range classes) and singular points of the functional components and algorithms
- all transitions of the state machines
- coverage of all characteristics of the functionality in context (e.g. Watchdog function is triggered)

It is uncertain if model coverage will provide the adequate mechanisms to detect, at model level, potential hazardous uncertainties of the specification, like Unintended Functions, that could lead to an unacceptable failure condition of the system. There are some reasons that can support this doubt:

- Model-based approaches are based on the use of some formalisms (like block or state diagrams) that are good for modelling some particular aspects of system behaviour, so at the end model verification for each one of this formalism may be excellent for catching some particular anomalous behaviour or Unintended Function, but could obviate other important Unintended Functions with the potential impact of causing a mishap.
- The use of model-based approaches needs to be supported by activities that assess the thoroughness of the verification, in particular by model coverage. In the case this model coverage activity is not performed correctly, Unintended Functions in the software may not to be detected. In addition, if tools are used in the MBD process (as it is generally the case), their use may not be so obvious for an engineer to use, and this could lead to the same state of uncertainty after model verification.

Then, it is deemed necessary to understand the real capacity and limitations of the different formalisms and supporting tools to ensure **model coverage**, in order to define a set of Acceptable Means of Compliance (AMC) criteria and recommended practices for the development of airborne safety critical software in relation to EASA Certification Specification (CS).

6. OBJECTIVES

The main purpose of the SOMCA project is to establish whether coverage analysis performed at model level can ensure an appropriate level of confidence for the detection of unspecified behaviours, with the aim of characterising the occurrence and impact of all Unintended Functions and to be able to repair them at specification level. This would allow addressing potential safety problems during early phases of the system and software development, with the advantage of reduced costs of repair for the industry. Hence, at the end, the aim of the study is to provide a sound scientific basis for preparing a certification policy update taking into account the Model-Based Development CRI and the Certification Memo, the forthcoming release of EUROCAE ED-12C, and to provide recommendations on the acceptance criteria for Model Coverage Analysis.

Several objectives were defined to guide this study. A first set of goals are related to the specific characteristics of the modelling notations used by the aeronautics industry, and the types of Unintended Functions associated with this new development process:

- Analysis of **Model-Based Development formalisms** employed for airborne software, focusing on those employed for the specification of Formalised Designs.
- For each notation, identification of which **kinds of Unintended Functions** with a potential safety impact present in the low-level software design can be analysed at model level. This analysis brought to the light the necessity of a clear definition of what an Unintended Function is, providing a taxonomy of Unintended Functions, objectives not initially foreseen in the project.

Another important set of goals concerns the usage of Model Coverage to guide the validation and verification activities, capable of bringing an appropriate level of confidence about the absence of unspecified behaviours in a given Formalized Design:

- Study of **Model Coverage Analysis techniques** for the model-based assessment of potential unspecified functions in the low-level requirements. This assessment needs to consider the different assurance levels defined in ED-12B (Level A, B, C and D).
- Definition of a **new set of Model Coverage Analysis criteria** capable of detecting the safety-related unintended functions identified in pervious phases. The goal is to find generic criteria applicable to different modelling notations and toolsets.

Finally, different objectives are defined for the assessment of the convenience and safety implications of using Model Coverage Analysis with respect to the traditional development processes defined in ED-12B:

- Assessment of the **effectiveness of the proposed MCA criteria** in the identification of Unintended Functions at model level, through the use of a variety of generic examples including basic models and real-world designs.
- Analysis of the **support of current commercial MBD toolsets** of the new proposed Model Coverage Analysis criteria.
- Study whether a subset of ED-12B objectives for **source code coverage could be replaced** by Model Coverage Analysis.

If deemed appropriate at the end of the project, recommendations for amending certification requirements and policies or acceptable means of compliance must be given at the light of the obtained results and conclusions about Model Coverage Analysis. It is worth noting that some of the goals initially defined for the project have been dynamically adapted through the course of the study to cover additional interesting topics not initially identified.

7. METHODOLOGY AND APPROACH

7.1. INTRODUCTION

This study is organised starting from a sound theoretical assessment of the types of formalism and model paradigms mostly used within the realm of the aerospace industry to characterise which are the “feared situations” in terms of incomplete requirements coverage that may have unexpected and undesired safety implications. This characterization must be based on the understanding of the applied formalisms and model-based technologies, and must have a high level of abstraction with respect to specific implementations. This coverage deals with, not just the lack of coverage of existing specifications, but also with the potential event in which the model or formalism used enforces additional functions which may not have been specified.

In subsequent phases, the theoretical analysis with its high level of abstraction shall be challenged against actual situations in which the presence of undesired functions or improper specifications coverage through modelling has been identified, by using literature references, GMV’s own experience as well as EASA inputs. Based on this dual view, conclusions can be derived about under which conditions the occurrence of such situations may or may not appear, and hence support the derivation of a set of criteria (preliminary) in order to implement a model-based coverage technique that allows responding to the strict safety criteria that could be applicable on a certification process.

At this point, the recommendations are applied over an existing piece of SW, developed and fully verified against DO-178B DAL-C equivalent standards based on conventions software development procedures. For that purpose, the same piece of software, will be designed and verified with a model-based approach following the identified criteria, assuming a DAL-A assurance level. The outcomes of this process can be then compared with the conventional development process outcomes, in order to assess how the proposed criteria behave in terms of: effective requirements coverage achieved, effective functional and performance behaviour achieved, and safety implications or oversights which may have been detected along the process (based on either of those methodologies).

Eventually conclusions may enable EASA to define a consolidated set of certification specifications about the usage of such model-based developments for certification of airborne equipment facilitating the overall process for developers, and granting an improved safety assurance process right from the specification phase.

7.2. DESCRIPTION OF THE APPROACH

Two methodological approaches were considered: bottom-up and top-down. The basics of the bottom-up approach is to start from concrete examples and then try to generalise the conclusions. On the contrary, the top-down approach begins with the generic concepts and then tries to demonstrate that they are applicable to concrete examples. Since both methodologies have pros and cons the approach selected combines both of them as shown in Figure 7-1.

Since the objective in to refine the preliminary definition of the Model Coverage Criteria and iterative approach has been selected to ensure that any conclusion obtained downstream is re-injected in the process.

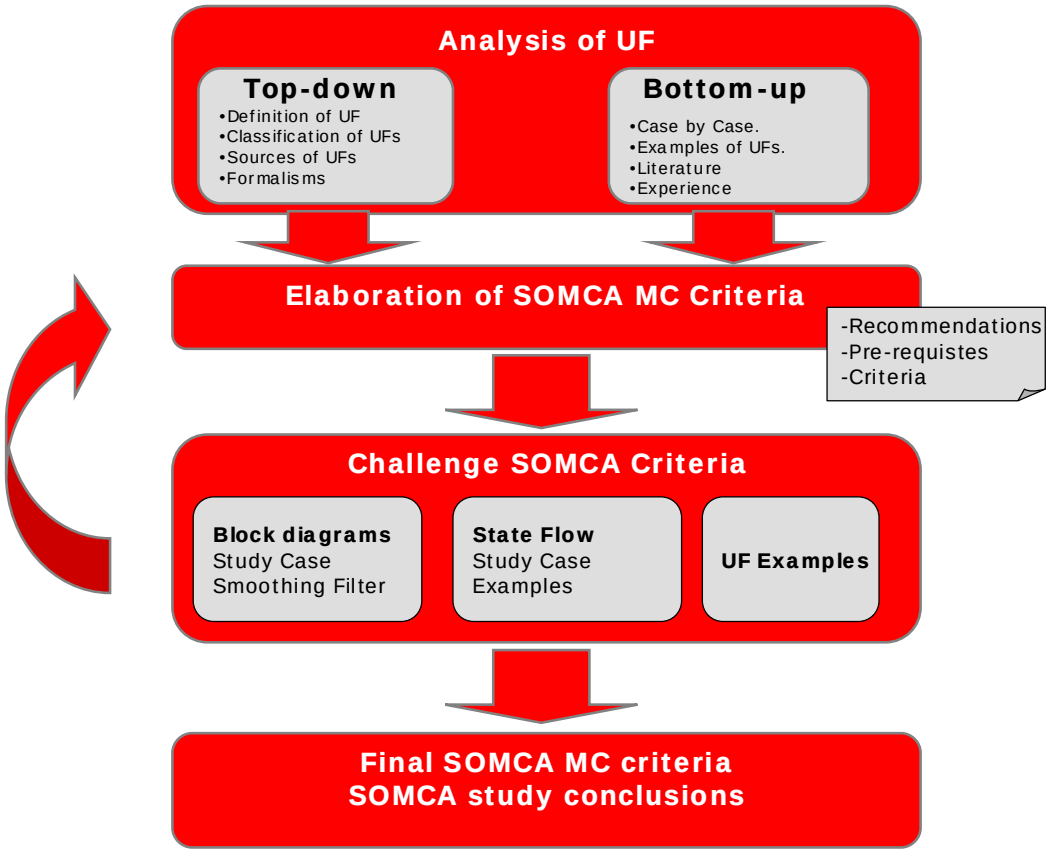


Figure 7-1: SOMCA study approach

The following sections provide a detailed description of the approach.

7.2.1. ANALYSIS OF UNINTENDED FUNCTIONS

The first step before starting the definition of the SOMCA criteria was to understand and analyse the Unintended Functions. The *Bottom-up* view consisted on a systematic search of real UF with the objective of finding a set of examples of unintended functions that could be used as input to try to challenge and refine the initial criteria provided by EASA. On the contrary, the objective of the *Top-down* perspective was to have a clear definition of the object being analysed, the Unintended Function, providing a clear shape of what can be considered and what cannot be considered an Unintended Function. The exercise of trying to define and understand what an Unintended Function is, brought to the light several questions (and associated conclusions):

- What is an Unintended Function? Can we provide a generic definition of Unintended Function?
- What is not an Unintended Function? And what are the differences between Unintended Functions and bad software design practices, errors in the software implementation, etc.?
- If a definition of Unintended Function can be provided, how can this definition help to refine the Model Coverage Criteria?
- Can Unintended Functions be classified and a complete taxonomy provided?
- How can this classification exercise help to increase the power of the already defined Model coverage Criteria?
- Do generic Unintended Functions exist? How can these generic Unintended Functions be extrapolated to all the models enhancing the Model-Coverage Criteria? Or on the contrary, it is only possible to perform a case by case analysis and the Model-Coverage Criteria needs to be updated every time that a new Unintended Function is detected.

- Is it possible to identify a set of possible sources of the Unintended Functions? How can help the analysis of these possible sources of Unintended Functions to refine the Model Coverage Criteria?

This analysis of the unintended function was implemented into a set of lines of work to organise and focus the effort.

Line of work 1: Analysis of formalisms

Unintended Functions, as understood in the SOMCA project, appear mainly as a consequence of the use of the MBD techniques (by several reasons), therefore the first line of work consists in the analysis of the different formalisms used in Model Base Design and particularly in those applied in the aviation industry. As natural consequence, the different notations and tools supporting these formalisms are also analysed.

For each formalism and tool, the main characteristics are analysed and provided in this document, the analysis put special attention on the following aspects:

- Can the intrinsic characteristic of the formalism/tool be prone to the generation of Unintended Functions?
- How the characteristics of the formalism/tool/notation can help to refine the Model-Coverage Criteria?

Line of work 2: Sources of unintended functions

As part of this line of work, several aspects of the SW development are analysed to try to find possible sources of Unintended Functions. These sources cover a wide range of possibilities: the formalism selected can be prone to a given type of Unintended Function, the nature and type of the requirements intended to be modelled, the type of validation is not the appropriate one, etc. The analysis process itself will provide valuable feedback in the consolidation of the concept and classification of Unintended Functions, as well as the different aspects that the Model Coverage Criteria shall consider: techniques used in the modelling, types of validation, nature of the requirements modelled, etc.

Line of work 3: Classification of unintended functions

The objective of performing the exercise of classifying the Unintended Functions is, again, to better understand what they are, the conditions that could favour the appearance of Unintended Functions and, what is more important, the means to detect them.
Providing a classification of Unintended Functions, and being able to allocate each new detected Unintended Function into a given type of the classification, will contribute to:

- Ensure that when challenging the Model Coverage Criteria, in the next phases of the project, all the range of Unintended Functions are covered.
- Define an absolute and generic Model Coverage Criteria. If the Unintended Functions could not be classified, then a generic Model Coverage Criteria would hardly be found.

It is important to stress, that the target is not the classification of the Unintended Functions itself, but to apply this analysis process as a means to refine the Model Coverage Criteria.

Line of work 4: Examples of unintended functions

The objective here is to apply a bottom up approach, identify a set of Unintended Functions coming from some specific examples and use them to challenge the Model Coverage Criteria. This will help to increase the detection power of the MC criteria. The examples of Unintended Functions come from the following activities:

- Analysis of basic blocks of the library of the tool.
- Examples of Unintended Functions found in other projects.

- Examples of Unintended Functions identified in the analysis performed in the different lines of work.

It is important to note that the activity of trying to detect some specific Unintended Functions (the search process itself) is, to some extent, the definition of a MC criteria. The examples found in this line of work will be used to challenge the Unintended Function definition, classification and Model Coverage Criteria.

7.2.2. DEFINITION OF SOMCA CRITERIA

The definition of the SOMCA criteria was driven by the following questions that brought up when the roadmap of the MC analysis was being defined:

- What is understood as Coverage Analysis when talking at model level?
- How the outputs of the UF analysis can be considered and can contribute to the definition of a SOMCA Model Coverage Criteria?
- Is it possible to define a set of Model Coverage Criteria capable of detecting all the UFs (at least those examples identified)?
- Is it possible to tailor a Model Coverage Criteria to the different SW DALs?
- How existing tools understand Model Coverage and how they support it?

Once again, the selected approach to answer these questions was a mix of bottom-up and top-down views. The top-down view consisted on the analysis of the concepts of “Model Coverage Analysis” and “Model Coverage Criterion” in order to have a clear understanding when defining the set SOMCA Model Coverage Criteria. The activities for this analysis were a survey of coverage criteria at source code level and a survey of coverage criteria for block diagrams and state diagrams and investigation on how modelling tools support this analysis.

The bottom-up perspective was based on the analysis of the outputs of previous phases. During the analysis phase of Unintended Functions two main outputs were generated: a set of examples of UFs and a set of recommendations compiled throughout the analysis phase. All this information was processed and the result was a set of criteria, prerequisites and recommendations:

- SOMCA Criterion: an objective activity vital for detecting some UF types, and that must be enforced and analysed for all model elements. Part of the MCA criteria.
- Prerequisite: an objective activity that should be applied to the Formalized Design before performing the Model Coverage Analysis, that is not enforced per model element but for the whole design or group of model elements. Intended to be part of an additional verification activity related to MCA.
- Recommendation: a recommendation that could be useful to avoid introducing further some UF types, or to ease detecting them, but too prescriptive to be added as an MCA criterion or prerequisite. Cannot be considered a verification activity, and can refer also to recommendations related to the source code.

7.2.3. CHALLENGE SOMCA CRITERIA

From the methodological point of view, once the SOMCA criteria are defined they should be tested in order to assess different properties of the criteria: detection power and effectiveness, completeness and implementation feasibility. To do this, the following aspects were analysed:

- Analysis of the detection capability of the SOMCA criteria using the examples of UFs gathered during the study.
- Analysis of the degree of coverage of the SOMCA criteria of the different types and sources of UFs identified in the study.
- Application of the SOMCA criteria to real and representative piece of critical SW. The SOMCA criteria are applied to a real operational example to assess detection power of remaining UFs after V&V activities.

The outputs resulting from the SOMCA criteria challenge were used to further refine the SOMCA criteria.

7.2.4. ELABORATION OF CONCLUSIONS

This is the natural final phase in any study. In this last project step all the conclusions elaborated throughout the project are compiled, processed and matured bearing always in mind the safety and certification aspects. During the study, intermediate results and conclusions are compiled in the so-called Interim Reports and final conclusions are presented in the Final Study Report, this document.

8. IMPLEMENTATION

This section describes the implementation of the methodology proposed in the previous section. This description covers the work breakdown structure, programmatic aspects and technical deliverables.

The SOMCA project is as study project with duration of seven months, and the activities were divided into three main Work Packages (WP) as described in Figure 8-1, which main objectives are:

WP1_MF&MCA (Model Formalism and Model Coverage Criteria): This WP implements the analysis of the Unintended Functions described in section 7.2.1 covering the four lines of work identified there and the definition of the Model coverage criteria, described in 7.2.2. The objective of this WP is twofold, firstly to investigate the different model formalisms for identifying a generic set of unintended functions which may impact on system safety and secondly to set of model coverage analysis criteria. This WP is grouping two major tasks:

- Task 1 — Model Formalism: Investigate model formalisms.
- Task 2 — Model Coverage Criteria: Define a set of model coverage analysis criteria.

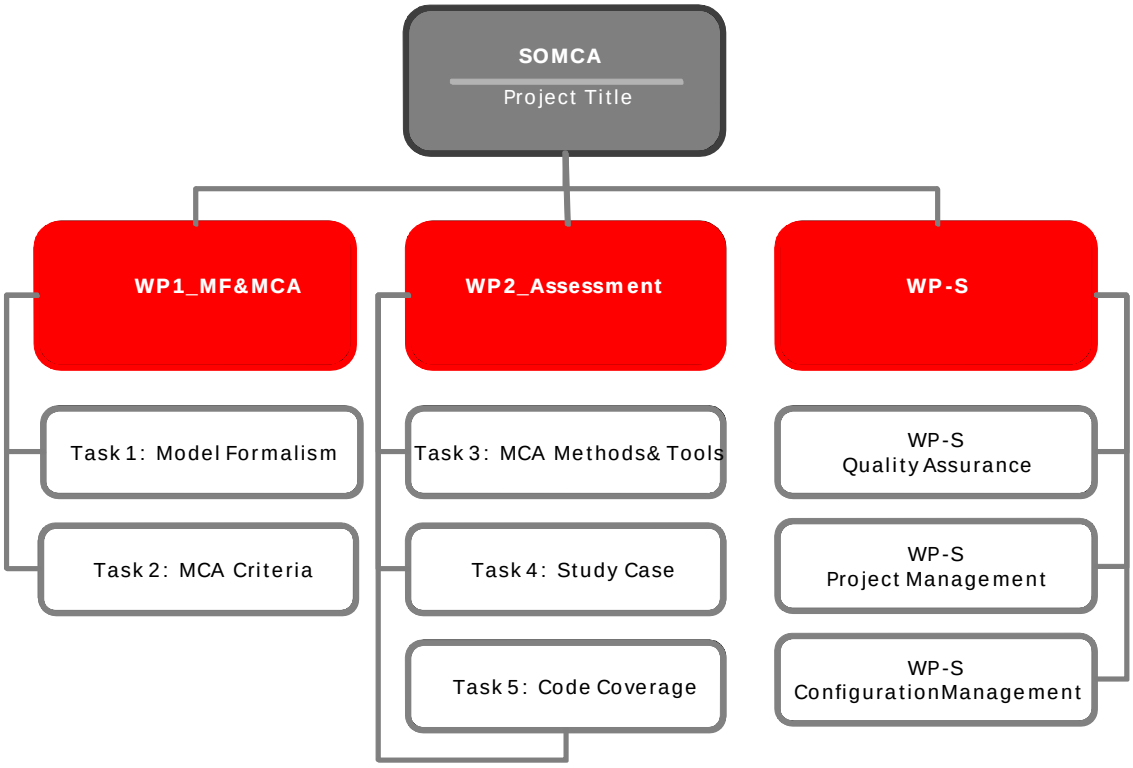


Figure 8-1: SOMCA Work breakdown structure

WP2_Assessment: This WP address the assessment of the SOMCA Model Coverage criteria as described in section 7.2.3. The objective of this WP is to assess the Model Coverage Criteria identified in WP1 from different perspectives: existent methods and tools used in the development of avionic critical software, completeness and effectiveness and practical application in a set of study cases and relation between model coverage and structural code coverage. Three tasks will cover these diverse aspects of this assessment:

- Task 3 MCA Methods and Tools: Summary of MCA methods and tools and how commercial tools implements and assess model coverage.
- Task 4: Study Case: Preparation of generic examples and real study case.

- Task 5: Analysis of the relation of the code coverage and model coverage.

WP-S (Support): the objective of this WP is to provide support to Project Management and it includes all process supporting project activities: Quality Assurance, Project Management and Configuration Management. The support activities of this WP will be present during the complete duration of the project, as it is deemed essential to control the adherence to Project Management Plan.

The technical outcomes of the WP1 andWP2 were included into two Interim Reports, and then a Final Study Report (this document) compiled major findings of the project. Each of these reports was subjected to a review and approval process by EASA. The following table describes the logic between the task and the reports:

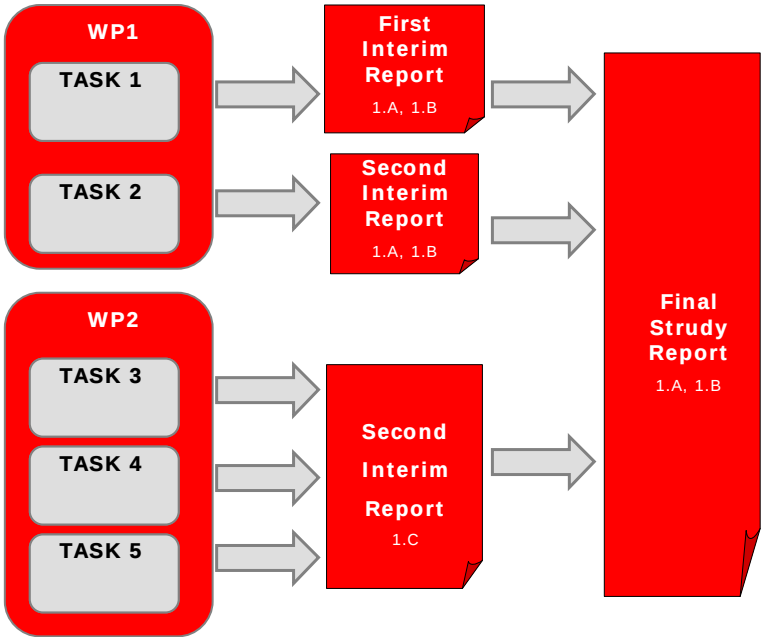


Figure 8-2: SOMCA project technical deliverables

EASA monitored the evolution of the project by means of a Project Progress Report delivered by GMV every month and by a set of checkpoints (Project Progress Reviews) distributed in time as indicated in the following table, being T0 the contract signature:

Meeting	Date
Kick Off Meeting	T0+1week
Project progress review meeting 1	T0+1 month
Project progress review meeting 2	T0+3 months
Project progress review meeting 3	T0+5 months
Final Presentation	To+7 months

Table 8-1: Progress Meeting schedule.

The following section summarises the activities performed in the context of each task.

8.1. DETAILED DESCRIPTION OF TASKS

8.1.1. TASK 1: MODEL FORMALISM

This first task of the first WP implements the initial step of the study methodology described in section 7.2.1, aimed at analysing the Unintended Functions and Model Formalisms. This task was programmed for two months and the outputs were compiled in the First Interim Report, issues 1.A and 1.B.

At proposal time the focus of the task was twofold:

- A comprehensive survey on the different model formalisms, which are used nowadays to support the development of safety-critical airborne software.
- Once those formalisms are identified, the assessment of potential situations which, upon application of those formalisms, may induce the generation of unintended (and, particularly, undesired) functions.

However, as described in section 7.2.1, it was detected the necessity to better understand and define the concept of unintended function and the context of the problem, hence the task was organised into four lines of work.

Line of work 1: Analysis of formalisms. The different modelling notations relevant for the development of safety-critical software within the aviation industry were analysed. Secondly, a survey of the features provided by the toolsets supporting each notation was made, and finally it was analysed the possibility of each formalism to introduce Unintended Functions in the model. A secondary objective was to provide a set of key terms related with model formalisms in the scope of this project, which have sometimes different meanings within the industry.

More concisely the following analyses were performed:

- Formalised Requirements: Analysis of the different modelling notations and tools used in the production of Formalised specification in the aerospace industry. The following formalism were analysed: SCR (Software Cost Reduction), RSML and RSML^{-e} (Requirements State Machine Language without Events), SpecTRM (pronounced “spectrum”, and abbreviation of Specification Toolkit and Requirements Methodology), and SysML (Systems Modelling Language).
- Formalised Design: the different formalised modelling notations were analysed, mainly block diagrams and state machines. For each of these notations the main implementations were also analysed.
    - For the block diagrams the following implementations were analysed: Statemate ActivityCharts, by I-Logix (now part of IBM); SAO, an in-house development tool by Aérospatiale (now EADS/Airbus); SCADE Data Flow diagrams, by Esterel Technologies; LabVIEW, by National Instruments; BEACON Signal Flow notation, by Applied Dynamics International; and Matlab’s Simulink toolbox, by The MathWorks.
    - On the sate machine side, the following implementations were analysed: Statemate StateCharts, by I-Logic (now IBM); SCADE SSM notation (Safe State Machines), by Esterel Technologies; BEACON Control Flow notation, by Applied Dynamics International; Rational Rhapsody UML state machine notation, by IBM; and Matlab’s Stateflow, by The MathWorks.

After this preliminary survey of the market only two implementations were selected Simulink (and Stateflow) and SCADE. The reasons for the selection of these tools were the availability of licences and the large representation of the tools in the aeronautic industry.

The analysis of these formalisms were performed with the final objective of detect which characteristics of these notations could favour the injection of unintended functions, which recommendations could reduce the probability of injecting them and how this information can contribute to the refinement of the Model Coverage Criteria.

Line of work 2: Sources of Unintended Functions. The SW lifecycle was reviewed with the aim of identifying engineering practices that could increase the possibility of injecting unintended functions or could reduce the possibility of detecting them; these two types of practices were called “sources of Unintended Functions”, and they were organised into four groups: modelling mistakes, formalism or toolsets, aspects external to the model, and misdetection.

Line of work 3: Classification of Unintended Functions. A preliminary taxonomy for the UFs was provided, this would help to assess the completeness of the study in the following terms: each detected UF can be allocated into a given type, there is at least one UF of each type and the final Model coverage criteria covers this taxonomy.

Line of work 4: Examples of Unintended Functions. The activity of finding examples of UFs was carried out during the whole project, and whenever a new UF was detected the list of examples of UFs was updated. However, this line of work was specifically aimed at searching examples of UFs. The search of UF was based on the following resources: available literature, GMV's previous experience, and analysis of basic blocks and elaboration of intended examples aimed at demonstrating the existence of basic UF.

Each activity performed during TASK 1 was developed with the aim of contributing to the definition of the SOMCA Model Coverage Criteria, therefore whenever a valuable conclusion was found a "recommendation" was generated. These recommendations were processed in the following phases.

8.1.2. TASK 2: MODEL COVERAGE CRITERIA

This second task of the WP1 implements the second step of the study methodology described in section 7.2.2, it is aimed at providing a first refinement of the Model Coverage Criteria. The planned duration of this task was three months and the outputs were compiled in the Second Interim Report, issues 1.A, 1.B, and 1.C.

This task started with a sound analysis and processing of available literature to establish a definition of Model Coverage Analysis and Model Coverage Criteria. This preliminary phase of the task studied the Structural Code Coverage case because some concepts used in Model Coverage are inherited from this field. In addition, the different between coverage in block diagrams and state machines were also considered.

The next analysis to contribute to the definition of the MC criteria was to analyse how commercial tools understand Model Coverage, in particular SCADE and Simulink (including Stateflow).

All the outputs produced in the previous phases of the project (Analysis of Formalisms and Unintended Functions) in the form of recommendation were processed and for each recommendation it was generated: a pre-requisite, a recommendation, a criterion.

With all the inputs generated in TASK 1 and TASK 2 a first version of the SOMCA Model Coverage Criteria was generated.

It is important to highlight, that the definition of the SOMCA Model Coverage Criteria has followed an iterative approach. The seed for the first iteration was the Model Coverage Criteria provided by EASA in the certification Memo, then, as part of the TASK 2, the SOMCA Model Coverage criteria was defined, and finally the finding of the TASK 4 produced the final refinement of the Model Coverage Criteria as presented in this document.

8.1.3. TASK 3: MCA METHODS AND TOOLS

This third task of the WP2 implements part of the second step of the study methodology described in section 7.2.2 , its objective is to analyse how commercial modelling tools support the defined SOMCA Model Coverage Criteria. The planned duration of this task was three months and the outputs were compiled in the Second Interim Report, issues 1.B and 1.C.

The initial objective of this task was, considering the MCA acceptance criteria established as consequence of Task 2, to perform a survey of the existent MCA methods (in commercial software development toolkits for aerospace) as well as possible evolutions, through a consultation of several relevant aircraft / equipment manufacturers and solution/tool vendors.

GMV proposed to analyse at least the most widely used commercial model-based development tools:

- SCADE Suite, by Esterel Technologies
- Simulink & Stateflow, by MathWorks
- BEACON, by Applied Dynamics International

However, during the evolution of the project, the scope of the task was aligned with the outputs and activities performed in previous tasks, in order to maximise the quality of the assessment of the criteria proposed. Eventually, it was decided to analyse the two most representative modelling tools, SCADE and Simulink (State Machines), and in particular how they support model coverage compared with the SOMCA Model Coverage Criteria: adding or features of the tools aimed at assessing the model coverage. The output of this task provided valuable feedback in order to refine the initial SOMCA criteria increasing its feasibility, removing those aspects where the added value was not so evident.

8.1.4. TASK 4: STUDY CASE

This fourth task of the WP2 implements part of the third step of the study methodology described in section 7.2.2, it is aimed at assessing the SOMCA Model Coverage criteria in terms of: detection power, completeness and feasibility. The planned duration of this task was three months and the outputs were compiled in the Second Interim Report issue 1.C. and the Final Study Report.

To assess the SOMCA criteria decided to use the examples of UFs identified during the project but also exercise it over more complex examples coming from real examples. To achieve this objective, this task was organised into the following working threads:

- Study Case for Block Diagrams
- Study Case for State Machine
- Elaboration of Final Study Report.

Study Case for Block Diagrams

The objective of this study case is to assess the SOMCA criteria in a real development. To do so, an operational critical piece of SW developed with traditional methodology was selected, and the same subsystem was developed following a MBD techniques, finally the SOMCA criteria was executed within this model with assurance level A. To enrich the output of this activity the model was developed using two different tools: GMV used Simulink and ESTEREL provided the model developed by SCADE. The subsystem selected was a smoothing stage based on a Hatch filter.

The activities performed as part of this task are the following:

1. Definition of HiLR. Definition of a clear base line containing the higher-level requirements to derive the model and the test cases.
2. Elaboration of Simulink model based on the HiLR.
3. Execution of Model Advisor to check model correctness.
4. Definition of HiLR-Model traceability using DOORs tool, and requirement coverage analysis.
5. Definition of Test Cases based on HiLR.
6. V&V of the smoothing model based on the previously defined Test plan. Due to the definition of the UF itself, it is necessary to perform a complete verification of the model before applying the SOMCA criteria to assess the added value of the criteria in the detection of UFs.
7. Analysis of Model Coverage based on tools included in the Simulink Tool.
8. Application of the SOMCA criteria to the smoothing model, to check if there is any UF remaining in the model after V&V activities.
9. Definition of UFs to be artificially injected in the model. To stress SOMCA within the real example some UFs were artificially injected in order to exercise specific SOMCA criterion.
10. Application of SOMCA criteria to detect artificially injected UFs. Once the UF were injected the SOMCA criteria was analysed in the model.
11. Validation of the smoothing model within a pseudo-operational environment to assess remaining UFs after SOMCA criteria application. The model developed was exercised with the real inputs and expected output recorded from an operational environment, this allows to directly compare the behaviour of the model created with the real fully validated subsystem and extract the following conclusions:

a. Are there UFs still remaining in model after V&V activities and SOMCA criteria application?

b. If so, how could they have been detected? How the SOMCA criteria could be modified to detect them

12. Refinement of SOMCA Criteria with all the outputs generated in these activities.

ESTEREL was provided with the HiLR and the test plan, the model developed by ESTEREL was analysed during three days with onsite support, mainly to analyse the Model Coverage implemented in SCADE and the relation with the SOMCA criteria.

Study Case for State Machines

The real example selected to assess the SOMCA criteria did not contain any state machine complex enough to perform a real assessment of the SOMCA criteria related to state machines. Therefore it was decided to analyse a set of complex state machines to perform a realistic assessment. The activities executed in this task were the following:

1. Identification of a set of complex examples: from literature, examples already existing in the tools or created ad-hoc.
2. Application of the SOMCA criteria to these examples to assess existing UFs.
3. Definition of a set of UF to be artificially injected in the models.
4. Application of SOMCA criteria to assess detection capability of the UFs injected in the model artificially.
5. Refinement of SOMCA Criteria with all the outputs generated in these activities.

Final Study Report

The last activities of this task were the production of the final refinement of the SOMCA criteria, compilation of the main conclusions of the study, identification of the points that would need further analysis in the future. Finally, the Final Study Report was elaborated considering all this information.

8.1.5. TASK 5: CODE COVERAGE AND MODEL COVERAGE

This fifth task of the WP2 implements part of the second step of the study methodology described in section 7.2.2, it is aimed at analysing the relation between the model coverage and code coverage. The planned duration of this task was three months and the outputs were compiled in the Second Interim Report, issues 1.B, and 1.C.

The objective of this task was to initiate the assessment of the conditions under which a subset of ED-12B objectives for structural code coverage verification may be replaced by the model coverage and what are the related constraints or dependencies. As part of this task, GMV investigated how to define the means to obtain objective evidences to demonstrate that the assurance level obtained with SOMCA is enough for the certification process. The Structural Code Coverage has been accepted traditionally by the industry, therefore a possible means could be to compare the assurance level obtained by means of Structural Code Coverage with the one obtained with SOMCA criteria.

9. RESULTS AND OUTCOMES

This section presents the results obtained during the study, after performing the steps described in the previous section. After a brief discussion of modelling notations, we present our analysis about Unintended Functions in the context of MBD and their origin. Then, Model Coverage Analysis is discussed and a new set of SOMCA MCA criteria is presented, including an assessment of its effectiveness and an analysis of how far SCADE and Simulink / Stateflow currently support it. Finally, it is discussed the relationships between Model Coverage and Structural Code Coverage.

Just a few examples are presented in this section, but please note that the SOMCA criteria have been challenged against several models during the course of this study (both already existing examples and models specifically created for this project). For further information, please refer to the annexes for a detailed list of examples and a complete summary of the results.

9.1. MODELLING FORMALISMS

This section includes the results and outcomes relative to the modelling formalisms analyzed in this study. The complete analysis can be found in Annex C (section 14.)

The use of design modelling languages, such as Simulink and SCADE (Safety-Critical Application Development Environment), in the development of safety-critical systems appears to be part of a long-term trend. This popularity indicates that system developers see genuine value in their use. This study focuses on notations used for airborne development, specifically on those notations used for Formalized Designs. The recent EASA Certification Memorandum [RD.8] gives the following definition:

> *A **Formalized Design** may be a model produced by the use of a modelling tool or it may be a design stated in a formalized language. In either case, a Formalized Design should contain sufficient detail of such aspects as code structures and data / control flow for Source Code to be produced directly from it, either manually or by the use of a tool, without any further information.*

Over the last decade, synchronous, graphical modelling languages have been increasingly used in the development of safety-critical systems. Simulink and SCADE are by far the most widely-used development tools, at least for airborne software. Due to its popularity, these tools have been sometimes used for the specification of requirements [RD.9], however Simulink and SCADE main use continues to be the design and experimentation of a system, and the code generation. It is worth noting that not all High-Level Requirements are modelled in a Formalized Design (for example some non-functional requirements can be impossible to be modelled). Furthermore, they can intentionally be left outside the model due to the increased complexity to the design without any added value. In the following subsections this idea will be extended because, depending on the limitations of the specific notation, this could be a source of Unintended Functions.

Notations for Formalized Designs allow the simulation of the models to perform dynamic verification at design level, a feature not available in traditional development techniques. This dramatically changes the possibilities of the designers, allowing them to better understand the implications of a design decision with respect a wide variety of test scenarios, and thus removing many design errors early in the project. Obviously, this feature is critical to remove many types of Unintended Functions that were not previously possible with traditional techniques.

Code generation is also a common feature of these toolsets, usually to safe profiles of the Ada and C programming languages. In addition, some of these development environments include or can be extended with specific tools for model coverage analysis and static analysis for formal verification, requirements traceability (interfaced with DOORS or RequisitePro), or code generation to high-integrity programming languages like Spark. Even if all these features were also possible with traditional development techniques, MBD tools offer further integration among all them reducing the effort required to apply them and the possibilities of introducing mishaps, thus gaining increased confidence on the absence of Unintended Functions.

Both Simulink and SCADE provide two different types of notations for the development of Formalized Designs, which are in fact the most extended categories of formalisms used in commercial model-development tools: block diagrams for continuous control and state diagrams for discrete control. Each notation is focused on modelling different types of algorithms, and commercial tools usually support both notations, even allowing mixing these two types of formalisms in the same design. Usually a subset of the notation or libraries is used in

production software [RD.10][RD.11][RD.12]. Both types of formalisms are aimed at handling complex designs, providing different abstraction constructions where each block or state can be decomposed into a hierarchy of blocks or substates.

9.1.1. BLOCK DIAGRAMS

Block diagrams is a category of graphical notations for designing dynamic systems, i.e. whose outputs change over time like electrical circuits, mechanical or thermodynamic systems. The designer drags and drops blocks onto a canvas and connects the outputs of one block to the inputs of another block, to graphically depict the time-dependent mathematical relationships (data flows) among the system's inputs, internal variables, and outputs. Blocks can also be parameterised (usually displaying a wide variety of configuration options), and can be composed hierarchically from simpler blocks, allowing designers to organize complex system designs. New blocks can be defined by the developer and added to a reusable library.

The history of these block diagram models is derived from engineering areas such as Feedback Control Theory and Signal Processing. Block diagrams are a natural modelling framework for control engineers to design both digital and analogue circuits, by sampling sensors at regular time intervals, performing signal processing computations on their values, and outputting values often using complex mathematical formulae. Data is continuously subject to the same transformation as well. Block diagrams are easily amenable to simulation, and the virtual experimentation is a powerful and flexible tool for rapid prototyping which can be extended with specialised control blocks (like the basic constructs of a functional programming language) or blocks to support static analysis.

9.1.2. STATE DIAGRAMS

While block diagrams are a natural notation for the design of circuits, the state diagram family of formalisms is closer to the notation used by system and software engineers for modelling the behaviour of an embedded system. A state diagram is also a representation of a reactive (event-driven) system, but now the system is composed by a set of states (also called modes), input events, edges (usually called transitions) between states, and processing actions. The system fires a transition from one state to another if a guard condition holds, which could have processing actions associated with the state and / or transition.

State diagrams allow a discrete control changing behaviour according to external events originating either from discrete sensors and user inputs or from internal program events, e.g. value threshold detection. Discrete control is characteristic of modal human machine interfaces, alarm handling, complex functioning mode handling, or communication protocols. State diagrams are used to describe data flows and easily allow simulation, static analysis, and a later derivation into a software programming language. In addition, these notations usually allow having hierarchical diagrams, where a state is actually composed of sub-states and local transitions. This hierarchy of states facility is mainly for organisational purposes, but could also simplify the model in case all the substates have a transition to the same external state for a specific event. Another feature of state diagrams usually incorporated into these notations is the history mechanism: a variable records the substate being executed, in case the designer wants to resume execution to the last substate.

9.1.3. CONCLUSIONS ABOUT MODELLING FORMALISMS

The industry already recognises the advantages provided by Model-Based Development, and the usage of block diagrams and state diagrams in specific phases of a project is widely extended. Even if a variety of tools for Formalized Designs exists in the market, in the case of airborne software the most extended are SCADE and Simulink.

However, one of the main obstacles found during this study has been the differences found in the notations implemented in each tool. Both Matlab and SCADE notations are based on the same concepts, but the features offered by each tool and the notation supported greatly differs. Actually, some of the unique features of each notation or toolset can be dangerous for a safety-critical environment, being thus a possible source of unintended functionality. These differences make very difficult to establish common criteria valid for all notations. The first approach could be one of the following:

- Define a common language / formalism that should be supported by all tools.
- Define criteria specific for each tool.

None of these two solutions are viable. The first one would require a huge effort in the tools suppliers to adapt the tool for the new formalism. The second one will create specific rules for each tool and notation, introducing new problems about the comparison of the rules between similar notations.

The approach taken has been to define the common elements in the formalisms in a precise way, but valid for both notations, and develop the criteria over these generic definitions. These differences could generate possible coverage problems under some tools that should be studied independently. Most of these problems can be mitigated with the use of Design Standards. If we use generic criteria in combination with design standards specific for each notation, we will have a good combination between the benefits of generic coverage criteria and the elimination of the problems introduced by the specific notation issues (thanks to the rules included in the design standard).

The next section will present an analysis on the Unintended Functions in a Model Based Development approach, including a discussion on the UF types specific of each modelling formalism.

9.2. ANALYSIS OF UNINTENDED FUNCTIONS

9.2.1. WHAT IS AN UNINTENDED FUNCTION?

"Unintended Function" is a term commonly used within software certification to refer to any software behaviour not intended by the system designers. This generic term is not defined in ED-12B / DO-178B, however we felt that it was important to dedicate some effort to analyse the meaning of this concept in the framework of the SOMCA project, to properly understand the nature of these software problems. We wrote a definition to ensure all the important aspects of UFs were considered:

> **In the SOMCA project, an Unintended Function is any unspecified —not defined in the higher-level requirements— and uncontrolled behaviour of the software under the aeroplane operating and environmental conditions, not detected after the validation and verification has been performed.**

The goal of the Validation and Verification activities is to ensure that the software behaves as intended. This involves ensuring that all Intended Functions specified in the requirements are produced, and no Unintended Function (UF) can result during execution of the software.

Both Model Validation and Verification activities are based on the higher-level requirements, ensuring first that every intended Function is implemented. However, there may be parts of the model which have not been exercised by the requirements-based verification procedures. So, during simulation model coverage analysis should be performed and additional verification produced if deemed necessary. Model coverage will provide an assessment of how effectively the higher-level requirements-based verification covers the functionality in the model. Note that in case derived requirements are identified in the model, and until these have been properly verified, there will not be full model coverage. Once the model has satisfied its higher-level requirements-based verification and full model coverage, code could be manually or automatically generated.

An Unintended Function is not just an error in the software, but also any additional software characteristics not contemplated in the system specification, and thus prone to produce unexpected behaviours. This uncertainty in the system performance introduced by the Unintended Functions, which may also lead to unacceptable hazardous events, prevents the system to be compliance with CS 25.1309 [RD.3] and consequently to get airworthiness type certification. Regarding UF detection on airborne software leveraging Formalized Designs, as it was released quite some time ago (in 1992), ED-12B/DO-178B does not provide sufficient guidance to cover all aspects of a model-based development. For this reason, Certification Authorities have developed additional guidance like contained in the EASA Certification Memorandum [RD.8]. Such guidance introduces means to detect Unintended Functions, in particular the model coverage activity. The criteria for model coverage provided in the Certification Memo appeared to be useful over the past years, however the need for alternative or additional criteria has never been investigated so far.

In case that the Validation and Verification activities are correctly carried out, it will be guaranteed that the model is compliant with the higher-level requirements and design standards, as well as requirements are

correctly implemented and also bi-directional traceability, requirements / design, is ensured. However there are some unintended behaviours that cannot be caught only by means of requirements-based tests, for instance, when there is additional functionality in the model —no initially included in the specification— that only is revealed under certain environmental conditions. This is exactly the point on which this study is focused, on the provision of appropriate model coverage criteria to detect Unintended Functions in the model, not easily detectable after performing other model Validation and Verification activities.

9.2.2. SOURCES OF UNINTENDED FUNCTIONS

The objective of this section is to identify possible reasons leading to the introduction of Unintended Functions in safety-critical software, and related with the use of model-based development techniques, mainly Formalized Designs. Our methodology started with the enumeration of possible sources of errors based on our past experience —like mistakes in the model, problems with the toolset, or when interfacing with external code— as well as those problems identified in the literature review. After this, we studied the different language constructions of Simulink, Stateflow and Scade language, trying to extract examples of obscure or error-prone features. The outcome of this research was a set of interesting considerations like the subtle semantics of some constructions, the importance of proper configuration of the toolset, the need to validate the model in the final platform, and the scalability of designs and the effort needed to understand and work with a complex design. Finally, other possible reasons were raised after different brainstorming sessions about Unintended Functions.

Two different aspects are considered in this section:

- Activities that could directly inject UFs into the system, development activities, covered by section 9.2.2.1.
- Activities aimed at detecting defects or errors in the specification and/or system, verification and validation activities, addressed in section 9.2.2.2.

Note that the defects not detected by the validation and verification activities would become UFs. In this context, all the verification activities defined in ED-12B / DO-178B are understood as filters to detect UFs.

9.2.2.1. UF Injection

Many possible sources of Unintended Functions related with the use of model-based development have been observed, and that could be introduced at different levels of the development process. Not all of them are exclusive to the use of MBD techniques, though, and many of the errors enumerated in this report are also possible with traditional development techniques. But even if the probability of introducing some mistakes are less probable that when using traditional techniques, they are still possible within an MBD context and thus the model coverage analysis criteria must be ready to address them.

Modelling mistakes

The key of model-based development is abstraction, where the engineer is just modelling relevant aspects of the system. Models are thus designed, and therefore can contain errors—even if formal methods are used to prove some system properties. These human mistakes could be introduced when writing a Formalized Design. Let us first enumerate some possible sources of Unintended Functions directly attributed to mistakes introduced by the designer in the model:

- **Wrong understanding of the requirements**, due to an incomplete specification of requirements or insufficient domain experience of the designer. This problem is not exclusive of MBD techniques, and can be mitigated by requirements-based tests verification, or a more complete and less ambiguous requirements specification.
- **Incorrect subsystem usage**, due to an insufficient description of the role of each input / output (like accepted units or valid ranges) or to wrong assumptions about the internal behaviour in non-nominal scenarios (robust results under extreme conditions). This could happen with standard basic blocks, but it is more common with complex ad-hoc compound blocks or super-states composed of different abstraction levels. An extra effort is required for the proper documentation of the model component to reduce this

type of problems, covering different aspects of the block like range of inputs, dynamic response, safety properties…

- **Wrong configuration** of parameterised components, like inappropriate default values. Large and complex model components could be seen as black boxes, and not enough attention may be spend in properly setting their configuration options. Input Ports must be always connected with an appropriate data flow, this restriction shows an unconnected port as an obvious problem. However, configuration parameters have less visibility than input ports, and sometimes can be forgotten to configure them properly because they use to have default values. The use of empty configuration parameters (instead of setting improper default values) guarantees that the designer will be warned by the tool when a missing value is required, and thus will be forced to think about the proper configuration.
- **System-level interactions** not taken into account, like unexpected propagation of errors among apparently unrelated components. This problem is related with unneeded coupling within a design, and is especially common when using non-structured language features (see section "Formalism or Toolset issues" below). When interactions exist in a design this must be made explicit in through visible connections showing the inter-dependencies.
- **Coupling of logical and numerical flows**, leading to different (and unexpected) behaviours under slight changes in the input values. Again, this is a problem related with the complexity of a design, and can be difficult to detect if the tested inputs are always deterministic. Robust handling of events / signals, and the use of randomised input values in the test harness can contribute to detect this kind of errors.
- **Assumptions in model reuse**. Usually some of the environmental assumptions taken for granted in the original project do not stand in this new project, forcing the model to work under unexpected conditions. The use of modular designs, extensive programming defensive techniques, and explicit documentation of the environmental assumptions are required to reduce the probability of facing these problems.
- **Partial use of existing block due to model reuse** from other projects, inheriting in the process more functionality than required for the current design. From DO-178B:

> ***5.2.4 Designing for Deactivated Code***
>
> *Systems or equipment may be designed to include several configurations, not all of which are intended to be used in every application. This can lead to deactivated code that cannot be executed or data that is not used. This differs from dead code that is defined in the glossary and discussed in subparagraph 6.4.4.3. The activities for deactivated code include:*
>
> *a. A mechanism should be designed and implemented to assure that deactivated functions or components have no adverse impact on active functions or components.*
>
> *b. Evidence should be available that the deactivated code is disabled for the environments where its use is not intended. Unintended execution of deactivated code due to abnormal system conditions is the same as unintended execution of activated code.*
>
> *c. The development of deactivated code should comply with the objectives of this document.*

According to these premises, the execution of deactivated code should be considered as an Unintended Function, and should be taken into account as a possible source of uncontrolled behaviour. Finding these kinds of errors is usually one of the main goals of model verification, however some of them always fail to be detected by the performed simulations and quality reviews. As said above, complexity and coupling are determining factors for introducing this kind of errors, as well as the need to work on teams composed of different designers. Even if a model prototype is designed by a single person or a core team, when this design must be refined it is very difficult to properly communicate to a larger team the original intentions and assumptions, leading thus to misunderstandings and modelling mistakes. These problems were also present in traditional development techniques, and were actually more common because MBD techniques are more adequate for handling with complex designs, but are still present.

Formalism or Toolset issues

Besides the specific errors directly related with mistakes in the design, there are also specific problems related with limitations of a modelling notation or problems with the toolset. As notations must be supported by a

toolset, models are very vulnerable to unexpected errors in these complex software packages, or limitations of the provided products (e.g. generated source code). Within this list we can find at least:

- **Error-prone language constructions**, leading to hard-to-detect flaws in the model. Poor practices of classical programming language have equivalent constructions in some modelling notations, like Data Store Memory blocks (public global variables), or From / To blocks (goto statements). In addition, data type mismatches are not always warned by the tool, resulting in wrong connections for example due to different units. The UFs coming from this source can be mitigated by the use of modelling standards or secure subsets for a given notation.
- **Non-formalized language semantics**, resulting in non-deterministic behaviours. Some basic blocks can also present a random behaviour unless used in a design under strict conditions, and the behaviour of the model can change with a different version of the tool or when using a code generator from a different vendor. This is not tolerable for high-integrity code, and different research efforts try to formalise the semantics of notation subsets.
- **Use of obscure tool features**, modifying the behaviour of the model in unexpected ways. For example, the precedence order is a state diagram can be linked to the position of the boxes in the diagram, thus small reorganisations of a working design can lead to a different behaviour [RD.14]. Or the differences in the behaviour of hierarchical models [RD.15], where is not the same to configure a subsystem as virtual (just an abstraction feature) than as atomic (were the whole subsystem must be executed as a single block before processing other blocks).
- **Formalism not adequate** for expressing some types of requirements, producing over-complex models leading to mistakes on the design. The ability to mix different notations in the same design, e.g. block diagrams and state diagrams, is a way to use the best formalism for each type of problem, but can also difficult the understanding of the design if the models in each notation are not modular enough, effectively resulting in another source of problems.

All these sources of Unintended Functions should be taken into account when choosing the specific toolset, and depending on the criticality level of the specific software component could severely limit the set of adequate notations. In addition, adequate modelling guidelines and a language profile should be adapted to the specific project, using DO-178B/ED-12B considerations as a starting point to avoid injection of unexpected functionality. Finally, it should be noticed that many of these sources of Unintended Functions can actually contribute to introducing or not detecting mistakes from category “Modelling mistakes” above.

Aspects external to the model

Another identified set of possible sources of Unintended Functions is related to external aspects to the model, like limitations in the development process, wrong assumptions about the platform or the environment, or to the need of communicating with non-modelled parts of the system:

- **Inappropriate selection of the modelled requirements**, not covering in the model some key aspects of the software or system. Not all requirements are modelled, usually because is not easily allowed by the notation, or some parts of the system are left out of the model to avoid overly complex designs; those functional requirements not considered in the model could introduce unexpected behaviours, like not considering the need to interact with those other system parts under non-nominal scenarios.
- **Inaccurate modelling of target platform characteristics,** like precision requirements not fulfilled by the embedded floating-point unit. The same test inputs usually lead to different results in the final target hardware than with respect simulations of the model, so the verification process must clearly specify when the differences are considered negligible due to the slight changes in the execution platforms. In addition, non-functional aspects of the system are also difficult to model. For example, as these formalisms tend to assume that operations are instantaneous, tests in the target platform must ensure that the execution time is within the limits.
- **Interfacing with components external to the model**, like hardware drivers from actual sensors / actuators or reused source code from old projects. Notations usually provide some gateways to interface with external entities, for example wrappers can be created to simulate the functionality of these external components. For example, a plain text file could simulate sensor inputs (instead of a complex hardware device) or the application configuration (which is actually handled remotely, and changed at runtime). Identification of safety and dependability barriers associated to these low-level functions is vital to be modelled in the wrapper. However, it is not possible to completely replicate the actual behaviour of those external components, and therefore the verification simulations performed are not completely accurate.

When interfacing with these external components the model should be especially robust to tolerate as many unexpected behaviours as possible, and a comprehensive verification in the final system is vital to properly test the actual behaviour of these components.

- **Synchronisation between the model and the generated source code**, especially during system maintenance. The need of interfacing the sources generated from the model against manually written source code for non-modelled parts implies that when the model is updated the newly generated sources must be carefully checked to ensure the changes do not introduce any improper interaction with those external parts of the code. It's important to clarify that, in case of use of automatic code generators, if the generator is not qualified, the source code generated needs to be verified and validated to ensure that no error has been introduced in the code generation.
- **Configuration Management of the modelling tools**, during all the life cycles of the project including maintenance. These toolsuites are complex software applications, and the possibility of introducing incompatible changes between successive versions must be taken into account. Slight changes in the simulations or code generator could lead to unexpected consequences. Therefore, modelling toolsets must be handled like other development tools like compilers, minimising version changes once the project has started.

As the whole system and environment is not completely modelled—either because is not feasible (due to limitations of the formalism, or to completely characterise some aspects of the environment) or not desirable (it was considered that there is no added value in modelling some parts of the system)—it is always required to properly verify the model in the actual target platform and to connect the generate code with other manually written parts. This leads to whole new categories of possible sources of Unintended Functions, and must be taken into account for properly ensuring the validity of the model, and a set of guidelines is required when interfacing with external parts of the model to be robust enough to handle unexpected interactions. It is thus important to characterise how a general model-coverage analysis criteria can identify the set of Unintended Functions caused by sources external to the model.

9.2.2.2. UF Misdetection

We have described some reasons or conditionals about the injection of UFs. But there is another source of problems directly related with the detection (or misdetection) of UFs. These issues cannot result in injecting an UF by themselves, but they can difficult the detection of UFs injected by other sources.

These problems are related to verification activities, as opposed to previous sources seen, which are related to development activities.

- **Incomplete validation and/or verification of the model**, because an inappropriate development process is followed. Too much confidence is placed in models that have not been properly validated and/or verified under adequate non-nominal scenarios or stress conditions, without comparing differences between the results obtained in the simulator and the target platform... This must be mitigated by detailed regulations and industry-wide development practices, like the MBD supplement in ED-12C.
- **Inadequate sample time** for the required resolution of a block, leading to imprecise results in the simulation. Especially problematic when testing non-nominal scenarios, avoiding to detect wrong behaviours under specific types of inputs. Designers dedicate a lot of effort in giving appropriate values to the step time of each block or subsystem and solver settings, and even after validation many design flaws can still be hidden in the model. The use of inherited sample time in some blocks can contribute to not detecting these incorrect values, until a change is required in the design during maintenance.
- **Inadequate configuration** of simulations, invalidating the performed model-in-the-loop simulations or process-in-the-loop tests. Toolsuites feature a wide-variety of configuration options (some of them can even overlap) not easy to understand for lay engineers. Therefore, it is not trivial to deploy adequate configuration values for a specific project. Configuration values like the simulation or step time can have dramatic effects in the behaviour and precision of the results, and should be adjusted by a well-trained engineer.
- **Bugs in the simulator**, effectively invalidating some tests performed to verify the model. These tool suites are extremely complex software programs, and especially when advanced or uncommon features are employed in a complex design we can expect that sometimes the simulation can give wrong errors. Developers must not be overconfident on the model-in-the-loop simulations.

9.2.3. TOWARDS A TAXONOMY FOR UNINTENDED FUNCTIONS

This section proposes a taxonomy of Unintended Functions. This exercise of classifying has been performed with the objective to better understand the main characteristics of the Unintended Functions, and to obtain feedback that could be valuable for the definition and challenge of the Model-Coverage criteria. Some key questions about Unintended Functions must be answered:

- What are they? Which is their impact on system performance?
- Which is their origin? What are conditions or elements that favour their appearance?
- How to detect them and also how to avoid their existence.

Since the SOMCA study is oriented to define an absolute and generic Model Coverage Criteria, applicable to any formalism used in Model-Based development approach, being able to provide a generic classification will contribute to the definition of this Criteria covering all the range of Unintended Functions, and it seems clear that if the Unintended Functions could not be classified, then a generic Model Coverage Criteria would hardly be found.

This classification exercise is supported by the analysis process of many aspects involved in Formalized Designs:

- Model-Based development methodologies: workflow, inputs and outputs of each stage, major activities performed during the software life cycle, essential processes supporting development (configuration management, Quality and Safety assurance, Validation and Verification…).
- Formalisms and toolsets supporting them.
- Knowledge of system architecture, complexity and coupling of functions or dependencies.
- System environment: operation concepts, HW platform (target / development), system interfaces and level of integration with other systems.

In short, this exercise demands a good understanding of system characteristics and the development strategy as well as the analysis process which is highly valuable for the subsequent definition of the Model Coverage Criteria.

Varied error taxonomies exist for specific software domains, and even multiple taxonomies are possible for the same types of software errors depending on the goal of the classification. As many software issues could be considered as an Unintended Function, and furthermore, error occurrences can be classified according to different points of view, it is also possible to create different sets of categories for the same group of Unintended Functions.

In the context of the SOMCA project, this report will consider the Unintended Functions in a Formalized Design with respect to its higher-level requirements. We have developed an initial taxonomy of Unintended Functions types with the purpose of properly defining the Model-Coverage Analysis criteria (MCA) with specific Unintended Function examples for all the categories identified. Under the following classification, there are five main types of Unintended Functions, which could be further refined into more specific categories.

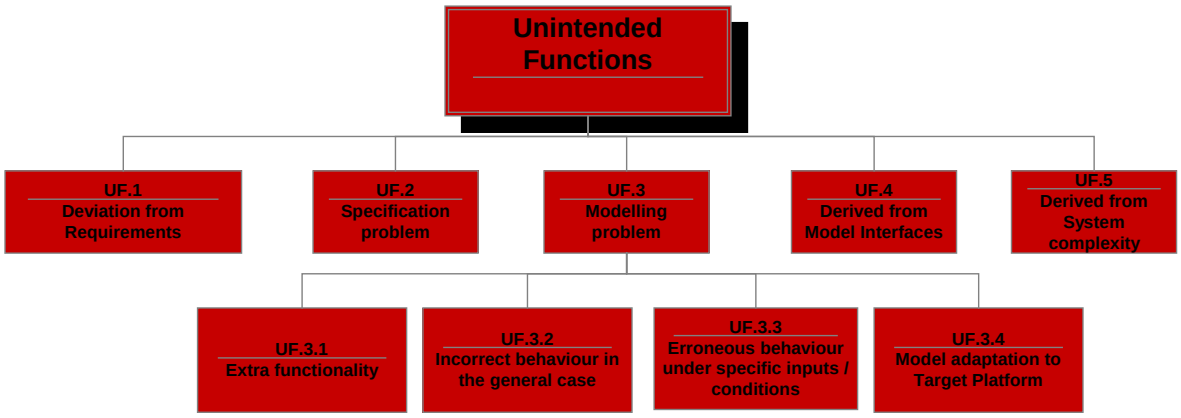


Figure 9-1: SOMCA Taxonomy of Unintended Functions

For specific examples of each category, please refer to the table at section 9.6.2.2.1.

- **[UF.1] Deviation from requirements in the implementation**: This is a modelling problem where the model does not represent the complete set of requirements. In this case the *specification is complete*. Verification should catch these kinds of problems, probably through requirements traceability. For this kind of UF, the design is not properly implementing the specification due to an engineers' misunderstanding of the requirements. This doesn't mean that the behaviour of the software will be useless, buggy or unsafe, but it will not follow the specification.

- **[UF.2] Specification problem**: The Model does represent the set of requirements, but the *specification is not complete*. The Model introduces derived requirements. This kind of problems can be detected by increasing coverage, or by model review, new derived requirements should be included in the specification.

  Requirements traceability will not detect this kind of Unintended Functions unless performed by an independent team that correctly interprets the requirements. However, if the original designers misunderstood the intent of the requirements, the risk of the verification team also failing to correctly interpret the specified behaviour should be taken into account. The use of Formalized Requirements is a way to reduce the ambiguity, and thus to prevent the misunderstanding of the requirements.

- **[UF.3] Modelling problem**: This category embraces many types of Unintended Functions. The common assumptions for all of them are the following: Correct and *complete specification*, including a *correct understanding of the specification*.

  - **[UF.3.1] Extra functionality**: The model implements *more functionality than specified* and this additional functionality should not exist. This Unintended Function can be mitigated through the use of *bi-directional traceability*.

    This kind of UF involve having more functionality than specified in the software low-level requirements, like **adding more features** in the design "to improve the system", or due to **code reuse** of existing library blocks. While all other Unintended Function categories are caused by mistakes in the design, this category could be intentional or unintentional. These unspecified features are outside the safety analysis of the system, and therefore can have safety implications.

  - **[UF.3.2] Incorrect behaviour in the general case** (model does not meet expected behaviour as specified in higher-level requirements). This kind of UFs should be easily detected in validation. The nature of these errors is variable: Design error or problem associated to formalism or tool, incorrect assumptions about the behaviour / allowed inputs / adequate configuration parameters. One possible mitigation strategy is to improve the verification.

In the case of Unintended Functions from this category the engineer correctly understands the requirements but introduced an error in the design, for example creating a block that does not completely follow its specification. The difference between this category and the next one, is that the behaviour is not the desired one in the general case, e.g. not enough precision of the outputs.

This kind of Unintended Functions can be generated because of an insufficient knowledge of the block or by an incomplete definition of the basic block behaviour or characteristics. To reduce the problems derived from UF.3.2, software engineers need a deep understanding of the problem domain, the formalisms and the tools used in the model definition.

The more detailed the specification of a given module the lesser the probability to face unexpected behaviours in an operational environment. This characterisation of a given block must describe the functional aspect as well as its performance and external interactions with other modules or the environment, including: static (input types and range), dynamic (delays inside the block), frequency and time answer in case of filters, stability conditions in case of transfer functions, expected output/behaviour for well known input patters (inputs like, step, saw, sine, etc.)...

- **[UF.3.3] Erroneous behaviour under specific inputs / conditions** (model meets expected behaviour as specified in the higher-level requirements and according to verification, but under some circumstances an incorrect behaviour can arise). Not easily detected by validation, probably unless non-nominal scenarios are defined. Possible causes: Design error or problem associated to formalism or tool, incorrect assumptions about the behaviour / allowed inputs / adequate configuration parameters.

  This category is similar to the previous one. Also, in the case of Unintended Functions from this category the engineer correctly understands the requirements but introduced an error in the design, for example creating a block that does not completely follow its specification. But for this specific case, the functionality fails just under specific conditions, like inexact results for some inputs or unexpected side effects into related blocks in a given system mode.

- **[UF.3.4] Model adaptation to Target Platform**: Hardware- and Operating System–related Unintended Functions.

  This is a wide category, including both the hardware and Operating System problems. They have been included in the same category because both make reference to the environment conditions in the target platform that are external to the model. It's important to remark that these environment conditions should be taken into account in the model design, but it's assumed that the conditions that could generate an Unintended Function are difficult or impossible to include in the model.

  These kinds of UFs are very hard to predict, being quite difficult to detect by MC analysis, and one key reason to require verifying the software in the final target platform.

- **[UF.4] Derived from Model Interfaces**: This kind of UF is related with the correct understanding and analysis of the system environment (interfaces). If the environment specification is not addressed correctly and completely some problems could arise at different levels: Incomplete / incorrect specification; inappropriate validation (deficiencies in test cases or inadequate definition of verification scenarios) or model implementation errors not considering the adequate environment interfaces. For any of these cases it is required to ensure that the Interface Specification is correctly addressed at all levels, but especially at model level input / output control should be also ensured.

- **[UF.5] Derived from System complexity** (highly coupled functions and data coupling problems).

  The complexity of the system to be modelled may derived in a complex model architecture where data or control functions are highly coupled. In this case, this concept is similar to the one defined in DO-178B / ED-12B, being data coupling related to the degree of dependence of a model component on data not exclusively under control of that model component. Additionally, we could think of control coupling as the degree by which one model component influences the execution of another model component.

The higher the degree of coupling the higher probability of facing unintended behaviours in case of implementation errors exists. See for instance the example of 'Highly coupled algorithmic architecture'.

Coupling derived errors could be found during requirements-based testing with an appropriate set of test cases where nominal and non-nominal verification scenarios are defined. Additional reviewing of test cases and results could improve coverage for avoiding these problems.

There are so many kinds of software flaws that can be left unspecified in the software requirements, that the number of possible types of Unintended Functions is actually unbounded. However, to properly exercise a set of Model Coverage Analysis criteria, it is important to test as many different types of Unintended Function as possible. To gain confidence on the selection of the specific problems chosen, we have developed a taxonomy of Unintended Functions so at least one type of each category is analysed. Even if this taxonomy does not completely cover all the possible problems that can be faced (which is not an easy task), the mind exercise would be useful at least to find new Unintended Functions.

Now that we have a better understanding of what types of Unintended Functions exists and their origin, the next sections will focus on the usage of Model Coverage Analysis for the UF detection at model level.

9.3. MODEL COVERAGE ANALYSIS

> *When drafting DO-178B/ED-12B, it was realized that **requirements-based testing cannot completely provide this kind of evidence with respect to unintended functions.** Code that is implemented without being linked to requirements may not be exercised by requirements-based tests. **Such code could result in unintended functions**. Therefore, something additional should be done since **unintended functions could affect safety**. A technically feasible solution was found in structural coverage analysis.*
>
> *The rationale is that if requirements-based testing proves that all intended functions are properly implemented, and if structural coverage analysis demonstrates that all existing code is reachable and adequately tested, these two together provide a **greater level of confidence that there are no unintended functions**.*
>
> —DO-248B, "*3.43 What is the intent of structural coverage analysis?*" [RD.6]

Coverage, in general, gives a measure of the completeness of a verification activity, and as such coverage metrics are usually a percentage of the activity accomplished. In the case of Formalized Designs, the following definition is proposed for the scope of the SOMCA project:

> ***Model Coverage** is the degree to which a Formalized Design has been exercised through Verification Cases and Procedures.*

The coverage of a Formalized Design can be performed both through simulations of the model, and by means of testing the derived code in the hardware platform. One of the key concepts of the definition is that the model has to be exercised through verification cases derived from the Higher-Level Requirements, as structural testing is not a good approach for UF detection. Indeed, if the simulation and test cases are derived just from the Formalized Design itself it cannot be detected if some requirements are not implemented in the model or if additional function/behaviour has been introduced in the Formalized Design.

The coverage metrics are obtained through a specific set of criteria:

> ***Model Coverage Criteria** is the set of coverage objectives that must be satisfied during the verification for different aspects of the model. Each coverage objective is defined according to the following characteristics:*
>
> 1. *Coverage elements: Components of the model that must be considered in the analysis*
> 2. *Coverage procedure: Establishment of a requirement to decide if a coverage element has been executed*

Finally, the coverage ratio is the number of coverage elements exercised with respect the total number of coverage elements in the Formalized Designs that need to be exercised to achieve the coverage objective. In

safety critical software the goal is to exercise all the source code, and thus the coverage ratio is always the 100% in this industrial domain.

An example of an MCA criterion could be:

- Coverage elements: transitions of the diagram
- Coverage procedure: all transitions have been triggered

This set of criteria can be specific for a modelling notation, or for a given DAL.

Finally, this leads to the definition of MCA [RD.8]:

> ***Model Coverage Analysis*** *is an analysis that determines which requirements expressed by the model were not exercised by verification based on the requirements from which the model was developed.*

Model Coverage Analysis implies analysis of the completeness of the verification of the model, and therefore verification of the model is thus incomplete until the whole model is covered. When the existing set of Verification Cases and Procedures do not cover the complete model it is necessary to perform an MCA resolution, which will either point to deficiencies in the higher-level requirements (the requirements from which the model was developed), the selected verification cases, or the Formalized Design. Depending on the conclusions of the analysis it may be required to improve the requirements specification, the number of verification cases, to justify deactivated parts of the model, or to remove an UF, iterating until the Model Coverage reaches the whole Formalized Design.

It should be noted that a model coverage analysis does not directly detect the Unintended Functions present in a given Formalized Design, besides some specific problems like the identification of dead code. At the end, the MCA is mainly an activity to increment the number of verification cases and procedures until the whole model is properly tested, and thus it is important to understand that UF occurrences are normally identified through the simulation and execution of the verification cases (or other techniques like static analysis) but not by the model coverage analysis itself.

The next subsections give some details about the coverage criteria analysed for each formalism.

9.3.1. ANALYSIS OF COVERAGE CRITERIA APPLIED TO BLOCK DIAGRAMS

The criteria defined for Structural Code Coverage in ED-12B/DO-178B are not directly applicable to the Block Diagram notation. In data-flow oriented notations there is no simple equivalence to “statements”, and actually a few input vectors usually exercises all the blocks in the model. The model elements that are considered for coverage are:

- *Model Block*: Both basic blocks (predefined in the notation) and model component blocks (block composed of other blocks) are considered.
- *Model Link*: The interconnection between a block output with one or more block inputs.
- *Logic Path*: The sequence of contiguous model links of Boolean type. The same model links can be part of different logic paths.

Please, note that “blocks” are commonly known as “operators” in SCADE, while “component blocks” are named “subsystems” in Simulink.

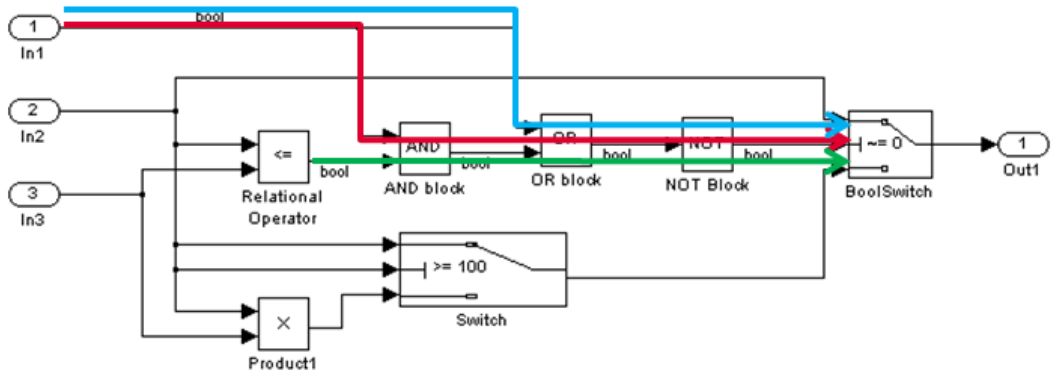


Figure 9-2: Example model showing different Block Diagram elements

The above example block diagram contains different blocks, links connecting the blocks, and three different logic paths (marked by the colour arrows). As can be seen, the links connecting numeric blocks (e.g. between the product and the switch block) are not logic paths because do not contain a Boolean signal. In addition, it is worth noting that the same link can be part of different logic paths, e.g. the link between the 'And' and the 'Or' blocks is part of two logic paths, while the link connecting the not block and the switch is the last link of the three logic blocks. The first link of the bottom logic path is the output of the relational operator, and is part of just that logic path.

A literature review was performed for Model Coverage technique applicable to block diagrams. Besides those works focused on the comparison of model coverage and structural code coverage [RD.16][RD.17], some researchers have proposed additional coverage criteria for data-flow notations. Specifically, a research project defined coverage criteria specific for the Lustre language, and thus applicable to SCADE and other Block Diagrams [RD.18]. These new Lustre criteria are relevant for the SOMCA project, and are also based on the masking effect (as the MTC Masking MC/DC described above). All of them are based on satisfying the "activation condition" of all the blocks in a logic path, to analyse whether a specific input vector activates a given logic path. That is, it is checked if the values of a specific input vector do not mask the input of a logic path so its value affects the last link of said logic path. If the block doesn't mask the logic paths that pass through the considered input, then the activation condition of that block is satisfied and therefore those links are activated for that block. If all the links of a path are activated, it is said that the input (the first link) of the logic path affects the output of that logic path (its last link).

9.3.2. ANALYSIS OF COVERAGE CRITERIA APPLIED TO STATE DIAGRAMS

The State Diagram notation cannot neither directly apply the source code coverage criteria. Namely, the following coverage elements are considered:

- *States*: **parent states** and **substates** should be exercised, as well as the **state actions** associated with them. Also, specific techniques are needed to exercise the state **History** mechanism, a feature that records the substate being executed when a parent state is exited, so when returning to the parent state the execution is resumed to the last active substate (instead to a default init substate).
- *Transitions*: transitions need to be triggered, the **transition decision** (composed of logical expressions) guarding each transition needs to be verified, and the **transition actions** associated with each transition should also be exercised.
- *Transition paths*: a **transition path** is a specific sequence of transitions, beginning and ending in specific states (which could be the same).
- *Events*: external **events** guarding a transition or a state action should be verified.

By definition a parent state contains different substates, and is active whenever one of its substates is active. But note that state diagrams are hierarchical, and a substate can also be a parent state if it contains new

substates. While in Stateflow the History mechanism is a property of a parent state, in SCADE the history mechanism is a property of transitions, not just recording the last substate that was active but also the whole contents of the memory operators implementing the state actions.

Each transition contains an explicit or implicit Transition Decision composed by a Boolean expression, and when this expression is evaluated as “true”, the transition is triggered. Transitions may also have associated a Transition Action, a set of statements that are executed when the transition is triggered. Similarly, a state (or substate) can also have associated a State Action, another set of statements that are executed when the state is active. Transition Decisions and State Actions can be guarded by Events, if the transition should not be triggered or the action not executed until the external event has been received.

The “event” element is a bit problematic, because there are several understandings and implementations of what an event is (also called signal) between different tools. This difference makes very complicate the definition of common criteria to cover them.

An event is a special variable that has the following properties:

- Is a Boolean variable: True or False (or Activated and Not Activated)
- The event can be emitted (set to true), but no deactivated (set to false).
- No matter how many elements in the model activate it at the same time, it’s only processed once (if several different model elements are reactive to them, all of them process the signal, but only once).
- The variable only remains activated (true) for one execution step. A signal emitted in one step is not active at the following one.
- A transition can have one or more events associated to it. The condition (if exists) is only evaluated when any of the associated events are received.

The elements categorized as events for different tools are the following:

- In Matlab Simulink, they are called “Events”, and they can be used only inside Stateflow, with some particularities. All the properties associated to block diagrams are not applicable to them. The behaviour of a Stateflow machine is associated to events and it should be used with care (this will be further explained in SOMCA Prerequisite 17: Requirement of Design Standard)
- In SCADE, they are called “signals”. They have some differences with the Simulink ones, and can be used both in Block Diagrams and State Machines.

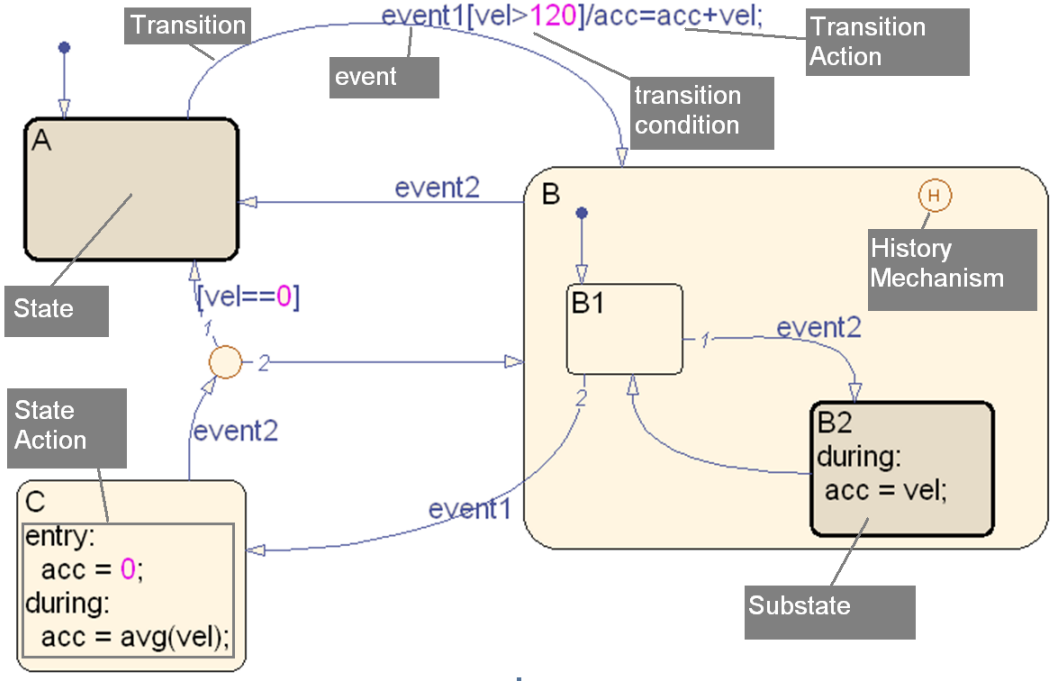


Figure 9-3: Example model showing different State Diagram elements

In the above figure the Stateflow model contains different transitions activated by two events (`event1` and `event2`). The transition at the top also has a decision (`vel > 120`) that must be true to be triggered, in which case the associated transition action (`acc = acc + vel`) will be executed before activating the state `B`. This parent state `B` uses the history mechanism (encircled H), and thus the precise substate activated (`B1` or `B2`) would the substate active when `B` was exited for the last time. Substate `B2` contains a state action (`acc = vel`), like the state `C`, which specify different actions when entering the state than when staying within the state. Finally, the state `C` has two exit transitions both triggered by the `event2`, the first one if `vel` equals zero, and the second one if `vel` is not equal to zero.

The following testing strategies for state machines have been found in the literature:

1. *All events testing*: each event of the state diagram has been exercised
2. *All states testing*: each state of the diagram is exercised at least once
3. *All actions testing*: each action is executed at least once
4. *All transitions testing*: each transition is exercised at least once
5. *All Round-Trip Paths*: every sequence of transitions beginning and ending in the same state is exercised at least once.
6. *Exhaustive testing*: Every path over the State Diagram is exercised at least once

The Exhaustive testing strategy can be considered equivalent to the path coverage criterion for source code, and due to the explosive number of combinations to exercise, this analysis is not practicable. And like Path Coverage, the interest of describing the Exhaustive testing technique lies in the understanding of the limited number of transition paths actually exercised by feasible coverage criteria with respect to the total number of possible combinations allowed in a State Diagram.

9.4. SOMCA MODEL COVERAGE

This section presents one of the main outcomes of the SOMCA Study, describing the selected SOMCA Model Coverage Analysis activity, including the motivations and justifications that led to the current proposal. The different criteria are based on previous work from the EASA Certification Memorandum [RD.8], a specialized literature review, the support in current toolsets for Model Coverage Analysis, and the experience gained analysing how to detect specific UF occurrences within a Formalized Design.

Besides selecting a given set of coverage criteria able to detect as much UF occurrences as possible, during the study it was decided to also list some other development and V&V activities not directly related with coverage analysis but that can improve the UF detection rate. We have three groups of activities:

- *Criterion*: a coverage objective for evaluating the completeness of the verification cases and procedures of a given Formalized Design. Part of the MCA criteria and must be enforced and analysed for all model elements for detecting specific UF types.
- *Prerequisite*: a verification or development activity that is applied to the Formalized Design before performing the Model Coverage Analysis, that is not enforced per model element but for the whole design or group of model elements (e.g. a component). Intended to be part of an additional verification activity related to MCA.
- *Recommendation*: a recommendation that could be useful to avoid introducing further some UF types, or to ease detecting them, but too prescriptive to be added as an MCA criterion or prerequisite. Cannot be considered a verification activity, and can refer also to recommendations related to the source code.

Therefore, before performing the Model Coverage Analysis it is required to ensure that the model conforms to the SOMCA prerequisites. This is required to ensure some specific UF types are identified. The list of recommendations is not required to be applied, are just best practices.

9.4.1. SOMCA PREREQUISITES

These prerequisites are part of the model verification, and could be applied at early phases of the project, as established in the goals defined at the beginning of section 9.4.3. The application of these prerequisites would help to detect different UF types, and this ensures that the MCA criteria focus on the detection of those UF categories not easily identified by other verification activities.

- **SOMCA Prerequisite 1: ED-12B Applicability**

  ED-12B guidance has been followed except for those activities related with Structural Code Coverage.

  For those activities described in the ED-12B but *not addressed* in this SOMCA Study, the ED-12B guidance should be followed.

- **SOMCA Prerequisite 2**: **Source Code Coverage**

  Imported source code *must be verified according to ED-12B.*

  The structural coverage of the source code included in the model is difficult to evaluate, because in some cases, the border between the model itself and the imported code is not clear, and can lead into confusion. For example, the actions defined in Stateflow are always typed manually, like short code snippets without flow control, directly inserted in the model. These statements cannot be considered as imported code but as part of the model.

  The source code coverage can be evaluated in two different ways:

  - Included in the model coverage during the execution of the verification cases, using SOMCA Criterion 4: Activation coverage (for short source code snippets without flow control).

- Obtained independently with external Structural Code Coverage tools during the execution of the verification cases (for imported source code, following traditional Structural Code Coverage techniques).

In order to fulfil this prerequisite, the procedure to evaluate the coverage of source code must be specified for each project, depending on the tools used and the applicable modelling rules. It's not necessary to apply the same procedure in the entire model, it can be chosen depending on the situations. E.g. Structural Code Coverage for external functions and state machine actions are considered covered by SOMCA Criterion 4: Activation coverage.

The main variable to elaborate this evaluation method is the tool chosen to develop the model. It would be interesting to generate predefined rules for each tool, but that is out of the scope of this study.

- **SOMCA Prerequisite 3: Model - Requirements Traceability**

All model elements must be precisely traced to specific higher-level requirements, or identified as derived requirements of a specific requirement.

Both studied tools have traceability support as part as the model definition. Depending on the tool, the traceability from the model to requirements can be a bit different. The trace should be as detailed as possible (limited by the detail of the requirements and the tool support for traceability).

- **SOMCA Prerequisite 4: Requirements – Model Traceability**

All higher-level requirements must be traced to model elements, precisely identifying the portions of the model that represent a given requirement.

In order to get bi-directional traceability, the trace must se also from the requirements to the model. This will allow obtaining a traceability matrix to see the requirements coverage.

- **SOMCA Prerequisite 5: Simulation configuration**

An adequate tuning of sample time and other simulation parameters has been performed.

The simulation parameters can alter the behaviour of the model if they are not properly configured. The configuration parameters must be specified and explicitly documented to avoid UFs coming from this potential source.

- **SOMCA Prerequisite 6: Modelling of Target Platform**

Specific simulations and tests should be exercised to ensure the target platform is accurately modelled.

The verification process must clearly specify when the differences obtained between the simulations and the tests are considered negligible due to the slight changes in the execution platforms. Differences in the results of a specific verification case between a simulation and a test usually indicate a model error. However, slight differences in the results are expected mainly when working with floating point values, so it must be specified when the differences are negligible.

- **SOMCA Prerequisite 7: Analysis of simulation differences**

A clear process must be defined for a Formalized Design or a model component to analyse when the differences between the results of a simulation and a test are considered a modelling error.

In many cases, there can be small differences between the simulation results and the final software running on the target. These differences must be explicitly documented to avoid UFs, e.g. if the arithmetic precision of the target machine is lower due to the resolution in the input values, the allowed precision loss must be specified.

- **SOMCA Prerequisite 8: Type check**

  There should be no data type mismatches.

  All data types must be explicitly specified to achieve the strongest typing possible. Once all types have been specified, a type check should be done over the model. If the tool allows it, the enumeration definitions should not be counted as normal integer values, and the connection of different enumerated values should generate an error. Automatic resolution of types is not allowed, and all types must be explicitly declared.

- **SOMCA Prerequisite 9: Uniform Model Verification Strategy**

  All model components shall be simulated or tested while exercising the complete Formalized Design.

  As far as possible, the verification and coverage results should be done in the model as a unique element [RD.15]. Partial model verification or coverage analysis should be avoided if possible, to avoid behavioural differences in the execution of independent model components.

- **SOMCA Prerequisite 10**: **Definition of Tool versions**

  The same version of the modelling tools, Formalized Design, and derived source code must be employed for all the design, production, validation and verification activities.

- **SOMCA Prerequisite 11: Tool and notation suitability**

  The appropriate modelling tool and notation must be evaluated for the project.

  Depending on the model purpose, there are some notations that suits better for each situation. The most appropriate notation should be used in each situation. E.g. an algorithmic data processing filter is easier to be done with block diagrams that with State Machines.

- **SOMCA Prerequisite 12: Transfer function stability**

  Stability analysis has been performed for transfer functions.

  Due to the complex behaviour of transfer functions, an analysis must be performed to specify restrictions or risks depending on the input data. See section 13.5.1 for an example.

- **SOMCA Prerequisite 13: Model documentation**

  All model elements should be properly documented, including range of inputs, dynamic response, non-functional requirements, and safety properties.

- **SOMCA Prerequisite 14: Barrier identification**

  Safety and dependability barriers should be identified and attached to external model entities.

- **SOMCA Prerequisite 15**: **Environment definition**

  The environment conditions must be explicitly documented, emulated, and verified.

- **SOMCA Prerequisite 16: Warning free simulation**

  All tool warnings during a simulation should be considered as errors except if the warning is properly justified.

Dangerous warnings can be raised silently during the simulations, hiding a potential unintended function that will remain in the target executable. Because of this reason, all simulation warnings must be considered errors. However, in some exceptional situations, it's possible that the warning is a false positive, or a known and desired behaviour. Under these circumstances, the warning can be justified and not removed.

- **SOMCA Prerequisite 17: Requirement of Design Standard**

  A design standard must be defined.

  Due to the big differences between different notations and formalisms used by each tool, and the purpose of modelled applications, it's not possible to define universal design rules or restrictions. Due to that, a design standard (including rules and restrictions) must be defined and used for all models.

  Some rule examples (they are only examples and should not be taken as a real restriction/rule):

  - The usage of early return logic in Stateflow is not allowed.
  - No local events can be used in any Stateflow diagram.
  - Weak transitions in SCADE cannot be used.

- **SOMCA Prerequisite 18: Explicit priority for Transitions**

  Explicit priority must be assigned to transitions. The usage of implicit priority based on position or display rules is not allowed.

  In the State Machine mechanism, the outgoing transitions of a state always have an associated priority to specify which condition the order of the evaluation of the conditions. Depending on the tool used, this priority can be specified in several ways. Usually, this priority is explicitly associated to the transition with a number specifying the order of evaluation. But in some cases, this priority is implicit, and the tool solves it in an undetermined way (creation order, graphical position, etc.). Due to the risks introduced by the uncontrolled resolution order, the priority must be always declared as explicit and displayed in the state diagram.

- **SOMCA Prerequisite 19: State Reachability**

  All States must be reachable at least by one transition.

  The initial state is not taken into account for this prerequisite. A state without input transitions will never be executed, and all the actions associated to that state and the possible outgoing transitions will end as dead code.

- **SOMCA Prerequisite 20: Recursive SOMCA Applicability**

  For all actions specified in a State Machine (both in States and Transitions) all SOMCA Criteria should be applied as for any other block.

  In the State Machines, Some actions can be associated to states and transitions. These actions are functionality that can be designed in several ways. Depending on the tool, you can associate a block diagram, a complete subsystem or some source code (e.g. Stateflow only allows source code associated as action to a state, but SCADE allows associating block diagram elements in the State as part of the State action). This difference forces the SOMCA Criteria to be applied in a different manner for each formalism.

  As no specific criterion for actions can be defined due to the flexibility of them, the best solution is to apply recursively all the SOMCA Criteria to the actions, using the appropriate criterion for each formalism used in the model. A recommended strategy is to define rules for this applicability in the Design Standard defined for SOMCA Prerequisite 17: Requirement of Design Standard.

- **SOMCA Prerequisite 21: Input / Output assertions**

For those components with restrictions to the input or output values, assertions should be included to control these restrictions.

For some components, there can be assumptions about ranges, or specific properties, like PRE/POST directives, in their input or output values. These kinds of assumptions should be checked if the tool used provides support for it.

- **SOMCA Prerequisite 22: External timing management.**

  The total execution time of model executions steps must be managed.

  It has been observed that some UFs are related with uncontrolled execution times that violate the timing constrains of the software, leading to an uncontrolled state of the system (time consumed, blocking calls, etc.). As the studied formalisms do not support this kind of restrictions over the execution time, they must be done outside of the model, once the source code has been generated, controlling the execution time of each model step to check against a given value and raise a warning or error in case that this barrier is activated.

  This management is a way to detect UFs once they have appeared, not a way to eliminate them. Even if this management is done, the model must be designed to match the timing requirements specified. In some cases, due to platform or specification restrictions, this management cannot be done. In these cases, a justification is accepted instead of the timing management.

9.4.2. SOMCA RECOMMENDATIONS

This section is a compilation of the modelling recommendations gathered during the study. They have been grouped into two lists: those aimed at reducing the introduction of Unintended Functions, and the recommendations to help detecting existing UF occurrences in a Formalized Design.

Note that these recommendations are not considered part of the MCA but “best practices” that would help to reduce the number of UFs.

9.4.2.1. Unintended Function Introduction Avoidance

- **SOMCA Recommendation 1**: Usage of Formalized Requirements avoids ambiguity and could reduce the possibility of introducing some UF types.
- **SOMCA Recommendation 2**: The set of modelled requirements should be suitable for modelling, and non-modelled requirements (i.e. not part of the Higher-Level Requirements) must be reviewed to ensure are compatible with the Formalized Design.
- **SOMCA Recommendation 3**: Automatic code generation could reduce the presence of some UF types.
- **SOMCA Recommendation 4**: Notations with formalized semantics are recommended since could avoid some UF types.
- **SOMCA Recommendation 5**: The most adequate notation should be used for each part of the model.
- **SOMCA Recommendation 6**: Modelling notations should discourage the use of non-structured notation features.
- **SOMCA Recommendation 7**: Modelling guidelines have to be defined to forbid the error-prone features of a specific notation.
- **SOMCA Recommendation 8**: The software designers should receive adequate training about the problem domain, notation, and modelling tools.
- **SOMCA Recommendation 9**: Consider whether a configuration value should be an explicit block input or a default configuration property of the block (if the notation allows it) initialised to a safe value, which could lead to the introduction of Unintended Functions.

- **SOMCA Recommendation 10**: Component interactions should be clearly identified in a Formalized Design.
- **SOMCA Recommendation 11**: Formalized Designs intended to be reused should be modular, document explicitly the environmental assumptions, and exhibit extensive defensive mechanisms.
- **SOMCA Recommendation 12**: A Formalized Design should reduce coupling.
- **SOMCA Recommendation 13**: A Formalized Design should limit complexity.
- **SOMCA Recommendation 14**: Proper data types should be chosen to accurately represent the handled values, specifically adequate data units and format must be selected.
- **SOMCA Recommendation 15**: Prototypes of Formalized Designs should be documented in detail to properly communicate the original intentions and assumptions when refined into a final design.
- **SOMCA Recommendation 16**: All changes in the model must be identified, and carefully translated into changes into the source code.

9.4.2.2. Unintended Function Detection

- **SOMCA Recommendation 17**: Usage of Formalized Requirements could detect subtle errors thanks to formal analysis tools and other checkers.
- **SOMCA Recommendation 18**: Executable models can help in the correct design of a model.
- **SOMCA Recommendation 19**: Fuzz testing could be employed with all model components.
- **SOMCA Recommendation 20**: Modelling guidelines should be automatically checked for the model.
- **SOMCA Recommendation 21**: Static analysis can detect data type mismatches and potential precision loss in a Formalized Design.
- **SOMCA Recommendation 22**: Formal Methods could be used as additional measures to verify some safety properties.
- **SOMCA Recommendation 23**: As much verification cases as possible should be executed in the final platform (and optionally simulated too).
- **SOMCA Recommendation 24**: Include debug elements as extra blocks / transitions instead of as an additional conditions within an existing decision.
- **SOMCA Recommendation 25**: For those tools that aren't qualified, the use of any optimisation is not recommended.

9.4.3. SOMCA CRITERIA

Finally, this subsection describes the SOMCA MCA criteria, which is one of the main outputs of the study. Each criterion is a coverage objective for evaluating the completeness of the verification cases and procedures for a given Formalized Design. Must be enforced and analysed for all model elements for detecting specific UF types. The criteria have been refined several times during the present study, and the output of this refinement is included in this version of the document as the final proposed Model Coverage Criteria. Different goals were self-imposed at the beginning of the study for the definition of the SOMCA MCA criteria:

1. *UF detection*: The coverage criteria must be powerful enough to identify as many UF as possible, giving confidence about the intended behaviour of the model and the completeness of the verification process.
2. *Complete model*: The criteria should cover the whole model, including complex notation features and imported elements.
3. *At model level*: The complete MCA criteria can be applied before the source code has been developed, being able to verify a Formalized Design at early development phases.

4. *Scalability*: The verification cases required to accomplish the whole criteria must not be too time-consuming. The MCA should not be a burden for the developer, even when applied to complex Formalized Designs.
5. *DAL specific*: It was expected that each Design Assurance Level will require a different set of model coverage criteria, to balance the effort required to verify a model with respect to the criticality of the model and the consequences of an unidentified Unintended Function.
6. *Easy to learn*: When choosing between different criteria with a similar degree of UF identification, this secondary goal should prefer the easier to understand to the developer. The next sections will analyse different coverage criteria and testing strategies based on previous work, and finally the proposed SOMCA MCA criteria will be defined considering the above goals and considerations.

9.4.3.1. Range Coverage

SOMCA Criterion 1: Range coverage

All the significant values of the inputs and outputs of each model component must be exercised.

This criterion includes the following:

- All singular points of the functional components and algorithms
- All equivalence classes (valid/in-range and invalid/out-of-range classes), including internal data types
- Continuous and discontinuous input signals, including transitions between the maximum and minimum in-range values and periodic signals (e.g. angle between [0 .. 2 · pi)).

This criterion is intended to be applied at model-component level, i.e. the analysis of the different input vectors should not be done for each basic block but for a the inputs and outputs of a state machine or complete block component encapsulating a specific functionality. In the case of analysing general criteria over State Diagrams some clarifications should be done:

- The singular points that must have taken into account are the ones associated to all operations over the input parameters of the state diagrams. These operations include both the conditions associated to each transition and the ones included in the actions associated to transitions and states.
- The case of equivalence classes is the same as for the singular points.

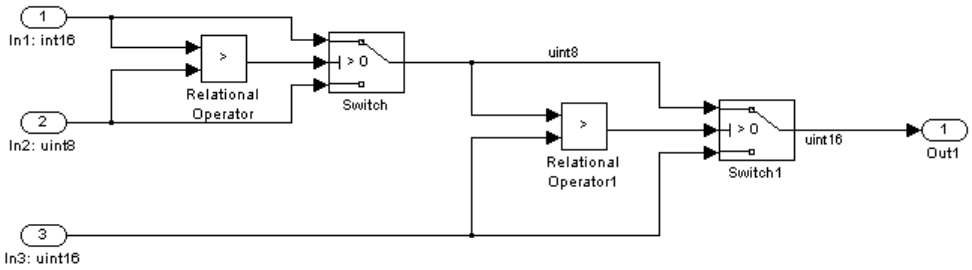


Figure 9-4: Range Coverage criterion example.

For example, consider the above model component: a 3-input Max block. To satisfy the Range coverage criterion, it is needed to analyse the inputs and outputs of this component (called “subsystem” in Simulink):

- *Singular points*: there are no singular points; all input numbers behave as surrounding values.
- *Discontinuous input signals*: for example, sawtooth signal between the minimum and maximum values for the each input.
- *Equivalence classes*: Different equivalence partitions have been taken into account as explained bellow.

If the equivalence classes are constructed following the criterion (i.e. the boundaries of internal computations need to be taken into account), the following partitions are considered for each port:

- *Input 1*: besides the range of the int16 input signal (from −32768 to 32767), the analysis yields that the input can be squeezed into an uint8 link (from 0 to 255). This should raise and alarm, and detect an UF present in the model (in some situations, the input 1 will be saturated to the maximum value that can hold in a uint8 type, i.e. 255). However, let's continue with the application of the whole Range coverage criterion for illustrative purpose. In this case, 5 partitions are considered: [−∞ .. −32769], [−32768 .. −1], [0 .. 254] [255 .. 32767], [32768 .. ∞]
- *Input 2*: just the boundaries of this uint8 signal must be considered, so [−∞ .. −1], [0 .. 255], and [256 .. ∞] are the 3 partitions considered.
- *Input 3*: this input is always routed through uint16 links (from 0 to 65535), so again just its boundaries are considered in the equivalence classes, i.e. the 3 partitions [−∞ .. −1], [0 .. 65535], and [65536 .. ∞].
- *Output*: Even if the output has type uint16, intermediate computations can be performed with types uint8 and int16, and therefore the 6 partitions are [−∞ .. −32769], [−32768 .. −1], [0 .. 255], [256 .. 32767], [32768 .. 65535], and [65536 .. ∞]. However, the two partitions of negative values can never be satisfied because the link has an unsigned type, nor the last partition because the signal is just 16-bit wide.

Note that when out-of-range values are provided as inputs, extreme in-range values will be provided because the model is configured with saturation semantics.

Now, different number of verification cases can be selected, depending on the test set selection policy [RD.22][RD.23]. If the minimal test set is selected, a minimum of max(5, 3, 3, 3) = 5 verification cases are needed, so for example the following input vectors cover all the equivalence classes of the 3 inputs and the output (but the out-of-range output partitions):

- T1: { −32769, −1, −1} = 0
- T2: { −32768, 0, 0} = 0
- T3: {0, 0, 300} = 300
- T4: {255, 255, 32768} = 32768
- T5: {32768, 256, 65536} = 65535

This criterion has been derived from the Cert Memo, but note that the Range Coverage criterion has been extended to detect further UF types.

9.4.3.2. Functionality Coverage

SOMCA Criterion 2: Functionality coverage

All characteristics of the functionality in context must be exercised for each component.

This criterion includes:

- Activation of characteristics (e.g. Watchdog function is triggered)
- Reaching internal conditions (e.g. saturation block fed with an input over the upper limit)
- Stability of transfer functions (e.g. standardized input signals for transfer functions: step wave, sine wave, square signal, and sawtooth wave)

The specialisation of the Functionality Coverage criterion for different element types is intended to ease the coverage assessment, due to the wide range of possible ways to classify the functionality of a model component. This criterion has been derived from the Cert Memo.

For example, consider the functionality provided by a limiter block with dynamic upper and lower limits. Besides the different input values (nominal, above the higher limit, below the lower limit), other situations need to be considered, like the transition modifying the upper limit while.

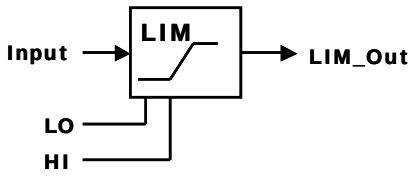


Figure 9-5: Functionality Coverage example.

9.4.3.3. Modified input coverage

SOMCA Criterion 3: Modified input coverage

All inputs of every block and state diagram have changed at least once.

This criterion tries to find problems in the integration of different components or blocks. It is required for all inputs to change at least once during the verification cases, or to be justified.

Please, note that Modified input coverage is not equivalent to Range Coverage, because Range Coverage will not force to exercise different values if there is just one equivalence class. In addition, this criterion applies to every single basic block, while the Range Coverage criterion is intended for the inputs *and outputs* of each *model component*. Also, the Modified input coverage applies to Boolean values too.

For example, in the next state machine the DebugMode value never changes because it is connected to a constant. In some cases it is needed to add a justification why that constant is required, but in this example the criterion finds a debug element that the designer forgot to remove from the model.

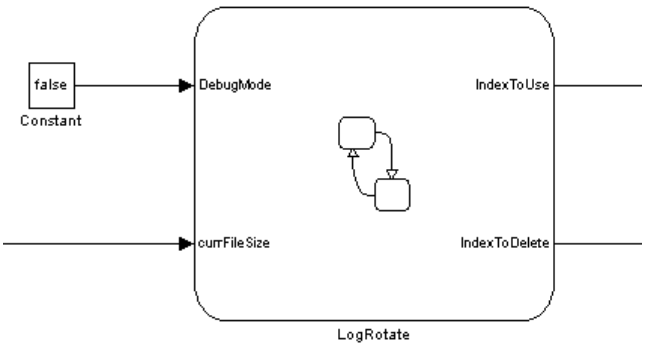


Figure 9-6: Modified input coverage example.

9.4.3.4. Activation Coverage

SOMCA Criterion 4: Activation coverage

All model elements whose execution depends on an external rule or signal must be activated.

This criterion covers:

- Enabled components: The signal that enables the block execution must be activated.
- State machine actions: Those actions in a State Machine associated to specific event (e.g. entry, during, exit) must be executed.

This basic criterion covers both State Machines and Block Diagrams, and the differences between both of them could be enough to divide it into two different criteria. It has been keep as a single one because the final objective is the same in both situations: to check that all elements of the model have been executed.

For example, in the next SCADE model the `inc` block is wrapped within a conditional block, and thus the inc block will be executed just when the activation decision is True (when input signal 'in' is less or equal to 7). Therefore, to satisfy the Activation coverage this conditional block needs to be enabled at least once during the execution of the verification cases, otherwise the `inc` block would never been exercised. An equivalent example in Simulink would involve enabled subsystems.

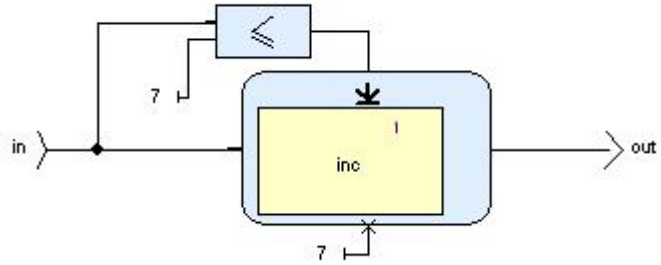


Figure 9-7: Activation coverage example.

9.4.3.5. Local Decision Coverage

SOMCA Criterion 5: Local Decision Coverage

Every block decision and Boolean output of a basic block has taken on all possible values at least once.

This criterion applies the classical Decision Coverage to all blocks in a block diagram that depends on a Boolean expression, including:

- **Logical operators**: the output of each logical operators has taken both the True and False values (e.g. and, or, xor, not...)
- **Relational operators**: the output of each relation operator has taken both the True and False values (e.g. equals, greater than, less than...)
- **Switch blocks**: the decision of the switch / if block has taken both the True and False values, activating both inputs.
- **Selectors**: the decision of the selector has taken both the True and False values, enabling the processing of the contents of each branch.

For example, in the next model, during the execution of the verification cases the local outputs of each of the 4 logical blocks is recorded. The output of both relational blocks and the output of the 'nand' operator have taken both Boolean values, satisfying the Local Decision Coverage for these three blocks. However, this criterion detected that the output of the 'and' operator has just taken the True value, due to a missing case of the scenario. This was resolved adding a new verification case to exercise that situation.

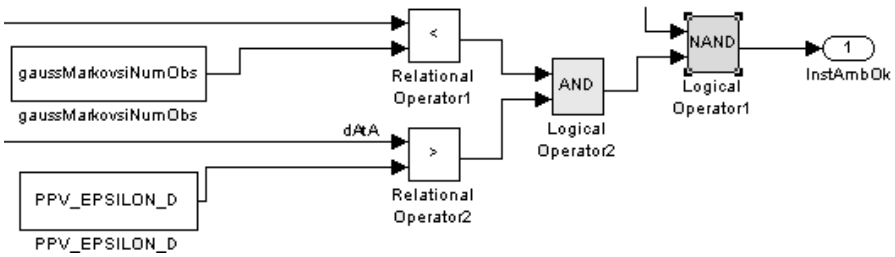


Figure 9-8: Local Decision Coverage example.

Note that in this case the Modified input coverage criterion will also detect the problem because one of the inputs of the 'nand' block would never change. But in other cases the inputs can change while the decision can remain the same, like with relational operators, or the decision of a switch or selector.

The Local Decision Coverage criterion is also applied to selectors because the Activation Coverage criterion only requires executing all model components of activate constructions, but does not require that the construction is at least once *not* activated. For example, in the following SCADE model the Activation Coverage criterion will be satisfied if the first two branches are activated, but it does not require activating the last branch. However, the Local Decision Coverage will detect that the second decision (`A > 0`) has never taken the False value. This is equivalent to statement coverage vs. decision coverage from Structural Code Coverage, where statement coverage is a basic coverage criterion while decision coverage is meant for higher assurance levels.

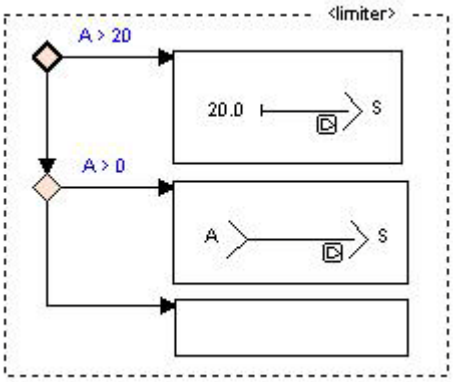


Figure 9-9: Local Decision Coverage example (selector).

9.4.3.6. Logic Path Coverage

SOMCA Criterion 6: Logic Path coverage

The input of every logic path has been shown to independently affect the output of the logic path.

For satisfying this criterion it is necessary to verify that a change in the input of a logic path modifies the output of that path when all the links of the path are active (i.e. no block masks any of the links of the path). This requires at least two different verification cases where all links of each logic path are active, and also that the output link has both the True and False values.

For example, consider the diagram from section 9.3.1, and let's focus on one of the three logic paths. To satisfy this criterion for the middle logic path (starting at the In1 port, passing through the 'and', 'or', 'not' block, and ending at the decision of the switch block), we require exercising two different input vectors for In1, In2, and In3 to yield both the True and False values at the decision of the switch block, while no block masks any of the input links of the logic path.

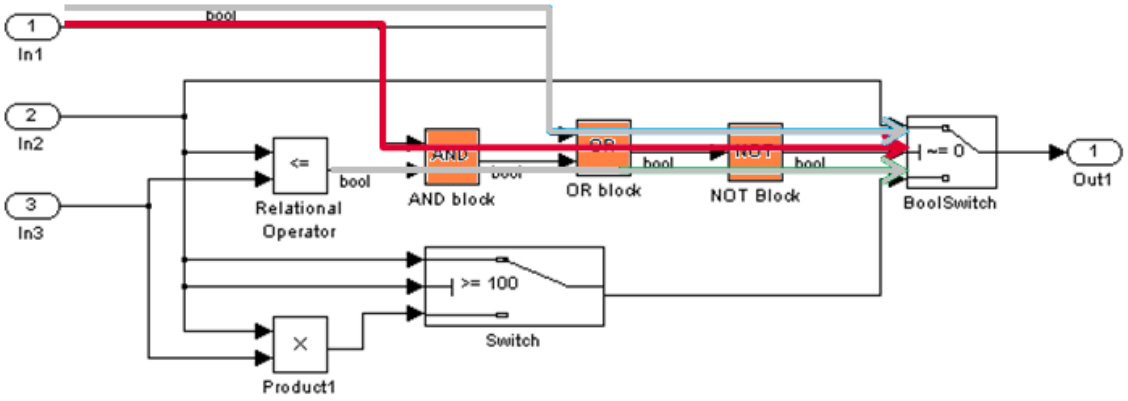


Figure 9-10: Logic Path Coverage criterion example

For example, in the next table we evaluate the data flow for the input vectors {In1=False, In2=0, In3=0} and {In1=True, In2=0, In3=1}, showing in italics the inputs and outputs of each block involved in the logic path. In this case the first input vector activates all the links of the middle logic path, resulting in a True value at the end of the path. However, for the second input vector the OR block masks the middle path (the True value at I1 for the OR block will always result in a True value, regardless of the value of I2), so even if at the end of the logic path the False value is get it is not enough to satisfy the Logic Path Coverage criterion for this path. Usually, not satisfying the criterion means modifying a verification case or adding an additional verification case to exercise that part of the model, but actually in this case it is not possible to satisfy the criterion because the False output can be obtained just when the path is not active.

Input ports			Relational block			AND block			OR block			NOT block		Switch
In1	In2	In3	I1	I2	O	*I1*	I2	*O*	I1	*I2*	O	*I*	O	*I2*
False	**0**	**1**	0	1	True	*False*	True	*False*	False	*False*	*False*	*False*	*True*	***True***
True	**0**	**1**	0	1	True	*True*	True	*True*	**True**	***Masked***	True	True	False	**False**

Table 9-1: Activation example middle logic path.

9.4.3.7. Parent State Coverage

SOMCA Criterion 7: Parent State coverage

All states and substates have been entered and exited (except for those without exit transitions), and all substates have been active at least once when parent state exits.

The Parent State Coverage criterion not only requires that all substates have been activated, but also that the parent state (which contains different substates, and thus are active whenever one of its substates is active) has been exited when each specific substate was active. This tests the behaviour of the state machine when the parent state is interrupted at every possible situation. It is worth noting that some parent states do not have a global exit transition but each sub-state can have specific transitions to other states, and thus this criterion does not apply to those sub-states.

For example, consider the next Stateflow model with two concurrent state machines. The 'On' state contains to substates ('TempOK' and 'TempNeedsChange'), which can terminate and transition to the PressOff state whenever the Button is pressed. Thus, to satisfy the Parent Coverage criterion the 'On' state has to be exited from both substates.

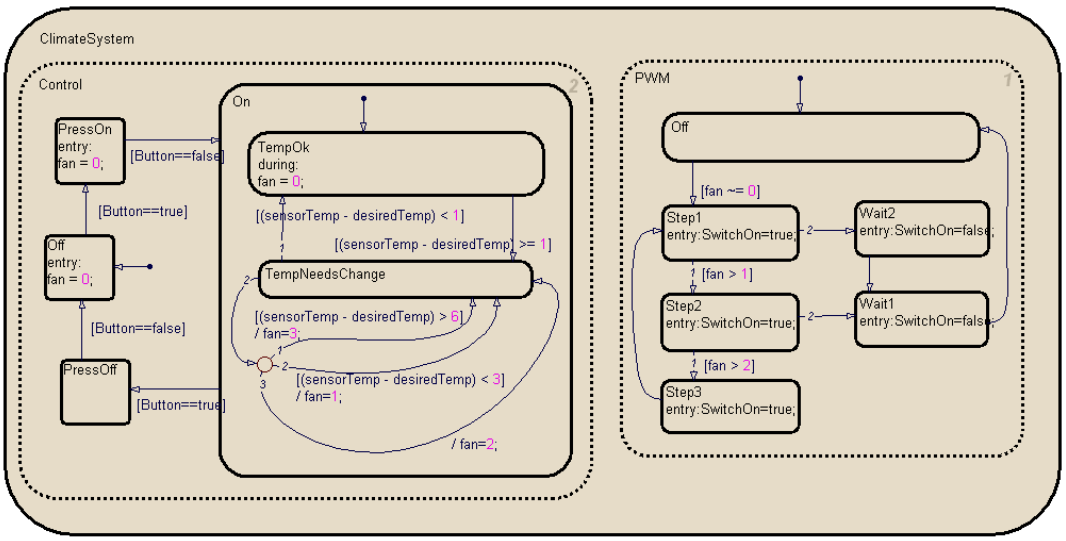


Figure 9-11: Parent state coverage example.

9.4.3.8. State History Coverage

SOMCA Criterion 8: State History coverage

All states and substates have been entered and exited, and all substates that were active when parent state exits have been later re-entered.

The State History Coverage criterion is similar to the Parent State Coverage one, but also requires that the parent state is later re-entered and the last active substate reactivated. This is aimed at testing the history functionality present in different state diagram notations. Of course, for states without an exit transition it is not required to show that the state has been exited.

For example, in the next Stateflow diagram the State History Coverage criterion requires both:

- The `Sender` state is exited while in the `Waiting` substate, and later re-entered from the `RequestActive` state.
- The `Sender` state is exited while in the `Transmitting` substate, and later re-entered from the `RequestActive` state.

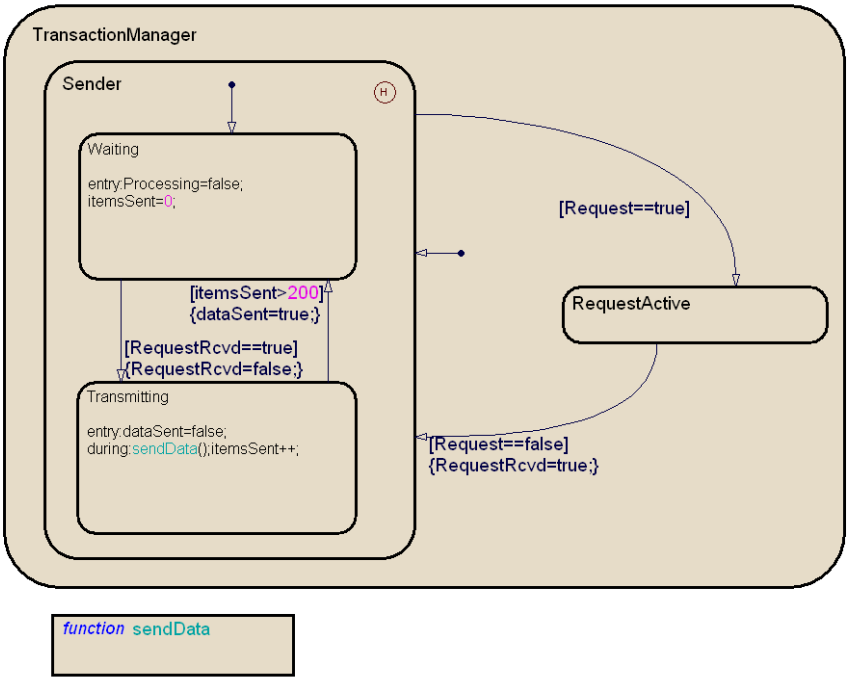


Figure 9-12: State History coverage example.

9.4.3.9. Transition Coverage

SOMCA Criterion 9: Transition coverage

All transitions of the diagram have been exercised.

This criterion checks the coverage over all the transitions in the model. During this study, it has been seen that when SOMCA Prerequisite 18 and this criterion are met, all states have been also covered, and it isn't necessary to check it by Criteria.

For example, consider the following model, which contains a state machine with some debug elements that the designer forgot to remove. In this case, the additional constant (`False` block), input port (`DebugMode` input), and transition (exit transition 1 of state `NeededNewFile`) controlled by the constant, allows easily experimenting with certain functionality of the state machine instead of waiting during hours until the exact conditions occur nominally. The Transition Coverage criterion would easily identify the extra debug elements of this example: Either the `DebugMode` is set as True so both the transition 2 and 3 in the diagram will never be exercised (just the transition 1, triggered when `DebugMode` equals to 1), or the constant was set as False so the debug transition would never be triggered.

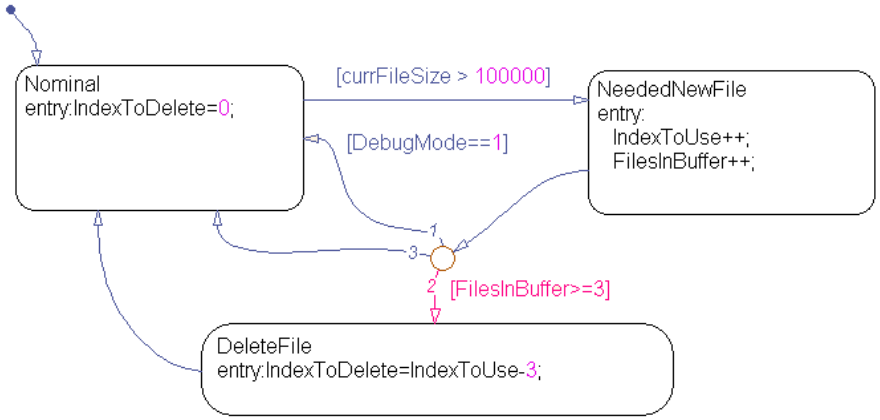


Figure 9-13: State machine with extra `DebugMode` transition.

Note that this UF could be also found thanks to the **SOMCA Criterion 3: Modified input coverage**, because the DebugMode input never changes, generating a coverage problem.

9.4.3.10. Transition Decision Coverage

SOMCA Criterion 10: Transition Decision Coverage

Every decision in the transition decisions has taken all possible outcomes at least once.

This criterion improves the coverage provided by the previous one by checking the decisions associated to the transitions.

Consider the state machine example of the Transition Coverage criterion, but with some modifications. Now there is no extra debug transition and the DebugMode is evaluated in the last exit transition of the state NeededNewFile. In this case, when the input constant is False the Transition Coverage criterion will not detect the extra debug element because all the transitions will be fired at some point. However, the decision involving the DebugMode input will always be True, and thus the Transition Decision Coverage won't be satisfied because this transition decision has never taken the False value, which would detect the UF.

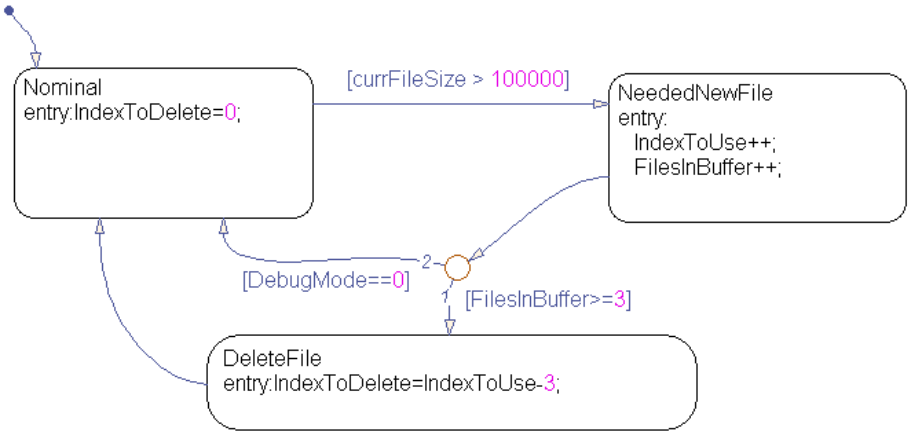


Figure 9-14: State machine with extra `DebugMode` decision.

9.4.3.11. Transition MC/DC

SOMCA Criterion 11: Transition MC/DC

Every condition in a transition decision have taken on all possible outcomes at least once, and every decision in a transition decision have taken all possible outcomes at least once, and each condition in a transition decision has been shown to independently affect the transition decision's outcome.

This is the adaptation of the classic Modified Condition / Decision coverage for the State Machines formalism. This criterion affects all the conditions evaluated in all the diagram transitions and it's only applicable to State Machines and not to Block Diagram formalisms.

The application of MC/DC on transition conditions is very important and necessary for High Software Assurance Levels (DAL-A) because these kind of conditions are evaluated almost in the same way as in source code. These similarities allow that applying MC/DC to transitions at model level provides similar coverage than the traditional code coverage techniques. However, the relationship between the coverage level provided by transition MC/DC applied at model level and the one provided by MC/DC over the generated code based on the model is not direct, so no extrapolation can be done between these two different coverage criteria (it's not assured that source code coverage provides model coverage and vice versa).

Consider a modification of the state machine from the previous section. Now suppose there is no extra debug transition, but the DebugMode input is added to the decision of an existing transition to enforce taking or not a specific transition (see next figure). In this case, if the DebugMode is set as True then that transition will never be taken, and thus the UF will be detected too by means of the Transition Coverage criterion. However, if the DebugMode constant has with the value False (which is more probable, otherwise the difference in the system behaviour would be easily noticed during the requirements-based tests) both transitions will be executed nominally so would not be detected by Transition Coverage criterion. In that case, Transition MC/DC will detect that the extra DebugMode condition never changes, identifying the problem.

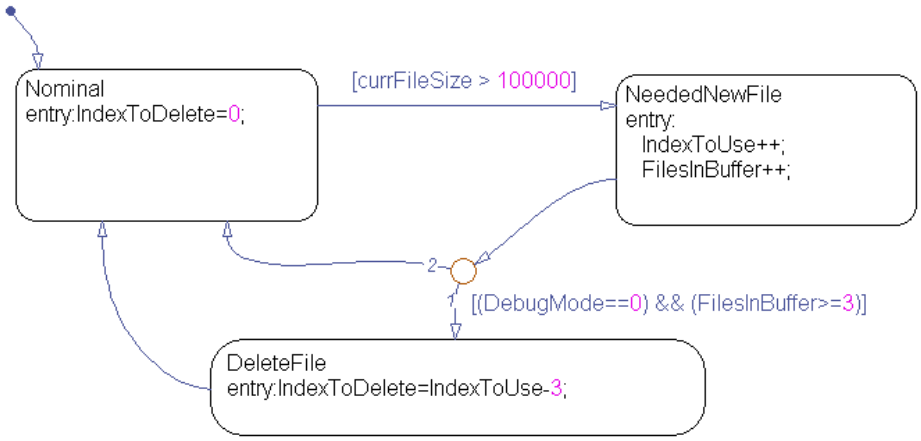


Figure 9-15: State machine with extra DebugMode condition.

9.4.3.12. Event Coverage

SOMCA Criterion 12: Event coverage

All external events are received at least once in each state that has transitions associated to them.

This criterion covers the reception of events when there are transitions associated to them, providing coverage over the events that trigger them. It has been separated from the condition and decision evaluation because the events have particularities that must be evaluated independently.

For example, to satisfy this criterion in the next SSM both the `'Start` and `'Force` events must be received while in the Red state, even if just the reception of one of them is enough to satisfy Transition Coverage if the decision `in < 100.0` is also satisfied.

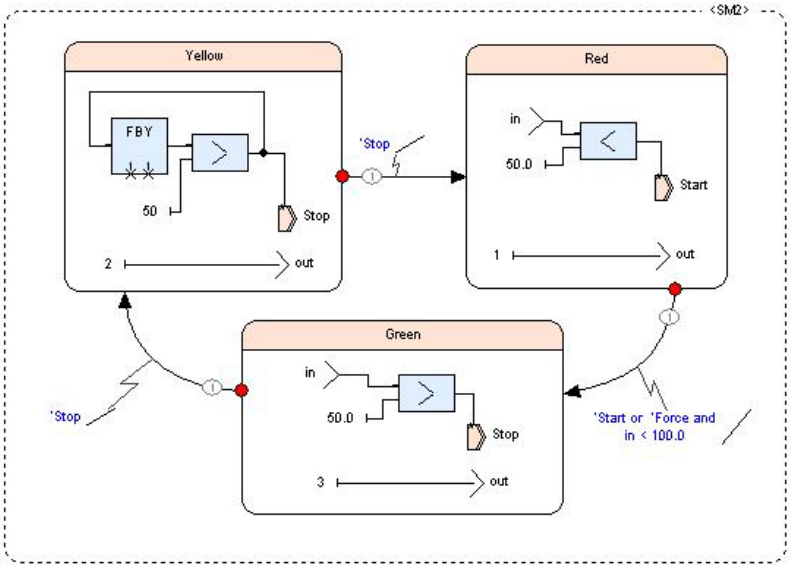


Figure 9-16: Event coverage.

Please note that this hole in the verification can also be detected by the Transition MC/DC criterion, however this criterion is meant just for high assurance levels while the Event Coverage criterion is designed for lower levels.

9.4.3.13. Activating Event Coverage

SOMCA Criterion 13: Activating event coverage

All external events activate a transition at least once in each state that has transitions associated to them.

This criterion is similar to the previous one, but adds the requirement that the event received activates the associated transition at least once. This criterion only requires that the event activates each transition associated to it. It doesn't require both cases (received and activated, and received but not activated) for each event and transition (activated and not activated). This means that the event is not taken as another Boolean value in the condition associated to it. This is done to prevent a high number of combinations in the verification cases needed to fulfil this criterion, it also improves compatibility between different formalisms. The main difference between this criterion and the previous one is that this one requires that the received event triggers the transition. Consider the next Stateflow example with two events: `ONOFF` (user button) and `UPDATE` (temperature has changed). All transitions are guarded by a single event, except the one to the `Overheat` state, which is implicitly evaluated when *any* event arrives. Therefore, to satisfy the Activating Event Coverage criterion at this transition both the `ONOFF` and `UPDATE` events must have been fired this transition. This showed a hole in the verification because just the `UPDATE` event was tested during the verification cases to trigger this transition.

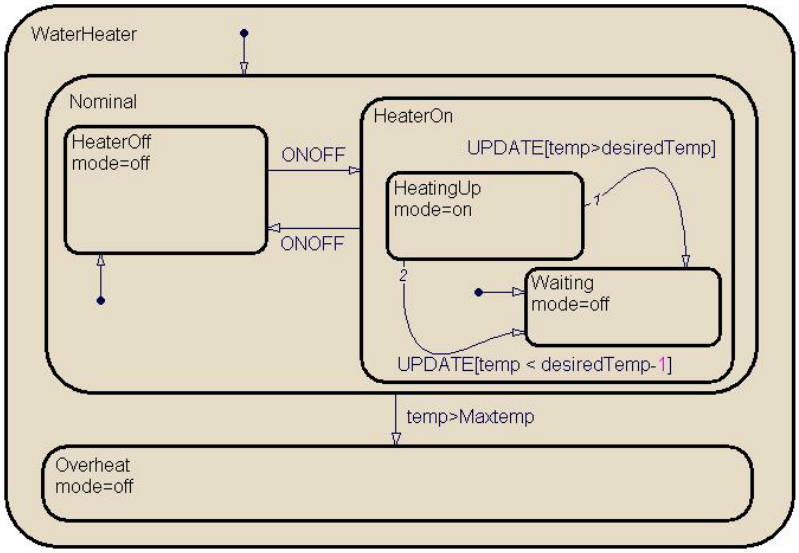


Figure 9-17: Activating event coverage.

This concept will be explained in a more detailed way with an example in the Assessment of SOMCA MCA Criteria (see section 9.6.1.8: Water Heater Control).In the definition of the criterion, we remark that the events checked are not taken as Boolean values. In this example, the problematic transition will be a perfect example of this simplification.

If we consider all events as Boolean variables (true or false), the condition of the transition to *Overheat* state will become the following expression:

(ONOFF || UPDATE) && (temp>Maxtemp)

If we analyze this transition with the Transition MC/DC criterion, the test cases needed are the following:

ONOFF	UPDATE	ONOFF || UPDATE	Temp > Maxtemp	Result
False	False	False	True	False
False	True	True	True	True
True	False	True	True	True
True	False	True	False	False

However, If the events are not taken as Boolean values, but only as indicated in this criterion, the result is the following (the condition will be analysed with transition MC/DC also):

ONOFF	UPDATE	Temp > Maxtemp	Result
-	Activated	False	False
-	Activated	True	True
Activated	-	False	False

The events and conditions are only taken into account when they are activated, simplifying the analysis process. It also remove the necessity of having scenarios where no events are activated (this is impossible for some formalisms, like Stateflow).

9.4.3.14. Level-N Loop Coverage

SOMCA Criterion 14: Level-N Loop coverage

All loops of depth N, with same inputs values maintained for a given number of iterations (m), present in the model have been executed.

A loop of depth *N* in the model is a transition path of length *N* that starts and ends in the same state. The "static input" qualifier refers that the loop is present for a fixed combination of input data. The cycle must be completed without any change of the input signals to the model.

The states machines are not usually designed to execute transitions every cycle. These kind of non-stable behaviours are usually produced by an error in the model. The objective of this criterion is to detect those internal loops that could lead to unstable situations of the model outputs for the same inputs. In some cases, these loops are part of the nominal behaviour of the state machine. The criterion doesn't force the removal of these kinds of loops, only their inclusion in the verification procedure. This criterion is similar to the All Round-Trip Paths testing strategy found in the literature, as stated in section 9.3.2.

If the tool allows the usage of events, the analysis gets more complicated. In this case, they are not taken into account in the application of this criterion.

The application of this Criterion needs a previous identification of the model loops. There is not yet an automatic method to find them, but they maybe could be found by arithmetic resolution of the transition condition combinations. The typical size of this kind of loops is 1, 2 or 3. The depth of the analysis will depend on the DAL level.

Sometimes, a loop can exist in the model but it's impossible to reproduce due to the preconditions on the input data or the model inputs. In these cases, a justification is needed to fulfil this criterion.

The number of iterations necessary to check that the loop has been covered (*m*) depends on several variables, and it should be determined specifically for each model. Typical numbers are 10, 50 or 100 iterations, but it can be higher or lower.

Let's see an example:

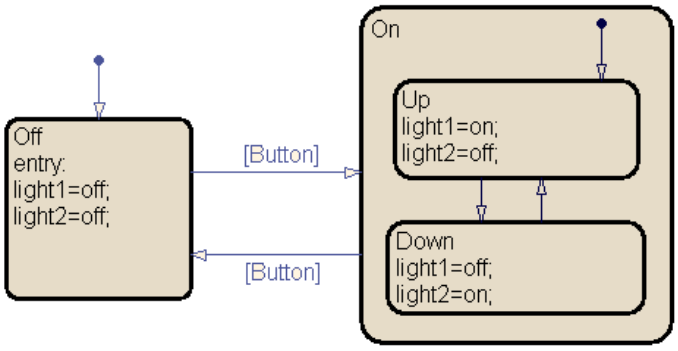


Figure 9-18: Loop Criteria Example

This figure presents a very illustrative example of function loops. The desired behaviour of this state machine is to enter state `On` when `Button` is pressed, and switch to `Off` after the button is pressed again. During the `On` state, the lights 1 and 2 blink alternatively.

The definition of a loop is "*A loop of depth N in the model is a transition path of length N that starts and ends in the same state".* In this case we have two level 2 loops:

- If the `Button` input variable is true, a loop activates between `Off` and `On` states, generating an UF that makes that when you press the button some time, you never know in which state you will end.
- If Button is false and the state `On` is active, we can find a loop between `Up` and `Down` states, because the transitions have no condition associated and are always executed. This is an example of a loop that is *not* an Unintended Function, because that's desired behaviour.

The main objective of this example is to show that a loop is not an undesired element in the model, only a risky one, and this criterion needs that all model loops have been executed (not eliminated) to check if the behaviour is the desired one.

9.4.4. CRITERION APPLICABILITY FOR EACH SOFTWARE ASSURANCE LEVEL

Now, once all the criteria have been defined, it is proposed to apply different set of criteria for each software criticality level in the following way. This tries to balance the number of verification cases required with the criticality of each software DAL.

Criterion Name	Block Diagrams DAL					State Diagrams DAL				
	A	B	C	D	E	A	B	C	D	E
SOMCA Criterion 1: Range coverage	√	√	√			√	√	√		
SOMCA Criterion 2: Functionality coverage	√	√	√			√	√	√		
SOMCA Criterion 3: Modified input coverage	√	√	√			√	√	√		
SOMCA Criterion 4: Activation coverage	√	√	√			√	√	√		
SOMCA Criterion 5: Local Decision Coverage	√	√				NA	NA	NA	NA	NA
SOMCA Criterion 6: Logic Path coverage	√					NA	NA	NA	NA	NA
SOMCA Criterion 7: Parent State coverage	NA	NA	NA	NA	NA	√	√			
SOMCA Criterion 8: State History coverage	NA	NA	NA	NA	NA	√				
SOMCA Criterion 9: Transition coverage	NA	NA	NA	NA	NA	√	√	√		
SOMCA Criterion 10: Transition Decision Coverage	NA	NA	NA	NA	NA	√	√			
SOMCA Criterion 11: Transition MC/DC	NA	NA	NA	NA	NA	√				
SOMCA Criterion 12: Event coverage	NA	NA	NA	NA	NA	√	√	√		
SOMCA Criterion 13: Activating event coverage	NA	NA	NA	NA	NA	√	√			
SOMCA Criterion 14: Level-N Loop coverage	NA	NA	NA	NA	NA	√ (3)*	√ (2)*	√ (1)*		

(* The number in brackets means the N level used in the coverage criterion)

Table 9-2: SOMCA MCA criteria

The SOMCA Criterion 14 is applicable for A, B and C DAL levels, but not in the same way. For each criticality the depth level that should be used to apply the criteria is different, e.g. For DAL-B, the criteria should be applied with level 2.

9.5. MCA SUPPORT IN COMERCIAL TOOLS

Nowadays, commercial MBD toolsets support Model Coverage for both Block Diagrams and State Diagrams as a basic verification activity. Each toolset provides a different set of coverage criteria, although both the criteria offered within the SCADE Suite and Simulink / Stateflow are mainly based on those from ED-12B for Structural Code Coverage, namely Decision Coverage and MCDC.

This section first describes an overview of the predefined criteria offered by the most widely-used toolsets, and also analyses the support for the SOMCA criteria in these commercial packages.

9.5.1. SCADE MTC

The SCADE Suite, by Esterel Technologies, provides the Model Test Coverage (MTC) tool for assessing the coverage of the model. The SCADE MTC tool offers two operation modes: Batch mode and Interactive mode; the purpose of interactive mode is to understand and refine the Verification Cases, while batch mode chain is qualified and intended for producing the final coverage report for the model. In addition, the SCADE MTC provides direct support for coverage resolution through “justification records”, an user-introduced textual rationale used to force the coverage of a given criterion for a set of model elements.

9.5.1.1. Block Diagrams

The criteria provided for Boolean blocks are different from those offered for numeric elements. For Boolean basic blocks (predefined operators) the predefined criteria include:

- **Operator Decision Coverage**: The Boolean output of a block has taken both possible outcomes (SCADE MTC handles the outputs of comparison operators (<, <=, !=, ==, >=, >) as decisions).
- **Operator MC/DC**: MC/DC is applied locally to the inputs and output of a basic block. Therefore, if a decision is composed by the results of different blocks, the masking effects of a block are not taken into account for the coverage of the previous blocks.
- **Masking MC/DC**: The input of a specific Logic Path affects its output. In this case, the Masking Effects of a block are considered, like in one of the MC/DC variants for Structural Code Coverage (see section 12.). Thus a Boolean block is covered just if its output is not masked by the next blocks. Thus, a connected set of Boolean blocks is covered when all their logic paths haven taken both the True and False outcomes.

After the introduction of Masking MC/DC and the possibility of globally checking MC/DC for a compound network of Boolean blocks, the Operator MC/DC criterion is considered obsolete because it analysis is local to a single Boolean block. These criteria are applied to the following Boolean blocks: AND, OR, NOT. It also applies to the switch (If...Then...Else) and case blocks and to init if the inputs are Boolean. N-gates AND and OR operators are supported..

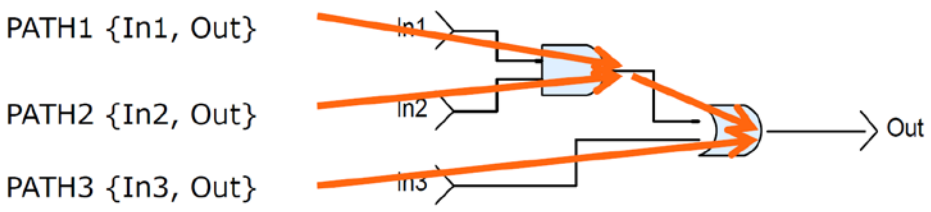


Figure 9-19: Masking MC/DC in SCADE (courtesy of Esterel Technologies)

For numeric blocks, there are basic predefined criteria defined that check whether the inputs and outputs of the block change, but the user can define his own criteria by defining the conditions to be checked during testing.

These custom coverage criteria are typically used for ensuring that all the functionality of a given block has been exercised, e.g. reaching the upper and output limits and the reset operation of a block.

9.5.1.2. State Diagrams

The SCADE Suite provides in the MTC tool provides predefined criteria for State Diagrams, and also allows the user to define custom criteria as for Block Diagrams. The predefined coverage criteria for Safe State Machines can be categorised into:

- State coverage: All the states have been active
- Transition coverage: All the transitions have been triggered

Decision Coverage and MC/DC criteria can also be applied to transition actions and transition conditions. This ensures the complete activation of states, activation of each decision path in a transition, as well as the activation of the actions within states and transitions, and the activation of transitions conditions. MTC does not provide predefined support for the History mechanism.

9.5.1.3. Integration Coverage

In addition to the coverage provided for Block and State diagrams, SCADE offers an additional set of coverage criteria called “Integration Coverage”. These criteria try to cover the possible errors introduced due to the integration of operators in higher level ones. In this category two criteria can be found:

- Control Activation: Covered when the instance has been activated.
- Data Activation: Covered when each input has been changed.

The main difference between the Integration coverage and the other criteria is that the coverage is measured for each instance of the block, instead of measuring globally as happens for the rest of criteria.

These criteria have different details on its behaviour depending on the type of the block that is going to be analysed. Iterators and polymorphic operators are examples of these criterion specializations.

9.5.1.4. Custom Coverage Properties

In addition to the already existent coverage criteria provided by the SCADE Suite, additional custom properties that are included in the coverage analysis can be defined for any operator. This properties must be defined manually (using also block diagrams and state machines).

9.5.2. SIMULINK VALIDATION AND VERIFICATION TOOLBOX

The MathWorks markets the “Simulink Verification and Validation” toolbox add-on for measuring the model coverage of simulation cases. This optional 1st-party toolbox is applicable to both Simulink models and Stateflow diagrams. Other third-party tools exists for measuring the coverage of Simulink and Stateflow models like Reactis Tester, which provides a slightly different set of coverage criteria [RD.16].

The “Simulink Verification and Validation” toolbox provides some support for coverage resolution: Some elements can be removed from the coverage analysis, which can be used to justify deactivated components/blocks/transitions.

It is worth noting that the reports also specify if some optimisations that could affect the coverage results were enabled during the simulation of the model. For example, the “block reduction” option allows the simulator to replace a set of block by an equivalent simplified block (or remove them entirely if detected as dead blocks) which would alter the coverage of those blocks.

9.5.2.1. Block Diagrams

The following coverage criteria are implemented in the Simulink Verification & Validation toolbox (the last two criteria are available since the Matlab 2010b release), and any combination of them can be activated globally for a given model:

1. **Decision coverage**: The Boolean expression of a block has taken both possible outcomes
2. **Condition coverage**: All the Boolean inputs of a logic block has taken both possible values
3. **MC/DC**: MC/DC is applied locally to the inputs and output of a basic logic block
4. **Look-up Table coverage**: Values accessed in a look-up table block
5. **Signal Range coverage**: Maximum and minimum output numeric values of a block
6. **Design Verifier coverage**: All the user defined checks introduced in the model have been satisfied.

The last coverage criterion is optional, related with the Design Verifier package, an independent Simulink toolbox to perform static and dynamic analysis within a model, usually with the help of user-defined annotations through verification blocks (named test conditions/objectives, proof assumptions/objectives). If this package is installed in addition to the Simulink Verification & Validation toolbox, the user can define her own coverage criteria thanks to the availability of the test objective blocks. If these expressions are satisfied by the Verification Cases, the Design Verifier coverage criterion will be considered as covered for these user defined criteria.

MC/DC in Simulink is equivalent to 'Operator MC/DC' from SCADE MTC, where the analysis is applied locally to logical blocks, and there is no equivalent criterion to 'Masking MC/DC' in the V&V toolset. However, even if these criteria are applied to a single bock, logical operators in Simulink can have an arbitrary number of inputs, so it would be equivalent to applying the criterion to multiple connected logical operators when all the blocks are of the type (e.g. if two 2-input And gates connected to a 2-input And gate are converted to a single 4-input And gate the coverage analysis would consider all 4 inputs).

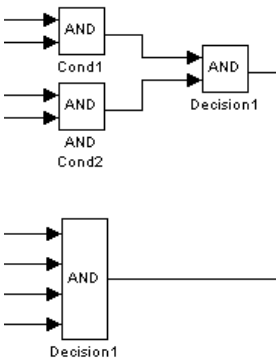


Figure 9-20: The three AND gates can be converted to a single AND block

However, when the decision is composed by expressions of different logical operators it is not possible to get the same as Masking MC/DC (e.g. two 2-input And gates connected to a 2-input Or gate cannot be converted to an equivalent 4-input logical operator, so the coverage of each block would be local to each one of the three blocks).

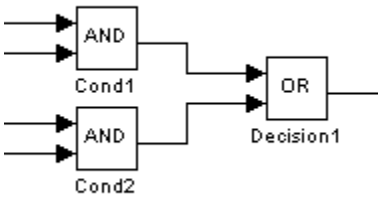


Figure 9-21: The coverage analysis would be local to each block

Condition Coverage and MC/DC can be applied to Logical operators (And, Or, Xor, Not, Nand, Nor, Nxor) and module-enable blocks (enabled and triggered subsystems in Simulink terminology). The Condition Coverage criterion can also be applied to the Combinatorial Logic block (truth tables).

The Decision Coverage does not apply to the Logical operators, but can be checked for the Combinatorial Logic block and the module-enable block, as well as to other Boolean blocks like the If block and the While Iterator block. Interestingly, Decision Coverage in Simulink is not just applied to logic blocks, but also to those predefined library blocks that internally contain a Boolean expression, for example:

- Abs block: Has the input received a negative value?
- Dead Zone block: Has the input value between the lower and upper limits?
- Discrete-Time Iterator: Has the external reset input been activated? Has the input value reached the upper/lower limits?
- For Iterator: Has the iteration value above the iteration limit?
- Min block: Has each input port received the minimum value?
- Rate Limiter: Have the rising and falling slew rate been reached?
- Relay: Has the input signal reached the switch on point?
- Saturation block: Has the input ever reached the upper/lower limits?

Relational operators (==, ~=, <, <=, >, >=, IsInf, IsNaN, IsFinite) are not considered to contain a decision or a condition in the Simulink Verification & Validation toolbox, even if these Boolean expressions are defined as a condition in ED-12B. Therefore, the output of relational operator blocks is not analysed in the coverage reports, regardless of the activated criteria (including DC, CC, nor MCDC). To check whether both outcomes have been produced for these blocks, their Boolean output has to be connected to another block checked for coverage, like a logical block (which is usually the case in a design). Other possible approach is to introduce a test objective block at the output of the relational operator (see next figure), annotating that both the {true, false} values are expected to be obtained in the simulations, which will be analysed in the report if the Design Verifier coverage criteria is active.

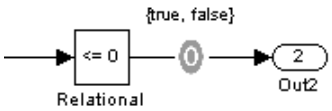


Figure 9-22: 'Test objective' annotation to analyse the output of a relational block

Relational operators are typically the beginning of Logic Paths (the output is the first link of the path, usually connected to a logic block), and thus is important to analyse if both Boolean outcomes have been produced. It is surprising that the Decision Coverage criterion applies to numeric blocks (like the Abs block), but no criterion can directly analyse the behaviour of relational operator blocks.

9.5.2.2. State Diagrams

The Simulink Validation & Verification toolbox provides specific coverage criteria for Stateflow's charts embedded in Simulink models. However, the naming of these criteria is the same as for Block Diagrams, and thus the objective of each criterion can be misleading:

- Decision Coverage: This coverage option actually activates four different criteria.
  - All the transitions have been triggered
  - All the states have been active
  - Exited from all the substates when the parent state exits
  - All previously active substates have been entered due to the history mechanism
- Condition Coverage: All the conditions of a transition decision have taken both possible outcomes
- MCDC: All the conditions within a transition decision have taken both possible outcomes, and have been shown to independently reverse the outcome of the transition decision

9.5.3. SUPPORT OF SOMCA CRITERIA BY COMERCIAL TOOLS

The coverage criteria currently included in commercial tools do not fully support the SOMCA Criteria, but has some common points and can be used to evaluate some of them. Table 9-3 shows how the evaluated tools (Simulink and SCADE) support the proposed SOMCA criteria with their own built-in Coverage Analysis. The Support can be classified as:

- *Compatible*: The tool already includes an equivalent coverage criterion that is the same (or almost the same) than the proposed criterion. The name can be different.
- *Partially Compatible*: The tool has a similar built-in criterion which results almost cover all the functionality of the SOMCA Criterion. Most of the times, the results of the built-in criterion can be analysed to generate the SOMCA ones.
- *User-defined check*: The tool doesn't support the criteria, but it allows defining manually some properties that can be analysed during the model coverage to generate the SOMCA criterion results.
- *Not supported*: The tool does not support the SOMCA Criterion.

Criterion	Simulink Support	SCADE Support
SOMCA Criterion 1: Range coverage	**User-defined Check** User criteria can be defined for each operator with Verification and Validation toolbox. Signal range analysis provides partial support for this criterion, but no check is done.	**User-defined Check** User criteria can be defined for each operator
SOMCA Criterion 2: Functionality coverage	**Partially Compatible** Some library blocks includes in the condition coverage an small functionality coverage (like abs block) Also user criteria can be defined for each operator with Verification and Validation toolbox.	**User-defined Check** User criteria can be defined for each operator
SOMCA Criterion 3: Modified input coverage	**Not supported**	**Compatible**
SOMCA Criterion 4: Activation coverage	**Compatible** Activation parameters are checked with Decision Coverage.	**Compatible**
SOMCA Criterion 5: Local Decision Coverage	**Compatible**	**Compatible**

Criterion	Simulink Support	SCADE Support
SOMCA Criterion 6: Logic Path coverage	**Not supported** Local MC/DC is applied to all blocks, that can be equivalent to this criterion under some circumstances, but in general the criterion is different.	**Compatible** Provided by Masking MC/DC
SOMCA Criterion 7: Parent State coverage	**Compatible**	**Not supported**
SOMCA Criterion 8: State History coverage	**Compatible**	**Not supported**
SOMCA Criterion 9: Transition coverage	**Compatible**	**Compatible**
SOMCA Criterion 10: Transition Decision Coverage	**Compatible**	**Compatible**
SOMCA Criterion 11: Transition MC/DC	**Compatible**	**Compatible**
SOMCA Criterion 12: Event coverage	**Partially Compatible** Covered by Decision Coverage	**Partially Compatible** MTC can check if a signal has been emitted. Reception of the signal is covered by MC/DC
SOMCA Criterion 13: Activating event coverage	**Partially Compatible** Covered by Decision Coverage	**Partially Compatible** MTC can check if a signal has been emitted. Reception of the signal is covered by MC/DC
SOMCA Criterion 14: Level-N Loop coverage	**Not supported**	**Not supported**

Table 9-3: Support of SOMCA Criteria by Simulink and SCADE

This partial support of the proposed criteria will help in the application of it to the proposed examples (both real and theoretical).

9.6. ASSESSMENT OF SOMCA MCA CRITERIA

This section exercises the SOMCA model coverage criteria with different Unintended Functions types. To evaluate the effectiveness of each criterion, MCA has been performed on a wide variety of UF examples using the SOMCA criteria. These examples have been selected from the different categories of the UF taxonomy, with the intention of ensuring that the proposed Model Coverage criteria are able to detect a wide variety of Unintended Functions.

This section is organised in the following subsections:

- Illustrative examples: A few indicative UF examples are presented and analysed with the SOMCA criteria, evaluating whether the criteria and prerequisites were able to detect the UF.
- Assessment results: A summary of all the UF examples is provided, indicating the detection rate and effectiveness of each SOMCA criteria and prerequisite.

9.6.1. ILLUSTRATIVE UF EXAMPLES

This section contains a few complete examples about applying the SOMCA MCA Criteria. Only the most illustrative examples have been selected, including typical occurrences of unintended functions that can be easily identified as well as some cases that cannot adequately detected through simulation. For additional examples, including examples for every category of the UF taxonomy, please refer to appendix 13.

9.6.1.1. Software Shutdown Button

In the frame of the operational systems, it's common to have a software shutdown button that allows to shutdown the system in a controlled way. This section will describe several different models that implement this functionality is different ways, presenting several UFs caused by different reasons.

The example of the behaviour of a Shutdown button is very common in operational systems and critical systems. This kind of buttons used to have several requirements associated to them, so it seems to be a good example to be analysed.

This example contains three versions of the same State Machine. The differences are very few, but all of them present a different UF. For each one of them the UF will be analysed independently. They don't follow a strict evolutive process; the different injected UFs are independent.

9.6.1.1.1. Forced shutdown not included in the Requirements

The figure included in this section shows the implementation of a Start / Stop button that controls the main switch of a small embedded system. The main objectives of this State Machine are the following:

- Power on the system when the button is pressed
- When the system is on and the button is pressed more than 100 milliseconds, send a signal to the software to shutdown.
- Once the software has finished and confirmed the shutdown request, power off the system.

But during the model implementation, someone adds an additional feature to this small state machine: As a emergency mechanism, if the button is pressed for more than 5 seconds, the system is powered off without software confirmation. This new behaviour is not included in the requirements, but as it's an additional feature and the nominal behaviour is not affected, it probably won't be detected in the verification phase.

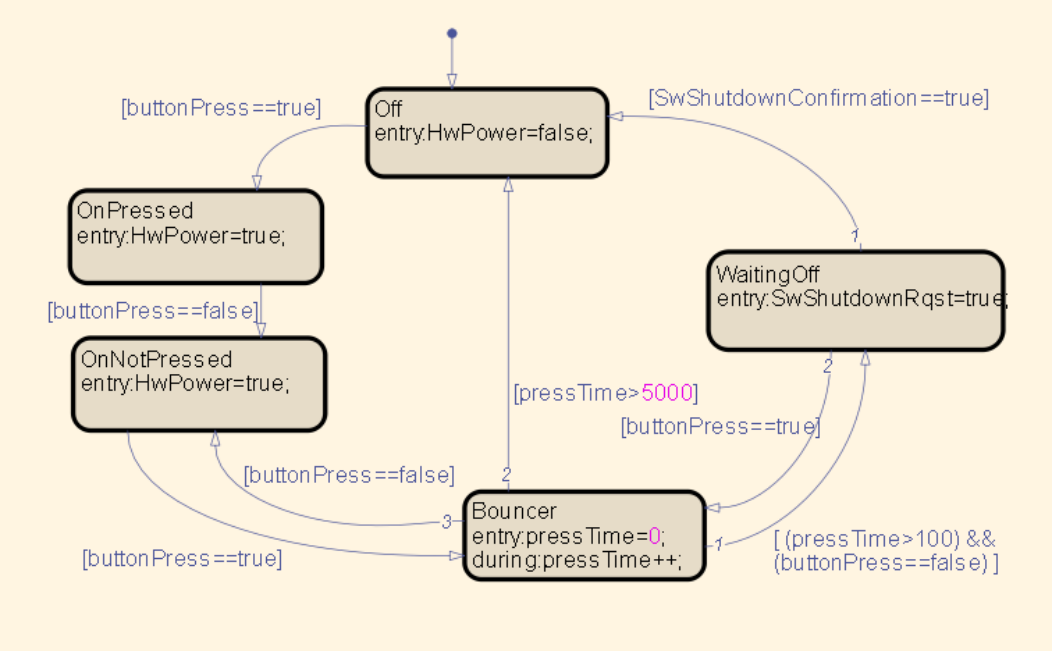


Figure 9-23: Software Shutdown Example (1)

This UF is a *[UF.3.1] Extra functionality*, and it's easily detected in the coverage analysis thanks to the **SOMCA Criterion 9: Transition coverage**, because the transition that adds the new feature is not executed in the verification process.

For this UF, the **SOMCA Prerequisite 3: Model - Requirements Traceability** could have been very useful. However, the usual association of model elements with requirements integrated in the tools doesn't include the transitions as an "associable to requirements" element, so this prerequisite won't be helpful in this situation.

9.6.1.1.2. Forced shutdown not working

In this second example, the forced shutdown is already included in the specification, but an error in the conditions associated to the transitions does not allow the “forced shutdown” transition to be activated. The “software shutdown” condition will be always activated before the second one.

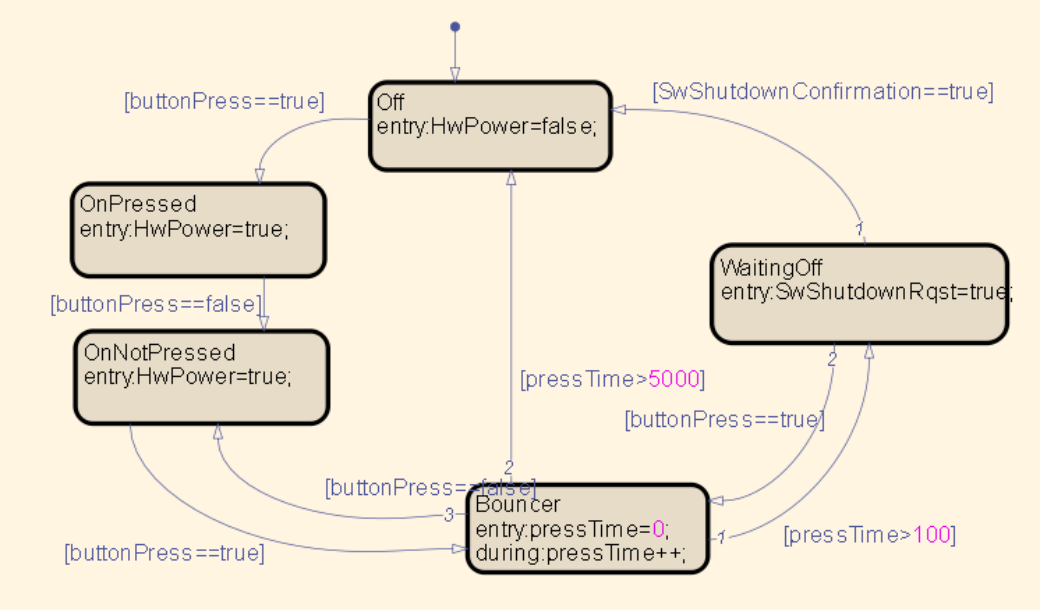


Figure 9-24: Software Shutdown Example (2)

In this case, the specification is correct, and the error is in the implementation of the model. This is an example of a *[UF.3.2] Incorrect behaviour in the general case*. If the verification is not able to detect the UF, the **SOMCA Criterion 9: Transition coverage** will easily find this UF, because the transition is never executed in the verification process.

9.6.1.1.3. Software shutdown not working

In the third variation of this example, the error would be in a small typing error in the condition of the “forced shutdown” transition. The point used to separate groups of digits has been typed incorrectly, changing the value 5000 by 5. Because of this, the state `WaitingOff` would never be reached.

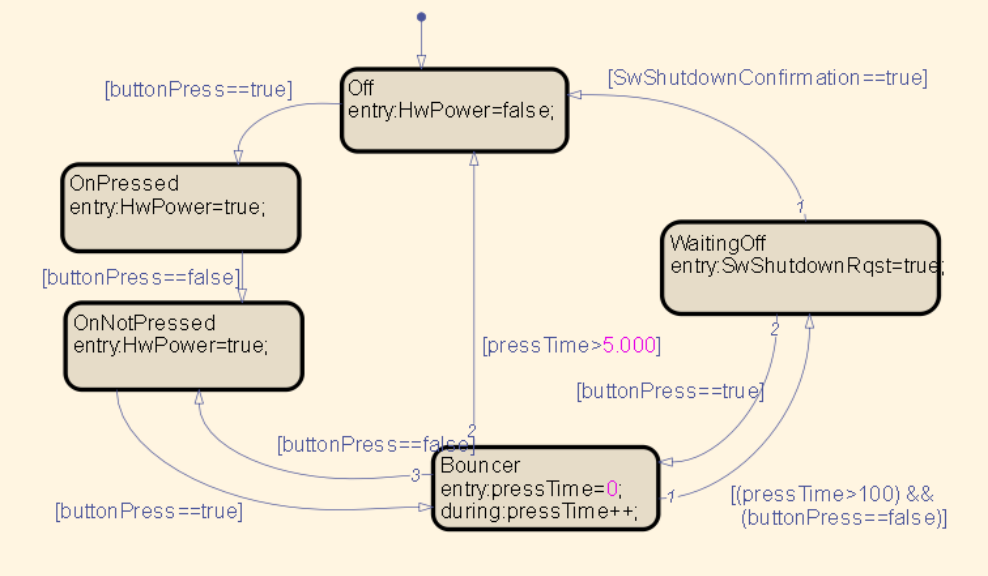


Figure 9-25: Software Shutdown Example (3)

This is an example of *[UF.3.2] Incorrect behaviour in the general case*. If the verification is not able to detect the UF, the **SOMCA Criterion 9: Transition coverage** will easily find this UF, because the transition is never executed in the verification process.

9.6.1.2. Division by zero in Gauss-Markov Filter

One of the filters used in the Smoothing filter (developed as part of the next tasks of this project) is a Gauss-Markov filter, used to discard the input data that does not follow a given statistical distribution. In this module, several mathematics blocks are used: Summation, multiplication and division blocks are very common. The next figure shows the model component we are describing in this example.

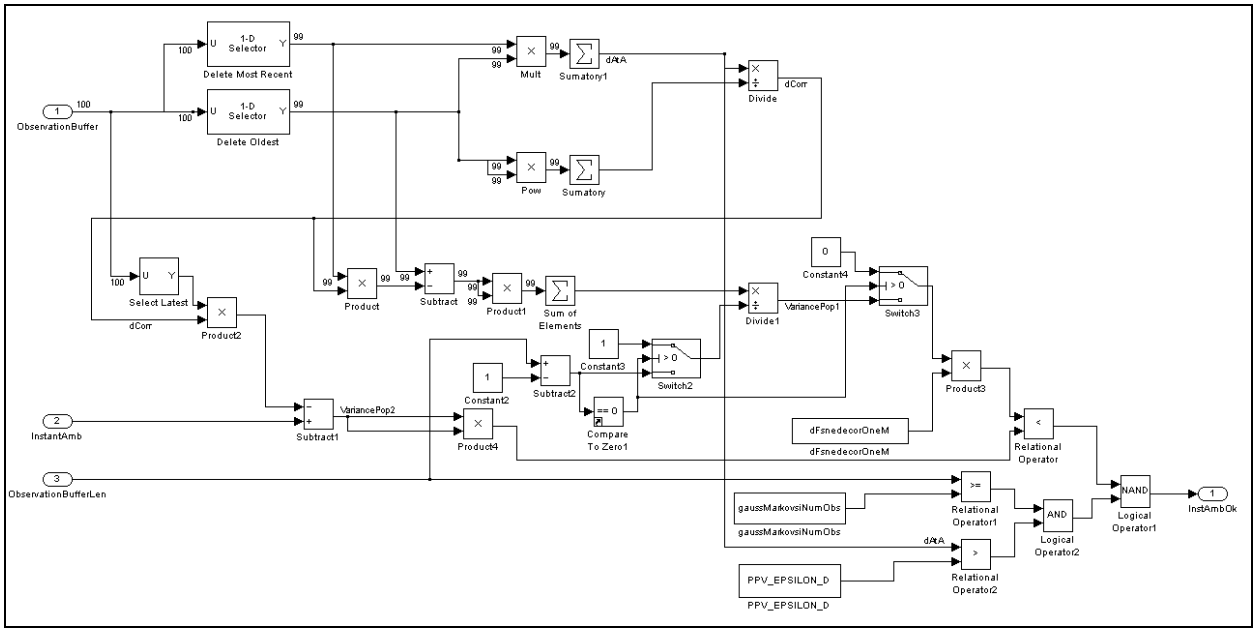


Figure 9-26: Gauss-Markov Filter

In this model, the value dCorr (top-centre in the figure) is calculated by dividing the signals dAtA and dAtB. In nominal behaviour, the observation buffer is full, and the values used in the division do not give any problem. But when the buffer is reset and is filled with zeros, the dAtB term (the divisor in the previously mentioned division operation used to calculate dCorr) is zero, and a division by zero happens.

In Simulink, this kind of errors is warned by the Matlab console, but the execution continues using a zero value as the result of that operation. If this warning is missed, the model "seems to work".

The usual coverage criteria do not analyse this kind of mathematical block, and this error can be missed in the simulations and injected in the generated source code as an Unintended Function. When this situation happens in the source code generated based on the model, an exception is raised and the program probably crashes. Even if the generated code would be able to ignore these exceptions, the value of dCorr could be undetermined, possibly affecting the subsystem output.

This example of UF can be easily avoided by two different mechanisms:

1. By applying the SOMCA Prerequisite 16: Warning free simulation. This prerequisite requires executing all simulations without warnings. In this example this is the easiest solution, and the UF would be avoided. But this solution is only available when the UF has happened in the simulation. If the circumstances under this UF happens would be harder to happen, and the simulation cases do show this UF, the error wouldn't be detected, and the UF would be introduced in the executable.
2. A second solution would be applying SOMCA Criterion 1: Range coverage in order to analyse the singular points of the division blocks (divisor = zero). If this analysis would have been performed, the UF would have been found even if the test cases do not show it. However, it's very common to justify that a block divisor is has been never zero.
3. The last solution would be to analyse formally the subsystem to check if there is any combination of inputs that could lead to a division by zero. The result of this analysis would have shown that, even if the value has not been executed in the test cases, the singular point can be achieved for a given combination of inputs. However, this analysis is only a suggestion and it's not covered in this study. See section 9.9.2 for further information.

In the same module, other divisions blocks are protected against this circumstance, and do not have this UF.

To achieve the best protection against this kind of UFs, both mechanisms should be required.

9.6.1.3. Precision loss caused by hardware support

For this example we can use any block that includes floating point operations in its internal design. For example, an "Add" Block with 3 inputs. The block has been designed and properly configured, and it has been used in several different models in the past without any problem. We include this block in a new design, and based on the previous experience, we think that we won't find any unexpected function here.

But the new design is deployed on a new hardware, a small controller that doesn't have hardware support for floating-point operations and these instructions are emulated (very common in small microcontrollers). What is unknown by the developer (but specified in the hardware documentation), is that the algorithm used in this emulation, in order to improve the process time for floating point instructions, introduces an small error in the result of these operations, giving the result quicker, but with lower resolution that the expected one.

This precision loss, not taken into account in the model because of a bad understanding of the hardware limitations, introduces an Unintended Function in the *[UF.3.4] Model adaptation to Target Platform*.

In this example, the problem is related with the limitations of a hardware component, but the real problem was on the understanding of the hardware limitations and the model restrictions that should have been introduced based on the hardware analysis.

If this restriction wouldn't have been known and documented for the hardware, the Unintended Function will be the same, but the problem wouldn't be in the model, but in the hardware element that introduced the precision loss.

In this case the error is external to the model. However, in this specific example the prerequisite of performing specific simulations and tests to ensure the target platform is accurately modelled will uncover the UF in the target's floating-point library early in the project (**SOMCA Prerequisite 6: Modelling of Target Platform**). This would allow applying different solutions even before the Formalized Design is modelled:

- The model will be adapted to accurately simulate the numeric precision, and designed taking into account the precision limitations
- The target platform will be modified, for example fixing the floating-point library to achieve the desired precision or choosing another CPU with hardware floating-point support.

As noted, the other MCA criteria cannot help in the identification of this UF just through simulation. In case the prerequisite does not discover the precision problems in the platform, the UF will not be detected until the source code is available and the verification cases through simulations are compared with the tests in the platform. For this purpose, it would be needed to apply the **SOMCA Prerequisite 7: Analysis of simulation differences**, defining a process to establish when the differences between a simulation and a test are considered an error.

9.6.1.4. Climate Control

The following example will present a State Machine modelled in Simulink made for control a Climate Control. The model should switch on a fan (the velocity is controlled by quick small pulses) to achieve the desired temperature (obtained as a model input).

We will present three different versions of the model, each one of it has an Unintended function, and we will demonstrate how SOMCA Criteria can find the UFs. In all three examples the Verification demonstrated that the model met the requirements.

9.6.1.4.1. Button Press instability

The first version of this example is represented in the following figure:

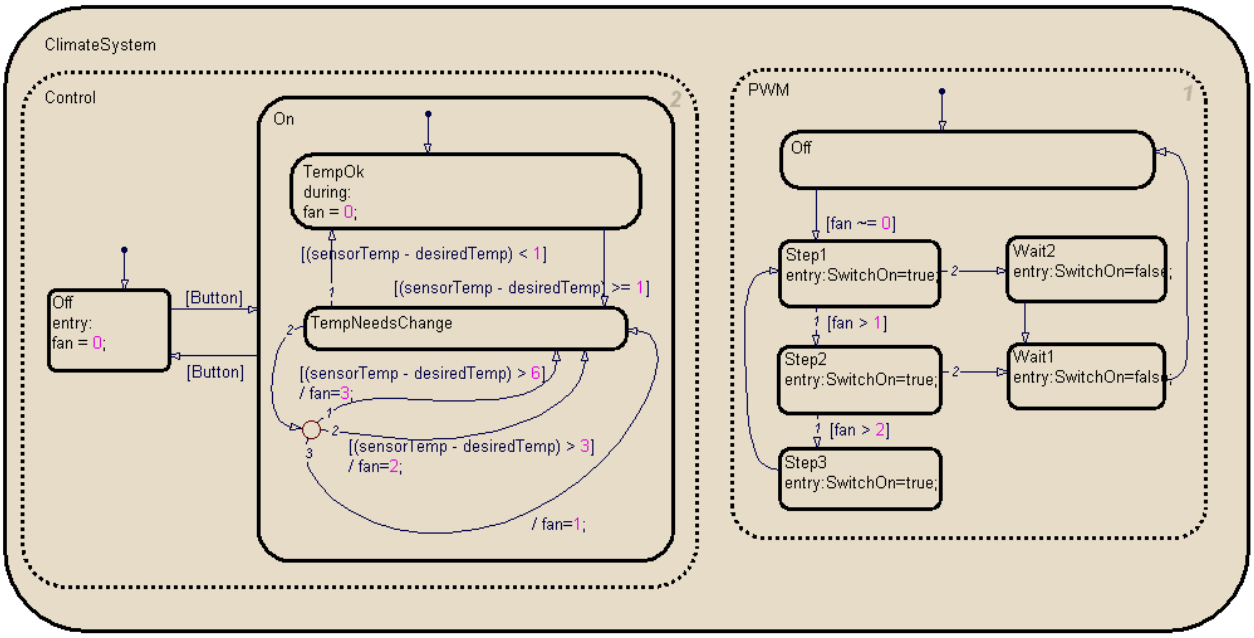


Figure 9-27: Climate System State Machine (1st Version)

In this model, the result of the application of the SOMCA Criteria is the following:

Criterion	Result
SOMCA Criterion 1: Range coverage	Not executed for this example
SOMCA Criterion 2: Functionality coverage	Not executed for this example
SOMCA Criterion 3: Modified input coverage	Not executed for this example
SOMCA Criterion 4: Activation coverage	OK: All activable elements have been activated
SOMCA Criterion 5: Local Decision Coverage	Not applicable (only for block diagrams)
SOMCA Criterion 6: Logic Path coverage	Not applicable (only for block diagrams)
SOMCA Criterion 7: Parent State coverage	**Not OK**: The "On" State has never exit when the substate "TempNeedsChange" was active.
SOMCA Criterion 8: State History coverage	Not applicable (no history junctions in the model).
SOMCA Criterion 9: Transition coverage	OK: All transitions have been executed.
SOMCA Criterion 10: Transition Decision Coverage	OK: All decisions have take all possible values
SOMCA Criterion 11: Transition MC/DC	OK: MC/DC is covered for transition decisions.
SOMCA Criterion 12: Event coverage	Not applicable (no events in the model)
SOMCA Criterion 13: Activating event coverage	Not applicable (no events in the model)
SOMCA Criterion 14: Level-N Loop coverage	**Not OK**: Level 2: There is a loop between Off and "On" states that haven't been tested.

Table 9-4: SOMCA Results on Climate Control Example (1)

In this subsection will analyse the problem detected by **SOMCA Criterion 14: Level-N Loop coverage**: When the input "Button" is active, the state diagram changes continuously between "On" and "Off" states, generating that when the activation button is pressed, the system is only switched on 1 out of 2 times.

This problem was not detected in validation because the verification test case used a pulse to switch on the state diagram that was immediately deactivated. This UF can be classified as *[UF.3.3] Erroneous behaviour under specific inputs / conditions*.

9.6.1.4.2. Fan speed locked when button is pressed

The next version of the example has solved the problem present in the previous version, and added a requirement about the behaviour of the system when the button is pressed. This second version is represented in the next figure.

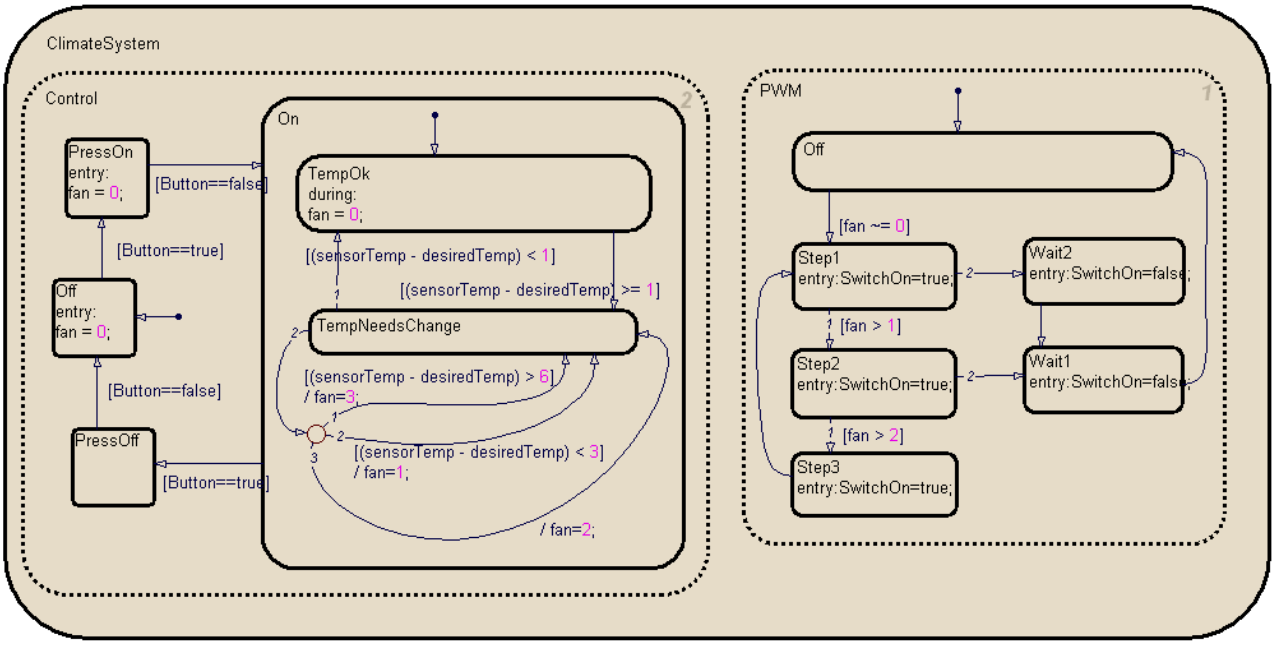


Figure 9-28: Climate System State Machine (2nd version)

The same SOMCA Criteria is applied to this second version, and this is the output:

Criterion	Result
SOMCA Criterion 1: Range coverage	Not executed for this example
SOMCA Criterion 2: Functionality coverage	Not executed for this example
SOMCA Criterion 3: Modified input coverage	Not executed for this example
SOMCA Criterion 4: Activation coverage	OK: All activable elements have been activated
SOMCA Criterion 5: Local Decision Coverage	Not applicable (only for block diagrams)
SOMCA Criterion 6: Logic Path coverage	Not applicable (only for block diagrams)
SOMCA Criterion 7: Parent State coverage	**Not OK**: The "On" State has never exit when the substate "TempNeedsChange" was active.
SOMCA Criterion 8: State History coverage	Not applicable (no history junctions in the model).
SOMCA Criterion 9: Transition coverage	OK: All transitions have been executed.
SOMCA Criterion 10: Transition Decision Coverage	OK: All decisions have take all possible values
SOMCA Criterion 11: Transition MC/DC	OK: MC/DC is covered for transition decisions.
SOMCA Criterion 12: Event coverage	Not applicable (no events in the model)
SOMCA Criterion 13: Activating event coverage	Not applicable (no events in the model)
SOMCA Criterion 14: Level-N Loop coverage	Not applicable (no loops present in the model)

Table 9-5: SOMCA Results on Climate Control Example (2)

Now, as the loop has been solved, the **SOMCA Criterion 14: Level-N Loop coverage** is no longer applicable, and the only coverage problem detected is by **SOMCA Criterion 7: Parent State coverage**. It seems that the verification test cases never switched off the system while it was working.

When the verification is improved with this circumstance, a very interesting thing is observed: If the system is switched off, the fans stop. But while the button is pressed, the fans work at their previous speed independently of the current temperature. Looking at the model, the problem is related with the deactivation of fan when the button is pressed. The state that controls the fan speed is not active, but the fan speed is not reset until "Off" state has been reached, generating the UF.

This UF can be classified as *[UF.3.3] Erroneous behaviour under specific inputs / conditions*.

9.6.1.4.3. Missing intermediate fan speed

Based on the previous example, we introduce a modification in the action statements of the “Wait2” substate. We change the execution of the statement from “entry” to “during”. The next figure shows this final version of the State Machine.

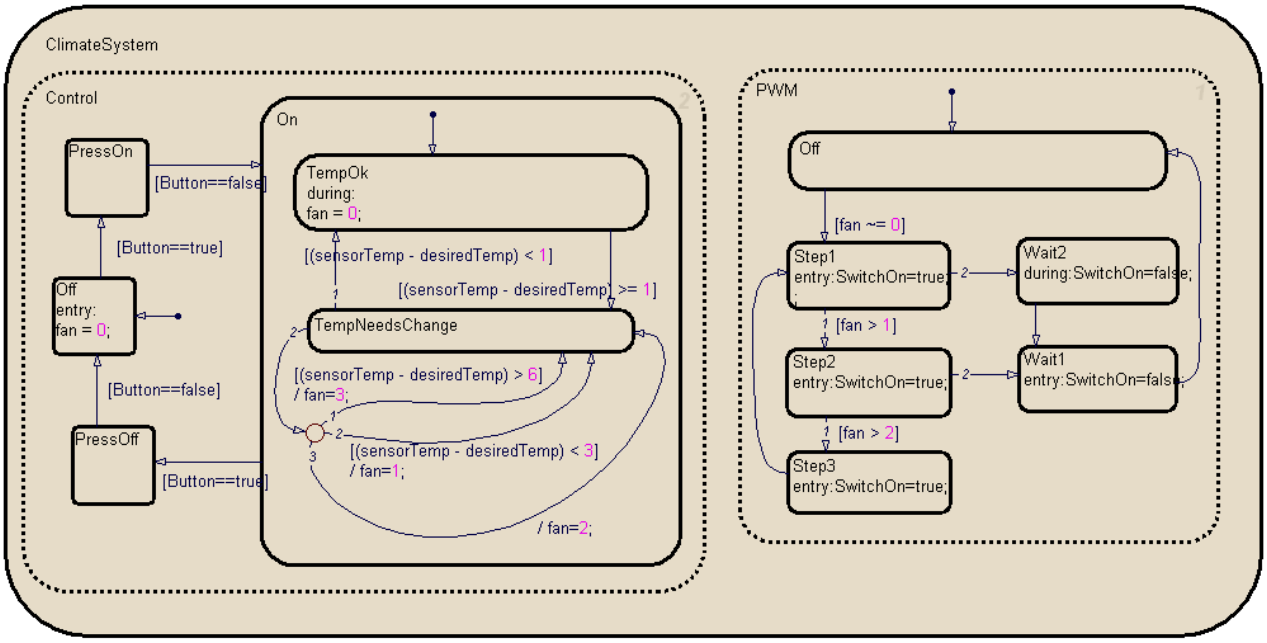


Figure 9-29: Climate System State Machine (3nd version)

The application of SOMCA Criteria on the third version of the State Machine gives the following results:

Criterion	Result
SOMCA Criterion 1: Range coverage	Not executed for this example
SOMCA Criterion 2: Functionality coverage	Not executed for this example
SOMCA Criterion 3: Modified input coverage	Not executed for this example
SOMCA Criterion 4: Activation coverage	**Not OK**: The statements in “Wait2” substate haven’t been executed.
SOMCA Criterion 5: Local Decision Coverage	Not applicable (only for block diagrams)
SOMCA Criterion 6: Logic Path coverage	Not applicable (only for block diagrams)
SOMCA Criterion 7: Parent State coverage	**Not OK**: The “On” State has never exit when the substate “TempNeedsChange” was active.
SOMCA Criterion 8: State History coverage	Not applicable (no history junctions in the model).
SOMCA Criterion 9: Transition coverage	OK: All transitions have been executed.
SOMCA Criterion 10: Transition Decision Coverage	OK: All decisions have take all possible values
SOMCA Criterion 11: Transition MC/DC	OK: MC/DC is covered for transition decisions.
SOMCA Criterion 12: Event coverage	Not applicable (no events in the model)
SOMCA Criterion 13: Activating event coverage	Not applicable (no events in the model)
SOMCA Criterion 14: Level-N Loop coverage	Not applicable (no loops present in the model)

Table 9-6: SOMCA Results on Climate Control Example (3)

The problem with SOMCA Criterion 7 is described in the previous section and it’s not going to be addressed here. The interesting thing is the problem shown by **SOMCA Criterion 4: Activation coverage**. The sentence associated to the “during” action hasn’t been never activated. Analysing the results, it can be observed a problem not detected during the verification: One of the intermediate speeds was missing, executing always the higher one. The verification did not detect it because the temperature went down, but quicker than it should be.

This UF can be *classified as [UF.3.3] Erroneous behaviour under specific inputs / conditions.*

9.6.1.5. Variance calculation error

In the checkCodeSmoCodeVar there is a block called "100ElementsVariance" that calculates the variance for any number of elements between 1 and 100. This variance is used to check that the difference between the input code and the smoothed code follow a given statistical distribution. This check is one of the main barrier checks implemented in the Smoothing filter.

In the original model, this filter was supposed to be working, and the filter seemed to work for the given data and configuration, but once the real data and configuration were included the barrier was being activated too frequently, resetting the filter continuously when the input data was correct.

Investigating the variance block, an UF was found in the algorithm. Some values that should be ignored were being included in a summatory that polluted the result of the block. The verification was not able to find the problem because the highly tolerant values set in the algorithm configuration allowed the filter to work in a similar way than the normal one.

This is a clear example of an error in block functionality. In Structural Code Coverage, this problem would have been found by the use of unit testing. In model coverage, the already existent coverage criteria are not able to find any problem in the block. If SOMCA Criteria could have been applied to the entire model, this UF would have been detected thanks to **SOMCA Criterion 2: Functionality coverage**.

This UF can be classified as *[UF.3.2] Incorrect behaviour in the general case*, because the averages are always erroneously calculated in the first 100 seconds.

The detection of this UF is related with another UFs detected in earlier stages of the UF analysis. In particular, the UF explained in section 13.1.2.2.3 (Coupled functionality between Filter reset conditions) is strongly related with this UF.

9.6.1.6. Smoothed Observable masked by Status

This UF is a consequence of a decision taken during the model design which described in section 13.1.2.2.1 (Output Observable and Status). In that section, a problem was found in the requirements about the management of the Smoothed Observable depending on the Smoothed status. The decision taken seemed to be the most appropriate one. But when the data has been compared with the real one, the behaviour on those situations was not the appropriate one.

This error is clearly derived from a uncertainty in the requirements, and can be classified as *[UF.2] Specification problem*. Due to its nature, this UF can't be detected by the SOMCA Criteria, because the functionality developed is working and it's according to requirements, or at least the understanding of the incomplete requirements.

9.6.1.7. Flowchart error within a Stateflow model

This state diagram of a stop watch, extracted from a paper by Grégoire Hamon and John Rushby [RD.14]. As can be seen in the figure, the stopwatch is modelled as a Stateflow's state diagram with two main states (`Stop`, and `Run`) and 4 sub-states:

- `Reset`: The clock is stopped. Display and internal clock have the same time.
- `Running`: The clock is running and the display is updated.
- `Lap`: The clock is running, but the display is not updated.
- `Lap_stop`: The clock is stopped. Display and internal clock can have different times.

In addition, there are 3 external events used to trigger the transitions and to guard some state actions:

- `TIC`: clock tick (100 ticks/second)
- `START`: Start button
- `LAP`: Lap button

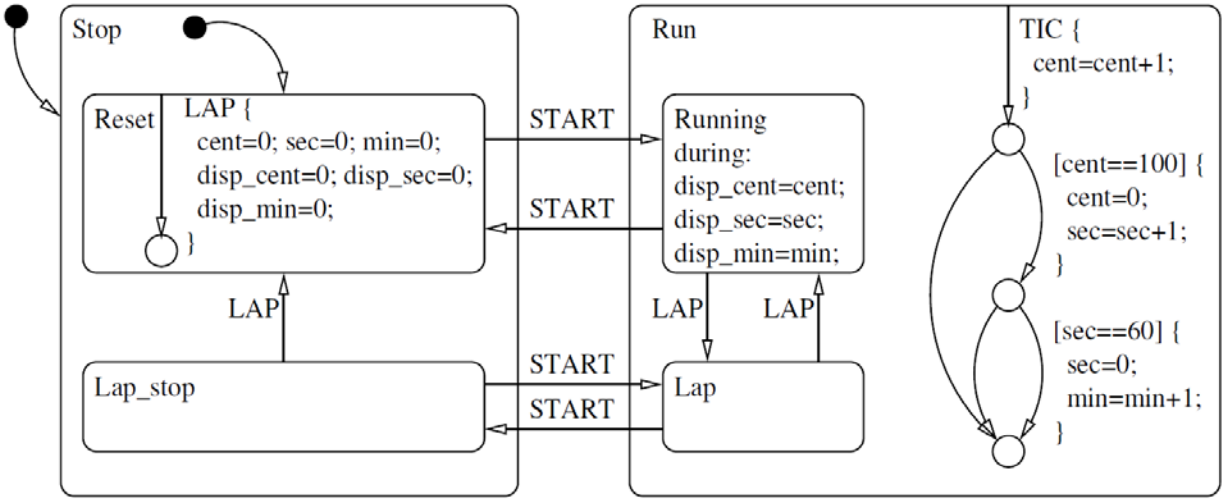


Figure 9-30: Stopwatch in Stateflow (Hamon and Rushby, 2007)

Analysing by visual inspection, the design seems to be correct, but, as explained in the paper, the SAL model checker easily finds a bug in the model: The display (`disp_cent`, `disp_sec`, and `disp_min` variables) is not properly updated when several LAP and START events occur between two clock ticks.

Then, the UF is characterized by an incorrect display value under a burst of external events: if the "Lap button" is pressed so fast that the transitions between substates do not permit to execute the actions defined while staying in the `Running` state (i.e. just the enter and exit actions for this state can be executed, because the state machine is not enough time within the state to execute the during actions), the display will not be updated even if the `Running` state was entered. Note that this kind of obscure bugs is very difficult to discover with traditional testing techniques, while formal analysis techniques like model checkers can easily find a counter example.

As noted, in said paper this bug in the model was detected by static analysis. Now let's try to identify this UF by model coverage analysis. From the set of prerequisites, the one related with documenting the environment conditions would establish the maximum rate allowed for external events. In addition, the prerequisite of

running the verification cases in the final platform ensures that even if the UF can just be replicated with the source code (e.g. due to the different timings), it could be detected through tests even if it is impossible to be detected through simulations. Now, let's start with the application of the different criteria.

In this case, Range Coverage is applicable just to the output values (`disp_cent`, `disp_sec`, `disp_min` variables), because there are no input ports, and the Functionality Coverage criterion requires considering the different features of the stopwatch (start count, freeze count, stop count, reset count), but neither helps for detecting the UF. The different criteria for transitions (Transition Coverage, Transition Decision Coverage, Transition MC/DC) and the different coverage criteria for states (Parent State Coverage, and State History Coverage) are not useful for detecting this type of UF, because these criteria do not exercise the events enough. Not even the event criteria (Event Coverage and Activating Event Coverage) are adequate for the identification of the model error.

If the Loop Coverage criterion is applied and the `LAP` event is forced to execute in burst (continuously transitioning between the `Running` and `Lap` substates), the Activation Coverage criteria may not be satisfied for the 'during' actions of the `Running` substate. However, even if the UF is reproduced, it is not guaranteed that the chosen verification case will never exercise the actions of the `Running` substate ever once, and in any case the designer performing the MCA resolution wouldn't probably notice that this is an error, and choosing a new (trivial) verification case for exercising those actions.

Therefore, we can consider that it is extremely unlikely that this UF could be detected by the SOMCA MCA criteria, even if the techniques required are available. As stated above, Model Coverage just exercise a limited number of situations, and thus testing / simulation alone cannot guarantee that no error remain in the design. All in all, for detecting this type of modelling problems it is better to rely on Formal Methods, as done by Hammon and Rushby to spot this UF in their paper.

9.6.1.8. Water Heater Control

In this case we consider a Water Heater control implemented with Stateflow. This State machine has the following inputs:

- `ONOFF` (user button): Stateflow event
- `UPDATE` (temperature has changed): Stateflow event
- `Temp`: Stateflow input value
- `DesiredTemp`: Stateflow input value
- `MaxTemp`: Stateflow input value

To summarize the behaviour of this control state machine, If the machine is switched on, the HeaterOn State keeps the temperature as indicated by the input value. A special state is designed to switch off and block the heater when an overheat is detected. In the resulting State Machine all transitions are guarded by a single event, except the one to the `Overheat` state, which is implicitly evaluated when *any* event arrives.

Despite the verification didn't detect any problem, exhaustive testing showed that in some cases the controller got blocked after pressing the `ONOFF` Button and the `DesiredTemp` was very close to the `MaxTemp`.

After applying the SOMCA Criteria, the **SOMCA Criterion 13: Activating event coverage** showed that during the verification the transition to Overheat state was only triggered by the `UPDATE` event, and not by `ONOFF` event. This discovered the UF: When the temperature is higher than the MaxTemp, but the machine is manually switched off, the transition to Overheat state was activated, instead of going to the Off State..

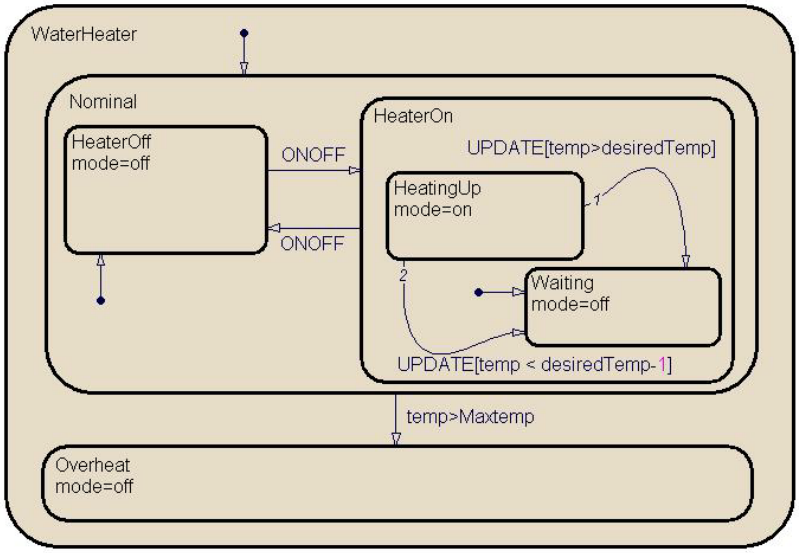


Figure 9-31: Water Heater Control.

This UF is a clear example of *[UF.3.3] Erroneous behaviour under specific inputs / conditions.*

9.6.2. ASSESSMENT RESULTS

One of the main objectives of this study is to prove the power of the SOMCA MCA Criteria. This section presents the results obtained by the MCA criteria for the identification of Unintended Functions in the examples included in section 9.6.1 and 13. The results have been organized in different ways before extracting any conclusions about the effectiveness of the SOMCA MCA activity. The assessment distinguishes between a set of abstract UF examples (called “theoretical examples”), and another set of concrete UF examples (referred to as “real-world examples”):

- Summarizing the overall criteria effectiveness for theoretical and real examples.
- Establishing relationships between UF types, Criteria and effectiveness.
- Showing the evolution from the Theoretical examples to the Real examples.
- Describing which specific Unintended Functions were not successfully detected and how external aspects should be addressed.
- Providing some conclusions over these results.

The section has been structured in subsections following the natural order used in the elaboration of the results and conclusions:

- First of all, the Theoretical examples from previous versions with the results of the last version of SOMCA Criteria.
- After the theoretical examples results, the real ones, in order to establish a continuous timeline.
- Once all the results have been presented, the analysis of the relationship between both of them, to show the evolution of the project in time.
- At the end, the conclusions of the MCA Assessment, based on the three previous subsections.

9.6.2.1. Theoretical UF Examples

9.6.2.1.1. Summary of Theoretical UF Examples

Table 9-7 presents the classification of the UFs presented in the theoretical examples based on the UF Type. Each column contains the following contents:

- *Type*: Category within the UF taxonomy, as found in section 9.2.3.
- *UF*: Name of specific UF example
- *Detected by SOMCA criteria / prerequisite*: Indicates whether a SOMCA MCA criterion / prerequisite was able to identify the UF.
  - - : Not applicable.
  - *Yes*: The UF has been detected by the proposed MCA criteria.
  - *Partially*: The indicated criteria or prerequisites can help in the identification but are not guaranteed to detect the UF.
  - *No*: The UF has not been detected by the proposed MCA criteria.

Type	UF	Detected by SOMCA Criteria / Prereq
[UF.1] Deviation from requirements in the implementation	-	-
[UF.2] Specification problem	13.5.9. Satellite clock stability check	Partially / Yes
	13.5.13. Event log length	Partially / Yes
[UF.3.1] Extra functionality	13.5.10. Descoped functionality interference: Reuse of message decoder	No / Partially
	13.5.11. Debug elements present in operational versions	Yes / -
[UF.3.2] Incorrect behaviour in the general case	13.5.2. Unsynchronized naming of Input / Output Ports	Yes / Yes
	13.5.14. Internal precision loss in arithmetic block	No / Yes
[UF.3.3] Erroneous behaviour under specific inputs / conditions	13.5.1. Unstable System – transfer function	Partially / Yes
	13.5.3. Implementation error in max block	Partially / Yes
	9.6.1.7. Flowchart error within a Stateflow model	Yes / -
	13.5.4. Filter block Inputs	Yes / -
	13.5.5. Memory-latch error in a limiter	Yes / Partially
	13.5.6. Commanding for multiple blocks	Yes / -
	13.5.7. System time adjust delay	Yes / -
	9.6.1.2. Division by zero in Gauss-Markov Filter	Yes / -
[UF.3.4] Model adaptation to Target Platform	13.5.8. Cache conflicts in concurrent blocks	No / No
	9.6.1.3. Precision loss caused by hardware support	Partially / Yes
[UF.4] Derived from Model Interfaces	13.5.16. Error in model inputs	Partially / Partially
[UF.5] Derived from System complexity	13.5.12. Highly coupled algorithmic architecture	Partially / -
	13.5.15. Protocol control coupling	Yes / Yes

Table 9-7: Classification of Theoretical UF examples based on UF Type

9.6.2.1.2. Criteria effectiveness for Theoretical Examples

This table establish a relationship between the proposed Criteria and the Theoretical examples included in Appendix B: Detailed UF Examples.

Criterion	UF	UF Type	Result
Generic Criteria			
SOMCA Criterion 1: Range coverage	13.5.3. Implementation error in max block	[UF.3.3] Erroneous behaviour under specific inputs / conditions	UF Detected by Criteria
	13.5.4. Filter block Inputs	[UF.3.3] Erroneous behaviour under specific inputs / conditions	UF Detected by Criteria
	13.5.7. System time adjust delay	[UF.3.3] Erroneous behaviour under specific inputs / conditions	UF Detected by Criteria
	13.5.12. Highly coupled algorithmic architecture	[UF.5] Derived from System complexity	Partially detected
SOMCA Criterion 2: Functionality coverage	13.5.1. Unstable System – transfer function	[UF.3.3] Erroneous behaviour under specific inputs / conditions	UF Detected by Criteria
	13.5.2. Unsynchronized naming of Input / Output Ports	[UF.3.2] Incorrect behaviour in the general case	Partially detected
	13.5.13. Event log length	[UF.2] Specification problem	Partially detected
	13.5.15. Protocol control coupling	[UF.5] Derived from System complexity	UF Detected by Criteria
SOMCA Criterion 3: Modified input coverage	13.5.11. Debug elements present in operational versions (b)	[UF.3.1] Extra functionality	UF Detected by Criteria
SOMCA Criterion 4: Activation coverage	-	-	-
Block Diagrams Criteria			
SOMCA Criterion 5: Local Decision Coverage	-	-	-
SOMCA Criterion 6: Logic Path coverage	13.5.2. Unsynchronized naming of Input / Output Ports	[UF.3.2] Incorrect behaviour in the general case	Partially detected
	13.5.6. Commanding for multiple blocks	[UF.3.3] Erroneous behaviour under specific inputs / conditions	UF Detected by Criteria
State Machines Criteria			
SOMCA Criterion 7: Parent State coverage	-	-	-
SOMCA Criterion 8: State History coverage	-	-	-
SOMCA Criterion 9: Transition coverage	13.5.11. Debug elements present in operational versions (a)	[UF.3.1] Extra functionality	UF Detected by Criteria
SOMCA Criterion 10: Transition Decision Coverage	-	-	-
SOMCA Criterion 11: Transition MC/DC	13.5.11. Debug elements present in operational versions (b)	[UF.3.1] Extra functionality	UF Detected by Criteria
SOMCA Criterion 12: Event coverage	-	-	-

Criterion	UF	UF Type	Result
SOMCA Criterion 13: Activating event coverage	-	-	-
SOMCA Criterion 14: Level-N Loop coverage	-	-	-

Table 9-8: SOMCA Criteria and Theoretical Example UFs associated to them

As we can see in the table above, the Theoretical examples do not exercise the most refined criteria, as expected. This is caused by the time they were identified. When these examples were defined, the only available criteria were the ones proposed in the Certification Memo [RD.8], which is quite similar to the criteria covered by the Theoretical Examples.

The results of the UF detection are summarized in the following table.

Detection case	Occurrences	%	Examples
Completely detected by SOMCA MCA	6	32%	13.5.4. Filter block Inputs 13.5.6. Commanding for multiple blocks 13.5.7. System time adjust delay 13.5.11. Debug elements present in operational versions 13.5.13. Event log length 13.5.15. Protocol control coupling
Partially detected by SOMCA MCA criteria, detected by pre-requisite	3	16%	13.5.1. Unstable System – transfer function 13.5.3. Implementation error in max block 9.6.1.2. Division by zero in Gauss-Markov Filter
Not detected by SOMCA MCA criteria, but by pre-requisites or recommendations	4	21%	13.5.5. Memory-latch error in a limiter 13.5.9. Satellite clock stability check 13.5.14. Internal precision loss in arithmetic block 9.6.1.3. Precision loss caused by hardware support
Partially detected by SOMCA MCA criteria, partially detected by pre-requisites or recommendations	5	26%	13.5.2. Unsynchronized naming of Input / Output Ports 13.5.8. Cache conflicts in concurrent blocks 13.5.10. Descoped functionality interference: Reuse of message decoder 13.5.12. Highly coupled algorithmic architecture 13.5.16. Error in model inputs
Not detected at all	1	5%	9.6.1.7. Flowchart error within a Stateflow model
Total	19	100%	

Table 9-9: Summary of UF identification for Theoretical Examples

9.6.2.1.3. Analysis of results for Theoretical Examples

From the 19 examples, just one of them was not detected at all with the criteria or the prerequisites (5%):

- 9.6.1.7. Flowchart error within a Stateflow model

This UF type is hard to identify, because in order to reproduce the error it's necessary a combination of inputs values very difficult to achieve. For this example, the Formal Analysis used in the paper written by John Rushby [RD.14] has proved to be useful to detect this UF. It could be interesting to study if other kind of formal analysis is able to find this UF.

The following four UF examples are partially found of mitigated by SOMCA Criteria or Prerequisites (26%):

- 13.5.2. Unsynchronized naming of Input / Output Ports
- 13.5.8. Cache conflicts in concurrent blocks
- 13.5.10. Descoped functionality interference: Reuse of message decoder
- 13.5.12. Highly coupled algorithmic architecture
- 13.5.16. Error in model inputs

The rest of the examples can be found or eliminated with the SOMCA Criteria and Prerequisites (68%).

9.6.2.2. Real-World Examples

9.6.2.2.1. Summary of Real-World Examples

The following table present all the UFs identified in the real examples ordered by UF type, and indicating if the SOMCA Criteria or prerequisites have been able to found them.

Type	UF	Detected by SOMCA Crit.	Comment
[UF.1] Deviation from requirements in the implementation	13.1.3.1.4: Input value of CodeSmoothedCodeVar check	No	Mitigated by Prerequisite
[UF.2] Specification problem	13.1.2.2.1: Output Observable and Status	No	
	13.1.2.2.2: Units in input parameters	No	
	9.6.1.6: Smoothed Observable masked by Status	No	
	13.1.3.1.3: Gap processing	Partially	Logic Path analysis maybe could find the UF, but the analysis have not been done.
[UF.3.1] Extra functionality	13.1.2.2.3: Coupled functionality between Filter reset conditions	Yes	
	9.6.1.1.1: Forced shutdown not included in the Requirements	Yes	
	13.4.2: Petition Queuing in Transaction Manager (b)	Yes	
[UF.3.2] Incorrect behaviour in the general case	13.1.2.1.1: KSM Unlimited	No*	*Eliminated by prerequisite.
	13.1.2.1.3: Complexity in Algorithm Details	No*	*Eliminated by prerequisite.
	13.1.2.1.4: Gauss-Markov Buffer length Check	Yes	
	9.6.1.5: Variance calculation error	Yes*	*Depends on the implementation of Functionality coverage.
	13.1.3.1.4: Input value of CodeSmoothedCodeVar check	No*	*Mitigated by Prerequisite
	9.6.1.1.2: Forced shutdown not working	Yes	
	9.6.1.1.3: Software shutdown not working	Yes	
	9.6.1.4.3: Missing intermediate fan speed	Yes	

Type	UF	Detected by SOMCA Crit.	Comment
[UF.3.3] Erroneous behaviour under specific inputs / conditions	13.1.2.1.2: Gauss-Markov Division by Zero	No*	*Eliminated by prerequisite.
	13.1.2.1.5: Cycleslip Init	Yes	
	13.1.2.1.6: Missing Abs block	Partially	The detection is not assured.
	13.1.3.1.5: ASP Calculation minor problem	No	UF cause not detected due to the absence of data.
	13.2.1: Gap not detected	Yes	
	13.2.2: Cycleslip not detected	Yes	
	13.3.2.1: Unstable output when simultaneous inputs are activated	Yes	
	13.3.3.1: Error in Standby Condition	Unknown	Not checked.
	9.6.1.1.2: Forced shutdown not working	Yes	
	9.6.1.4.1: Button Press instability	Yes	
	9.6.1.4.2: Fan speed locked when button is pressed	Yes	
	13.4.1: Serial Port Controller	Yes	
	13.4.2: Petition Queuing in Transaction Manager (a)	Yes	
	9.6.1.8: Water Heater Control	Yes	
	13.4.3: Text Line Editor Modes	Yes	
[UF.3.4] Model adaptation to Target Platform	---		Not applicable (no target platform)
[UF.4] Derived from Model Interfaces	-	-	-
[UF.5] Derived from System complexity	13.1.2.1.3: Complexity in Algorithm Details	No*	*Eliminated by prerequisite.

Table 9-10: Real Examples UFs ordered by type.

9.6.2.2.2. Criteria effectiveness for Real-World Examples

The following table present all the UFs identified in the real examples ordered by UF type, and indicating if the SOMCA Criteria or prerequisites have been able to found them.

Criterion	UF	UF Type	Result
Generic Criteria			
SOMCA Criterion 1: Range coverage	13.1.2.1.6: Missing Abs block	[UF.3.3] Erroneous behaviour under specific inputs / conditions.	The UF is partially discovered (not always)
SOMCA Criterion 2: Functionality coverage	9.6.1.5: Variance calculation error	[UF.3.2] Incorrect behaviour in the general case	The UF would have been discovered if criteria would have been automatically applied.
SOMCA Criterion 3: Modified input coverage	13.1.2.1.4: Gauss-Markov Buffer length Check	[UF.3.2] Incorrect behaviour in the general case	UF Detected by Criteria
	13.2.1: Gap not detected	[UF.3.3] Erroneous behaviour under specific inputs / conditions.	UF Detected by Criteria

Criterion	UF	UF Type	Result
SOMCA Criterion 4: Activation coverage	9.6.1.4.3: Missing intermediate fan speed	[UF.3.2] Incorrect behaviour in the general case.	UF Detected by Criteria
Block Diagrams Criteria			
SOMCA Criterion 5: Local Decision Coverage	13.1.2.1.4: Gauss-Markov Buffer length Check	[UF.3.2] Incorrect behaviour in the general case	UF Detected by Criteria
SOMCA Criterion 6: Logic Path coverage	13.1.2.1.5: Cycleslip Init Behaviour	[UF.3.3] Erroneous behaviour under specific inputs / conditions.	UF Detected by Criteria
	13.1.2.2.3: Coupled functionality between Filter reset conditions	[UF.3.1] Extra functionality	UF Detected by Criteria
	13.1.3.1.3: Gap processing	[UF.2] Specification Problem	Possible detection, but not assured (coverage not executed).
	13.2.2: Cycleslip not detected	[UF.3.3] Erroneous behaviour under specific inputs / conditions.	UF Detected by Criteria
State Machines Criteria			
SOMCA Criterion 7: Parent State coverage	13.4.1: Serial Port Controller	[UF.3.3] Erroneous behaviour under specific inputs / conditions.	UF Detected by Criteria
	13.4.2: Petition Queuing in Transaction Manager (a)	[UF.3.3] Erroneous behaviour under specific inputs / conditions	UF Detected by Criteria
	9.6.1.4.2: Fan speed locked when button is pressed	[UF.3.3] Erroneous behaviour under specific inputs / conditions	UF Detected by Criteria
SOMCA Criterion 8: State History coverage	13.4.2: Petition Queuing in Transaction Manager (b)	[UF.3.1] Extra functionality	UF Detected by Criteria
SOMCA Criterion 12: Event coverage			
SOMCA Criterion 13: Activating event coverage	9.6.1.8: Water Heater Control	[UF.3.3] Erroneous behaviour under specific inputs / conditions	UF Detected by Criteria
SOMCA Criterion 9: Transition coverage	9.6.1.1.1: Forced shutdown not included in the Requirements	[UF.3.1] Extra functionality	UF Detected by Criteria
	9.6.1.1.2: Forced shutdown not working	[UF.3.3] Erroneous behaviour under specific inputs / conditions.	UF Detected by Criteria
	9.6.1.1.3: Software shutdown not working	[UF.3.2] Incorrect behaviour in the general case.	UF Detected by Criteria
SOMCA Criterion 10: Transition Decision Coverage			
SOMCA Criterion 11: Transition MC/DC	13.4.3: Text Line Editor Modes	[UF.3.3] Erroneous behaviour under specific inputs / conditions	UF Detected by Criteria
SOMCA Criterion 14: Level-N Loop coverage	13.3.2.1: Unstable output when simultaneous inputs are activated	[UF.3.3] Erroneous behaviour under specific inputs / conditions	UF Detected by Criteria
	9.6.1.4.1: Button Press instability	[UF.3.3] Erroneous behaviour under specific inputs / conditions	UF Detected by Criteria

Table 9-11: SOMCA Criteria and Real Example UFs associated to them

The results of the UF detection of the real world examples are summarized in the following table.

Detection case	Occurrences	%	Examples
Completely detected by SOMCA MCA	18	63%	9.6.1.1.1: Forced shutdown not included in the Requirements 9.6.1.1.2: Forced shutdown not working 9.6.1.1.3: Software shutdown not working 9.6.1.4.1: Button Press instability 9.6.1.4.2: Fan speed locked when button is pressed 9.6.1.4.3: Missing intermediate fan speed 9.6.1.5: Variance calculation error 9.6.1.8: Water Heater Control 13.1.2.1.4: Gauss-Markov Buffer length Check 13.1.2.1.5: Cycleslip Init 13.1.2.2.3: Coupled functionality between Filter reset conditions 13.2.1: Gap not detected 13.2.2: Cycleslip not detected 13.3.2.1: Unstable output when simultaneous inputs are activated 13.4.1: Serial Port Controller 13.4.2: Petition Queuing in Transaction Manager (a) 13.4.2: Petition Queuing in Transaction Manager (b) 13.4.3: Text Line Editor Modes
Partially detected by SOMCA MCA criteria, detected by pre-requisite	0	%	-
Not detected by SOMCA MCA criteria, but by pre-requisites or recommendations	3	11%	13.1.2.1.1: KSM Unlimited 13.1.2.1.2: Gauss-Markov Division by Zero 13.1.2.1.3: Complexity in Algorithm Details
Partially detected by SOMCA MCA criteria, partially detected by pre-requisites or recommendations	2	8%	13.1.3.1.3: Gap processing 13.1.2.1.6: Missing Abs block
Not detected at all	5	18%	13.1.2.2.1: Output Observable and Status 13.1.2.2.2: Units in input parameters 9.6.1.6: Smoothed Observable masked by Status 13.1.3.1.4: Input value of CodeSmoothedCodeVar check 13.1.3.1.5: ASP Calculation minor problem
Not analyzed	1	5%	13.3.3.1: Error in Standby Condition
Total	29	100%	

Table 9-12: Summary of UF identification for real world examples

9.6.2.2.3. Prerequisite effectiveness for Real-World Examples

As we did for Criteria in the previous section, the following table presents a relationship between SOMCA Prerequisites and UFs detected by them.

Prerequisite	UF Related with it	Comment
SOMCA Prerequisite 1: ED-12B Applicability		
SOMCA Prerequisite 2: Source Code Coverage		
SOMCA Prerequisite 3: Model - Requirements Traceability	9.6.1.1.1: Forced shutdown not included in the Requirements	No: It cannot detect the UF due to traceability tool restrictions
SOMCA Prerequisite 4: Requirements – Model Traceability	13.1.3.1.4: Input value of CodeSmoothedCodeVar check	Partially, traceability does not use to have the necessary level of detail.
SOMCA Prerequisite 5: Simulation configuration		
SOMCA Prerequisite 6: Modelling of Target Platform		
SOMCA Prerequisite 7: Analysis of simulation differences		
SOMCA Prerequisite 8: Type check	13.1.2.1.3: Complexity in Algorithm Details	Yes: The Summation block doesn't have size restriction, but it should have found it.
SOMCA Prerequisite 9: Uniform Model Verification Strategy		
SOMCA Prerequisite 10: Definition of Tool versions		
SOMCA Prerequisite 11: Tool and notation suitability		
SOMCA Prerequisite 12: Transfer function stability		
SOMCA Prerequisite 13: Model documentation		
SOMCA Prerequisite 14: Barrier identification		
SOMCA Prerequisite 15: Environment definition	13.1.2.2.2: Units in input parameters	
SOMCA Prerequisite 16: Warning free simulation	13.1.2.1.2: Gauss-Markov Division by Zero	Yes: Eliminates the UF
SOMCA Prerequisite 17: Requirement of Design Standard		
SOMCA Prerequisite 18: Explicit priority for Transitions		
SOMCA Prerequisite 19: State Reachability		
SOMCA Prerequisite 20: Recursive SOMCA Applicability		
SOMCA Prerequisite 21: Input / Output assertions	13.1.2.1.1: KSM Unlimited	Yes: Eliminates the UF
SOMCA Prerequisite 22: External timing management		

Table 9-13: SOMCA Prerequisites and Real Example UFs associated to them

9.6.2.2.4. Analysis of results for Real-World Examples

Based on the data presented in previous sections, we can extract some interesting conclusions:

- Most of the detected/injected UFs have been detected by SOMCA Criteria or Prerequisites (number in parenthesis includes partially detected results):

Number of examples	Detected by Criteria	Partially detected by Criteria	Detected by prerequisites	Partially detected by prerequisites	Undetected	Not analysed	Total detected

29	18	2	3	0	5	1	23

Table 9-14: Summary of Real-world examples assessment

- Some UFs could not be detected by SOMCA Criteria:
  - 75% (3 out of 4) of the undetected Unintended Functions belong to *[UF.2] Specification problem* (75%). This is very significant, because the specification problems are the ones most difficult to identify.
  - The last undetected Unintended Function found (13.1.3.1.5: ASP Calculation minor problem) is a very minor problem, very close to the error tolerance level, and the source of this Unintended Function hasn't been found, so no conclusions can be extracted from this example.
- There are several UFs detected or eliminated by Prerequisites. This shows us that meeting these prerequisites before the verification activities will remove several unintended UFs that are very difficult to identify by MCA.
- The most common types of UFs found in the real examples are
  - *[UF.3.2] Incorrect behaviour in the general case*
  - *[UF.3.3] Erroneous behaviour under specific inputs / conditions*

  And almost every UF classified as any of these two have been found by the SOMCA Criteria / Prerequisites.
- Some criteria have not been exercised in any example. Refer to the next section.

9.6.2.3. Assessment conclusions

Finally, after challenging the SOMCA criteria against the different types of UFs, both real-world and from the lab, these are the final conclusions (the not analysed examples have been removed in this summary table):

Examples	Number of examples	Detected by Criteria	Partially detected by Criteria	Detected by prerequisites	Partially detected by prerequisites	Undetected	Total detected
Theoretical	19 (100%)	6 (32%)	3 (16%)	4 (21%)	5 (26%)	1 (5%)	18 (95%)
Real-World	28 (100%)	18 (64%)	2 (8%)	3 (11%)	0 (0%)	5 (18%)	23 (82%)
Total	**47 (100%)**	**24 (51%)**	**5 (11%)**	**7 (14%)**	**5 (11%)**	**6 (13%)**	**41 (87%)**

Table 9-15: Summary of SOMCA Criteria assessment

1. For the theoretical examples (19 examples):
   a) Just one of them was not detected at all by the criteria or the prerequisites (5%). This state machines example comes from the literature [RD.14], where a formal analysis tool was used to detect it.
   b) The rest of the examples can be found or eliminated with the SOMCA Criteria and Prerequisites (95%), although in five of them the criteria or prerequisites is not always guaranteed to avoid the problems.
2. For the real-world examples (28 examples analysed):
   a) 15% was not detected by the criteria. In this case, 3 out of the 4 non-detected UFs were related with problems in the specification. The remaining one was not identified because its magnitude is so low that it is in the range of the tolerance due to the different implementation between the Operational example and the modelled one.

b) The rest of the examples can be found or eliminated with the SOMCA Criteria and Prerequisites (85%), but three of them are not guaranteed to be detected.

The results obtained after applying the SOMCA Criteria to these varied UF examples show a high detection rate: 89% of the 45 examples, including both real-world examples and theoretical ones. As explained above the results from the real-world examples are considered more indicative, so the actual detection rate can be seen as slightly lower: an 85%. Therefore, it can be considered with a good level of confidence that the SOMCA Criteria is comprehensive enough to properly exercise a Formalized Design, giving better results than the current set of criteria defined on the Certification Memorandum.

After analysing all the proposed theoretical and real examples, these are the results of the Criteria effectiviness on the proposed examples:

Criteria	UFs Fully detected		UFs Partially detected		Total	
	Theor.	Real	Theor.	Real	Theor.	Real
SOMCA Criterion 1: Range coverage	3	0	1	1	4	1
SOMCA Criterion 2: Functionality coverage	2	1	2	0	2	1
SOMCA Criterion 3: Modified input coverage	1	2	0	0	1	2
SOMCA Criterion 4: Activation coverage	0	1	0	0	0	1
SOMCA Criterion 5: Local Decision Coverage	0	1	0	0	0	1
SOMCA Criterion 6: Logic Path coverage	1	3	1	1	1	4
SOMCA Criterion 7: Parent State coverage	0	3	0	0	0	3
SOMCA Criterion 8: State History coverage	0	1	0	0	0	1
SOMCA Criterion 9: Transition coverage	1	3	0	0	1	3
SOMCA Criterion 10: Transition Decision Coverage	**0**	**0**	**0**	**0**	**0**	**0**
SOMCA Criterion 11: Transition MC/DC	1	1	0	0	1	1
SOMCA Criterion 12: Event coverage	**0**	**0**	**0**	**0**	**0**	**0**
SOMCA Criterion 13: Activating event coverage	0	1	0	0	0	1
SOMCA Criterion 14: Level-N Loop coverage	0	2	0	0	0	2

Table 9-16: Summary of SOMCA Criteria effectiveness

As it can be seen in this table, there are two criteria that have not contributed to detect an UF in any of the examples analysed:

- **SOMCA Criterion 12: Event coverage** has not been used in any example:

  This doesn't mean that this criterion is not useful. The reason because they haven't been executed is because events have not been used widely in the proposed examples. Most of the examples developed have used Simulink as the main tool to analyse, and the usage of Simulink events (inside Stateflow) changes the periodic execution on the State Machines, making the analysis harder. The SCADE events (called *signals*) are not used also in the imported examples.

  Even without a real proof of its efficiency, we consider that these criteria can add some added value to the MCA, and it should be taken into account. The fact that the more restrictive version of this criterion (**SOMCA Criterion 13: Activating event coverage**) has been proved useful in the examples, indicates that this criterion should be kept for lower criticality levels.

- **SOMCA Criterion 10: Transition Decision Coverage** has not been exercised by any example

  There were several examples that could be found by these Criteria. However, all of them are also detected by a simpler one, the **SOMCA Criterion 9: Transition coverage** or the most restrictive one: the **SOMCA Criterion 11: Transition MC/DC**. Even with no examples associated to them, these criteria are very important, because the conditions must be strongly checked to find any possible UF related with them. As more and less restrictive criteria have been proved useful, an intermediate level between the other two criteria can be very useful, even if it doesn't have any associated UF in this study.

The conclusion is that, although these 2 criteria above described not been exercised and their added value cannot be demonstrated by examples, they are still considered necessary to guarantee coverage assurance level.

9.7. MODEL COVERAGE VS. STRUCTURAL CODE COVERAGE

The definition of the SOMCA MCA criteria raises different questions with respect to the applicability of Structural Code Coverage criteria as mandated by ED-12B. The main one is whether Model Coverage Analysis would be sufficient to get airworthiness certification for MBD projects without requiring Structural Code Coverage as currently requested by ED-12B. This boils down to the following points:

- Do Model Coverage and Structural Code Coverage address the same type of problems, considering that they are referred to different levels of abstraction?

- Does Model Coverage provide the same level of correctness assurance than Structural Code Coverage as proposed in ED-12B?
  - Is it possible to trace Model Coverage to Structural Code Coverage?
  - Can be compared the completeness assurance level provided by Model Coverage with Structural Code Coverage?
  - Does the application of SOMCA MCA criteria at model level guarantee a given level of Structural Code Coverage in the associated source code?
  - In which conditions the SOMCA MCA criteria can provide the same protection as Structural Code Coverage?

The goal behind these questions is to demonstrate the conditions under which the coverage criteria properties can be preserved in the transformation from model to code.

In the frame of SOMCA Project, no conclusions have been extracted on this subject. Further research will be necessary in order to obtain answers to the questions presented above or in order to obtain conclusions.

9.8. CERTIFICATION MEMORANDUM AMENDMENT

This section contains an analysis of the recommendations proposed to the EASA Certification Memorandum [RD.8] considering the outputs of the SOMCA Model Coverage Analysis activity, including the criteria, pre-requisites and recommendations. All of them are related to chapter 23 ("The Validation and Verification of Formalised and Model-Based Software Requirements and Designs"), which provides guidance material for certification applicants employing a Model Based Development approach. The next table analyses which changes would be needed to integrate each SOMCA criterion and pre-requisite, while note that all the SOMCA recommendations from section 9.4.2 are linked with the usage of a Formalized Design Standard.

#ID	SOMCA Items	CM Section	Cert Memo Modification	Justification
0	SOMCA Criteria (all 14 criteria)	23.2.9 Coverage of formalized designs	Replace: The following criteria may be used to assess the coverage of the Formalized Design: • all the conditions of the logic components • all equivalence classes (valid/in-range and invalid/out-of-range classes) and singular points of the functional components and algorithms • all transitions of the state machines • Coverage of all characteristics of the functionality in context (e.g. Watchdog function is triggered) By: The following criteria may be used to assess the coverage of a Formalized Design components stated through Block Diagrams: • Range coverage • Functionality Coverage • Modified input Coverage • Activation Coverage • Local Decision Coverage • Logic Path Coverage The following criteria may be used to assess the coverage of a Formalized Design components stated through State Diagrams: • Range coverage	As the main output of the SOMCA study, the assessment from section 9.6.1.8 shows that the SOMCA criteria from section 9.4.3 ensure that the model is thoughtfully exercised, being more effective in the identification of Unintended Functions in a Formalized Design. In addition, as stated in the FSR the different formalisms are so different that specific criteria should be employed for each notation to effectively exercise each design model. Therefore, the original MC criteria for Formalized Designs has been detailed with the SOMCA criteria for Formalized Designs stated through Block Diagrams and for Formalized Designs stated through State Diagrams, while another generic set of criteria has been defined for other notations. Note that the original CM criteria about "all the conditions of the logic components" has been reused in the generic criteria for other notations, while the other original criteria have been replaced by the equivalent SOMCA criteria, to leverage their more detailed definition, while adding the Modified input coverage and Activation coverage, which could be generically used in other notations.

#ID	SOMCA Items	CM Section	Cert Memo Modification	Justification
			• Functionality Coverage • Modified input Coverage • Activation Coverage • Parent State Coverage • State History Coverage • Event Coverage • Activating Event Coverage • Transition Coverage • Transition Decision Coverage • Transition MC/DC • Level-N Loop Coverage The following criteria may be used to assess the coverage of a Formalized Design stated through other notations: • all the conditions of the logic components • Range Coverage • Functionality Coverage • Modified input Coverage • Activation Coverage In addition, at the end of section 23.2.9, add new subsections 23.2.9.1 to 23.2.9.14 describing each criterion as they appear in the FSR section 9.4.3, and add the table specifying the mapping of criteria and criticality levels from FSR section 9.4.4.	
1	SOMCA Prerequisite 1: ED-12B Applicability	23.1 Background	(no change needed)	Prerequisite already covered in the certification memo.
2	SOMCA Prerequisite 2: Source Code Coverage	23.2.10.6 Structural Coverage of Source / Object Code.	Replace: should be demonstrated to the degree required for the DAL of the software.	The Structural Code Coverage of source code imported in a Formalized Design has to be verified externally to the model using the same applicable

#ID	SOMCA Items	CM Section	Cert Memo Modification	Justification
			By: should be demonstrated to the degree required for the DAL of the software. Applicants should identify if the structural coverage of imported source code within the Formalized Design will be demonstrated through simulations.	criteria from ED-12B, but can be measured through simulations in some situations.
3	SOMCA Prerequisite 3: Model - Requirements Traceability	23.2.10.1 Traceability and Granularity of Requirements / Design Elements	(no change needed)	Prerequisite already covered in the certification memo.
4	SOMCA Prerequisite 4: Requirements – Model Traceability			
5	SOMCA Prerequisite 5: Simulation configuration	23.2.8.3 Considerations on simulation cases, procedures and results	Add at the end of the enumeration: - An adequate tuning of sample time and other simulation parameters has been performed.	The simulation parameters can alter the behaviour of the model if they are not properly configured.
6	SOMCA Prerequisite 6: Modelling of Target Platform	23.2.8.3 Considerations on simulation cases, procedures and results	Add at the end of the enumeration: - Specific simulations and tests should be exercised to ensure the target platform is accurately modelled.	The simulations should be as close as possible to the executions in the final platform, and thus it must be configured to take into account the specific characteristics of the target platform like numeric formats available.
7	SOMCA Prerequisite 7: Analysis of simulation differences	23.2.8.3 Considerations on simulation cases, procedures and results	(no change needed)	Prerequisite already covered in the certification memo (*The simulation results are correct and that discrepancies between actual and expected results are explained*)
8	SOMCA Prerequisite 8: Type check	23.2.10 General principles and activities	Add new subsection 23.2.10.x with this prerequisite.	This is a general verification activity for the model.
9	SOMCA Prerequisite 9: Uniform Model Verification **Strategy**	23.2.8.3 Considerations on simulation cases, procedures and results	Add at the end of the enumeration: - All model components shall be simulated and tested while exercising the complete Formalized Design.	Partial model verification or coverage analysis should be avoided if possible, to avoid behavioural differences in the execution of independent model components.
10	SOMCA Prerequisite 10: Definition of Tool versions	23.2.2 The system / software planning	(no change needed)	Prerequisites already covered in the certification memo (*State which tools will be used during the*

#ID	SOMCA Items	CM Section	Cert Memo Modification	Justification
11	SOMCA Prerequisite 11: Tool and notation suitability	process		*development and verification of their conventionally developed software components and which will be used for development and verification of their formalized system / software components*)
12	SOMCA Prerequisite 12: Transfer function stability	23.2.10 General principles and activities	Add new subsection 23.2.10.x with this prerequisite.	This is a general verification activity for the model.
13	SOMCA Prerequisite 13: Model documentation	23.2.10 General principles and activities	Add new subsection 23.2.10.x with these prerequisites about model documentation.	This is a general verification activity for the model.
14	SOMCA Prerequisite 14: Barrier identification			
15	SOMCA Prerequisite 15: Environment definition			
16	SOMCA Prerequisite 16: Warning free simulation	23.2.8.3 Considerations on simulation cases, procedures and results	Add at the end of the enumeration: - All tool warnings during a simulation should be considered as errors except if the warning is properly justified.	Warnings during simulations usually point to modelling errors. Unless justified (e.g. a false positive), all simulation warnings must be considered errors.
17	SOMCA Prerequisite 17: Requirement of Design Standard	23.2.2 The system / software planning process	(no change needed)	Prerequisite already covered in the certification memo (*If a Formalized Design will be used, identify the design standards with which the Formalized Design should comply*)
18	SOMCA Prerequisite 18: Explicit priority for Transitions	23.2.10 General principles and activities	Add new subsection 23.2.10.x with these prerequisites.	This is a general verification activity for the model.
19	SOMCA Prerequisite 19: State Reachability			
20	SOMCA Prerequisite 20: Recursive SOMCA Applicability	23.2.9 Coverage of formalized designs	Add at the end of the section: Hierarchical constructions within a Formalized Design the criteria shall be applied recursively for the adequate notation unless otherwise specified in a criterion.	New paragraph to handle Formalized Designs composed of multiple notations.
21	SOMCA Prerequisite 21: Input / Output assertions	23.2.10 General principles and activities	Add new subsection 23.2.10.x with these prerequisites.	This is a general verification activity for the model.
22	SOMCA Prerequisite 22: External timing management			
23	SOMCA Recommendations	23.2.2 The system / software planning process	(no change needed)	The SOMCA recommendations should be part of “the design standards with which the Formalized Design should comply”.

Table 9-17: List of Cert Memo changes to integrate the SOMCA criteria.

9.9. POSSIBLE CRITERIA EVOLUTIONS

During the course of the present study the SOMCA criteria were refined in different iterations, discarding the least effective new criteria. However, some of the envisioned Model Coverage Criteria were not mature enough to be included in the final SOMCA criteria, although could be candidates for a future evolution. This section introduces some of those possible evolutions for an advanced set of Model Coverage Analysis criteria, with the aim of inspiring further research in this area.

9.9.1. FORMAL MODEL VERIFICATION

Some tools include features that allow performing formal model verification for a given set of properties. This is the case of SCADE Design Verifier.

This kind of analysis is not addressed in this study, but is clearly a powerful and interesting verification activity within the scope of Unintended Function detection. Currently this kind of analysis is not included in ED-12B / DO-178B, but the upcoming ED-12C / DO-178C will include a specific supplement about Formal Methods. However, it would be interesting to analyse the implications of using formal proofs within a MBD approach.

9.9.2. COVERAGE OF NUMERIC FLOWS

While the coverage of model elements handling logical values can be considered comprehensive enough to ensure a proper verification of a Formalized Design, the verification of data flows involving numeric types is not so mature. We feel that this topic should be improved, maybe refining an existing criteria like Range coverage or with the creation of a new specific criteria.

The **SOMCA Criterion 1: Range coverage** is focused on the analysis of local range coverage, making the analysis independent for each model component. But, in the same way the Logic path concept described in section 9.4.3.6 introduces a relationship between the local coverage of each block, for arithmetic blocks it could be done the same way. The result of this will be the concept of “Arithmetic Path”. This will increase the scope of the path coverage to all variable types.

As an example of the utility of this criteria improvement, we can analyse the part of the model used in inject a UF example in section 13.1.2.1.6. The following figure shows the model part to be analysed.

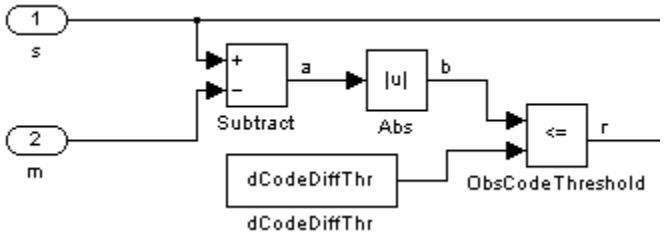


Figure 9-32: Missing Abs block (previous to UF injection)

If we analyse all blocks independently, we can achieve full coverage without covering all the possible situations of the entire subsystem. We will take `dCodeDiffThr` as a constant (value 10) to explain the problem. Consider the following two test cases:

Test Case	S	M	A	B	R
#1	50	1	49	49	False
#2	2	3	-1	1	True

With these two test cases we have achieved full coverage. The Abs input value has taken both possible ranges, positive and negative, and the ObsCodeThreshold comparison have taken both possible output values. However, the subsystem functionality has not been fully tested. There are combinations of values that not have been tested, like an A negative value higher than the threshold that generates a negative output value.

The main difficulty in the definition of this procedure is that for logic variables the possible values are restricted to two: True and False. Enumerated values could be evaluated also with the same procedures, but integer and real values have infinite possible values, which forces us to follow another approach. The approach proposed is to make an arithmetic resolution of the Ranges and singular points of all blocks, starting from the last one until all the range rules depends only on model input or constants.

This methodology has been tested on this example, generating the following intervals that characterize the entire subsystem (We are not taking the zero value as a singular point, but it could be easily included in this analysis).

- $S \geq M$ and $S - M \leq d$
- $S \geq M$ and $S - M > d$
- $S < M$ and $M - S \leq d$
- $S < M$ and $M - S > d$

Based on these conditions, we can update the verification cases to fulfil the coverage criteria:

Test Case	S	M	A	B	R
#1	50	1	49	49	False
#2	2	3	-1	1	True
#3	3	2	1	1	True
#4	1	50	-49	49	False

These four conditions fully cover all the possibilities that the described subsystem could manage. The analysis is simple, and allows generating more efficient criteria for the coverage of arithmetic blocks.

This approach is situated between the formal analysis and the model coverage, providing the best of both. However, it needs a more deep investigation to analyse if it can be used in bigger models and subsystems in an efficient way. This is an open point for future studies.

10. CONCLUSIONS

The SOMCA study has followed an iterative approach for the elaboration of a Model Coverage Criteria starting from the Certification Memo [RD.3] provided by EASA and finalising with the production of the SOMCA Model Coverage Analysis, approach defined in section 7. The work was organised into a set of Work Packages and Tasks as described in detail in section 8. A summary is provided here:

- Task 1: Model Formalism, aimed at investigating the modelling notations and tools used in MBD in the aeronautic industry. The concept of UF was also investigated.
- Task 2: Model Coverage Criteria. Define a set of model coverage analysis criteria.
- Task 3: MCA Methods and Tools. Summary of MCA methods and tools, and how commercial tools implement and assess model coverage.
- Task 4: Study Case. Preparation of generic examples and real study case.
- Task 5: Analysis of the relation of the code coverage and model coverage.

The first step was to establish a definition for the Unintended Function concept, including a taxonomy of UFs and the review of the software lifecycle to identify possible sources of UFs. This first step included also a systematic search of examples of UFs using available literature, GMV's experience, and the analysis of basic modelling blocks. This information was very valuable not only to understand the problem under analysis but also to ensure that the assessment of the SOMCA Model Coverage Criteria was complete.

The second step was the definition of the first SOMCA criteria version started with a sound analysis of available literature to establish a definition of Model Coverage Analysis and Model Coverage Criteria, and the already existing coverage strategies at structural and model level:

> **Model Coverage Analysis**: *is an analysis that determines which requirements expressed by the model were not exercised by verification based on the requirements from which the model was developed.*

This definition from the Certification Memo [RD.3] brings to light the first important remark that was later on confirmed by the experimentation with real examples: a model coverage analysis is a way to measure the completeness of the V&V activities, determining which parts of the model have not been properly exercised by the verification cases and procedures, not a technique to directly spot the problems contained in the model.

To complement the definition of the Model Coverage Analysis, the two most representative modelling tools were analysed, SCADE and Simulink, and in particular how they support Model Coverage analysis compared with the SOMCA Model Coverage Criteria. Important conclusions were extracted from this activity. Both tools support the state diagrams and the block diagrams formalisms, but each tool implements its own notation for these design formalisms, and the way this notation is translated into code depends on the tool; additionally, each tool measures and understands model coverage in a different way. Finally, it was found that, due to its intrinsic nature, different sets of criteria for block diagrams and for state diagrams would be needed.

The following step was to perform the initial definition of the SOMCA Model Coverage criteria; this process was guided by the following goals: high capability of UF detection, applicable to all the model elements and characteristics, scalability, adjustable for the different software criticality levels, applicable before source code has been developed, and easy to learn.

This first definition was based on three main inputs: first the Certification Memo, second the previous existing work, and third, and most important, the processing of the outputs of the TASK 1. During the activities developed as part of this task, a set of recommendations was generated as input to the following phases of the project. The processing and refinement of these recommendations provided three types of outputs:

- Criterion: an objective activity vital for detecting some UF types, and that must be enforced and analysed for all model elements. Part of the MCA criteria.
- Prerequisite: an objective activity that should be applied to the Formalized Design before performing the Model Coverage Analysis, that is not enforced per model element but for the whole design or group of model elements (e.g. a component). Intended to be part of an additional verification activity related to MCA.

- Recommendation: a subjective recommendation that could be useful to avoid introducing further some UF types, or to ease detecting them, but too prescriptive to be added as an MCA criterion or prerequisite. Cannot be considered a verification activity, and can refer also to recommendations related to the source code.

All this information was compiled into the SOMCA criteria; the following table provides the list of the SOMCA criteria with the applicability to the different DAL levels, note that Prerequisites and Recommendations are not included in this table:

Criterion Name	Block Diagrams DAL					State Diagrams DAL				
	A	B	C	D	E	A	B	C	D	E
SOMCA Criterion 1: Range coverage	√	√	√			√	√	√		
SOMCA Criterion 2: Functionality coverage	√	√	√			√	√	√		
SOMCA Criterion 3: Modified input coverage	√	√	√			√	√	√		
SOMCA Criterion 4: Activation coverage	√	√	√			√	√	√		
SOMCA Criterion 5: Local Decision Coverage	√	√				NA	NA	NA	NA	NA
SOMCA Criterion 6: Logic Path coverage	√					NA	NA	NA	NA	NA
SOMCA Criterion 7: Parent State coverage	NA	NA	NA	NA	NA	√	√			
SOMCA Criterion 8: State History coverage	NA	NA	NA	NA	NA	√				
SOMCA Criterion 9: Transition coverage	NA	NA	NA	NA	NA	√	√	√		
SOMCA Criterion 10: Transition Decision Coverage	NA	NA	NA	NA	NA	√	√			
SOMCA Criterion 11: Transition MC/DC	NA	NA	NA	NA	NA	√				
SOMCA Criterion 12: Event coverage	NA	NA	NA	NA	NA	√	√	√		
SOMCA Criterion 13: Activating event coverage	NA	NA	NA	NA	NA	√	√			
SOMCA Criterion 14: Level-N Loop coverage	NA	NA	NA	NA	NA	√ (3)	√ (2)	√ (1)		

Table 10-1: SOMCA Criteria applicability depending of SW-DAL

Once a preliminary set of SOMCA criteria was provided, the next phase was the refinement and assessment of the criteria, this refinement was performed by challenging the criteria against different cases:

- Theoretical UFs, mainly identified during the first stage of the project (Task 1).
- Study Case with real example for block diagrams. The objective of this study case is to assess the SOMCA criteria in a real-world development. To do so, an operational critical piece of SW developed with a traditional methodology was selected, and the same subsystem was developed following MBD techniques. Finally the SOMCA criteria were applied to this model with assurance level A. To enrich the output of this activity the model was developed using two different modelling tools: GMV used Simulink and Esterel Technologies provided the model developed by SCADE. The subsystem selected was a smoothing stage based on a Hatch filter.
- Study case with real examples for state machines. The real-world example selected to assess the SOMCA criteria for block diagrams did not contain any state machine complex enough to perform a real assessment of the SOMCA criteria related to state diagrams. Therefore it was decided to analyse a set of complex state machines to perform a realistic assessment.

After these activities a complete assessment of the SOMCA criteria was performed with a set of maps indicating the detection capability of the different types and examples of UFs (real and theoretical) and the types of UFs covered by the SOMCA criteria.

As indicated in previous sections the outputs of all these activities were compiled into two Interim Reports and this Final Study Report.

10.1. PARTIAL CONCLUSIONS

During the elaboration of the SOMCA criteria in addition to the main conclusion, namely SOMCA MC criteria, prerequisites and recommendations, a set of secondary conclusions were identified. This section provides a summary grouped by topics:

SOMCA criteria definition

1. This study confirms that Model Coverage is a efficient way of detecting Unintended Functions introduced at the level of a Formalized Design.
2. It also confirms that criteria defined initially by EASA were necessary however the study also revealed that they were not sufficient to address all types of Unintended Functions.
3. The study provides a set of recommendations for the update of the EASA Certification Memo,that is based on concrete evaluation of theoretical and real-world examples involving UF.
4. This study could not establish whether the SCA and MCA could be equivalent under given conditions. This will require additional research and, due to the inherent difference between those two analyses, it is not trivial to establish such an equivalency.
5. The Certification Memo has been successfully refined into a more general criteria, the main reason is that the UF concept and scope have been widen in this study, creating a more generalized concept of UF.
6. There is a set of pre-requisites and recommendations that, although not part of the criteria, would enable the SOMCA criteria to detect the UFs or would increase its detection power. Therefore, before applying the SOMCA criteria a set of pre-requisites should be checked.
7. Lack of homogeneity in Tool Formalisms: The different tools present in the market, even based in the same theoretical formalisms, implement in different ways the block diagrams and state diagrams, making very difficult to establish relationships between them, and the different features of each of them suggest that some criteria should be specially designed or adapted to those special features.
8. Each commercial tool measures and understands model coverage in a different way, and the analysed toolsets do not directly support the criteria specified in the Certification Memorandum or the SOMCA MCA criteria.
9. Any Model Coverage Criteria, to be complete, should consider specific criterion to address block diagrams and state diagrams.
10. The possibility of having source code embedded in the model should be considered by the pre-requisites or criteria.
11. Most of the complex UFs to detect are those related with blocks with arithmetic components and managing integers and real numbers. The section 9.9.2 includes an initial study trying to enhance the detection of this kinds of UFs by extrapolating the concept of Logical Path to “Arithmetic path”, however the analysis would need further effort and has not been included as part of the final SOMCA criteria.
12. Some of the proposed examples of UF show that the timing constrains are external to the model design. In critical and real-time systems, this kind of constrains are very common and used to be very restrictive. It could be interesting if the tools could provide this kind of functionality to include it in the model.

SOMCA Assessment

1. The SOMCA criteria have been successfully tested against the different types of UFs, both real-world and from the lab, and the final conclusion is the following:

Examples	Number of examples	Total detected	Undetected
Theoretical	19 (100%)	18 (95%)	1 (5%)
Real-World	28 (100%)	23(82%)	5 (18%)
Total	**47 (100%)**	**41 (87%)**	**6 (13%)**

Table 10-2: Summary of SOMCA Criteria assessment

c) For the theoretical examples (19 examples):

i. A 95% of the UF examples were detected or eliminated by the SOMCA Criteria and Prerequisites.

ii. Just one of them was not detected at all by the criteria or the prerequisites (5%). This state machines example comes from the literature [RD.14], where a formal analysis tool was used to detect it.

d) Real-world examples (28 examples):

i. The SOMCA Criteria and Prerequisites detected or eliminated an 85% of the UFs.

ii. 15% was not detected by the criteria. In this case, 3 out of the 5 non-detected UFs were related with problems in the specification. The remaining one was not identified because its magnitude is so low that it is in the range of the tolerance due to the different implementation between the Operational example and the modelled one.

2. The proposed criteria can detect much more UF occurrences than the original Certification Memo criteria.

3. Most of the UFs not detected are related with problems in the specification, this is relevant because the analysis of the correctness of the specification (higher level of abstraction) is a complex problem, it would need further investigation but it is expected that notations for Formal Requirements would greatly help to reduce this type of UFs.

4. A significant number of UFs were detected by the application of the pre-requisites. This means that good development practices, adequate experience of the designer, and employing design- and specification- standards / guidelines are a must to reduce the probability of injecting UFs.

5. Some SOMCA Criteria have detected no UF within the real-world examples, as indicated in section 9.6.2.3, but the previous analyses encourage maintaining them as part of the criteria because their added value is clear for properly exercise a design model.

Relation of SOMCA criteria with other V&V activities

1. Formally speaking, the SOMCA criteria should be applied after the rest of the V&V activities, but an early execution of it would provide valuable feedback on the quality of the model.

2. As the definition of the Model Coverage Analysis states, the exercise of SOMCA criteria in real examples has revealed that Model Coverage Criteria are, at the end of the day, a measure of the quality and completeness of the V&V activities, providing an assessment on whether the defined test input vectors are comprehensive enough to exercise the whole model (i.e. the level of coverage achieved). In fact, during the analysis of the model generated to compare with the operational SW, the set of Test Cases defined was relaxed in order to allow UFs to pass through V&V activities to challenge the Criteria. Therefore if the V&V campaign were perfect no MC would be needed, and in the other way around, any UF detected by the MC criteria reveals a weak point in the V&V activities, mainly in the definition of the test input vectors.

3. The introduction of UFs to check the verification procedures can detect itself new hidden UFs. Once the model have been developed and verified, the UF injection discovered several new unexpected UFs.

Relation between Structural Code Coverage with Model Coverage

1. For the time being, and in the scope of the SOMCA project, it is not possible to define a direct relationship between structural and model coverage. Model Coverage and Structural Code Coverage not necessary address the same type of problems and they are referred to different levels of abstraction. Therefore, the open questions are:

   e) How can be compared the completeness and assurance level provided by Model Coverage with Structural Code Coverage?

   f) Does the application of SOMCA MCA criteria at model level guarantee a given level of Structural Code Coverage in the associated source code?

   g) In which conditions the SOMCA MCA criteria can provide the same protection as Structural Code Coverage?

10.2. FINAL CONCLUSION

The SOMCA study, as requested in the objectives, has provided a deep analysis for Formalisms used in MBD techniques in the context of the aeronautic industry, focusing in the two most representative tools SCADE (Esterel) and Simulink (MathWorks). Also a sound analysis of the UF concept (definition, taxonomy and sources) has been performed providing interesting inputs to the definition of the SOMCA Criteria.

The main conclusion of this study is that Model Coverage is an efficient way of detecting Unintended Functions introduced at the level of a Formalized Design. The Model Coverage Provided by EASA, as part of the Certification Memo [RD.8] section 23.2.9, was necessary although the study has revealed that they were not sufficient to address all the types of Unintended Functions. Therefore, it was necessary to complement these criteria to extend the power of detection of Unintended Functions in a Formalised Design.

The refinement of the EASA criteria is the SOMCA Model Coverage Criteria provided in section 9.4. The Model Coverage Analysis is composed by a set of criteria, pre-requisites, and recommendations that conforms the SOMCA criteria. These three outputs: criteria, pre-requisites and recommendations, are considered the set of recommendations to be used to amend the certification material and guidelines. As such, section 1.1 lists the changes needed in the Certification Memo to include the SOMCA criteria, which is therefore an essential input to the elaboration of the future ED-12C.

Since nothing has been identified that prevents this objective, and the successful result of the SOMCA assessment (refer to section 9.6), the SOMCA criteria is in the good way to be accepted as solely Acceptable Means of Compliance within EASA regulatory framework in order to get the airworthiness type certification for airborne equipment and systems developed using MBD techniques.

Additionally, it has been found that the SOMCA criteria provides an important added value to assess and enhance the quality of the V&V activities performed at model level specially the quality of the test input vectors, and hence of the final product, advancing the detection of UFs and verification flaws to early stages of the project.

Finally, this study could not establish whether the Structural Coverage Analysis and Model Coverage Analysis could be equivalent under given conditions. This will require additional research and, due to the inherent difference between those two analyses, it is not trivial to establish such an equivalency.

Future work

Among the possible steps after SOMCA the following points are highlighted:

- Investigation of the equivalence between Structural Coverage Analysis and Model Coverage Analysis and under which conditions could be possible.

- Development of the “Arithmetic path” concept and associated coverage criteria to increase the detection power of the SOMCA criteria for blocks containing arithmetic operations, since UFs related with this type of operations have been the most difficult to detect.
- Formal Specification and Formal Verification methods and the associated reduction of injection and misdetection of UFs.
- Use information coming from the real world for final minor refinement of the SOMCA criteria.
- Finally other interesting analysis could be: Formalized Requirements notations, feedback from relevant aircraft / equipment manufacturers, analysis of data coupling and control coupling at model level, etc.

11. KEY TERMS

This section defines some key concepts of the study, in order to better understand this document.

Formalized Design or Design Model: System and / or software design described using a specialised modelling notation or formal language. In other words, a Design Model prescribes software component internal data structures, data flow and/or control flow. A Design Model includes Low-Level Requirements and/or architecture. In particular, when a model expresses software design data, regardless of other content, it should be classified as a Design Model. This includes models used to produce code.

Formalized Requirements or Specification Model: System and / or software requirements specification by means of a modelling notation or formal language. In other words, a Specification Model is an abstract representation of externally observable properties of a software component, such as interface, functional, performance or safety characteristics. The Specification Model should express these characteristics unambiguously to support an understanding of the software functionality. It should only contain detail that contributes to this understanding and does not prescribe a specific software implementation or architecture except for exceptional cases of justified design constraints. Specification models do not define software design details such as internal data structures, internal data flow or internal control flow. Therefore a Specification Model may express high-level requirements but neither low-level requirements nor software architecture.

High-Level Requirements (**HiLR**): A set of requirements at a higher-level of abstraction that capture the requirements for the Formalized Requirements or Formalized Design. They should be developed and stated in a different manner than is used for the Formalised Requirements or Formalized Design. The “Higher-Level Requirements” are those requirements from which the model has been developed.

Low-level requirements (**LLR**): Any requirement contained in a Design Model.

Validation: determination that requirements are correct and complete. May be performed by reviews, analyses, simulation or a combination thereof.

Verification: determination that the implementation meets the requirements. May be performed by test, simulation, analyses or a combination thereof.

Requirements-based Testing: The process of exercising software with test scenarios written from the High-Level Requirements, not from the source code (cf. Structural Testing).

Structural Testing: The process of exercising software with test scenarios written from the source code, not from the requirements (cf. Requirements-based Testing).

Testing: The process of exercising the executable object code to verify that it satisfies specified requirements and to detect errors.

Simulation: The process of directly exercising a model or model component to verify that it satisfies specified requirements and to detect errors.

Test Case: A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

Test Procedure: Detailed instructions for the set-up and execution of a given set of test cases, and instructions for the evaluation of results of executing the test cases.

Simulation Case: A set of simulation inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular model target or to verify compliance with a specific requirement.

Simulation Procedures: Detailed instructions for the set-up and Simulation of a given set of Simulation Cases, and instructions for the evaluation of results of simulating the Simulation Cases.

Verification Cases and Procedures: A set of Test Cases, Test Procedures, Simulation Cases, and Simulation Procedures.

Model-Based Development: Development process in which models represent software requirements and/or software design descriptions to support the software development and verification processes and from which other artefacts, for example source code, test cases or even simulation cases, might be manually produced or automatically generated.

Model Component: A self-contained part or combination of parts of a model, which performs a distinct function.

Model Element: A unit from which a model is constructed. The main elements used in Block Diagrams are Basic Blocks, Component Blocks, and data Links. For State Diagrams the main elements used are States, and Transitions. Depending on the tool and the notation used, some elements may differ.

Formal Methods: the use of mathematical notation to describe, in a precise way, the properties that a system must have, regardless of the particular implementation. Formal methods are based on the use of logic and discrete mathematics in specification, design and construction of computer systems and software.

Unintended Function (**UF**): any unspecified —not defined in the higher-level requirements— and uncontrolled behaviour of the software under the aeroplane operating and environmental conditions.

Coverage Analysis: The process of determining the degree to which a proposed software verification process activity satisfies its objective.

Requirements-based Test Coverage Analysis: Analysis that determines which software requirements were verified by Requirements-based Testing, including normal and robustness testing.

Structural Code Coverage: Measurement of the degree to which the structure of the source code has been executed through Test Cases and Procedures.

Structural Coverage Analysis: Analysis that determines by which degree the source code structure were not exercised by Test Cases and Procedures based on the requirements.

Model Coverage: Measurement of the degree to which the structure of a Formalized Design has been exercised through Verification Cases and Procedures.

Model Coverage Analysis: Analysis that determines which low-level requirements expressed by a Formalized Design were not exercised by verification based on the Higher-level Requirements. Model Coverage Analysis implies analysis of the completeness of the verification of the model.

12. APPENDIX A: STRUCTURAL CODE COVERAGE

This appendix describes the concept of coverage analysis at source code level. This was used as an input for the definition of the Coverage Criteria at model level, but please also note that these criteria need to be employed in the coverage of those Formalized Designs importing external elements.

Several Structural Code Coverage techniques have been defined for source code. ED-12B defines the following Structural Code Coverage criteria, each one required for different criticality levels:

1. **Statement Coverage**: Every *statement* in the program has been invoked at least once.
2. **Decision Coverage**: Every point of entry and exit in the program has been invoked at least once and *every decision* in the program has taken on all possible outcomes at least once.
3. **Modified Condition/Decision Coverage (MC/DC)**: Every point of entry and exit in the program has been invoked at least once, *every condition* in a decision in the program has taken all possible outcomes at least once, *every decision* in the program has taken all possible outcomes at least once, and *each condition in a decision has been shown to independently affect that decision's outcome*. A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions.

Where these coverage criteria are defined based on the following source code elements:

- **Condition**: A Boolean expression containing no Boolean operators.
- **Decision**: A Boolean expression composed of conditions and zero or more Boolean operators.

One important consideration from the ED-12B definition of MC/DC is that if a condition appears more than once in a decision, each occurrence is a distinct condition (coupled conditions). This is called “Unique-cause MC/DC”, and it means that just decisions with uncoupled conditions can be shown to independently affect a decision, even for correct code. To be able to analyse coupled conditions, “Masking MC/DC” leverages the masking effect of some logic blocks [RD.19]. For example, an “and block” masks the logic paths that pass through its first input if for a given input vector the value of any of the other inputs of the “and block” is False, and thus this first input does not contribute to the result of the logic path regardless of its value (and thus is “masked”). In contrast, if all the other inputs of the “and block” are True for a specific input vector, the block doesn’t mask the logic paths that pass through the first input, thus contributing to the result of the logic path.

For each software Design Assurance Level a different set of coverage criteria is requested:

Criterion name	DAL				
	A	B	C	D	E
Statement Coverage	√	√	√		
Decision Coverage	√	√			
Modified Condition / Decision *Coverage*	√				

Table 12-1: ED-12B Structural Code Coverage criteria

Furthermore, for Level A any extra object code added by the compiler not directly traceable to the source code must be analysed for necessity and correctness, and the compiler libraries need to be verified as the application source code for level A and B.

Besides the Structural Code Coverage criteria defined in ED-12B, there are more demanding coverage techniques identified in the literature review. The goal of Path Coverage criterion is to exercise *all possible paths* in the source code, which means exhaustively testing the software. Actually, the number of possible paths may be infinite if the software contains unbounded loops. Just trivial programs can be exhaustively tested, and Path Coverage is not a feasible coverage criterion in practice, but it is interesting for comparison

purposes with other criteria: The coverage criteria actually used industrially do not ensure that all the program behaviours have been tested (that would require using the Path Coverage criterion), and thus testing alone cannot ensure that no error remain.

In Multiple Condition Coverage every statement in the program has been executed at least once, and *all possible combinations of condition outcomes* in each decision have been invoked at least once. This requires 2^n test cases for each decision composed of n conditions (MC/DC requires $n+1$ tests instead), which exhaustively tests all the possible combinations. Even if the number of needed tests is finite, Multiple Condition Coverage is unfeasible too for non-trivial software programs. But like the Path Coverage criterion, is interesting for comparison purposes: Decision Coverage or MC/DC just test a reduced number of cases for each decision (but representative), so it is not guaranteed that all bugs have been removed.

Please note that in those Formalized Designs with external elements imported into the model –like Ada source code implementing the functionality of a block, or a transition action– the proposed SOMCA Model Coverage criteria have to be complemented with Structural Code Coverage techniques for the implementation of those external elements. A library of basic model elements used to produce a Formalized Design is subject to the full guidance of ED-12B, including the objective of Structural Code Coverage.

13. APPENDIX B: DETAILED UF EXAMPLES

In addition to the UF examples presented in section 9.6.1, this appendix contains the model examples used for challenging the SOMCA MCA Criteria. It contains both Theoretical UF examples and actual models based on real-world certified software.

The largest model is going to be generated using Simulink and SCADE. The major part of the Smoothing model is going to be modelled using Block diagrams due to the nature of the model (data processing). Small State Machines will be included to manage the internal state of the algorithm, but due to the small size of them, they won't be suitable to check the SOMCA Criteria associated with State Machines. Because of this, the SOMCA analysis will focus on the general and Block diagram criteria.

As the Smoothing model does not cover all the formalisms and UF types included in this study, some other smaller examples are going to be included in order to challenge the SOMCA Criteria not exercised by the Simulink. The original idea was to use the Cruise Control example provided by Esterel in the SCADE Suite as part of the main examples. However, that example hasn't been enough, so other examples with State Machines have been included to make a more complete analysis of the SOMCA Criteria UF detection power.

These examples includes some UFs already present in the models and other ones artificially injected to increased the number of Unintended Functions to be discovered. During all the modelling, verification and model coverage activities some interesting issues have also been found. This information has been also included, like some problems found in the coverage tool due to an incorrect configuration (e.g. section 13.1.2.3).

This section will describe all these examples in detail, the UFs found and injected, and the results of the application of the SOMCA Criteria to all these examples.

13.1. SIMULINK SMOOTHING MODEL

As part of the Task 4 of the SOMCA Project, once the criteria were defined, they were challenged against a real-world model, extracted from an actual project. The original real project from which this example has been extracted wasn't developed following a Model Based Design approach. The methodology proposed to develop the model is to generate a set of High Level Requirements (HiLR) based on the source code, and develop the model based on these HiLR. This approach has several benefits:

- The source to generate the HiLR has passed a complete Validation and Verification. This will help in the generation of a very stable set of Requirements, and this will avoid major changes in the design in later stages of the modelling process.
- The filter has been already tested with real data, and this data can be captured to check the model with real data and expected outputs that are correct. This information can be used after the verification process to check how many UFs still remain in the model after the execution of the SOMCA Criteria.

The model proposed to check the criteria is the Smoothing filter, a processing filter that will exercise most of the criteria associated to block diagrams. This filter process pseudo code measurements received from a satellite and generate smoothed and filtered measurements that can be used in higher level algorithms.

The Model Development Plan for the Smoothing model, as well as all the model elements and details, are described in IR2 [RD.2].

13.1.1. APPROACH

The main objective of this task was to check the effectiveness of SOMCA Criteria against a set of Unintended Functions present in the developed Smoothing model.

The initial approach of this task was to introduce some known UFs that were not detected by V&V and the SOMCA criteria was able to find. But after a few tests, we realize that the Verification scripts had a very good quality and they performed a very good analysis over the generated data, and most of the UFs introduced in the model were detected during the verification. This reinforces the fact than the coverage criteria are a tool that can find errors and weak points in the verification process, not a way itself to find problems in the model. Because of this, if the verification procedure has a very good quality and is very strict, the problems found by Coverage Analysis will be very few.

Based on this, we changed the approach to follow in this analysis. The new proposed approach to follow is similar, we will introduce manually some known UFs in the model, but additionally to already developed verification, we are going to reduce the strictness of the verification scripts, making it less powerful, in order to show the power of SOMCA criteria on weaker verification environments.

13.1.1.1. Tailoring the SOMCA Criteria

The proposed SOMCA Criteria is a merge between already existent Coverage methods and a set of new or modified ones. Because of this, the coverage tools used in the modelling process does not include any of these new criteria. This generates an implicit limitation to the application of the SOMCA Coverage, because most of the developed criteria are not automatic, and must be done semi-automatically or completely manually. For a detailed relationship between SOMCA Criteria and Simulink Coverage Tool support, see Section 9.5.3.

The fact that the criteria cannot be applied automatically over the entire model introduces the possibility of misdetection of UFs in the coverage analysis. Another possibility is that some of the non implemented criteria analysed by hand won't be able to detect the UF once automated. However, the analysis will be done under these premises to get a first set of conclusions about the proposed SOMCA Model Coverage Criteria.

13.1.2. SMOOTHING MODEL UNINTENDED FUNCTIONS

13.1.2.1. Unintended Functions to be artificially injected

This section describes the list of Unintended Functions to be injected in the model in order to challenge the SOMCA Criteria. For each UF the following analysis is provided:

- Description of the UF
- Results of SOMCA Criteria application with nominal verification
- Results of SOMCA Criteria application with poor verification (if needed)

13.1.2.1.1. KSM Unlimited

In `SmoothProcess` Block, the `AcsOkAndKsm` AND Boolean operator block check that the `KSm` parameters is always below the configured limit (`siMaxK`). If this block, by error, is set to an OR operation instead of an AND one, the check against the maximum value is masked. The system seems to work, and the verification does not complain about the output values, but the `averagedAmbiguity` calculated to check against the thresholds is erroneously configured, and could generate the rejection of valid measurements, generating an UF.

This can be classified as *[UF.3.2] Incorrect behaviour in the general case* because the KSM limit is not checked in any time.

Results with nominal verification

In a first analysis, the SOMCA Criteria does not detect any weak point in the verification process related with this UF. The source of this UF is pure functional, and all logic combinations in the logic blocks related with this UF are achieved. The only way to detect this UF would be to continuously check the ranges of internal variables. If the **SOMCA Prerequisite 21: Input / Output assertions** would have been applied, this UF could have been found, because the input KSM value could have been checked against it maximum value.

The verification is not able to find the UF. However, if the SOMCA Prerequisites would have been met, this UF could have been detected.

13.1.2.1.2. Gauss-Markov Division by Zero

In the Gauss-Markov subsystem, if the Division block is not protected against division by zero, the simulation will run with warnings, and the verification scripts won't be able to detect the problem. But if the code is generated and run in the target platform, the division by zero will generate an exception that will halt the program.

This UF appears when the filter is reset, so it's classified as *[UF.3.3] Erroneous behaviour under specific inputs / conditions*.

Results with nominal verification

The verification is not able to find the UF. The SOMCA Criteria does not find also any problem with the coverage of the verification. However, if the SOMCA Prerequisites would have been met, this UF could have been found thanks to the **SOMCA Prerequisite 16: Warning free simulation**, because it avoids this kind of situations by forcing all warnings to be solved or justified.

13.1.2.1.3. Complexity in Algorithm Details

Due to the complexity of this algorithm, is complicate to verify completely all the values calculated by it (the only way would be to generate an equal independent algorithm in the verification, which is exaggerated). This could lead into algorithm problems not detectable by verification.

An example of this will be the following: If an average of a 100 elements buffer is calculated over the half of it (50 elements), and the result is used by the following stages of the algorithm, this problem will be very difficult to detect because the averages in most of the situations will be almost the same, and the UFs generated will remain undetected for a long time.

In the Simulink example, this could happen in the Gauss-Markov subsystem. If the "Delete Most Recent" and "Delete oldest" blocks would generate a vector of size 50 instead of 99, the averages would be very similar, and the verification won't detect it.

This problem could be mitigated with **SOMCA Prerequisite 8: Type check** (there should be no data type mismatches). All blocks should have the input types and sizes established statically to prevent this kind of errors. In this case, the Simulink Summation block does not allow specifying the size of input signal, so this Prerequisite can't be applied. In the case of SCADE, it performs automatically strict type checking, so this prerequisite would have avoided this UF.

This can be classified as *[UF.3.2] Incorrect behaviour in the general case* because the average is always erroneously calculated.

Results with nominal verification

The verification is not able to find the UF. The SOMCA Criteria is also unable to find any weak point in the verification, but this kind of problem could be mitigated with **SOMCA Prerequisite 8: Type check**.

All blocks should have the input types and sizes established statically to prevent this kind of errors. In this case, the Simulink Summation block does not allow specifying the size of input signal, so this Prerequisite can't be applied in Simulink, but maybe in other tools.

13.1.2.1.4. Gauss-Markov Buffer length Check

In the Gauss-Markov Check, there is a part of the model that checks that there are enough observations in the buffer to check if the Gauss-Markov filter can be applied. In the model is called "RelationalOperator1". If we change this operator from "higher or equal than" to "lower than", the check is disabled, because the condition to apply the rejection of observations is never activated.

The complete verification is not able to detect this problem, because there is another part of the model that checks the Instant Ambiguity and hides this problem. The discovery of this UF is the hint needed to discover a second one detailed in section 13.1.2.2.3: *Coupled functionality between Filter reset conditions.*

Results with nominal verification

The verification is not able to detect this UF. The SOMCA criteria are able to detect the weak point in the verification thanks to **SOMCA Criterion 5: Local Decision Coverage**. The criteria shows that all logic paths that end in "`InstAmbOk`" output parameter of the `GaussMarkov` subsystem never take the value *False*.

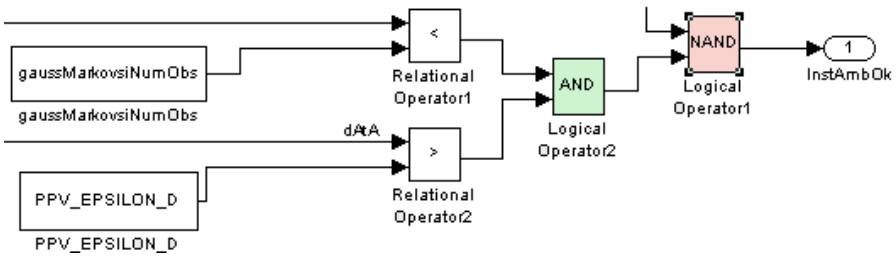


Figure 13-1: Gauss-Markov buffer length check injected UF

This problem would be also be found with **SOMCA Criterion 3: Modified input coverage** because the output value of this block is used as input for other ones, and as it never takes the "true" value, this criterion will complain about it. The Local Decision Coverage will complain also in other points of the model as a consequence of this UF.

13.1.2.1.5. Cycleslip Init Behaviour

The expected behaviour of the model when an `E_CS_INIT` is received in the `eCombinedCycleSlip` input is to reset the filter if the Phase Status is false. To inject an UF we will change this behaviour, making the filter not to be reset.

This Unintended function can be classified as *[UF.3.3] Erroneous behaviour under specific inputs / conditions*, because the Unintended Function only appears with a specific combination of the input parameters.

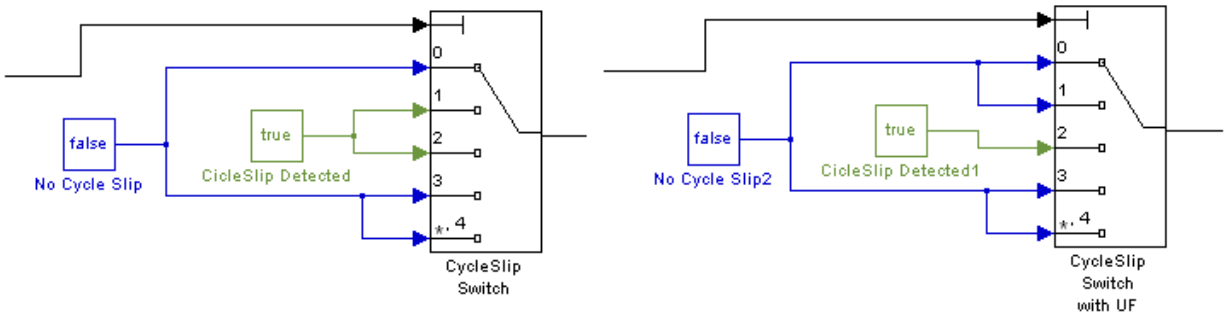


Figure 13-2: Cycleslip process injected UF

Results with nominal verification

The nominal verification tests all possible values of input enumerate and Booleans, detecting the problem when the `E_CS_INIT` is received but the filter is not reset. So, the nominal verification process is able to find the UF.

Results with weak test cases and verification

The nominal verification is able to detect the injected UF, but in a weak verification usually some of these values (or combinations) are not properly tested. In order to challenge the SOMCA criteria, we remove some test cases (and the associated verification) from the input data, generating a weak point in the verification related with the value `E_CS_INIT` for the `CombinedCycleSlipStatus` input Parameter. When this value is activated in the input parameters, the Phase Status is always true. Under these circumstances, the filter reset condition is never tested.

The SOMCA criteria are able to detect the weak point in the verification thanks to the **SOMCA Criterion 6: Logic Path coverage**, because the existing logic path that goes though value 2 of the `CycleSlipSwitch` to the Reset Switch blocks is never activated. This criterion only shows the weak point in the verification scenario, which does not test that condition. In fact, if no UF would have been injected, the output of the criteria would have been the same.

13.1.2.1.6. Missing Abs block

In the `checkCodeSmoothDiff`, when the threshold is checked, the value is converted to a positive value though an Abs block (absolute value). If we remove this block, a lot of measurements that should be rejected because of this check will be accepted. With the nominal verification test cases, this error is easily detected. However, if

in the measurements used for the test cases all the code measurements would have been smaller than the phase ones, this UF would have remain undetected.

This Unintended function can be classified as *[UF.3.3] Erroneous behaviour under specific inputs / conditions*, because the Unintended Function only appears when the difference between Code and Smoothed code is negative and over the configured threshold.

The next figure shows the block diagram without the unintended function:

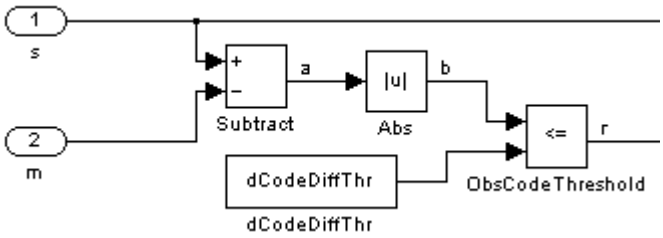


Figure 13-3: Missing Abs block (previous to UF injection)

And the following one shows the same model with the injected Unintended Function (note the missing Abs block):

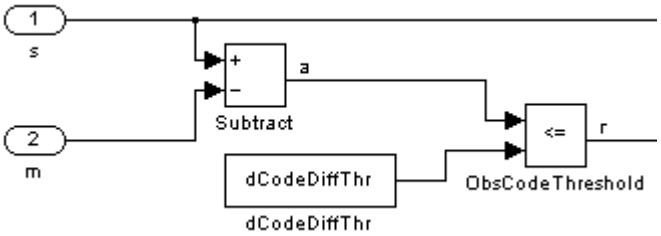


Figure 13-4: Missing Abs block (after UF injection)

Results with nominal verification

The nominal verification process is able to find the UF.

Results with weak test cases and verification

As detailed above, if the measurements used for the validation do not meet the described combination, this UF will remain undetected.

However, thanks to **SOMCA Criterion 1: Range coverage** the output of the *Subtract* block in the `checkCodeSmoothDiff` subsystem would have shown the problem, because the Criterion would have detected that no negative values had been in the *Subtract* block output.

In this case, the discovery of the UF is not ensured. The criterion depends on the ranges tested in the verification. If the ranges are very close to the one that shows the Unintended function (the difference is negative but below the threshold) the UF would remain undetected.

This example shows how local range analysis is not enough to detect all unintended functions for a given set of blocks that include arithmetical operations. See section IR2 [RD.2] for a more detailed analysis of the local range coverage problems.

13.1.2.2. Unintended Functions Already in the model

All UFs described in this section were present in the model, and they have been discovered after the verification process and before applying the SOMCA Criteria. They haven't been removed in order to be used in this study.

For each UF the following info is provided:

- Description and classification of the Unintended Function
- Analysis of which SOMCA Criterion should detect it.
- Results of SOMCA Criteria application (if possible)

13.1.2.2.1. Output Observable and Status

In the specification, the following conditions are covered:

- If the algorithm is not able to generate an output, the outputs should be 0.0 and False (Measurement and status)
- If the status is true, the observable should contain the smoothed value calculated.

But, what should be the value of the Observable when the algorithm is generating an observable but the status is false? (E.g. when the algorithm is converging) The current design set to smoothed value to zero as an interpretation of the first point mentioned above, but the specification is not clear about this point.

This is a clear example of *[UF.2] Specification problem*. The behaviour is not clear in the requirements, generating an undefined behaviour under some circumstances.

No SOMCA criterion, prerequisite or recommendation is able to detect this UF, because it has no symptoms, only a non-defined behaviour that could generate integration problems in the future.

13.1.2.2.2. Units in input parameters

The current specification does not specify the units of the Code and Phase input values. The designed algorithm is independent of input units, as long as all of them are the same for all measurements. These units can be used in two different units: meters and seconds (related by the speed of light).

The problem is that the configuration value for dMaxAmbChange, dCodeVarThr and dCodeDiffThr are stored in meters. If the input values are stored in seconds, the algorithm won't work as expected, because configuration parameters won't be suitable for that input ranges.

As well as for the previous UF, this can be classified as *[UF.2] Specification problem*. The requirement specification can lead into an erroneous configuration.

No SOMCA criterion, prerequisite or recommendation is able to detect this UF.

13.1.2.2.3. Coupled functionality between Filter reset conditions

The introduction of UF described in section 13.1.2.1.4 helped in the discovery of an already present UF in the model: There are two different checks of the Instant Ambiguity, based on different methods, with the same consequence: Reset the smoothing filter.

The Verification was not able to detect this because the consequence in the output and input/output parameters are the same. However, the application of **SOMCA Criterion 6: Logic Path coverage** was able to find a problem in the checkCodePhaseCoherence subsystem, showing that when the logic path that starts in the input parameters bInstantAmbOK and ends in the output parameter ACS OK is activated, the path never takes the value *false*. The result of the application of this criterion was that the GaussMarkov filter reset is always activated at the same time that a CheckCodePhaseCoherence filter reset, and the GaussMarkov reset is never activated alone.

This leads us to a new conclusion, very interesting in the terms of UF detection:

> The introduction of UFs to check the verification procedures can detect itself new hidden UFs.

This conclusion is very close to the principles of Fault Injection, a common technique used in software testing for coverage analysis, but in this case the reason of the injection is not related with increase the coverage but to test the SOMCA Criteria.

Even if the functionality of the Gauss-Markov filter is similar to the checkCodePhaseCoherence, there should be small differences that make the Gauss-Markov filter to be more precise. This UF can be originated by two reasons:

- Incorrect configuration of the input parameters: The Gauss-Markov filter as much as the CheckCodePhaseCoherence block strongly depends on the tuning of the configuration parameters values of the model. The most probable cause for this UF is an incorrect configuration.
- Not appropriate scenario for the verification: The scenario definition can have some problems in the noise introduced in the input data to force the checks to activate. Maybe these modifications of the nominal input data are not appropriate enough to force one check to activate and not the other.
- Real functionality coupling: The last possible reason of this UF is an incorrect definition of the algorithmic criteria to reset the Smoothing filter.

This UF can be classified as *[UF.3.1] Extra functionality*. Section 13.1.3 (Results obtained in a Pseudo-Operational Environment) adds some interesting information about this UF, detected in later stages of the UF analysis.

13.1.2.3. Issues in the coverage results provided by the Tool

During the analysis of the SOMCA Criteria, the Coverage Analysis already present in Simulink has been used as a help in the analysis, automating as much as possible. During this analysis, some unexpected problems were found in the analysis tool.

The results of the analysis were not coherent for some modules. Two identical blocks with the same inputs were reported with different coverage status.

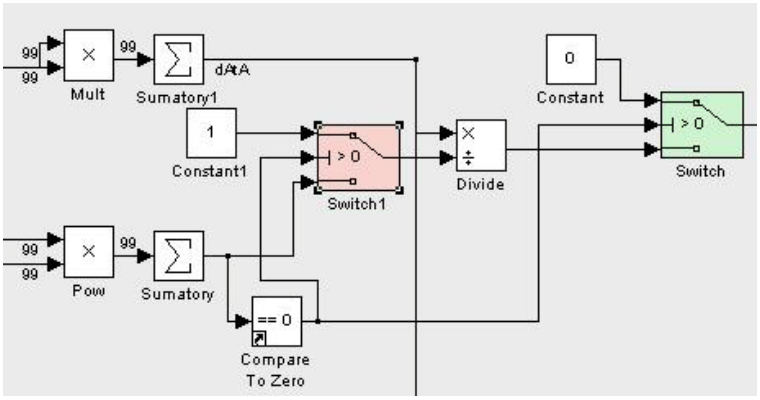


Figure 13-5: Simulink Coverage Error

This kind of issues is quite important, because introduce a new UF source and shows that non-certified coverage tools are not reliable, adding a new conclusion:

> The configurations of the simulation and coverage tool must be carefully specified and reviewed.

After some conversations with MathWorks support team, we discovered that an optimisation option included by default in the simulation configuration had a bug in the resolution of resolution dependencies, generating an error in the coverage reporting that could be easily solved by deactivating that optimisation feature.

As a conclusion of this example, a recommendation was included in Section 9.4.2 :

> For those tools that aren't qualified, the use of optimisations is not recommended.

13.1.3. RESULTS OBTAINED IN A PSEUDO-OPERATIONAL ENVIRONMENT

At this point, we have a Smoothed model developed in Simulink that has been successfully verified, and the SOMCA criteria have been applied over the verification process, tailored for this purpose.

As the last step in the evaluation of the SOMCA Criteria based on the Smoothing example, the model is going to be checked against the input and output values gathered from the system used as a reference. Since the operational software and the model are derived from the same HiLR, they should be equivalent, therefore the same operational inputs will be injected in both systems and the same outputs are expected. Any divergence on the behaviour would indicate that an UF is still remaining in the model.

13.1.3.1. Unintended Functions Detected

The testing against this new set of data has raised several new UFs, not detected by validation nor the applied SOMCA criteria.

13.1.3.1.1. Variance calculation error

In the checkCodeSmoCodeVar there is a block called “100ElementsVariance” that calculates the variance for any number of elements between 1 and 100. This variance is used to check that the difference between the input code and the smoothed code follow a given statistical distribution. This check is one of the main barrier checks implemented in the Smoothing filter.

In the original model, this filter was supposed to be working, and the filter seemed to work for the given data and configuration, but once the real data and configuration were included the barrier was being activated too frequently, resetting the filter continuously when the input data was correct.

Investigating the variance block, an UF was found in the algorithm. Some values that should be ignored were being included in a Summation that polluted the result of the block. The verification was not able to find the problem because the highly tolerant values set in the algorithm configuration allowed the filter to work in a similar way than the normal one.

This is a clear example of an error in block functionality. In Structural Code Coverage, this problem would have been found by the use of unit testing. In model coverage, the already existent coverage criteria are not able to find any problem in the block. If SOMCA Criteria could have been applied to the entire model, this UF would have been detected thanks to **SOMCA Criterion 2: Functionality coverage**.

This UF can be classified as *[UF.3.2] Incorrect behaviour in the general case,* because the averages are always erroneously calculated in the first 100 seconds.

The detection of this UF is related with another UFs detected in earlier stages of the UF analysis. In particular, the UF explained in section 13.1.2.2.3 (Coupled functionality between Filter reset conditions) is strongly related with this UF

13.1.3.1.2. Smoothed Observable masked by Status

This UF is a consequence of a decision taken during the model design which described in section 13.1.2.2.1 (Output Observable and Status). In that section, a problem was found in the requirements about the management of the Smoothed Observable depending on the Smoothed status. The decision taken seemed to

be the most appropriate one. But when the data has been compared with the real one, the behaviour on those situations was not the appropriate one.

This error is clearly derived from an uncertainty in the requirements, and can be classified as *[UF.2] Specification problem*. Due to its nature, this UF can't be detected by the SOMCA Criteria, because the functionality developed is working and it's according to requirements, or at least the understanding of the incomplete requirements.

13.1.3.1.3. Gap processing

The main two inputs of the model are the code and phase measurements. Associated to these inputs, there are both states that indicate if each measurement is valid. When the data is received properly and the status of one of this signals is set to false for a few epochs without errors indicated in the other model inputs, we have what we call a "gap".

The model should be robust against gaps, allowing small gaps of a configurable maximum time. This behaviour is specified in the requirements and modelled according to them. The validation has check that the model is robust against these gaps.

However, when the model has been exercised in the pseudo-operational environment, we have seen that the filter is reset with signal gaps. When this error is analysed deeply, the UF appears. The behaviour when both statuses are set to false at the same time is not defined in the requirements, and also the response of the model is not correct.

Analysing the real operational system that was used as reference for this model, a problem appears in the requirements. When this situation happens, the filter should do nothing, not even execute any of these checks apart from the gap size check. The requirements, developed based on the operational code, didn't capture this part of the algorithm because it is implicit in the code, but not in the requirements.

This is a clear example of *[UF.2] Specification problem* because a problem in the specification generates a typical situation not covered by the verification.

The detectability of this UF is not clear. In a first analysis, it seems that this kind of non-specified behaviour cannot be detected by any SOMCA Criteria. However, in some cases this kind of uncertainty in the specification can lead to an incomplete verification if the scenarios and verification process are completely requirements-based. In that case, the **SOMCA Criterion 6: Logic Path coverage** could find the problem, but not for sure.

13.1.3.1.4. Input value of `CodeSmoothedCodeVar` check

Previously, in the UF described in section 9.6.1.5 (Variance calculation error) the block described was as a filter that checked that the difference between the code and the smoothed code follows a specified statistical distribution. Once the problem described in that section was solved, the block started to work as expected, but the accuracy of the filter wasn't as good as expected.

Analysing the problem a new UF was found in the input values of this check. The value used for the check wasn't the difference between code and smoothed code, but the smoothed code itself. This error was probably introduced because the smoothed code is also a difference between two terms, and this could have been a bit confusing to the developer who designed this part of the model.

This UF is the perfect example of an Unintended Function. The specification is correct but the model does not follow it. This UF is quite difficult to handle when the verification is not able to find it, because it can't be found by any criteria. Someone could think that the **SOMCA Prerequisite 4: Requirements – Model Traceability** could be able to find this UF, but the requirements never have the level of detail necessary for a so detailed trace. This level of detail used to be in the low-level design, but in our case, the low-level design is the model itself, and no trace can be done.

This UF could have been mitigated if the verification scenarios would have reproduced properly the environment conditions and the working point (the usual ranges of model inputs in nominal behaviour) of the input data.

This UF can be classified as *[UF.3.2] Incorrect behaviour in the general case*, because the value used to check the barrier was incorrectly calculated in any case. It can be also classified as *[UF.1] Deviation from requirements in the implementation*, because the implemented check was using a different value than the specified one.

13.1.3.1.5. ASP Calculation minor problem

During the pseudo operational results comparison, another small difference was found. The ASP field in the Warm Start data (used for several barrier checks) usually doesn't have any difference, but after a gap, the difference between the obtained data and the pseudo-operational one grows a bit. This behaviour is systematic, but the amount of this difference is very low (between 12 and 14 magnitude orders below the value).

The origin of this UF has not been discovered, and it's not sure if the UF is in the model or in the operational system. In any case, the low difference won't affect the final output in a significant way, and it's not for sure if it can be classified as an Unintended Function. In any case, it can be classified as *[UF.3.3] Erroneous behaviour under specific inputs / conditions*, because the Unintended Function only appears with a specific combination of the input parameters.

13.1.3.2. Conclusions obtained from pseudo-operational environment

Once the simulation results have been compared to the results obtained from the pseudo-operational environment, we have obtained the following conclusions:

- Even with the coverage criteria showing a good compliance with the criteria and prerequisites, a great number of UFs detected in this environment were not detected in the original verification. Why is this? The first result of this analysis is that the test cases included in the verification were according to the requisites, but they weren't developed taking into account the final environment in which the model is supposed to work. A very interesting point in the verification process would be to put a big effort in characterizing the problem and the input data that will be present in the operational environment to focus the test cases in that input ranges.

> The working point (understood as the nominal range of input values expected in the operational environment) should be determined to focus the verification on it. Verification test cases should be representative of the real working conditions of the system.

- Another important conclusion obtained from this analysis is the importance of the **SOMCA Criterion 2: Functionality coverage**. For some kind of blocks with a clearly defined behaviour (like the one described in section 9.6.1.5: Variance calculation error) the functionality coverage is a very good way to detect UFs. The problem is that the functionality coverage means a big effort in model development and verification, and it's very difficult to provide full coverage in the entire model. Maybe the level of coverage can be related with the DAL level of the model. E.g. in a DAL-C only library blocks have to cover functionality coverage, but DAL-A means functionality coverage in all model subsystems.

- Sometimes during the modelling process, some details in the implementation of the requirements can be identified as risky or not clear. It would be a good idea to determine specific tests in the target platform to test the system under these circumstances to check if the behaviour is the appropriate one. Another possibility is to enable communication mechanisms between specification and design teams (even including co-engineering).

- Leaving apart the already identified UFs, the results obtained from the model were the same that the ones obtained as the expected ones, with error below the tolerance specified by the precision loss associated to the script generation.

13.2. SCADE SMOOTHING MODEL UF ANALYSIS

As a result of the collaboration with Esterel, a parallel Smoothing model has been developed in SCADE based on the same requirements used for the Simulink Smoothing model. The outputs of this collaboration are the following:

- Complete SCADE Model for the Smoothing Filter as detailed in the requirements.
- Adaptation of Matlab Test Scenarios for SCADE Suite.
- Model coverage based on these scenarios.

The model has not been verified in a so exhaustive manner like the Simulink one, however some UFs has been detected thanks to SCADE Verification and coverage analysis results.

Additional UF examples provided by Estereal are included in Section 13.6, but the results of these examples are not included in the global study results.

13.2.1. GAP NOT DETECTED

The inputs of the operator that manage the input gaps never changed, and the scenario includes some gaps that should be detected. It seems that the operator was not triggered when this situation happens.

In this case, the SCADE Suite detected that the input `psiGapSize` had never changed. The criterion used is very similar to the proposed SOMCA Criterion 3: Modified input coverage.

The preliminary classification is *[UF.3.3] Erroneous behaviour under specific inputs / conditions.*

No further analysis was done on this possible UF.

13.2.2. CYCLESLIP NOT DETECTED

The same happens with the cycleslips, in the scenario there are several cycleslip detections, and the operator input never change. It seems that the operator was not triggered when this situation happens.

The criterion that detected this problem was the SCADE "Masking MC/DC", similar to the proposed **SOMCA Criterion 6: Logic Path coverage**.

As in the previous UF, due to the absence of information, this classification is only preliminary, but it seems to be *[UF.3.3] Erroneous behaviour under specific inputs / conditions*.

No further analysis was done on this possible UF.

13.3. CRUISE CONTROL STATE MACHINE UF ANALYSIS

13.3.1. INTRODUCTION AND APPROACH

As the Smoothing model developed as part of the Task 4 is almost entirely done with Block diagrams and the State machines used for it are very simple, an additional model is going to be analysed as part of the collaboration with Esterel to exercise the SOMCA Criteria related with State Machines. This model is the "Cruise Control" example, included in SCADE suite as a very representative example of the SCADE State Machines. Since the example is supposed to be correct and with no UFs, a list of UFs to be artificially injected was provided to Esterel. A study of the proposed Unintended Functions to be injected was supposed to be done by Esterel.

The described approach only included the introduction of some known UFs in the SCADE Cruise Control Model in order to check if the SOMCA criteria were able to detect them. However, during the model analysis, a very interesting UF was found in the model, and also detectable by SOMCA Criteria.

13.3.2. UNINTENDED FUNCTIONS PRESENT IN THE MODEL

This section contains the results of the analysis of the unintended function already present in the model (not the injected ones).

13.3.2.1. Unstable output when simultaneous inputs are activated

During the first analysis of the model, a very interesting UF was discovered based on the premises described in **SOMCA Criterion 14: Level-N Loop coverage**. There is a level-2 loop between `Active` and `Interrupt` states. If the Brake is activated at the same time that Resume input value is set to True, the main State machine changes continuously between `Active` and `Interrupt` states. If the vehicle cruise speed is set to 0, the Throttle output is stable, but if the cruise control has a velocity different than zero established, the Throttle

output changes continuously from zero to a positive value, generating an unstable output that can generate big problems if the output is connected to a physical throttle.

This UF only affects the output under given circumstances, and it can be classified as *[UF.3.3] Erroneous behaviour under specific inputs / conditions*.

A very similar problem happens when the signals `On` and `Off` are activated simultaneously. This problem would be also detected by this criterion. In this case, one of the variables is the other one negated. Only one of them is necessary, and the other one can be calculated.

This UF cannot be detected by the default coverage criteria included in the SCADE Suite. However, the suite includes a tool called *Design Verifier* that allows verifying formally some properties in the model. With the use of this tool, this problem can be found if the appropriate rule is defined.

13.3.3. UNINTENDED FUNCTIONS TO BE ARTIFICIALLY INJECTED IN THE MODEL

The absence of verification for this model makes very difficult to introduce UFs, since an UF is by definition an unspecified behaviour not detected in a verification process, that's why the absence of a verification process complicates the introductions of UFs in the model, the border between a normal error easily detected at verification and an unintended function is not clear. The following section describes the UF proposed to be injected.

13.3.3.1. Error in Standby Condition

In the Cruise Control State Machine, inside the Active State, there is a logic OR operation that controls the Standby condition. This condition suspends the CruiseControl under certain circumstances (3). If this logic operation is changed to an XOR instead of an OR, the behaviour of the control model changes. The behaviour is the same except for one case: If the accelerator has a significant value (higher than the lower threshold) and the vehicle speed is out of the allowed range to be working, the standby condition is disabled, and the CruiseControl can activate overriding the `accel` input. This will generate that the vehicle is unable to reach higher velocities than `SpeedMax` even if the `accel` is pressed.

This UF only affects the output under given circumstances, and it can be classified as *[UF.3.3] Erroneous behaviour under specific inputs / conditions.*

13.4. STATEFLOW EXAMPLES

The Smoothing example selected to assess the SOMCA criteria did not contain any state machine complex enough to perform a real assessment of the SOMCA criteria related to state machines. The Control Cruise example contains a state machine but it's not big enough to exercise all the SOMCA Criteria. Therefore it was decided to analyse a set of complex state machines to perform a realistic assessment. The activities executed in this task were the following:

1. Identification of a set of complex examples: from literature, examples already existing in the tools or created ad-hoc.
2. Application of the SOMCA criteria to these examples to assess existing UFs.
3. Definition of a set of UF to be artificially injected in the models.
4. Application of SOMCA criteria to assess detection capability of the UFs injected in the model artificially.

5. Refinement of SOMCA Criteria with all the outputs generated in these activities.

13.4.1. SERIAL PORT CONTROLLER

The following example is a Serial port controller made with a State Diagram (Stateflow machine). This controller must read data from the serial port and print it to the screen. The state machine is represented in the next figure.

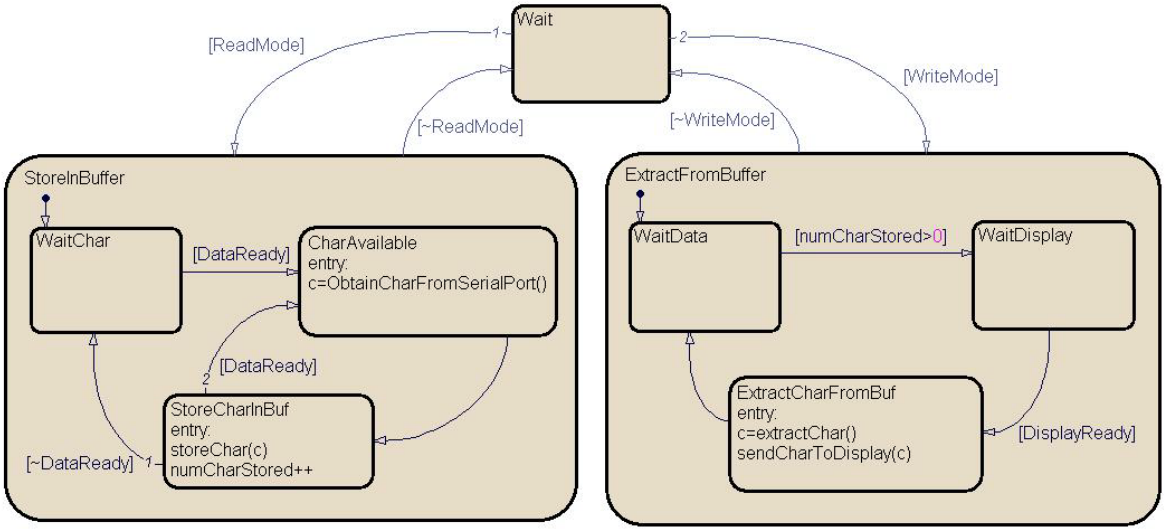


Figure 13-6: Serial Controller State Machine

The state machine can operate in three main modes: Wait, Read and Write Mode, controlled by two different signals called `ReadMode` and `WriteMode`. The nominal behaviour of this system is the following:

1. The system starts is in `Wait` mode.
2. `ReadMode` is activated, transitioning to `StoreInBuffer`
3. When data is available in the serial port, is read and stored in the temporary buffer
4. The `ReadMode` signal is deactivated, and the `WriteMode` one is activated.
5. The system produces all the information that the screen is able to display
6. The `WriteMode` signal is deactivated
7. Jump to number 2.

This State Machine works without problems and the data is read and printed in the simulations without any error. A preliminary coverage analysis shows that all states and transitions have been covered.

But when the system is tested in the real environment, some characters (0.1%) are lost. Why is this happening, if the simulations were OK?

The problem is that the `StoreInBuffer` State can be abandoned in any of their internal substates, and if this state is abandoned when the system is in the `CharAvailable` state, the character obtained in the call to `ObtainCharFromSerialPort` is not stored in the buffer.

This UF would have been detected by the **SOMCA Criterion 7: Parent State coverage**. This analysis would have shown that the `StoreInBufferState` hadn't been abandoned in all substates, indicating the weak point in the verification test cases that will detect this UF.

This UF can be classified as *[UF.3.2] Incorrect behaviour in the general case.*

13.4.2. PETITION QUEUING IN TRANSACTION MANAGER

This Example presents a State Machine that controls a sending mechanism that is activated under petition. If the Transaction is required, the model sends 200 packets and stops. To activate the transaction the input activation signal must raise and fall, and the sending is started after the signal fall.

The next figure shows this State Machine.

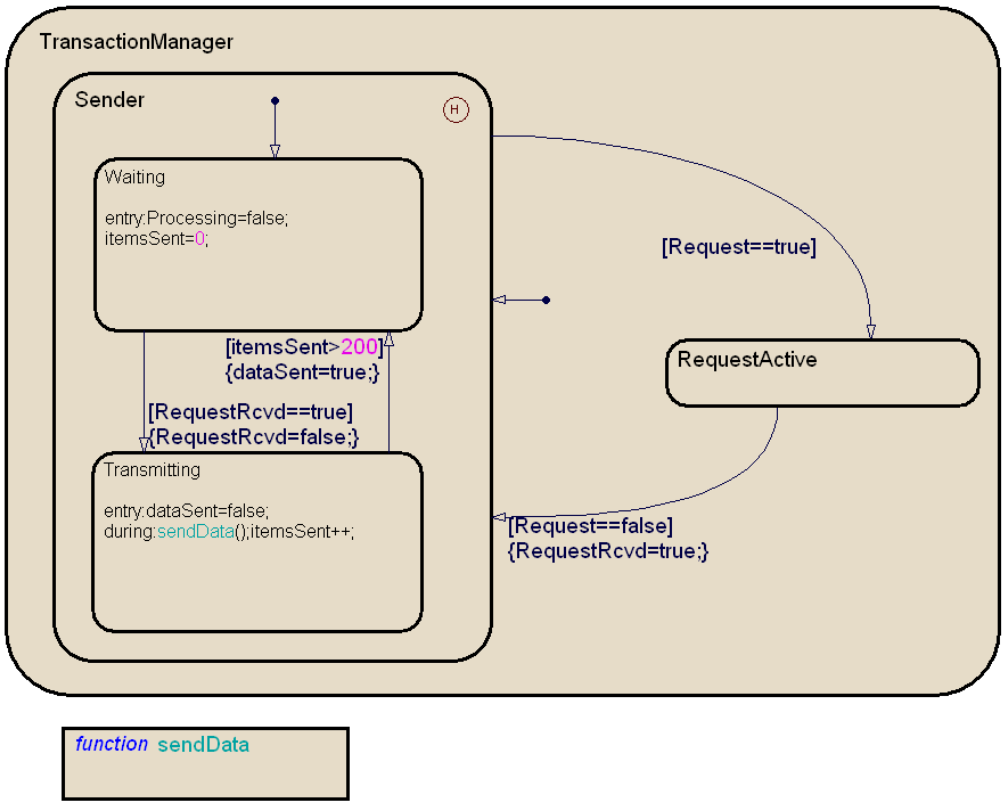


Figure 13-7: Transaction Manager State Machine

The first verification shows that the model behaves as expected. However, later, when the state machine is used on a real environment, the data sent by the model is not continuous, and the time taken to send the packets sometimes is higher than the specified. In addition to this, the model sometimes send two consecutives packages.

When the SOMCA Criteria is applied over the verification, this is the result:

Criterion	Result
SOMCA Criterion 1: Range coverage	Not executed for this example
SOMCA Criterion 2: Functionality coverage	Not executed for this example
SOMCA Criterion 3: Modified input coverage	Not executed for this example
SOMCA Criterion 4: Activation coverage	OK: All activable elements have been activated
SOMCA Criterion 5: Local Decision Coverage	Not applicable (only for block diagrams)
SOMCA Criterion 6: Logic Path coverage	Not applicable (only for block diagrams)
SOMCA Criterion 7: Parent State coverage	**Not OK**: The Transmitting State has never exited from a Sender transition.
SOMCA Criterion 8: State History coverage	**Not OK**: The Transmitting State has never been re-entered.
SOMCA Criterion 9: Transition coverage	OK: All transitions have been executed.
SOMCA Criterion 10: Transition Decision Coverage	OK: All decisions have take all possible values
SOMCA Criterion 11: Transition MC/DC	OK: MC/DC is covered for transition decisions.
SOMCA Criterion 12: Event coverage	Not applicable (no events in the model)
SOMCA Criterion 13: Activating event coverage	Not applicable (no events in the model)
SOMCA Criterion 14: Level-N Loop coverage	Not applicable (no loops present in the model)

Table 13-1: SOMCA Results on Transaction Manager

The application of the SOMCA Criteria shows that verification has not covered the situation that a Request is received when another one is processing. This situation is not covered by requirements. In this model, this absence of requirements over this situation has generated two UFs:

a) When the signal `Request` is active, the sending is stopped (i.e. *[UF.3.3] Erroneous behaviour under specific inputs / conditions*).

b) When a request is received during the sending, this request is queued and executed after the first send (i.e. *[UF.3.1] Extra functionality*).

13.4.3. TEXT LINE EDITOR MODES

For the proposed example, we will introduce a simple text editor for one line. The editor has two different modes: EditMode and InputMode. When the edit mode is activated, you can move forward and backward in the line and execute commands, when you activate the input mode, you can write in the current position.

The state machine that controls this editor is the one represented Figure 13-8.

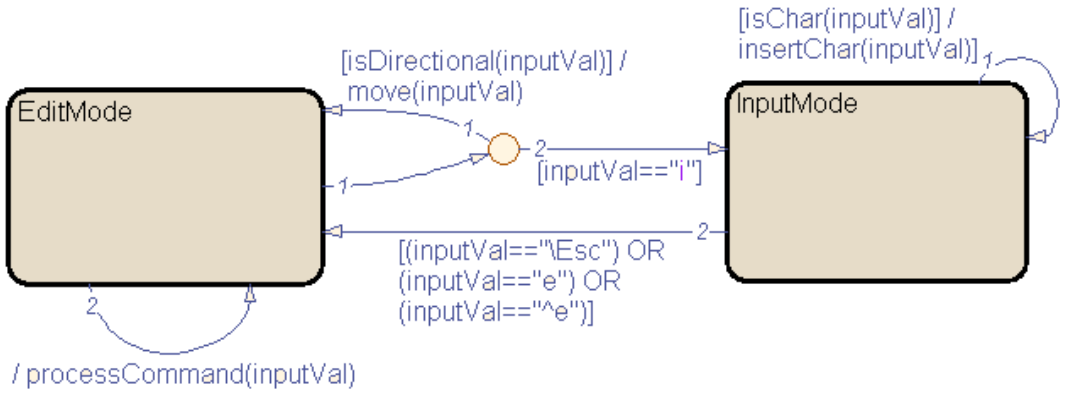


Figure 13-8: Example of text line editor

In this example, an UF is present that is not detected by validation. The Editor seems to work, but when you are in InputMode, you should leave if any of the three exit conditions are mateched. However, the second one do not trigger the transition.

Both **SOMCA Criterion 9: Transition coverage** and **SOMCA Criterion 10: Transition Decision Coverage** are not able to detect this UF because all transitions are executed and all decisions take both values. However, if we apply **SOMCA Criterion 11: Transition MC/DC**, the criterion shows that the second condition is never evaluated to true. When a deeper analysis is performed, it reveals that the condition of the first transition in the InputMode State always activates before the second condition of the second transition. This is because the function isChar returns true when called with character “e”.

This UF is a clear example of *[UF.3.3] Erroneous behaviour under specific inputs / conditions.*

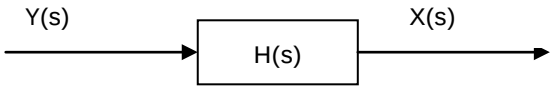
13.5. THEORETICAL UF EXAMPLES

Finally, this section provides the initial set of UF examples defined in the Interim Report 1 [RD.1] to explore and analyse the different types of Unintended Functions. They were conceived to study as many different UF kinds as possible, ranging from simple models to complex high-level protocol design errors, and including pathological errors external to the model or hardware specific. These examples were also used for challenging the different iterations of the MCA criteria, evaluating their effectiveness and to refine the initial SOMCA coverage criteria. Therefore, for the final SOMCA Criteria presented in this Final Study Report these theoretical examples are considered less interesting than the real-world ones, but are included here by completeness.

A detailed description of the example UFs can be found in Interim Report 1 [RD.1]. As explained in the First Interim Report, these examples have been selected from the different categories of the UF taxonomy, with the intention of ensuring that the proposed Model Coverage criteria are able to detect a wide variety of Unintended Functions.

13.5.1. UNSTABLE SYSTEM – TRANSFER FUNCTION

The Transfer Function is one of the basic blocks included in the default Simulink library. The Transfer Function of a system is the ratio of the output to the input of a system in the Laplace domain, in other words the Transfer Function is a model of the system in the Laplace domain. The Laplace transformation is frequently used in Control Theory because it facilitates the management of the differential equations that generally represents the behaviour of the systems.



$$H(s) = \frac{Y(s)}{X(s)}$$

Figure 13-9: Definition of Transfer Function

Transfer Functions along with MBD techniques are widely used to design and tune open- and close-loop controllers. One of the more sensible aspects of the design of controllers is the stability of the final system, since the final ensemble of the Controller plus the System can be also represented with a Transfer Function; the analysis of the stability of this final Transfer Function becomes critical.

The stability analysis is not straightforward and frequently is performed partially, testing the behaviour of the system with some pre-defined inputs: steps, square signal, etc. These tests only detect systems that are always unstable, but all the systems whose stability depends on certain conditions (type and value of the input, value of the Gain of the controller, etc) are not considered and hence can introduce Unintended Functions.

The following example shows a simple example of use of a Transfer Function block that leads to an unstable system, in this case the instability can be easily detected with a step signal.

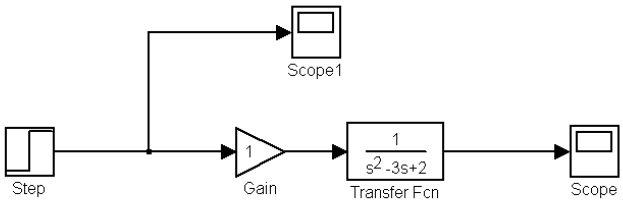


Figure 13-10: Model containing a Function Transfer block

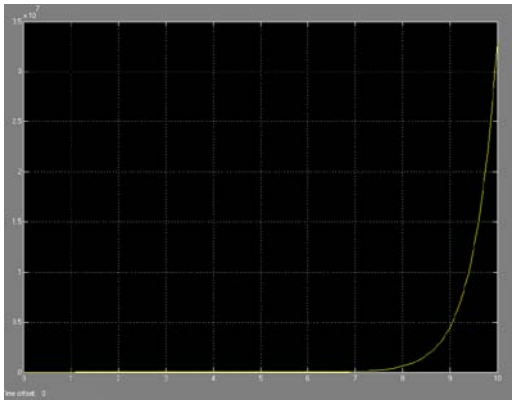


Figure 13-11: Output corresponding to the Function Transfer excited with a step signal

Figure 13-11 represents the unbounded output of the system showing the instability of the system.

However in the following case:

$$H(s) = \frac{1}{s^3 + 6s^2 + 11 \cdot s + 6}$$

Equation 1: Example equation for a Transfer Function.

The response of the system to some inputs (steps, square signals) shows that the system (although oscillatory) seems to be stable after some time:

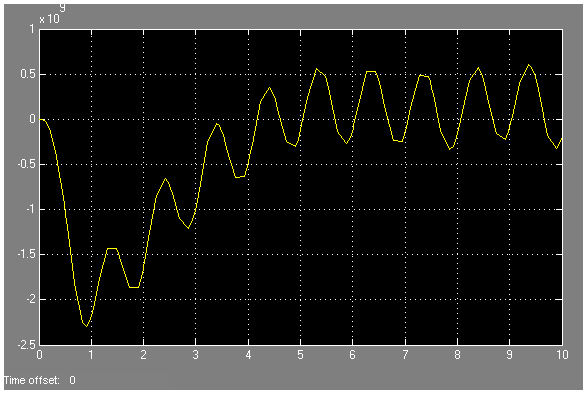


Figure 13-12: Response of the system represented by Equation 1 to a square signal.

However, when the Root Locus is analysed to assess the stability of the system, it is shown that only adding a Gain controller K, there could exist K values that could make the system to be unstable (the poles of the system move to positive values in the real axis).

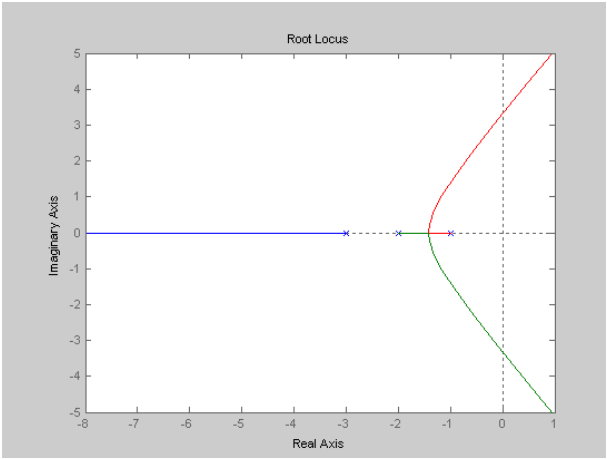


Figure 13-13: Root Locus diagram of the system represented in Equation 1.

Therefore, in this example of Unintended Function, the transfer function of this block diagram is not stable but oscillatory, and thus the outputs may present an unexpected behaviour under specific input signals. This UF example will be detected applying the prerequisite that the stability analysis must be performed for all transfer functions or define modelling guidelines that prohibits the use of transfer functions. The application of the **SOMCA Criterion 2: Functionality coverage** may detect stability problems when applying the standardized input signals, like the step or the square wave, but since the stability could depend on a configuration input value like the gain of the controller the check should be done for all the possible values of that parameter, what seems not be feasible.

The logic path coverage criteria cannot be applied in this case because only numeric blocks are involved in this UF example.

13.5.2. UNSYNCHRONIZED NAMING OF INPUT / OUTPUT PORTS

Each input (or output) of a Simulink subsystem can have two names: one “inside” identifier when editing the subsystem internals (giving a specific name for the input / output “knob” block), and another “external” identifier when displaying the subsystem as a block (editing the subsystem’s mask parameters). This can lead to hard-to-detect modelling errors if those names diverge, as experience showed in past projects.

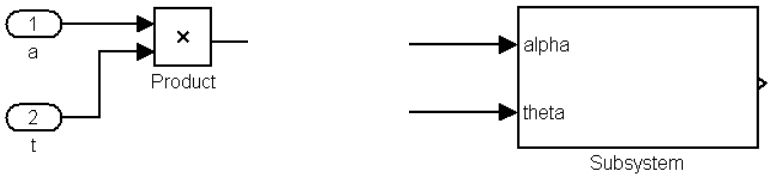


Figure 13-14: Example of incoherent naming in Simulink

Imagine a subsystem with two inputs. The external name for the first input is ‘`alpha`’, while the internal name is just ‘a’; while the second name is from the outside ‘`theta`’, and just ‘`t`’ internally. Suppose that due to a redesign of the subsystem’s internals, the names of the knob blocks are reversed, so input 1 is now connected as the parameter `theta`, and input 2 is acting as the parameter `alpha`. The internal knobs have been properly renamed, however the mask parameters have not been synchronised so the external names are still the old ones. Thus, the external connections to the subsystem from other subsystems appear as correct as the showed names are the expected ones. This led to an unintended behaviour which was very hard to catch. This kind of problems requires properly defining a set of restrictions to limit the number of error-prone features of the modelling notation (**SOMCA Recommendation 7: Modelling guidelines have to be defined to forbid the error-prone features of a specific notation**).

In this case, it seems unlikely that any coverage criteria could uncover in all models this error with a high probability, but different criteria and prerequisites may help. The **SOMCA Prerequisite 8: Type check** would detect the problem if the input ports have different types. If the data types are the same, the **SOMCA Criterion 2: Functionality coverage** criterion could detect the problem when executing with different equivalence partitions in each input. Also, in some situations the Functionality Coverage criterion could be incomplete even if apparently the provided vectors should properly exercise all the characteristics, uncovering the problem with the input ports connections / naming. The analysis performed for the various Path Coverage criteria could also help in uncovering the error if Boolean inputs, in case additional verification cases are needed to completely cover the buggy component. Specifically **SOMCA Criterion 6: Logic Path coverage** may detect the error with a high probability (many cases are needed to completely cover all links of a path).

In any case, this error is probably easier to detect analysing the model statically rather than running verification cases. Note that different recommendations from section 9.4.2 advocate for adhering to modelling standards to avoid introducing this kind of problems due to notations features. Obviously, the use of independent port naming would be one of those not recommended notation features (Could be included in **SOMCA Prerequisite 17: Requirement of Design Standard**)

13.5.3. IMPLEMENTATION ERROR IN MAX BLOCK

For the following example, we will define a block that implements a “Maximum” function. Imagine a simple block to compute the maximum of three input values of different types:

- `uf_max(int16, unit8, uint16) : uint16`

Where the different types of the block means the following:

- `int16`: 16-bit signed integer. Range: [-32.768 to 32.767]
- `uint8`: 8-bit unsigned integer. Range: [0 to 255]
- `uint16`: 16-bit unsigned integer. Range: [0 to 65.535]

The output value will be the highest one of the three input values.

However, the internal implementation incorrectly mixes different input types: The type of intermediate computations applied for each two inputs is the type of the second input, and thus the saturation is applied as the greater value allowed for this type. So as the max value of the second parameter is 255 (8-bit unsigned integer), when the first parameter (16-bit integer) is greater than 255 the intermediate value is saturated to 255, so the output can be wrong, for example:

- Correct result: `uf_max(12, 8, 1) = 12`
- Incorrect result: `uf_max(300, 15, 291) = max(255, 291) = 291` (Expected: `300`)

This UF example is caused due to a too-narrow numeric type in an internal computation. As shown in the figure, the inputs have different data sizes and signedness, but in the first switch block the value of the input 1 (an `int16`) could be converted into a `uint8` value.

Therefore, in some situations the value from input 1 will be saturated to the maximum value that can hold in a `uint8` type, i.e. 255. For example:

`uf_max (32767, 0, 0) = 255` *[Wrong output]*

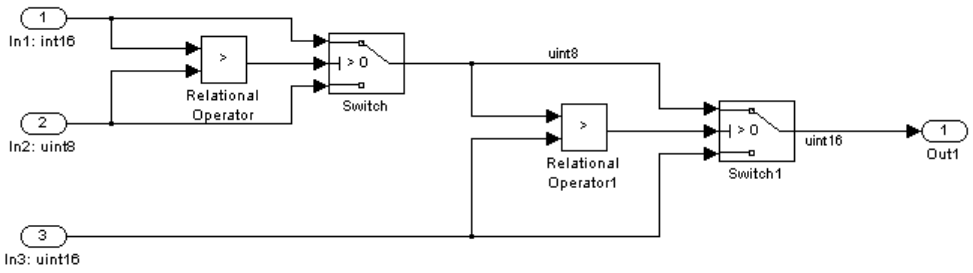


Figure 13-15: Implementation of `uf_max` block

The component contains two one-link logic paths, between the relational operators and the switches, and thus **SOMCA Criterion 6: Logic Path coverage** must be applied. The two verification cases {0, 0, 0} and {1, 0, 0} satisfy coverage of both logic paths for Multiple Condition Path Coverage. The error is not detected with these criteria.

The **SOMCA Criterion 2: Functionality coverage** can be defined for the component as "each input contains the maximum value", which will be exercised by the verification cases {1, 0, 0}, {0, 1, 0} and {0, 0, 1}. For the relational and switch blocks, which also have non-Boolean inputs, the same verification cases used for reaching path coverage can be used for achieving Functionality coverage. Neither verification case detects this UF example.

For Level B and A, **SOMCA Criterion 1: Range coverage** has to be applied too:

- *Singular points*: there are no singular points; all input numbers behave as surrounding values.
- *Discontinuous input signals*: depending on the values employed in the discontinuous signal, the problem could be detected (for example, sawtooth signal between the minimum and maximum values for the first input, and zero for the other inputs).
- *Standardized signals*: when a periodic signal is applied for the first input value the wrong output may be detected if the other two inputs have adequate constant values (like zero for both). See figure .

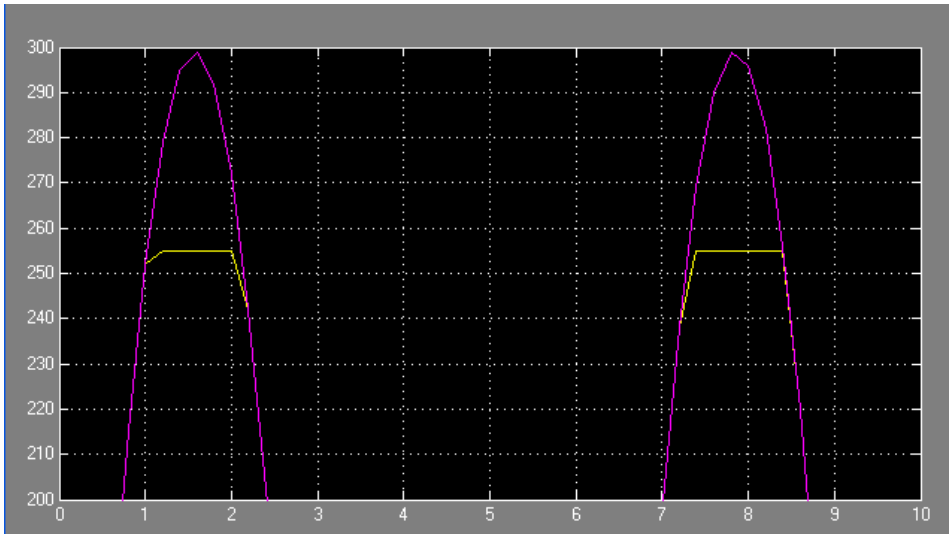


Figure 13-16: Result of `uf_max` block when applied a sine function.

- *Equivalence classes*: Different equivalence partitions have been taken into account to detect the problem, as explained bellow.

If the equivalence classes are constructed following the criterion (i.e. the boundaries of internal computations need to be taken into account), the following partitions are considered for each port:

- *Input 1*: besides the range of the `int16` input signal (from −32768 to 32767), the analysis yields that the input can be squeezed into a `uint8` link (from 0 to 255). This should raise and alarm, and detect the UF. However, let's continue with the application of the whole Range coverage criterion for illustrative purpose. In this case, 5 partitions are considered: [−∞ .. −32769], [−32768 .. −1], [0 .. 254] [255 .. 32767], [32768 .. ∞]
- *Input 2*: just the boundaries of this `uint8` signal must be considered, so [−∞ .. −1], [0 .. 255], and [256 .. ∞] are the 3 partitions considered.
- *Input 3*: this input is always routed through `uint16` links (from 0 to 65535), so again just its boundaries are considered in the equivalence classes, i.e. the 3 partitions [−∞ .. −1], [0 .. 65535], and [65536 .. ∞].

- *Output*: Even if the output has type `uint16`, intermediate computations can be performed with types `uint8` and `int16`, and therefore the 6 partitions are [$-\infty$ .. -32769], [-32768 .. -1], [0 .. 255], [256 .. 32767], [32768 .. 65535], and [65536 .. ∞]. However, the two partitions of negative values can never be satisfied because the link has an unsigned type, nor the last partition because the signal is just 16-bit wide.

Note that when out-of-range values are provided as inputs, extreme in-range values will be provided because the model is configured with saturation semantics.

Now, different number of verification cases can be selected, depending on the test set selection policy [RD.22]. If the minimal test set is selected, a minimum of $\max(5, 3, 3, 3) = 5$ verification cases are needed, so for example the following input vectors cover all the equivalence classes of the 3 inputs and the output (but the out-of-range output partitions):

- T1: $\{-32769, -1, -1\} = 0$
- T2: $\{-32768, 0, 0\} = 0$
- T3: $\{0, 0, 300\} = 300$
- T4: $\{255, 255, 32768\} = 32768$
- T5: $\{32768, 256, 65536\} = 65535$

However, even if these input signals satisfy the equivalence classes the UF is not detected (although other input vectors could detect it). Under the one-to-one test set selection at least one input vector per partition would be required, that is $5 + 3 + 3 + 3 = 14$ verification cases for this Max block. These 14 input vectors neither guarantee the detection of the UF, however.

Just if all the combinations of the equivalence partitions for the inputs need to be exercised, would guarantee the discovery of the model's mistake. However, that would require at least $5 \cdot 3 \cdot 3 = 75$ verification cases, certainly too many input vectors to be applied to non-trivial models. In fact, we have not found this test selection strategy for equivalence partitioning from any source, but has been created ad-hoc for the analysis of this UF example. We call it "Multiple test set" strategy.

In fact, if another set of equivalence classes are considered, not taking into account the boundaries of internal types (i.e. boundary-value testing), the following partitions would be considered:

- *Input 1*: [$-\infty$ .. -32769], [-32768 .. 32767], and [32768 .. ∞]
- *Input 2*: [$-\infty$ .. -1], [0 .. 255], and [256 .. ∞]
- *Input 3*: [$-\infty$ .. -1], [0 .. 65535], and [65536 .. ∞].
- *Output*: [0 .. 65535]

In this case, the chosen input vectors would equally may or may not detect the UF. For example:

- T1: $\{-32769, 256, 65536\} = 65535$
- T2: $\{32767, 0, 0\} = 255$ *[Wrong output]*
- T3: $\{32768, -1, -1\} = 255$ *[Wrong output]*

Therefore, in this case the requirement to use the types of internal computation is incrementing the number of verification cases needed but not the probability of detecting the UF by means of exercising the generated input vector. However, as noted above, the partitioning analysis considered with intermediate types actually lead to identify the data conversion between the first input port and a link of narrower precision. Therefore, for this example the Range Coverage criterion augmented with the analysis of internal data types detected the UF, even if finally the exercised verification cases may or may not point to the error.

In any case, as in the example 13.5.2, probably the easier way to detect this problem is applying the prerequisite of avoiding data type mismatches, which would avoid the introduction of the UF in the first place. The next figure contains a corrected model of the Max block, using explicit data conversion blocks to employ always the same type (`int32`) for internal computations.

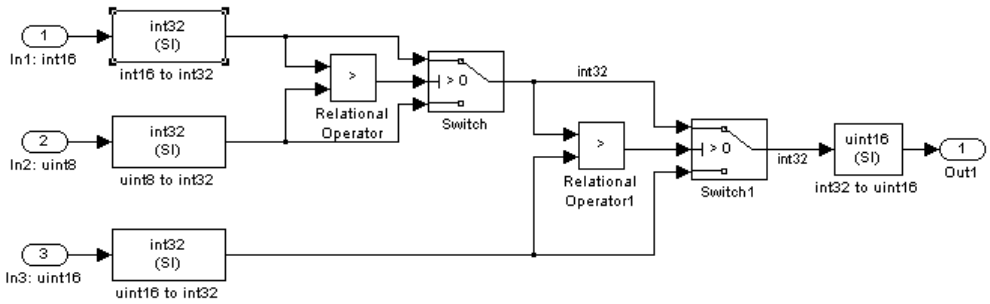


Figure 13-17: Correct implementation of uf_max block

13.5.4. FILTER BLOCK INPUTS

The following example has been provided by EASA as an example of Unintended Function (the original title of this example was “cyclic data as input to a filter block”). The example shows a block that implements a filter to smooth discontinuities in an input signal.

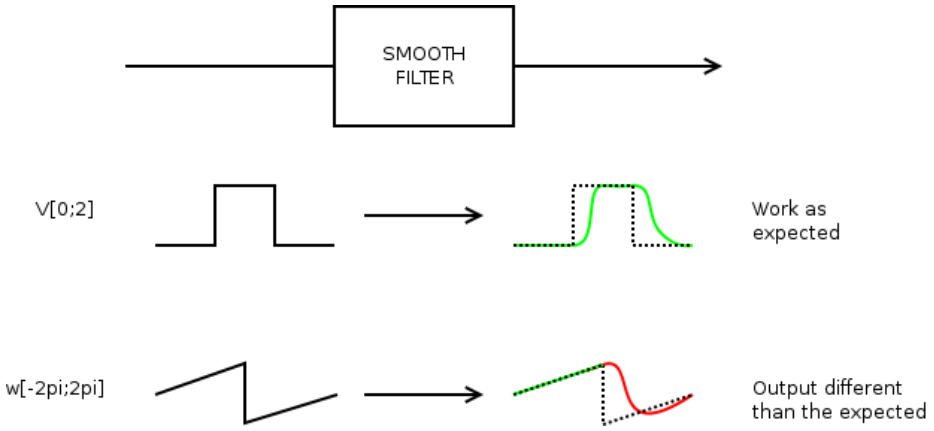


Figure 13-18: Smooth filter block

The filter works as expected when the input signal is continuous. But when used as a cyclic signal (e.g. if measuring the roll attitude of an A/C), the behaviour is not the expected due to a wrong assumption about the

physical units used to represent the external world. In this example the angular position of an antenna is represented in radians between 0 and 2∏, and thus there is a discontinuity when a whole revolution is performed (in the transition from 2 ∏ to 0). However, the model is designed assuming that the position values for the antenna are never discontinuous as its movements are always to a near position, something untrue. Specifically, a filter is directly feed with the position of the antenna, so the smoothed signal contains unexpected discontinuities.

The Unintended Function here is not an erroneous implementation of the smooth filter box but an incorrect use of it. The block has been designed as a smooth filter of the input signal, and the output is the expected one for that behaviour. The problem here is that the engineer that has designed this model has used a continuous signal filter to smooth a cyclic one, introducing an Unintended Function because of a partial knowledge of the block functionality.

This misunderstanding of the nature of the input signal should be covered /checked in the model, but currently defined criteria do not cover this kind of problems. New criteria or recommendations need to be defined to avoid this kind of errors.

The **SOMCA Criterion 1: Range coverage** criterion requires exercising the model with different input signals, including discontinuous ones. In this way, the antenna position will be required to transition between the maximum and minimum in-range values, which would force the discontinuity in the filter, and thus the UF detection.

13.5.5. MEMORY-LATCH ERROR IN A LIMITER

This Unintended Function has been also provided by EASA, and involves a limiter block. The expected behaviour of the limiter block of the next figure is the following:

- Input is in the closed interval [`LO`, `HI`], then `LIM_Out = Input`.
- If `Input > HI`, then `LIM_Out=HI`.
- If `Input < LO`, then `LIM_Out=LO`.

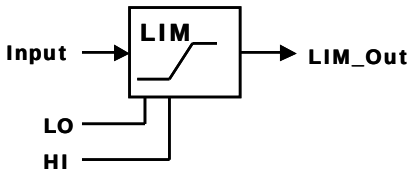


Figure 13-19: Limiter block diagram

From the functional point of view the block behaves as expected and no Unintended Function is detected. The Unintended Function appears when the block in embedded in the model and interacts with the rest of the blocks of the model generating an unexpected behaviour. When the value at the input of the block decreases from high values to low values the `Lim_Out` signal keeps the higher values for some time delaying, the final effect is that the device receiving the output of this block is feed with an input value higher than expected. If the device receiving `LIM_Out` is an actuator, this would mean an error with safety implications in the case of an aircraft.

This Unintended Function could be allocated in the type *[UF.3.3] Erroneous behaviour under specific inputs / conditions*, even if in this Block Diagram the functionality of the limiter block is correctly modelled, because when the signal decreases from the highest value the output is delayed for some time, having the signal a higher value than expected.

The problem in this UF example is that the behaviour of the limiter block is not the expected one by the designer; this could be due to one of the following reasons:

- Case 1: The dynamic response of the block has not been tested when isolated before injecting into the system. This case is covered by **SOMCA Prerequisite 15: Environment definition**. The dynamic behaviour of this element should have been tested and properly documented.
- Case 2: The dynamic behaviour of the model with the limiter already included in the model has not been correctly verified at model level. In this case the **SOMCA Prerequisite 21: Input / Output assertions** should be useful, because an assertion on the output values based on the input ones is easy to include in the design and can easily detect the UF.
- Case 3: The dynamic response of the limiter block was correctly checked when isolated but the block was not correctly documented, hence the designer was not aware of the real block behaviour. Case 3 is a clear example of **SOMCA Prerequisite 13: Model documentation**.

13.5.6. COMMANDING FOR MULTIPLE BLOCKS

In the following example we will analyse a small system based on three blocks: Two different processor modules, both accept the same inputs, but generate different outputs; and a Commander block, which function is to propagate external commands and gather internal responses:

- `MeasurementProcessor(MeasA, MeasB, Switch) : SmoothedMeas, SwitchResult`
- `ErrorProcessor(MeasA, MeasB, Switch) : BoundedError, SwitchResult`
- `Commander(ExtSwitch, IntResponses): SwitchResult`

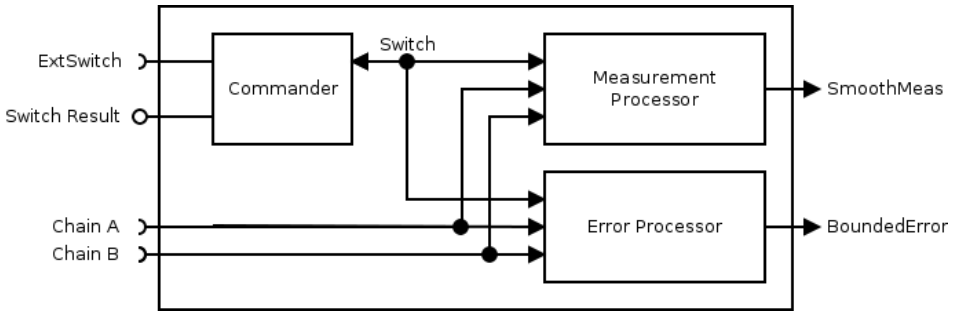


Figure 13-20: Dual Chain Processor

In the nominal behaviour, both processor components process the same data, and receive the same commands from the commander. The system can receive an external command that is sent to both processors. By default, both processors use measurements from chain A, but when the switch command is executed, they have to use the data received from chain B. The expected behaviour is that commands are always accepted or rejected equally for both processors. The validation is done under this premises and the system is validated.

But the blocks are not the same, and under some unpredictable circumstances, the behaviour is not the same. Due to some circumstances, the measurements received by chain B are incomplete, and the Error processor will not be able to switch to chain B because of that. When the switch is performed, the Commander sends the switch to both processors. The first block accepts the command, but the second one not. The global command is rejected, and no error is raised, but both processors are in an inconsistent state because they are using different input data: The first one is using chain B (it accepted the command) and the second one is using the Chain A (it rejected the command).

This Unintended Function is an error design, based on the (erroneous) premise that both blocks have the same behaviour for the same input. It's another example of *[UF.3.3] Erroneous behaviour under specific inputs / conditions*.

No one of the prerequisites of the MCA criteria is related with the root cause of this UF. Neither the Functionality Coverage criterion will detect the UF because probably the functionality considered in this context will be “switch chain successful” and “switch chain failed” (if in the design it was not considered that the confirmation could be different, certainly it will be assumed that way in the functionality of the model). Range coverage is also unlikely to identify this design error as the problematic chains are very specific values, and as the partitions are chosen at component level it is difficult to notice that a very specific equivalence class is needed for replicating that asymmetrical behaviour in the processor components (unless the equivalence classes are extracted automatically by an hypothetical tool).

In this case, as the confirmations for the switch command have to be processed by logic blocks, the Commander block contains specific logic paths to handle the result of the operation. However, the UF can be detected or not by the different path coverage criteria depending on how the Commander component is modelled. For example, suppose the result for the Switch command is computed by an “and block” fed by the Boolean confirmations of both processor components. In this case, the **SOMCA Criterion 6: Logic Path coverage** will detect the UF: this criterion requires input vectors that give both the False and True values for each link (including the links that carry the confirmation of both components), and to activate the logic path of a confirmation the other “and” input (the logic path of the other confirmation) must be True to avoid masking the result, therefore the Multiple Conditions Path Coverage will require the combinations were one confirmation is True and the other is False, uncovering the UF. An equivalent result will be obtained if the “and block” is and “or block” instead.

In summary, the proposed MCA criteria will be able to detect this UF applying the path coverage criteria, but just for the highest criticality levels (Level A, and maybe Level B). For other DALs this UF will probably remain unidentified in the model.

13.5.7. SYSTEM TIME ADJUST DELAY

The following example is the implementation of a “`SetSystemTime`” block, and is based on a real problem detected in GMV, in the scope of Galileo activities, related with the behaviour of a specific version of the RTOS in long system clock jumps.

The expected function of the previously mentioned block is to update the system time to the one passed as input. A simple behaviour with no outputs. The general purpose of this block is to synchronize the system time given an external time reference.

In the validation of this block, all jumps tested are in the normal range of this kind of synchronization. The function has been validated for seconds, minutes, hours and some days of time correction to the system time, and the time requirements of the block are met.

The problem is that the internal design of this block generates an internal delay in the execution of the block, not significant for the test cases, but directly proportional to the time difference between the current system time and the one given as input. The Unintended Function comes when a very long jump is performed (several years) due to a system clock reset and the time has to be synchronized. The previously mentioned delay for this case grows up to 5 minutes, freezing the system for that amount of time.

This is an example of a problem not detected in the validation of the block because of a bad criteria on the verification and testing in target environment, and can be classified in the proposed taxonomy as *[UF.3.3] Erroneous behaviour under specific inputs / conditions.*

The **SOMCA Criterion 1: Range coverage** mandates testing boundary values, and thus when the maximum clock jump is exercised (of thousands of years for this example) the UF will be easily detected as the operation would take a huge amount of time. It’s important to remark that this error is only present in the target platform with the generated code based on the model, so this error won’t be detected if the verification on the target platform is done with different scenarios than the ones used for the coverage of the model verification. In combination with **SOMCA Prerequisite 22: External timing management**, the detection of the UF in the target platform will be assured.

Other criteria are not as useful for detecting this UF: **SOMCA Criterion 2: Functionality coverage** will be satisfied with verification cases performing short clock jumps, and the different logic path coverage criteria are not effective because the UF is caused by numeric blocks and not Boolean ones.

13.5.8. CACHE CONFLICTS IN CONCURRENT BLOCKS

For this example, we have a matrix multiplication block. These are the inputs and outputs of the block:

- `MatMul(MatrixA, Matrix B) : MatrixR` (R = A·B)

The block multiplies A and B, and the output provided is the result of the operation. It has a time requirement depending on input matrix sizes. For this example, we will define this requirement as the following:

- Size of Input Matrix A: 20×20
- Size of Input Matrix B: 20×20
- Max time to produce result: 10 milliseconds

The block has been designed and tested, and the validation shows that the response time of the block is always below the time requirement (e.g. 4 ms). Which is unknown at this stage is that the internal design of the multiplying algorithm will use a buffer size that is 75% of the cache size.

The problem comes when this block is used in a concurrent model, and different threads execute this module at the same time. Each time one of the threads runs, it uses its own data, and the cache loads the new data, eliminating from the cache the data used by other threads. The concurrence of all threads provokes multiple cache errors, forcing continuous cache errors and updates, slowing the instruction execution. When this situation happens, the output time of the block is several times higher than the measured one, and giving a total process time that exceeds the requirement, showing an Unintended Function on its max time to produce result.

The problem could be an incorrect design that could lead to an Unintended Function on the block execution time. The problem can be defined as an unexpected behaviour due to the interaction with other functions or environment, and this kind of errors belongs to *[UF.3.4] Model adaptation to Target Platform*.

The UF in this pathological example can only appear when the software is executed in the final platform, and only when the size of the matrices cause continuous cache conflicts, radically slowing down the mathematical computations by various orders of magnitude. Slightly bigger or smaller matrix sizes, or a different platform with another cache policy or size, would not trigger the problem so the timing behaviour will be the expected one.

This UF occurrence is special with respect the other analysed examples because it cannot be replicated through simulations but just through testing in the final platform, and thus requiring that the source code have been already developed. Therefore, no proposed prerequisite or criteria would identify this problem as one of the goals of the SOMCA MCA is to be applicable early in the development phase, before the source code has been written. Even the prerequisite of exercising specific simulations and tests to ensure the target platform is accurately modelled will not uncover the error as not every hardware detail can (nor should) be replicated.

If the sizes of the matrices are constant in the system, the anomalous timing will be detected once it is tested in the final platform. If their sizes are variable, the Range Coverage criterion could help to select the problematic sizes when the equivalence classes are chosen. However, in both cases the UF will not be detected until the verification cases are executed in the final platform, that is, in the *late phases of the project*.

The **SOMCA Prerequisite 22: External timing management** will help in the detection of the problem, but it also won't help before the verification in the target platform.

This example shows that specific UF occurrences external to the model cannot be identified by the proposed MCA criteria: Even if the simulators are required to completely emulate the hardware behaviour, there are always hardware details impossible to replicate in the system, because usually their effect is negligible in the software. So it has no sense to add a prerequisite asking for this, because would radically slow down the simulations and complicate the implementation of the modelling tools (and thus, more prone to suffer bugs in the simulation) for little benefit. In contrast, this example emphasizes the need of running all the verification cases in the final hardware platform as the only way to properly verify the correctness of the software behaviour.

13.5.9. SATELLITE CLOCK STABILITY CHECK

The following is an implementation of a clock stability check block, and is based on a real problem detected in GMV, in the scope of Galileo activities, related with the detection of Feared Events.

In order to detect a feared event defined at system level affecting to the whole constellation of satellites, a block implementing a barrier check was designed and validated. That block checked the stability of the quadratic term of a clock for a given satellite. If this term changes in an unexpected way, the check set that satellite as “Failed”. This block was successfully validated since it was able to catch the feared event at satellite level in nominal conditions.

In non-nominal conditions, the situation was that the defined check was triggered for almost all the satellites, but for some of them the barrier was not triggered because the trigger threshold was not violated. This could lead to situations where most of the satellites present in the system were set to Failed and a few ones continued in Operational mode, hence the feared event at constellation level remains undetected..

This problem was found in a very advanced state of the validation, and it’s a perfect example of *[UF.2] Specification problem* caused by a not enough detailed requirement. In this case, an intense validation covering non-nominal scenarios found the Unintended Function.

The prerequisite of requirements validation is the best way for the detection of this UF, which should identify the differences in the outcome of the specified check for different satellites. This is covered by **SOMCA Prerequisite 4: Requirements – Model Traceability.**

13.5.10. DESCOPED FUNCTIONALITY INTERFERENCE: REUSE OF MESSAGE DECODER

While developing a critical message decoder, the development team finds that the State Diagram that it’s needed has been already developed some months ago for another module in the same project. The functionality of that module meets the requirements, so the module is included in the new model.

The validation and the model verification are done over the implemented model, but the target verification shows an unintended function in the check of a CRC that is not mandatory for the message acceptance. The analysis of the source of this unintended function shows that the CRC checking in the model was needed for the previous project, but not for the current one.

As described in section 9.2.2.1, this Unintended Function comes from the Partial use of existing block due to model reuse from other projects. This is a good example of *[UF.3.1] Extra functionality,* because this part of the model adds functionality not traced to requirements (at least to the ones in the baseline, the descoped ones are out of this analysis).

In this case, if bi-directional traceability between requisites and the model would have been properly done, the Unintended function would have been found during the Model Coverage Analysis.

Note that in this example, the descoped functionality is a part of an used block. In the case that the descoped functionality would be an entire non-used block, the error will be easily detected with traditional verification by traceability, and we wouldn't be an Unintended Function.

The UF contained in this State Diagram is due to model reuse: A model component has been copied from other part of the system, however some of the functionality provided is not needed. The reuse of already developed parts of a model always involves a risk about the introduction of errors and UFs related with the integration. This UF describes the involuntary introduction of correct, but not associated with requirements, code. The traditional Code Coverage approach can detect which parts of the model have not been executed and they should. But this kind of coverage is not able to detect which parts of the model are being executed and should not. Another approach is needed in order to detect this kind of UFs.

In these cases, requirements traceability can detect UFs if the requirements are detailed enough and the traceability is done at the lowest level. Unfortunately, Higher-level requirements are usually not as detailed as needed for a traceability deep enough to detect a partial descoping of a model component, just when the whole component should be removed. Therefore, the **SOMCA Prerequisite 3: Model - Requirements Traceability** and **SOMCA Prerequisite 4: Requirements – Model Traceability** can completely identify this UF when the whole component is descoped, but is not guaranteed to detect it for a partial descoping. Moreover, when the functionality is external to the model, additional consideration must be taken into account. The following prerequisite is required in those cases: **SOMCA Prerequisite 2**: **Source Code Coverage**. The detection depends on the coverage method used (as described in the prerequisite).

13.5.11. DEBUG ELEMENTS PRESENT IN OPERATIONAL VERSIONS

During the design and implementing phases of the project, a lot of debug blocks or modules can be used as helpers. There are also a lot of drivers and stubs used that should be deleted after they finish their utility. All these extra elements should be removed or disabled in the final versions of the product.

But sometimes (including actual airborne software), the removal of these debug elements are forgotten during the process, maybe because they don't affect to the functionality or because they are part of it. These elements include a functionality not traced to the requirements, implementing an Unintended Function.

An example of this kind of UFs would be the `LogRotate` system. A State Diagram is responsible of determine when a new file should be created and when old files should be deleted (keeping only the last three log files). During the design, a `DebugMode` input signal is added. This signal disables the deletion of files, helping the team during the implementation of the model. This signal is usually hardcoded to false. Once the model has finished, this signal is forgotten, and the verification does not detect any failure in this state diagram.

The State Diagram described in this example contains the following debug elements that the designer forgot to remove: an additional constant, input port, and transition controlled by the constant, which allows easily experimenting with certain functionality of the state machine instead of waiting during hours until the exact conditions occur nominally.

So the model contains a transition that is never executed, and it's not related with any requirement, generating an Unintended Function in the model. In this case, the coverage check over the state diagram transitions would have shown a transition not executed, revealing the Unintended Function. This is a real-work example of *[UF.3.1] Extra functionality*, because this part of the model adds functionality not traced to requirements.

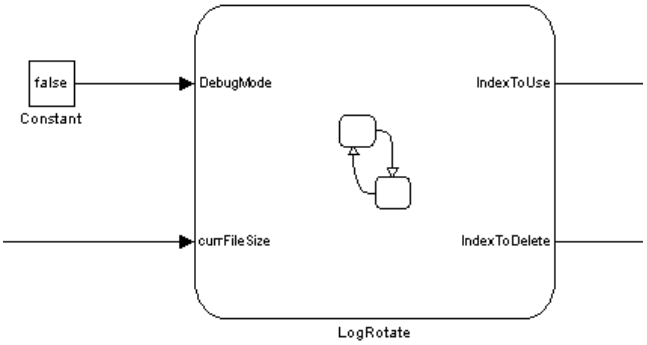


Figure 13-21: `LogRotate` inputs and outputs. Notice the `DebugMode` port.

In this case, the prerequisite about tracing the model components to higher-level requirements would probably uncover the extra debug functionality. The extra constant block, port, and transition should be easily spotted when performing the traceability analysis. However, as the prerequisite is applied to model components instead of unique elements, in some cases the traceability analysis may not identify every debug element.

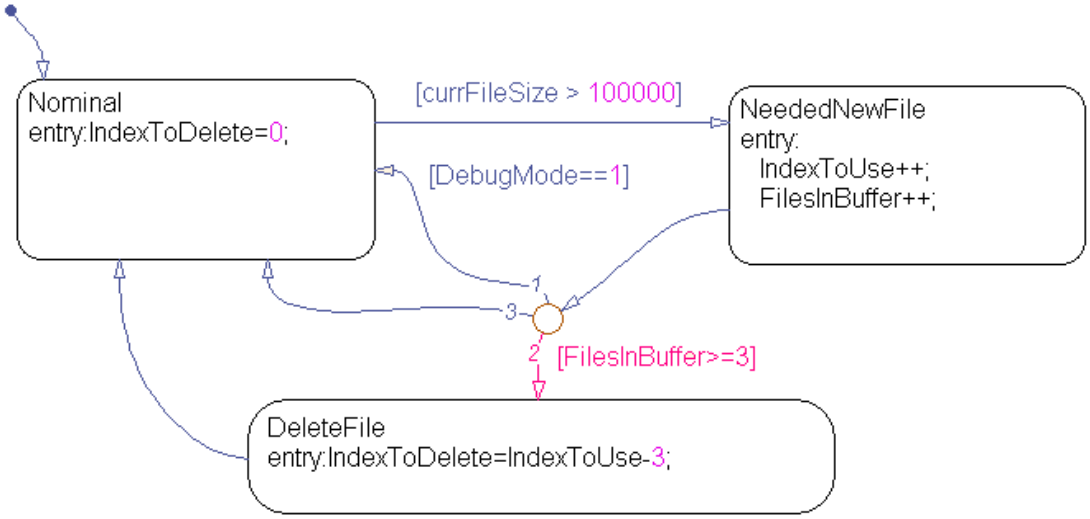


Figure 13-22: `LogRotate` state machine with extra `DebugMode` transition.

The proposed MCA criteria would also identify the extra debug elements of this example, namely the **SOMCA Criterion 9: Transition coverage**, which has to be applied for all DALs, will uncover the UF. Because either the `DebugMode` is set as True so both the transition 2 and 3 in the diagram will never be exercised (just the transition 1, triggered when `DebugMode` equals to 1), or the constant was set as False so the debug transition would never be trigger.

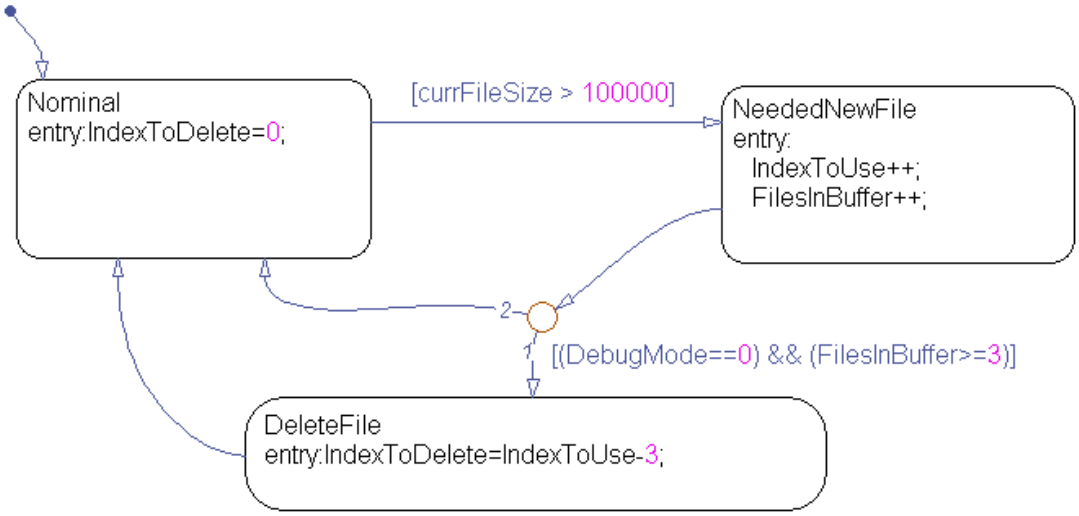


Figure 13-23: Modified `LogRotate` state machine with extra `DebugMode` condition.

Now suppose there is no extra debug transition, but the DebugMode input is added to the decision of an existing transition to enforce taking or not a specific transition (see above figure). In this case, if the DebugMode is set as True then that transition will never be taken, and thus the UF will be detected too for all software levels by means of the Transition Coverage criterion. However, if the DebugMode constant remained with the value False (which is more probable, otherwise the difference in the system behaviour would be easily noticed during the requirements-based tests) both transitions will be executed nominally so would not be detected by Transition Coverage criterion. In that case, the extra DebugMode condition in the decision will be identified if the model coverage would be performed as DAL A, by means of the MCDC analysis of the transition decisions (**SOMCA Criterion 11: Transition MC/DC**).

This UF could be also found, in both cases, thanks to the **SOMCA Criterion 3: Modified input coverage**, because the DebugMode input never changes, generating a coverage problem. This is an advantage over the MC/DC analysis because is a very simple analysis included in lower criticality levels.

In conclusion, there are different mechanisms available in the proposed MCA criteria and prerequisites to ease the identification and removal of extra debug elements. It is recommended to make all the debug elements dependant of a subsystem input instead of a global value.

13.5.12. HIGHLY COUPLED ALGORITHMIC ARCHITECTURE

Consider the following example of a complex algorithmic model with high data dependence between components. The final output of the algorithmic process is an assessment of the status of a series of elements. For instance, the assessment of the status of a set of receivers distributed all over the world, see Figure 13-24. This assessment is based in thousands of raw data measurements provided by these elements that are evaluated in other to determine with a given accuracy the capacity of each element to provide reliable information. Coming back to the example, the assessment of the status of the receivers was performed with the measurements performed by the receivers themselves.

Before using received raw data for the status assessment, there is a filtering stage aimed at rejecting outliers and noisy data that cannot be used due to the bad quality. This rejected data is not used as input in the following processing chain.

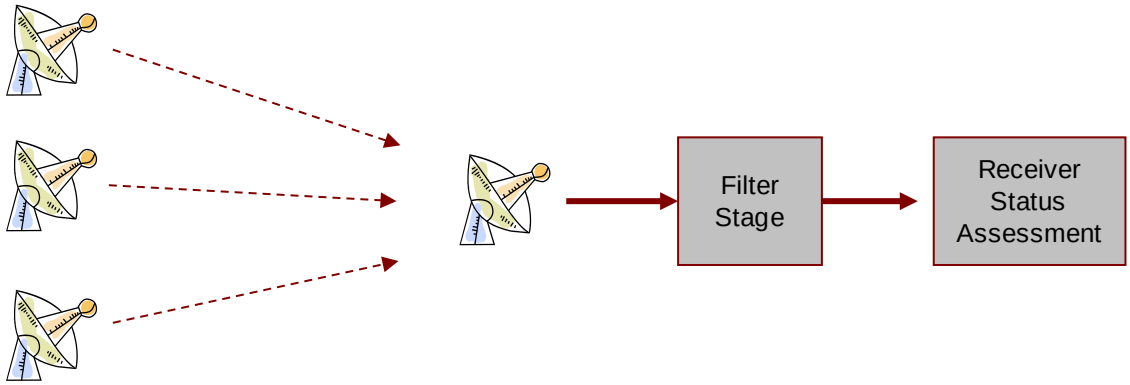


Figure 13-24: Example of Receivers Status Assessment

It was found that under certain non-nominal scenarios and error in the modelling of the initial data quality check filter was allowing to go through the processing chain incorrect raw data affecting a limited number of elements, as a result, many other elements were assessed to be unreliable as consequence of a contamination effect due to the extreme complexity of the algorithmic architecture.

The complex nature of the system makes that an error in input propagates across the entire system, generating several individual Unintended Functions that can be joined in a bigger one associated to the entire system and the scenario. This Unintended Function is *[UF.5] Derived from System complexity.*

This Block Diagram example behaves in an incorrect way under specific non-nominal conditions; specifically the filter component is not rejecting bad quality input data for some stations. When this incorrect raw data is processed by the stations, those elements are considered unreliable due to the contamination of the input data, producing a failure of the whole system. Therefore, in this example the effects of different Unintended Functions are joined together due to the complexity of the algorithmic architecture.

The root cause of this example is a set of wrong numeric computations performed at the filter stage, just in the event of non-nominal inputs. These non-nominal scenarios could be selected through the **SOMCA Criterion 1: Range coverage**, when searching all equivalence classes. However, for complex input data the complete set of equivalence classes could be too high.

SOMCA Criterion 2: Functionality coverage is not well suited for the detection of this problem because the different set of operations performed by the component can be completely exercised without the problematic input scenario, as thus cannot guarantee the detection of this UF. In addition, the different path coverage criteria are not applicable in this case because the problem is due to a numeric computation.

In summary, although the Range Coverage criterion could detect the problematic scenarios, for complex systems the MCA criteria is not guaranteed to detect this UF.

13.5.13. EVENT LOG LENGTH

The following example is based on an actual problem found in the scope of the Galileo activities in GMV. An imprecise requisite derived in implemented features not traced to requirements.

We have to create a model for a log system. Due to the nature of the problem, the formalism to use is State Diagrams. The requirements specify that the first 48 hours of execution should be stored and not deleted until an operator extracts that log (we will call this buffer the linear one), and the following events should be stored in another buffer, and this one should contain the last 48 hours of execution (this buffer will be the circular one). Thanks to this mechanism, an operator will be able to see the log from the system start, and the last part, and the disk space used is controlled and limited.

The problem came in the understanding of what the system should do after the operator extracts the log. Two options were available:

- Use both buffers again: We will store up to 96 hours of data (the first 48 in the linear buffer, and 48 more in the circular one)
- Use only the circular one: Only the last 48 hours will be stored in the log.

These decisions were not clear in the specification. When the model was done, the requirement was understood as the first option, but later, the inspection team understood that the second option was the best one.

This problem is a good example of *[UF.2] Specification problem*. Here, the code inspection raises this problem, and it was detected before the operational phase of the system, but this kind of errors is usually not detected, and the UF can be hidden in the model until it generates a problem.

The **SOMCA Prerequisite 3: Model - Requirements Traceability** (all model elements have to be traced to higher-level requirements) helps to identify this ambiguity in the specification: The model elements in charge of handling the buffers after the log extract operation need to be traced to a specific requirement. However, no requirement specifies the usage of the buffers after this operation. Furthermore, the **SOMCA Criterion 2: Functionality coverage** will ensure the model is exercised after the log extract operation, which would force to check the associated requirements for the assessment of the result of the verification cases with respect these buffer, helping to detect the specification problem. However, even if the MCA criteria could help to identify this UF, actually the prerequisite for requirements validation is needed to properly detect this kind of specification problems.

13.5.14. INTERNAL PRECISION LOSS IN ARITHMETIC BLOCK

For this example, any block with arithmetic process could be used. We will take, for example, a moving average block.

This moving average block has an input and an output, and the output calculates the average of the last 100 values read at the input. If less than 100 values have been read, the average is calculated over the available values. This value is validated using a predefined table of input and output values, stored with 4 decimal values. The block is validated based on this table.

But later, in operational environment, a small precision loss is detected in this block, an error not detected in the validation, but significant in the system where this block has been implemented. The detailed analysis show that the internal accumulator used in the block has 32 bits, but the inputs and the outputs has 64 bits.

In this example, a systematic error was not detected in validation because of the data used in the validation were not detailed enough. We have an example of *[UF.3.2] Incorrect behaviour in the general case*.

Epoch	Measurement in the input	Number of measurements stored	Expected output	Model output	Validation value	Error
1	1.4533234535	0	-	-	-	-
2	1.4533345635	1	1.4533234535	1.4533234239	1.4533000000	0.0000000296
3	1.4533233456	2	1.4533290085	1.4533289671	1.4533000000	0.0000000414
4	1.4535689567	3	1.4533271208	1.4533271194	1.4533000000	0.0000000015
5	1.4512389673	4	1.4533875798	1.4533875585	1.4533000000	0.0000000213
6	1.4533788968	5	1.4529578573	1.4529578090	1.4529000000	0.0000000483
7	1.4567856786	6	1.4530280306	1.4530280232	1.4530000000	0.0000000073
8	1.4535678458	7	1.4535648374	1.4535648227	1.4535000000	0.0000000147
9	1.4567856786	8	1.4535652135	1.4535651803	1.4535000000	0.0000000332
10	1.4556785679	9	1.4539230429	1.4539229870	1.4539000000	0.0000000559

Table 13-2: Inputs, outputs and error in the Moving Average Block

This UF occurrence is similar to the example analysed in section 13.5.2 above: The internal computations performed by an arithmetic block are performed with a data type with less precision (32-bit wide) than the input and output ports (64-bit wide), so the results will be inaccurate.

In this example the precision loss was not detected because the verification process used for this block used a table with pre-computed values with less precision than the output. However, as in the example of the Max block, the prerequisite of ensuring there is no data-type mismatches (**SOMCA Prerequisite 8: Type check**) is the easier way to detect both the UF in the model, and the precision problem in the verification table. Other phases of the MCA criteria are not appropriate for detecting this UF: The **SOMCA Criterion 1: Range coverage** will select different verification cases, all of them suffering the precision problem but it will not be detected because the table to verify the results is also wrong; the **SOMCA Criterion 2: Functionality coverage** could be satisfied once all the few functions implemented by this block have been exercised by verification cases, but again even if the outputs are not precise enough this criteria will not detect the UF; finally, the **SOMCA Criterion 6: Logic Path coverage** is not relevant in this case because the modelling problem is due to an error in a numeric path, and not in a logic one.

In conclusion, the prerequisite of solving data-type mismatches is the key to detect this UF, which also shows the dangers of performing the verification of the numeric computations in an inappropriate way due to the precision problems of the table with pre-computed values.

13.5.15. PROTOCOL CONTROL COUPLING

The following example has been extracted from a key module in the IPF RTMC project, in the scope of the Galileo activities. This example presents a system composed of multiple high-level communication blocks that use one low level communication block.

The protocol is a file transfer protocol for critical applications, with checks, confirmations and many control files. This protocol system is shared by four different higher- level modules, and for each one of them, the protocol is slightly different, activating or deactivating some features. The functionality provided by the low level communication block was configurable to an excessive degree, this is shown in the diagram with parameters p_1 to p_n.

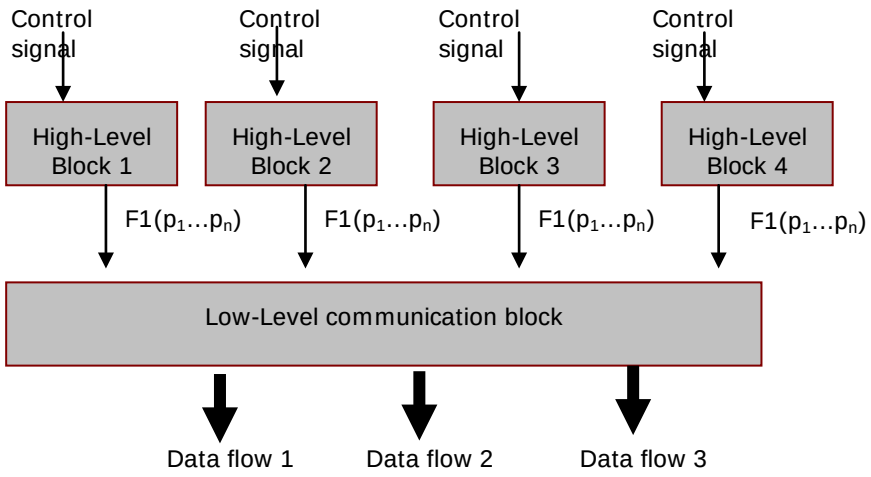


Figure 13-25: Example of protocol coupling

This kind of design is quite complex, and the four modules that share this system are highly coupled, because some parts of the behaviour implemented in the low level block are common but not the others. The experience has shown us that this is a continuous source of Unintended Functions from several types, like:

- Adding the use of a CRC in one of the upper modules introduced an Unintended Function in another one, because the second one must not use CRCs
- Adding a special control file for one of the modules affects the complete protocol behaviour, generating several Unintended Functions in the other ones.
- Changing the information included in one field of one control file removes an Unintended Function from one of the modules, but adding two new Unintended Functions on the other higher-level modules.

The nature of this protocol control element suggest that it should be designed as a State Diagram, with several states and substates to control each stage of the protocol. The final state diagram for this example is too big and the details of it are out of the scope of this section, so it won't be included here.

This is an example of how transition and state coverage can eliminate unintended functions from complicate models. The high number of states, substates and transitions elevates the complexity of achieving full coverage of the model, but making an effort in a more intense verification in order to improve coverage can be very useful in the early detection of UFs.

All these problems can be classified as derived from *[UF.5] Derived from System complexity*, because the source of all of them is the high level of coupling in the entire system, and the size of the State Diagram that controls this functionality.

Therefore, the Unintended Functions contained in this State Diagram are due to the complexity of the multiple protocols modelled, each one providing slightly different functionality. Furthermore, the maintenance of this model is problematic because a change in one protocol introduces regressions in other protocols.

The **SOMCA Prerequisite 3: Model - Requirements Traceability** (tracing the model elements to a specific higher-level requirement) is an effective way of detecting these protocol problems. As described in the IR1, when some functionality is just part of one of the protocols it is wrong to share that part of the model with the other protocols. Therefore, the requirements traceability can detect when a part of the state diagram should be specific for one protocol, but actually can be activated when exercising other protocols.

The main tool provided by the MCA criteria to detect the described UFs is the **SOMCA Criterion 2: Functionality coverage.** It ensures that all the different protocols are simulated, and thus possible regressions after a change in one protocol would be detected. The different state coverage criteria are not as useful in this case because these do not detect problems of states that shouldn't be shared among different protocols, although can identify regressions if a state of a specific protocol is no longer executed after a change in the model. The same considerations apply to the different coverage criteria for actions and transition decisions.

13.5.16. ERROR IN MODEL INPUTS

For this example, we will have a clock-correction system, with several inputs to the model detailed below:

- GPS Time, received by an antenna
- External time signal received in an IRIG-B (wired)
- Local clock

The main objective of the system is to keep synchronized the local clock, making corrections if necessary, but raising an alarm if any of the clocks differ from the others more than 100 milliseconds. The system is designed through a model based on state machines. A very intense validation is performed over the model and the result of this validation is successful, even on worst-case scenarios.

When this system is implemented and connected to real-time signals, the system boots and raises the alert every second, despite the received signals and the local clock are correct.

A first analysis of the problem shows the source of this Unintended Function. The time received in the GPS was 15 seconds greater than the IRIG-B one, but both of them were correct. The leap seconds (corrections done to the UTC time) were not being taken into account. The system was validated assuming that all external time signals are exactly in the same scale, but that wasn't the case:

- The External time signal is received in UTC, including leap seconds correction
- The GPS time was originally based on UTC, but no leap corrections have been applied since its start in 1980.

This is a good example of *[UF.4] Derived from Model Interfaces*, where the problem is located in the misunderstanding of the Model Interfaces.

In this state machine to synchronize the local clock with two external time signals: a GPS Time signal and an IRIG-B signal. It was assumed that both signals provided an UTC timestamp, however the GPS Time does not include leap seconds. This problem in the inputs specification was not detected until the system tests were executed in the final platform.

There is one prerequisite related with this Unintended Function: **SOMCA Prerequisite 15**: **Environment definition**. According to this prerequisite, the environment conditions must be explicitly documented, emulated, and verified, which would imply specifying the values received from each time signal, and also to possibly record the external signals to properly emulate at model level the environment.

In case the model is simulated with wrong time signals, the MCA criteria would not detect the problem at model level. The **SOMCA Criterion 1: Range coverage** would exercise the model with different time signals, but if their values are expected to be the same the behaviour of the state diagram will be considered correct. The **SOMCA Criterion 2: Functionality coverage** will also ensure that the case of unsynchronised time signals is simulated, but again the behaviour will seem correct. The **SOMCA Criterion 9: Transition coverage** and **SOMCA Criterion 12: Event coverage** would exercise again that the model containing the wrong assumption, but would not help either to invalidate the input signals in case they are not properly emulated.

13.6. ADDITIONAL UNINTENDED FUNCTIONS PROVIDED BY ESTEREL

This section has been included as provided by Esterel, and no modifications have been done in the original text.The results have not been included in the study results.

13.6.1. DIVISION BY ZERO COVERAGE

13.6.1.1. Introduction

This example is interesting because it addresses a feature of model coverage that is not limited to the usual Boolean operators and control structures. It addresses coverage of the behavioral classes of an arithmetic operator.

The library operator is a robust division by zero. If the denominator absolute value lies inside an interval with threshold (] -Tol, +Tol [) then it returns a default value, else it performs the normal division.

•

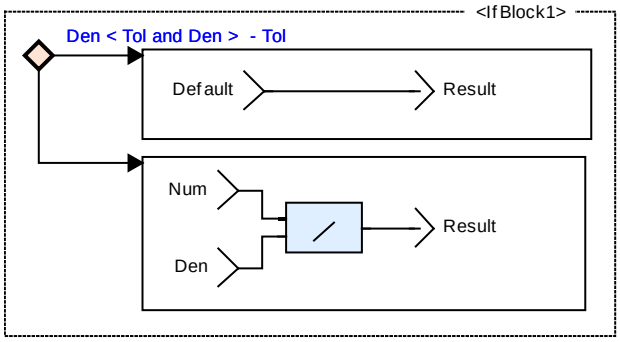


Figure 13-26: Robust Division

There are two instances of safe division with different threshold values ([-0.1 and 0.01).

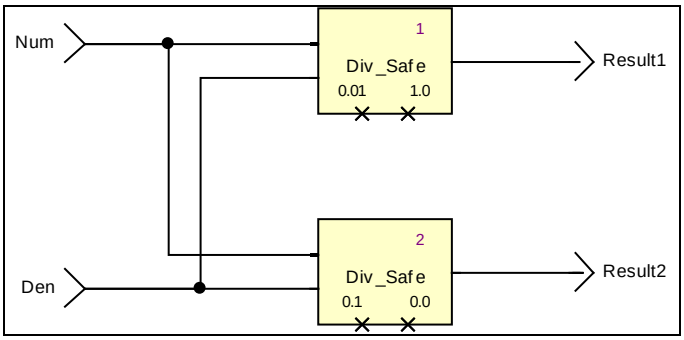


Figure 13-27: Uses of Robust Division

13.6.1.2. Testing

In the scenario the Denominator the denominator value enters the range [-0.1 ; 0.1] but never enters the range [-0.01 ; 0.01]. Thus only the second block is used in protected mode.

13.6.1.2.1. Test Case

SSM::set Root/Num 3.0	*SSM::set Root/Num -3.0*
SSM::set Root/Den 2.0	*SSM::set Root/Den -2.0*
SSM::cycle 1	*SSM::cycle 1*
SSM::set Root/Num 3.0	*SSM::set Root/Num -3.0*
SSM::set Root/Den 1.0	*SSM::set Root/Den -1.0*
SSM::cycle 1	*SSM::cycle 1*
SSM::set Root/Num 3.0	*SSM::set Root/Num -3.0*
SSM::set Root/Den 0.5	*SSM::set Root/Den -0.5*
SSM::cycle 1	*SSM::cycle 1*
SSM::set Root/Num 3.0	*SSM::set Root/Num -3.0*
SSM::set Root/Den 0.05	*SSM::set Root/Den -0.05*
SSM::cycle 1	*SSM::cycle 1*

13.6.1.3. Coverage Analysis with Predefined Logical Criteria

With purely logical criteria, even at MC/DC level, full model coverage is achieved as shown on the MTC report below.

2.3. Operator Criteria

Div_Safe/

Operator	Status	
IfBlock1:$and 1	PATH1={AND#1 i1, AND#1 o1} True	X
	PATH1={AND#1 i1, AND#1 o1} False	X
	PATH2={AND#1 i2, AND#1 o1} True	X
	PATH2={AND#1 i2, AND#1 o1} False	X
IfBlock1:$gt 1	True	X
	False	X
IfBlock1:$lt 1	True	X
	False	X

Root/

Operator	Status	
Div_Safe 1	Operator activated	X
Div_Safe 2	Operator activated	X

2.4. Control Criteria

Div_Safe/

Control	Status	
IfBlock1:else:	Activated	X
IfBlock1:then:	Activated	X

13.6.1.4. Introducing a User Coverage Criterion

In order to assess whether **each individual instance** of this operator has been used both in normal and protected modes, we introduce a specific user-defined coverage criterion, which has 2 coverage cases:

- Normal division
- Protection against Divide by Zero

When applying this criterion to the example, we can observe that the first safe division is never called in DivideByZero mode.

2.3. Operator Criteria

Div_Safe/

Operator	Status	
IfBlock1:$and 1	PATH1={AND#1 i1, AND#1 o1} True	X
	PATH1={AND#1 i1, AND#1 o1} False	X
	PATH2={AND#1 i2, AND#1 o1} True	X

	PATH2={AND#1 i2, AND#1 o1} False	X
IfBlock1:$gt 1	True	X
	False	X
IfBlock1:$lt 1	True	X
	False	X

Root/

Operator	Status	
Div_Safe 1	Normal divide	X
	Divide by Zero	
Div_Safe 2	Normal divide	X
	Divide by Zero	X

2.4. Control Criteria

Div_Safe/

Control	Status	
IfBlock1:else:	Activated	X
IfBlock1:then:	Activated	X

Resolution of this coverage shortfall leads to the introduction of additional test cases, to verify the behavior of the software when defensive behavior of the first division is activated, or to a justification that this cannot be activated and that this presents no safety risk.

13.6.1.5. Conclusion for this Example

This example shows that appropriate user criteria can capture the coverage of functional equivalence classes of library operators. This allows verifying that these equivalence classes are covered by test cases. An untested behavior could correspond to an unintended behavior.

13.6.2. MODEL INCONSISTENCY

13.6.2.1. Introduction

This example concerns section 13.2.1 of the GMV SOMCA report (GapNotDetected).

When analyzing the model coverage results obtained when running the SmoothingAlgorithm test cases, QMTC detected a number coverage shortfalls.

13.6.2.2. Coverage Analysis with Predefined Logical Criteria

When analyzing coverage of the caller of the RunningMode operator, MTC detected that the input psiGapSize of RunningMode has never changed. MTC also detected that PstInMeas.bCode in RunningMode was never false, which causes this lack of change of psiGapSize.

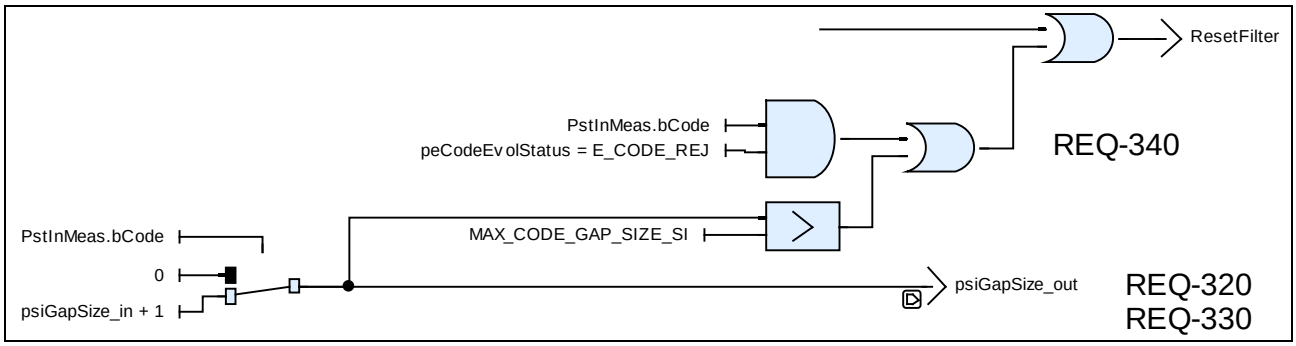


Figure 13-28:: Filter Gap Analysis and Reset

Analysis shows that this is due to a test of PstInMeas.bCode which cannot be satisfied. Indeed this test occurs within a branch which activation condition is “PstInMeas.bCode and PstInMeas.bPhase and PeCombCSStatus = E_CS_OK”. If this branch is activated PstInMeas.bCode is necessarily true, and the psiGapSize is never incremented.

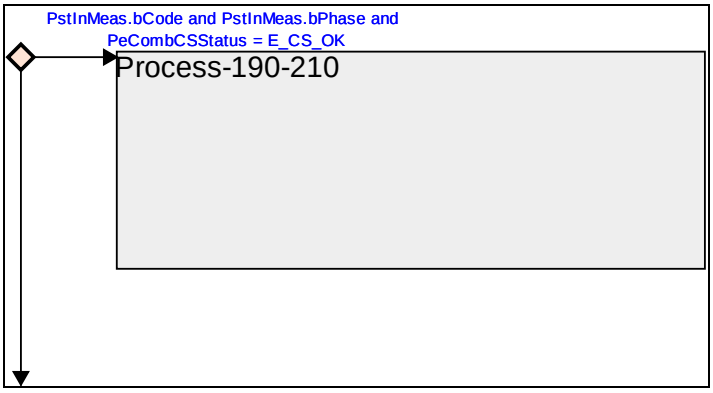


Figure 13-29:: Enclosing Branch Activation Logic

Analysis shows that this design is inconsistent. A re-analysis of the High-Level Requirements and of the model are needed to clarify the source of the problem and to resolve it.

13.6.2.3. Conclusion for this Example

This example shows that model coverage analysis may help in detecting and analyzing High or Low Level Requirements inconsistencies.

13.6.3. UNINTENDED ACTIVATION

13.6.3.1. Introduction

This example shows how MTC can help detecting lack of testing of an inhibition function.

There is a command, which unintended activation presents a safety risk. Thus the inhibition logic of this command is critical.

The HLR contains text describing the inhibition logic is a form such as:

Inhibit command X if C1 and not C2 or C3

This formulation is ambiguous and should normally be rejected during the review of the HLR, but real-life examples show that this type of text may exist.

The intent of the author of the HLR for the inhibition condition was:

Meaning A) Meaning A) C1 and ((not C2) or C3)

Assume the author of LLR understands and designs this requirement as follows:

Meaning B) C1 and not (C2 or C3)

This design contains an unintended functionality: command X may be activated in two cases where it should not. These are cases 6 and 8 of the truth table, where the inhibition function returns false instead of true.

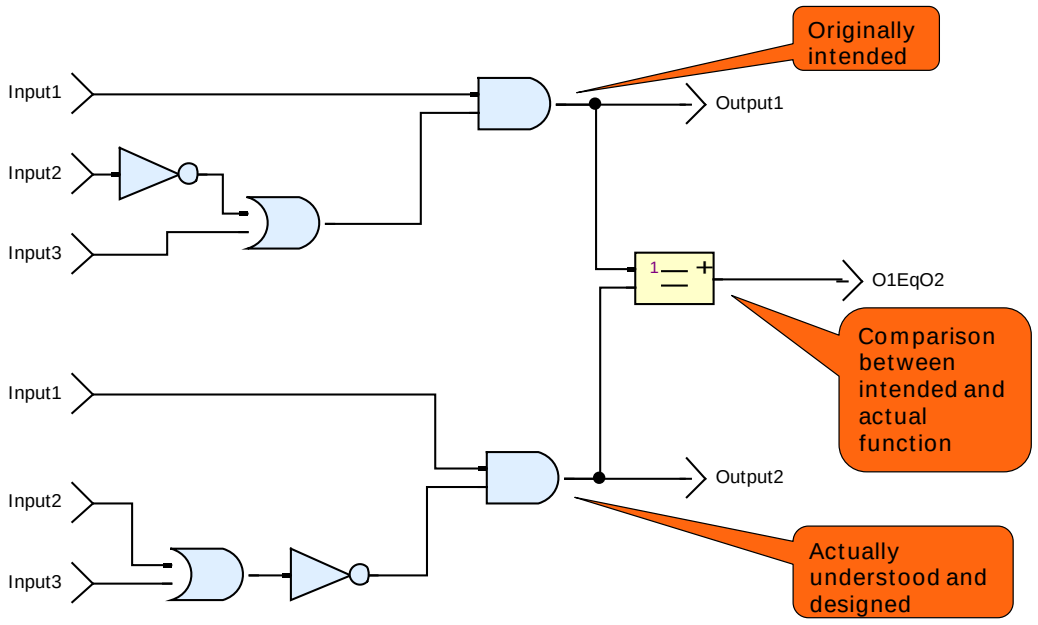


Figure 13-30: Actual versus Intended Inhibition Logic

Table: Truth Table of Actual versus Intended Inhibition Logic

TC	Input1	Input2	Input3	Output1	Output2	O1EqO2
1	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE
2	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE
3	FALSE	TRUE	FALSE	FALSE	FALSE	TRUE
4	FALSE	TRUE	TRUE	FALSE	FALSE	TRUE
5	TRUE	FALSE	FALSE	TRUE	TRUE	TRUE
6	TRUE	FALSE	TRUE	TRUE	FALSE	FALSE
7	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE
8	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE

Only test case 6 or 8 can detect a difference between actual and expected output of the inhibition.

Assume on uses a set of test cases covering all cases of the truth table except TC 6 and 8. These will not detect the non compliance.

13.6.3.2. Coverage Analysis with DC

Decision coverage is achieved with these test cases, as shown below.

Operator	Status	
$and 1	True	X
	False	X
$and 2	True	X
	False	X
$not 1	True	X
	False	X
$not 3	True	X
	False	X
$or 1	True	X
	False	X
$or 2	True	X
	False	X

13.6.3.3. Coverage Analysis with Local MC/DC

Local MC/DC is achieved with these test cases, as shown below.

Operator	Status	
$and 1	All True	X
	Only I1 False	X
	Only I2 False	X
$and 2	All True	X
	Only I1 False	X
	Only I2 False	X
$not 1	I1 True	X
	I1 False	X

$not 3	I1 True	X
	I1 False	X
$or 1	All False	X
	Only I1 True	X
	Only I2 True	X
$or 2	All False	X
	Only I1 True	X
	Only I2 True	X

13.6.3.4. Coverage Analysis with Masking MC/DC

Masking MC/DC analysis detects that the unmasked effect of “true” on Input3 on the output has not been shown.

Resolution of this shortfall requires the addition of TC 6 or 8, which will show that the function behaves incorrectly.

Operator	Status	
$and 1	PATH1={Input1, Output1} True	X
	PATH1={Input1, Output1} False	X
	PATH2={Input2, Output1} True	X
	PATH2={Input2, Output1} False	X
	PATH3={Input3, Output1} True	
	PATH3={Input3, Output1} False	X

13.6.3.5. Conclusion for this Example

This example shows that MTC may help in detecting untested unintended functionality.

14. ANNEX C: ANALYSIS OF MODEL FORMALISMS

This Annex has been extracted from Interim Report 1 [RD.1], Section 6.

The principal objectives of this section are firstly to study the different modelling notations relevant for the development of safety-critical software within the aviation industry, secondly to describe the features provided by the toolsets supporting each notation, and finally to analyse the possibility of each formalism to introduce Unintended Functions in the model. A secondary objective is to clarify the intent of some key terms related with model formalisms in the scope of this project, which have sometimes different meanings within the industry.

The main inputs for this section are GMV's experience working with some of these notations in different projects, a detailed literature review of model-based development, EASA inputs including the certification memorandum, and the public documentation provided by the tool vendors. Finally, an analysis of the characteristics of each formalism was performed in the scope of the SOMCA project to give a set of considerations regarding the possibility of introducing Unintended Functions.

14.1. INTRODUCTION TO MODEL FORMALISMS

Developing a model of the system and being able to experiment with it and analyse it early in the life cycle can really improve the **understanding of the system requirements**, but also can help to **find many errors in the requirements specification or design**. Ensuring the consistency of the requirements and the model verification coverage are key activities for safety critical system that should address a certification process, like airborne software systems, since they will allow the identification of hazardous inconsistencies. For example, the detection of Unintended Functions, that potentially could have a serious impact on the safety objectives of the system, for instance because of the lack of deterministic behaviour and the uncertainty that are associated to them.

There is a wide variety of model formalisms and toolsets, and many of them support the features described above, at least partially. However, it depends on the goals of the notation. Some formalisms are focused on specification of system requirements, describing what the system has to do but not how, so code generation is not an applicable feature but tools for dynamic tests and statically checking the internal consistency could be provided. Other notations are aimed at the design of hardware and software architectures, which usually focus on code generation.

Between these two categories, there are a wide range of different notations with varied purposes (like SDL for specification of communications protocols, UPPAAL for modelling concurrent software processes to find synchronisation errors…), different formality level (from a general-purpose UML tool for informal specification of requirements or use cases, to a formally-verified language semantics and theorem provers like Coq or PVS), and varied commercial support (small academic project with weak tool support and few university users, to full-featured commercial development suites used in thousands of industrial projects).

The use of design modelling languages, such as Simulink and SCADE (Safety-Critical Application Development Environment), in the development of safety-critical systems appears to be part of a long-term trend. This popularity indicates that system developers see genuine value in their use. This study focuses on notations used for airborne development, specifically on those notations used for **Formalized Requirements** or **Formalized Designs**. These two types of notations are complementary, and both approaches can be used in the same avionics project for different development phases. This gives different possible software life-cycles, depending on the formalisms used for the development of the different system and software parts. The recent EASA Certification Memorandum [RD.8] describes up to five different software development life cycles, using formalized notations or traditional techniques for these different system / software development processes. The following definitions for Formalized Requirements and Formalized Design apply:

> *A **Formalized Design** may be a model produced by the use of a modelling tool or it may be a design stated in a formalized language. In either case, a Formalized Design should contain sufficient detail of such aspects as code structures and data / control flow for Source Code to be produced directly from it, either manually or by the use of a tool, without any further information.*

> ***Formalized Requirements*** *may be produced by the use of a modelling tool or they may be requirements stated in a formalized language. They contain high level requirements that can later be implemented in either a Formalized Design or in a conventional software design. Formalized Requirements should neither contain Software design nor Software architecture details.*

It should be noticed that not all system requirements are modelled either in a Formalized Design or a Formalized Requirements specification (for example some non-functional requirements can be impossible to be modelled). Furthermore, they can be intentionally be left outside the design because add complexity to the design without any added value. In the following subsections this idea will be extended because, depending on the limitations of the specific notation, this could be a source of Unintended Functions.

14.2. FORMALIZED REQUIREMENTS

The main objective of a set of Formalized Requirements is to **avoid ambiguity** in the specification of requirements with respect the traditional way of using natural language to describe all the requirements. This is a key feature to eliminate Unintended Functions related with an incorrect interpretation of a requirement. In addition, the use of a formalism provides powerful possibilities, like the use **dynamic verification directly at requirements level**, without the need to wait until the system is implemented. Therefore, a requirements specification can be "debugged" very early, and thus a great share of requirements errors that were not detected until late phases of the project can now be fixed even before the software is designed, thus removing all those possible Unintended Functions related with the need to patch a design and implementation during system testing or even maintenance.

Furthermore, some of these notations leverage the power of formal methods, allowing to **prove the absence of unsafe properties** and to **detect inconsistencies in the requirements**, to increase the confidence on the validity of the specification. In addition, if both the requirements and design are formally modelled then it is possible to check its compatibility through formal analysis (at least in theory, we are not aware of any of such tools used in industry), which would also help to avoid Unintended Functions. Otherwise, it must be checked by manual review as with traditional techniques. This is an additional feature to ensure consistency on the system behaviour at all levels of the life cycle, thus reducing the Unintended Functions in a way previously not possible with traditional techniques.

It is worth noting that the boundaries between requirements and design notations can be fuzzy, and the ED-12B / DO-178B explicitly states that low-level requirements may be derived from design constraints in addition to higher-level requirements. However, even if sometimes Simulink or SCADE models are being used as a detailed requirements specification, a recent FAA industry survey showed that there is no consensus within the industry if these models really represent low-level software requirements, high-level software requirements, system requirements, or whether they represent requirements at all [RD.9]. This section will thus cover those notations originally developed for the specification of system and software requirements, while SCADE and Simulink will be covered in the subsection about Formalized Designs. Both textual notations and graphical formalisms have been considered for the requirements modelling of airborne systems.

Many modelling formalisms are available to specify software requirements, and the following formalisms have been (and are being) used in aerospace projects:

- SCR (Software Cost Reduction)
- RSML and RSML-e (Requirements State Machine Language without Events)
- SpecTRM (pronounced "spectrum", and abbreviation of Specification Toolkit and Requirements Methodology)
- SysML (Systems Modeling Language)

The **SCR** methodology (Software Cost Reduction), developed by Parnas and Madey, was one of the first notations for Formalized Requirements. Used for the specification of requirements of the U.S. Navy's A7E Corsair, the SCR methodology also captures assumptions about the environment and is based on the so-called four-variable model [RD.9]:

- Monitored variables (MON): environment values monitored by the system.

- Controlled variables (CON): environment values controlled by the system.
- Input variables (INPUT): the MON values provided by the system and seen by the software.
- Output variables (OUTPUT): the CON values computed by the software as provided to the software.

These variables are continuous functions of time, and the following five tables define the system requirements through mathematical relationships between the variables:

- NAT: natural system constraints, or environmental assumptions, imposed by the environment, such as the maximum rate of climb of an aircraft
- REQ: how the controlled variables (CON) respond to changes in the monitored variables (MON)
- IN: map between the input variables (INPUT) and the monitored variables (MON)
- OUT: map between the output variables (OUTPUT) and the controlled variables (CON)
- SOFT: bounds of the true software requirements without specifying the software design

This explicit definition of variables and equations of SCR models, not only completely eliminate ambiguity but also greatly reduce the gap between the requirements specification and the design, enhancing the validation. In addition, some of the tools available for SCR provide static on-button-checker [RD.25] (i.e. just write the specification, push a button and see the list of errors and warnings, there is no need to manually develop a proof) for statically checking the specification to analyse the internal coherence and possible errors (like unused variables) without requiring any specialised knowledge about the underlying logic. The animation of requirements can also be provided, that is the dynamic testing of a requirements specification with a battery of tests.

RSML (Requirements State Machine Language), developed by Nancy Leveson, is a requirements modelling language similar to SCR, with the addition of state machines (specifically to Harel's Statecharts [RD.26]see section 14.3.2.1), and was used in the U.S. for the TCAS II specification (Traffic Alert and Collision Avoidance System). One of the main goals of RSML is readability by non-computer professionals, such as system engineers, managers, representatives from regulatory agencies, or system users. In addition to variables and functions, RSML specifications also consist on hierarchical states and transitions.

RSML methodology has also evolved, and different variations exist including the **RSML^{-e}**, which is well suited for the most advanced formal verification techniques such as model checking and theorem proving. Thanks to a NASA project, translators of the RSML^{-e} were developed to the NuSMV model checker and the PVS theorem prover. Or the **SpecTRM** toolset and methodology, also developed by Leveson [RD.27]very similar to RSML and were a SpecTRM-RL specification also captures assumptions about the environment and documents human factors requirements to satisfies completeness criteria that ensure safety information is captured about inputs, outputs, state information, and modes.

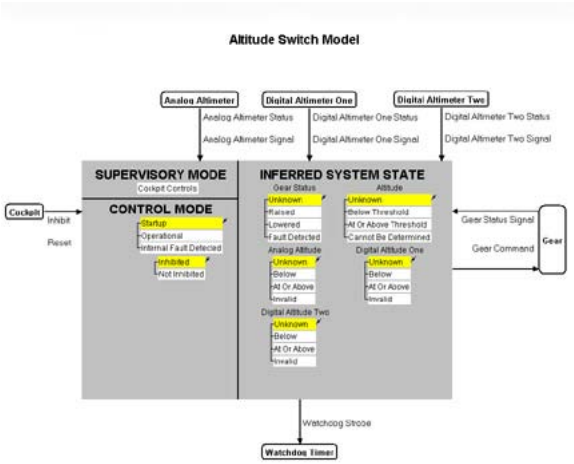


Figure 14-1: SpecTRM model

The **Systems Modeling Language (SysML)** is a general-purpose modeling language for systems engineering applications developed by the OMG (Object Management Group). It supports the specification, analysis, specification, design, verification and validation of a broad range of systems and systems-of-systems. This language is based on UML 2.0, but, as UML 2.0 is mainly designed for software design, several characteristics have been changed to give support to wider range of applications.

One of these new applications is the specification of formalized requirements in SysML (thanks to the inclussion of two new diagram types: requirement and parametric diagrams). Requirement diagrams are a special type of SysML diagram that allows to model the system requirements graphically. Using these diagrams, requirements allow the designer to associate the requisites with the model elements that implement them.

14.2.1. CONCLUSIONS

In summary, the use of Formalized Requirements is a very powerful tool to extensively document system or software requirements in a standardised way, including intent descriptions of each requirement, environment assumptions, human factors... in a high-level notation but making explicit the exact states and / or variables that the software will internally process. These notations completely avoid ambiguity and greatly reduce the gap between requirements specifications and designs, which reduces even more the possibility of introducing some kinds of Unintended Functions at design level. A disadvantage of this approach is that writing these formalized specifications usually takes much more time than the traditional approach of using just natural language and static diagrams. However, this force the system designers to put more attention to the requirements phase, resulting in less errors early in the development cycle, and as stated above experience shows that errors in the requirements is the main source of safety problems in airborne software.

Some toolsets include the ability of performing simulations to the specifications to easily experiment the implications of each requirement in the whole specification, which allows a deep understanding of the system and thus to reduce possible Unintended Functions. Furthermore, some of the most advanced toolsets provide powerful formal analysis checkers to analyse the internal consistency which really improve the detection of the most subtle errors.

However, not all requirements can be modelled (due to the specific goals of these notations and the generic intent of some system requirements) nor all of them should be formalized (due to the high effort of writing these specifications), which could lead to the introduction of Unintended Functions due to the interactions of the Formalized Requirements with the not modelled ones, or due to a wrong selection of the set of requirements modelled. And sadly, these notations and tools are much less extended than formalisms used for design modelling, and thus their commercial support is much less mature and the most advanced features are not so developed.

14.3. FORMALIZED DESIGNS

Over the last decade, synchronous, graphical modelling languages have been increasingly used in the development of safety-critical systems. Simulink and SCADE are by far the most wide-used development tools, at least for airborne software. Due to its popularity, these tools have been sometimes used for the specification of requirements, however Simulink and SCADE main use continues to be the design and experimentation of a system, and the code generation. Simulink and SCADE are widely used in automotive and avionics systems development.

Both Simulink and SCADE provide two different types of notations for the development of Formalized Designs, which are in fact the most extended categories of formalisms used in commercial model-development tools: **block diagrams** for continuous control and **state diagrams** for discrete control. Each notation is focused on modelling different types of algorithms, and commercial tools usually support both notations, even allow mixing these two types of formalisms in the same design. Both types of formalisms are aimed at handling complex designs, providing different abstraction constructions where each block or state can be decomposed into a hierarchy of blocks or substates.

These notations for Formalized Designs allow the simulation of the models to perform **dynamic verification** at design level, a feature not available in traditional development techniques. Like with Formalized Requirements, this dramatically changes the possibilities of the designers, allowing them to better understand

the implications of a design decision with respect a wide variety of test scenarios, and thus removing many design errors early in the project. Obviously, this feature is critical to remove many types of Unintended Functions that were not previously possible with traditional techniques.

Code generation is also a common feature of these toolsets, usually to safe profiles of the Ada and C programming languages. In addition, some of these development environments include or can be extended with specific tools for **model coverage analysis** and **static analysis for formal verification**, **requirements traceability** (interfaced with DOORS or RequisitePro), or code generation to **high-integrity programming languages** like Spark. Even if all these features were also possible with traditional development techniques, MBD tools offer further integration among all them reducing the effort required to apply them and the possibilities of introducing mishaps, thus gaining increased confidence on the absence of Unintended Functions.

The following subsections will further detail the characteristics of bock diagrams and state diagrams for the development of Formalised Designs.

14.3.1. BLOCK DIAGRAMS

14.3.1.1. Introduction

Block diagrams is a category of graphical notations for designing dynamic systems, i.e. whose outputs change over time like electrical circuits, mechanical or thermodynamic systems. The designer drags and drops blocks onto a canvas and connects the outputs of one block to the inputs of another block, to graphically depict the time-dependent mathematical relationships among the system's inputs, internal variables, and outputs. Blocks can also be parameterised (usually displaying a wide variety of configuration options, including format of internal data types), and can be composed hierarchically from simpler blocks, allowing designers to organize complex system designs. New blocks can be defined by the developer and added to a reusable library.

The history of these block diagram models is derived from engineering areas such as Feedback Control Theory and Signal Processing. Block diagrams are a natural modelling framework for control engineers to design both digital and analogue circuits, by sampling sensors at regular time intervals, performing signal processing computations on their values, and outputting values often using complex mathematical formulae. Data is continuously subject to the same transformation as well. Block diagrams are easily amenable to simulation, and the virtual experimentation is a powerful and flexible tool for rapid prototyping which can be extended with specialised control blocks (like the basic constructs of a functional programming language) or blocks to support static analysis.

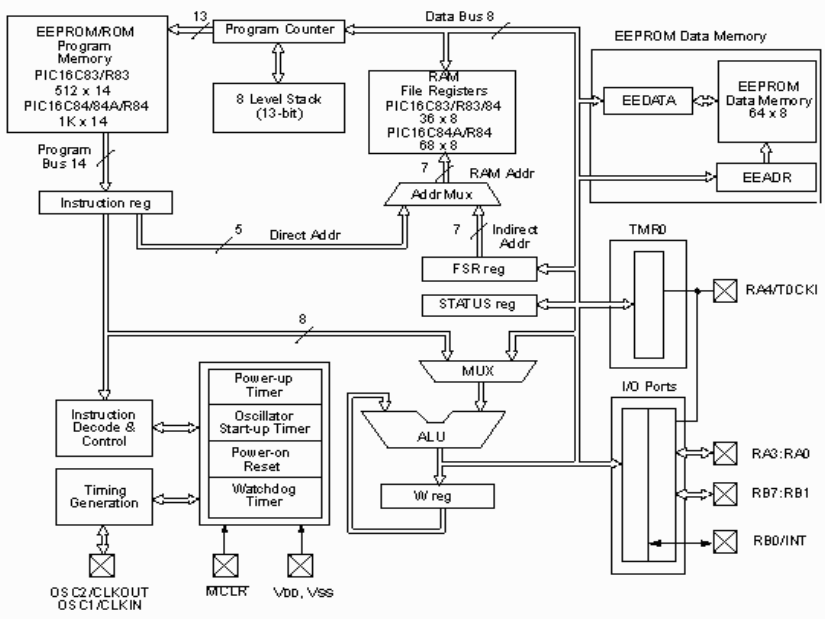


Figure 14-2: Block Diagram Example

14.3.1.2. Simulations for block diagram designs

This visual notation must be supported by specific tools to edit the diagram, simulate the system during a specific simulation time span, display the outputs over time, debug designs... a block diagram represents the instantaneous behaviour of a dynamic system. Determining a system's behaviour over time thus entails repeatedly solving the model at intervals, called time steps, from the start of the time span to the end of the time span. The process of solving a model at successive time steps is referred to as simulating the system that the model represents.

The tool to simulate the model can support fixed- and/or variable-step solvers. The accuracy of each simulated step depends on the size of the intervals between time steps. The user can specify the size of the time step in the case of fixed-step solvers, or the solver can automatically determine the step size in the case of variable-step solvers. To minimize the computation workload, the variable-step solver chooses the largest step size consistent with achieving an overall level of precision specified by the user for the most rapidly changing model state. This ensures that all model states are computed to the accuracy specified by the user. Simulating each step requires the use of numerical methods called ODE solvers. These various methods trade computational accuracy for computational workload.

It is important that these characteristics of the simulations are properly understood by the designers, to be aware of both the advantages and limitations of this approach. This is required first to be able to properly validate the model, and to avoid placing too much confidence on the results of the simulations. Otherwise, an inadequate validation would lead to the introduction of Unintended Functions.

14.3.1.3. Block diagram implementations

Each toolsuite implements its own notation, so actually there are many variants of the block diagram languages. Examples of tools that employ block diagrams include:

- Statemate *ActivityCharts*, by I-Logix (now part of IBM)
- SAO, an in-house development tool by Aérospatiale (now EADS/Airbus)
- SCADE *Data Flow* diagrams, by Esterel Technologies
- LabVIEW, by National Instruments

- BEACON *Signal Flow* notation, by Applied Dynamics International
- Matlab's Simulink toolbox, by The Mathworks

Note that each notation is incompatible between different tools, however some conversion utilities exist for the most extended notations, like Simulink. The changes are not just limited to the graphical representation, but also to the formality of the notation definition, the number of features, properties of the solvers... Each tool is aimed at specific markets, and thus the goals and properties of each notation are different. Even the basic blocks provided in the standard libraries could be similar, but the configuration options of each block can be very different. We'll first highlight the features of each toolset, analysing its adequacy for the development of safety-critical software.

Statemate is a modelling tool originally developed by I-Logix in late 1980s (acquired in 2006 by Telelogic, company which was two years later acquired by IBM) for the specification, analysis, design and documentation of reactive systems. It offers three distinct formalisms: *ModuleCharts*, for physical structure decomposition of the system; *StateCharts*, to specify the temporal behaviour and control relations of a system in a state-oriented model (a state diagram notation, see section 14.3.2 State diagrams); and *ActivityCharts*, for functional decomposition and information flow, an early block diagram notation were the function blocks were connected both by data-flow and control-flow arrows. Statemate was used in the fields of aerospace, communications, electronics, and transportation, although is being replaced by IBM Rhapsody. Therefore, its toolset is expected to enter in maintenance mode, being thus in disadvantage with other notations actively improved.

SAO (*Specification Assistée par Ordinateur*, Computer Aided Specification) is a visual programming notation, specifically a data-flow oriented formalism. SAO specifications are composed by a set of graphical sheets which describe transformational functions through the data flow model. The SAO environment used a symbol library and simple assembly rules. Each symbol of the library has an associated algorithm to compute its output values. The SAO functions communicate through discrete data exchange. SAO was created by Aérospatiale during early 1980s for the development of the fly-by-wire and flight control of the Airbus A320, technology that was later integrated into the product now called SCADE, and is thus interesting just from an historical point of view.

LabVIEW is a commercial package to process and measure laboratory equipment. LabVIEW's graphical notation is called *G*, a dataflow programming language where the engineer connects nodes by drawing wires. Its first version is from 1986, and has commercial acceptance for factory automation due to its focus on data acquisition and instrument control, but its usage in the aviation industry is much more limited.

BEACON is a modelling tool by Applied Dynamics International released in the 1990s, and has been used for the development of airborne embedded software for years. It provides both the *Signal Flow* and *Control Flow* notations (see section 14.3.2 State diagrams), and converted the designed diagrams into Ada, C, or Fortran source code. However, it has been recently discontinued, and currently just the Simulink code generator is being marketed.

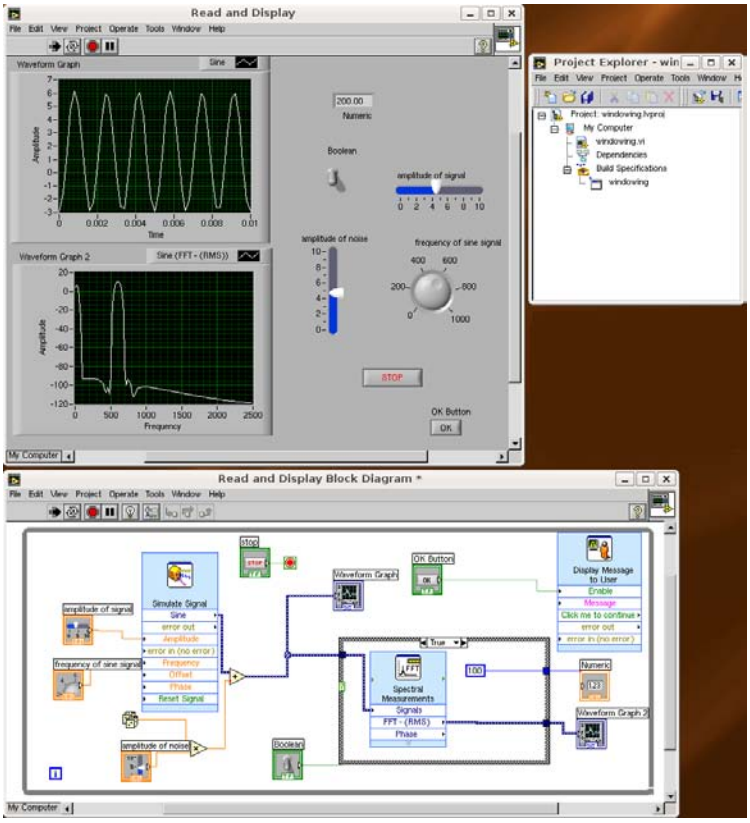


Figure 14-3: LabVIEW 8.2

Now, in the next subsections, let's analyse in detail the *de-facto* industry standards for Model-Based Development: Simulink and SCADE.

14.3.1.4. Simulink notation

Simulink is a widely-used Matlab toolbox, featuring block diagrams for the design and simulation of continuous reactive systems. The Simulink notation is a complex language (its User's Guide has nearly 1900 pages [RD.28]) with many features lacking any formal definition. Its documentation describes the semantics in informal operational terms supported by many examples, but the actual definition of the language is the "simulation semantics" given by its behaviour when simulated in the Matlab environment. This is a possible source of Unintended Functions, however researchers have formalized some subsets of the Simulink notation, defining proper syntax and semantics given a set of language features [RD.11]. A Simulink diagram is composed by blocks, signals (which represent mathematic entities and not physical connections), and data store objects (state variables to be shared among different blocks). However, data stores difficult readability and maintenance, being thus one another source of Unintended Functions, so its use is forbidden or restricted in the standard modelling guidelines [RD.12].

Blocks can be virtual (just a container to organise a diagram) or non-virtual (having an active role in the model), where a subsystem (group) of blocks can either be virtual (like a macro in the C programming language), with no impact in the execution whatsoever, or atomic where its blocks execute as a single unit. This non-obvious feature introduces differences in the behaviour of a model, so the engineers must be aware of their properties in the simulation to avoid introducing Unintended Functions.

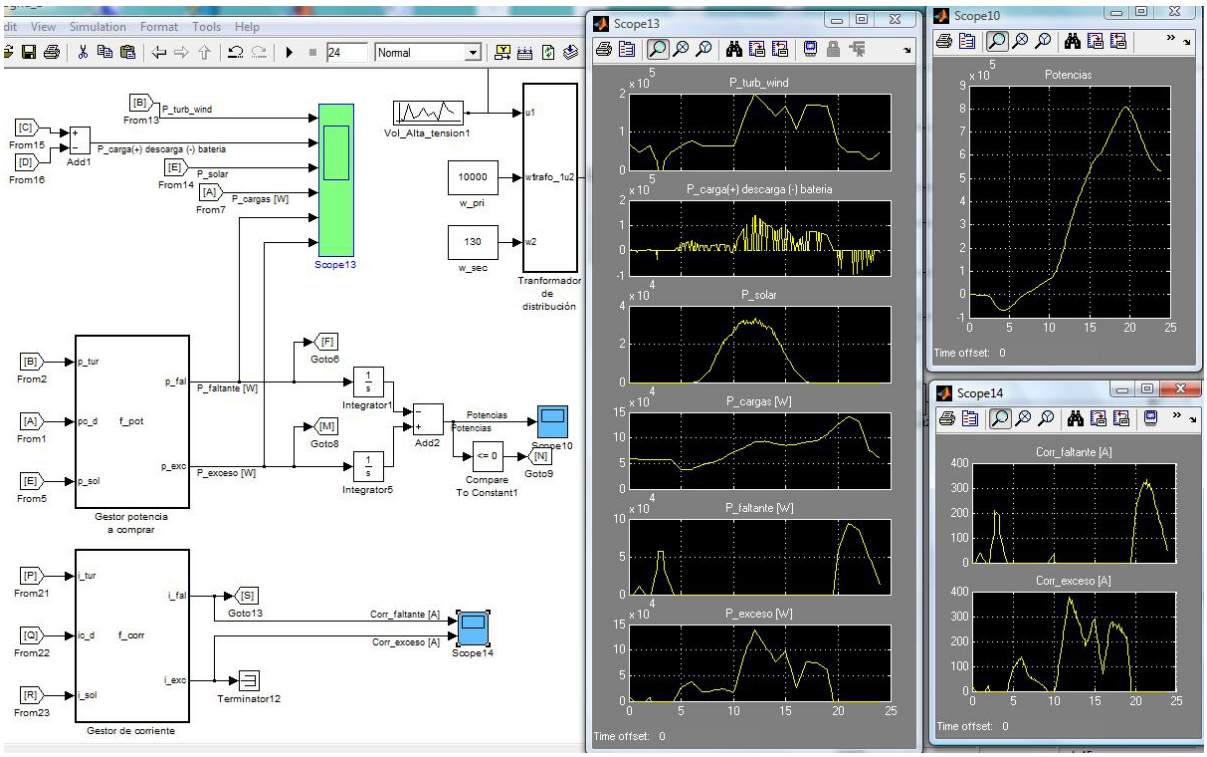


Figure 14-4: Simulink

Several additional tools for algorithm development, data analysis, data visualization, and numerical computation are provided, both by The MathWorks and by third parties. For example, the Verification & Validation toolbox provides requirements traceability (interfacing with DOORS or Office documents), model-coverage analysis, or Model Advisor (to check modelling guidelines, including ED-12B / DO-178B inspired guidelines). These toolbox demonstrates that The MathWorks has clear intentions to heavily pushing Simulink for the development of safety-critical systems, including airborne software. However, many rough edges still need to be addressed.

For example, executable code is usually generated from a Simulink model using the Real-Time Workshop add-on. This generator is not qualified due to the need to support a full featured model, including blocks with a non-deterministic behaviour, like the Merge block. Although the production of wrong code is rare, and when a problem appears the error is nearly always in the Simulink model and not the generated code, this code generator is thus not appropriate at all for the development of safety-critical code. Additionally, the behaviour of the code generated can be different than the behaviour of the simulated model if there are errors present in the model. This is an important consideration since the simulations are very slow especially for computational-expensive tests, so a common technique is to generate C code to run a battery of tests instead of running the simulations directly from the model.

There are specialised third-party code generators aimed at embedded / critical source code, like BEACON for Simulink, or dSpace Targetlink, which require that models adhere to a strict subset of the language (because many standard blocks are not supported, like matrix operations). Thus, models must be designed from the start for a specific code generator, and it is not usually feasible to modify an existing model to use another code generator (as usual when trying to certifying existing source code to high-integrity levels). These code generators are more suited for the development of airborne software.

Simulations on Simulink

An advantage of Simulink is that it can be simulated with fixed or variable-step solvers, allowing both the control system and the plant model (for example, the airframe) to be modelled within the same diagram. Many different solvers are provided by Simulink. Besides fixed- and variable-step solvers, Simulink solvers can also

be continuous or discrete (in the case all the blocks and subsystems of a model are discrete), and continuous solvers can either be explicit or implicit (for solving non-stiff or stiff problems, i.e. with extremely different time scales). In addition, matrix and vector operations are supported (backed by the underlying Matlab engine) while in other modelling tools must be codified by the user. Standard blocks are provided to solve and execute Matlab equations within Simulink diagrams, or for interfacing with general-purpose programming languages like Ada, C or Fortran.

The sample time is a key parameter of blocks, indicating when the outputs will be produced and its internal state will be updated during the simulation. Each block can be configured with a specific sample time parameter, while other blocks have an implicit sample time (like an Integrator, with sample time 0). Simulink also offers a wide-variety of sample times, which can be discrete, continuous, inherited, constant, variable, triggered, asynchronous, or fixed in minor simulation step. In addition, sample times are either block-based (same rate for input and output ports) or port-based (inputs and output execute at different rates). Even if this allows a better modelling of the environment for example, son many options can result in hard-to-understand models and simulations, which of course is an undesirable feature for the development of safety-critical software.

14.3.1.5. SCADE Data Flow diagrams

The SCADE environment is similar to Simulink, but it is not a general-purpose toolset but focused on safety-critical systems (originally developed for the design of aircraft systems), as explained before. Marketed by Esterel Technologies, two different products are offered: SCADE Suite for the avionics industry and SCADE Drive for the automotive industry. The SCADE suite has been used to develop Software for equipments that have been certified to ED-12B/DO-178B level A for Military and aerospace industries, certified to IEC 61508 and EN 50128 at SIL 3 (Rail transportation), and IEC 60880 compliant (Nuclear Energy).

The **SCADE data flow** notation is aimed at continuous control, and the diagrams are composed by blocks connected by data flows. As SCADE is aimed for the development of safety-critical systems, all its features are designed with determinism in mind: The blocks provided have unambiguous mathematical representation, and only fixed-step simulation is supported; Data flows have no functional dependency (timing and causality checks), are computed concurrently, and may carry discrete or continuous values; Initialisation of flows must be explicit; Scade blocks are hierarchical too, and a block can be composed of other blocks connected by local flows, but this hierarchy does not add any evaluation rules, is just an organization facility and the contents of a subsystem are replaced like a macro. All these characteristics are this well-suited for avoiding the introduction of Unintended Functions.

So even if at first sight SCADE can be similar to other block diagrams notations (but providing less useful blocks), many of the key differences are under the hood. The Scade notation can be either graphical or textual, and has precise formal semantics to allow formal analysis of SCADE models. Based on the Lustre programming language, the Scade textual language is a deterministic, declarative, and structured data-flow programming language. This strongly typed language has predefined and user-defined types (including real types), and provides temporal operators to access values from the past (like the fby operator, “followed by”). Scade textual operators are actually nodes / blocks in the graphical notation, while data flows are represented as arrows. Synchronous languages act as if programs react instantaneously to external events, resulting in deterministic programs from the temporal point of view. Based in a functional model, complex side effects are avoided and flows are parallel in nature. Lustre also provides extensive assertion constructions, being able to express safety properties in the design. This makes SCADE appropriate for the development of code of the highest criticality levels.

The SCADE Suite also provides a library of correct basic blocks, including limiters, edge detection, derivative, filters, flip-flops, vector operations… In addition, the SCADE Suite has many modules and kits for specific tasks of the life-cycle of safety-critical code development. For example, the SCADE Requirements Management Gateway links the model with specific requirements tools (DOORS, RequisitePro), documentation (PDF, Office documents), other design tools (Matlab, LabVIEW…) and source code files. In addition, model gateways to other notations allow translating to SCADE existing algorithms in SysML/UML, Simulink, or to translate SCADE models for testing into the LabVIEW environment. Finally, the SCADE Design Verifier (DV) is a model checker of safety properties based on formal methods, based on a powerful engine that can prove that a design is safe with respect to its requirements.

14.3.1.6. Conclusions about block diagrams

The block diagrams family of formalisms is a powerful notation, natural and easy to employ by the engineers. They allow fast prototyping and testing of the software design (as many blocks are already available), including modelling the interactions environment, which allow detecting and fixing design problems before the implementation phase. These simulations even allow detecting errors in the requirements early, and to increase the confidence in the completeness of the requirements, which usually reduces some types of Unintended Functions.

In addition, the tools for requirements traceability and model-coverage analysis further detects many types of Unintended Functions that was usually not detected through manual reviews. And as these modelled designs tend to be much more detailed than traditional designs, this allows the automatic generation of source code which is faster and far less error-prone than manually coding a design. And even if the source code is manually written like in the traditional development process, as the design is more detailed and refined thus its intent is also clearer for the engineer writing the source code.

However, block diagrams are less adequate to represent discrete control algorithms, like system modes, which sometimes leads to difficult to understand diagrams. In addition, while some notations like SCADE are appropriate for code development to the highest criticality levels, other notations contain features with subtle semantics, or that could result in surprising behaviours, like the difference between virtual and atomic subsystems in Simulink. And even if a model is considered valid after the battery of simulations is successful, the results could be different when executed on the target platform due to different reasons, for example non-formal notation semantics or toolset's bugs. Finally, these powerful and flexible toolsets are so highly configurable that this can be the source of Unintended Functions. Like with every development tool the user needs adequate training and previous experience, however the configuration of these modelling tools have much more implications on the obtained results than with other traditional development tools. Model simulations could be invalidated if wrong parameters are chosen (e.g. use of a configuration that must be later changed because is not adequate for later code generation). Or the configuration of the code generator could lead to sub-optimal sources, not adequate for the needed certification level or target platform. Therefore, designers must be aware of both the advantages and disadvantages of this approach to avoid placing too much confidence on the results of the simulations, otherwise some Unintended Functions would be introduced due to the inadequate validation of the model.

14.3.2. STATE DIAGRAMS

14.3.2.1. Introduction

While block diagrams are a natural notation for the design of circuits, the **state diagram** family of formalisms is closer to the notation used by system and software engineers for modelling the behaviour of an embedded system. A state diagram is also a representation of a reactive (event-driven) system, but now the system is composed by a set of states (also called modes), input events, edges (usually called transitions) between states, and processing actions. The system makes a transition from one state to another if a guard condition holds, which could have processing actions associated with the mode and / or transition.

In addition, these notations usually allow having hierarchical diagrams, where a state is actually composed of sub-states and local transitions. This hierarchy of states facility is mainly for organisational purposes, but could also simplify the model in case all the substates have a transition to the same external state for a specific event. Another feature of state diagrams usually incorporated into these notations is the history mechanism: a variable records the substate being executed, in case the designer wants to resume execution to the last substate.

In the past designers used **truth tables** to specify the relationships between the states and events / actions, but nowadays the most employed notations are graphical, including visual flowchart notations for the processing actions and the history flag. However, many tools also allow including truth tables in the model, mixing both state diagrams, truth tables, and even block diagrams.

State diagrams allow a discrete control changing behaviour according to external events originating either from discrete sensors and user inputs or from internal program events, e.g. value threshold detection. Discrete control is characteristic of modal human machine interfaces, alarm handling, complex functioning mode

handling, or communication protocols. State diagrams are used to describe data flows, which can be composed for Finite State Machines (FSM) and they allow easily simulation, statically analysing and a later derivation into a software programming language.

Finite state machines have been very extensively studied in the last 50 years, and their theory is well understood. However, in practice, they have not been adequate even for medium-size applications, since their size and complexity tend to explode very rapidly. Large applications contain cooperating continuous and discrete control parts and some properties should be under control: like **concurrency** and a proof **deterministic** behaviour. Determinism is a key requirement of most embedded applications. A system is deterministic if it always reacts in the same way to the same inputs occurring with the same timing. On the contrary, a non-deterministic system can react in different ways to the same inputs, actual reaction depending on internal choices or computation timings. Determinism is a must to develop flight control systems for a plane: internal computation timings should not interfere with the flight controls algorithms. The flight controls system should always obey the pilot commands and produce an identical output when identical inputs are generated in an identical context.. The same applies to man/machine interface and alarm handling.

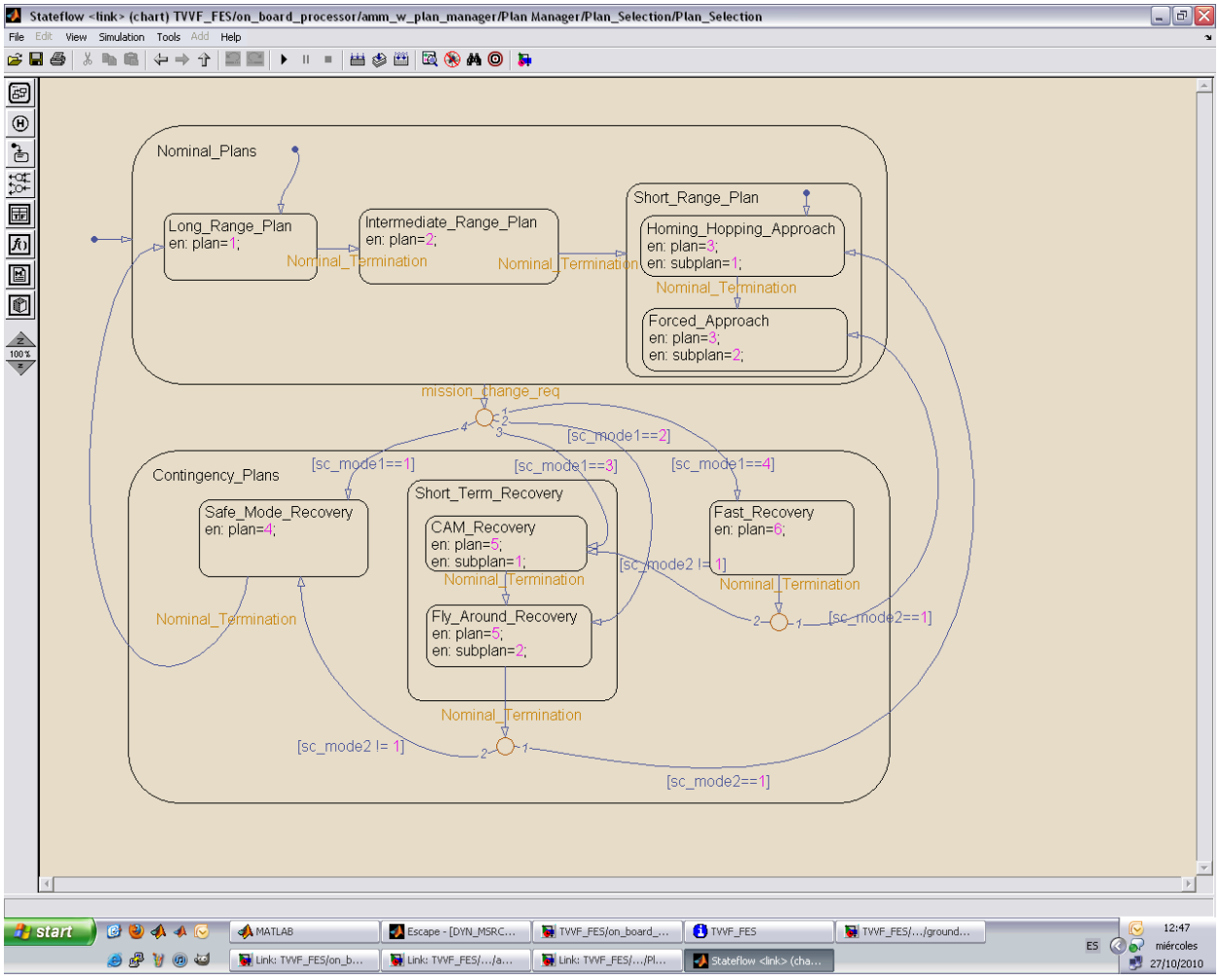


Figure 14-5: State diagram example

14.3.2.2. State diagrams implementations

Different commercial toolsets employ state diagrams:

- Statemate *StateCharts*, by I-Logic (now IBM)

- SCADE *SSM* notation (Safe State Machines), by Esterel Technologies
- BEACON *Control Flow* notation, by Applied Dynamics International
- Rational Rhapsody *UML state machine* notation, by IBM
- Matlab's Stateflow, by The MathWorks

It is worth noting that many of these tools were already covered in section 14.3.1.3, as commercial development environments usually support both block diagrams and state diagrams. As happened with the block diagram implementations, each development tool implements its own variation of the state diagram formalism, event if based in the same underlying theoretical foundation. There are many variants of the state diagram formalism, depending on the targeted marked the definition could be more formal or provide a different number of features.

StateCharts is a successful family of state diagrams. This visual formalism, defined by Harel during the 1980s [RD.26], extended the finite state machines and state diagrams of the time with hierarchical states, concurrent processing, and broadcast communication to concurrent elements. Ordinary state transition diagrams were flat, unstructured, and inherently sequential, causing state explosion when modelling systems with parallel threads of control. In Statecharts each state consists of a hierarchy of possibly concurrent states, and communicating transitions. Several slight variations of statecharts exist in the literature [RD.29] and in commercial packages, like I-Logix **Statemate** in the late 1980s, which was one of the first toolsets to implement this state diagrams notation as well as other formalisms (please, refer to section 14.3.1.3 for more information about this tool).

Nowadays Harel's statecharts are gaining popularity due to its standardization as *UML state machines* as part of the UML language. UML state machines provide some further extensions to the original Harel's definitions—like orthogonal states—and is supported by tools like IBM **Rhapsody**, which is used in some aerospace projects, although with even further slight differences [RD.30][RD.31]These packages allow testing the designs thanks to the animation of diagrams, and to generate source code.

Besides statecharts, the most widely-used state diagram notations used for the development of safety-critical systems are those from SCADE and Matlab.

14.3.2.3. Stateflow notation

Stateflow is the application inside the Simulink toolbox to model discrete reactive systems within the Matlab environment. Stateflow is another variant of the statecharts defined by Harel. This Matlab chart notation is a complex language with numerous, complicated, and often overlapping features lacking any formal definition. Scattered over the 1400 pages of its User's Guide [RD.28]the documentation describes the semantics in informal operational terms supported by many examples, but the actual definition of the language is the "simulation semantics" given by its behaviour when simulated in the Matlab environment. Therefore this can be a source of Unintended Functions, however formal definitions of Stateflow subsets exist [RD.14], which allow performing formal analysis in models that adhere to this language profiles.

Stateflow charts are introduced in Simulink models by means of the 'stateflow' block. Stateflow events can come from and propagate to other Simulink blocks or be local to a Stateflow chart, as well as share data objects with other Simulink blocks (a feature which is discouraged in different Simulink modelling guidelines, as explained in section 14.3.1.4, and thus highly undesirable for the development of safety critical systems). A Stateflow chart is composed of both graphical and not graphical objects, some of them not appropriate for the development of safety-critical software:

- Graphical objects:
  - *States*, representing the system modes. States can be active or inactive at any given time.
  - *Transitions*, a connection between states. Can be guarded by condition, and have associated processing actions.
  - *Connective junctions*, which allow different destination paths for a transition depending on the condition.

- *History junctions*, a flag in superstates to indicate that the next substate to execute depends on historical information about past active substates.
- *Functions*, a reusable graphical function composed of flowchart transitions and junctions.

- Non-graphical objects:
  - *Events*, which trigger transitions
  - *Data*, variables to store and share information with other Stateflow levels or Simulink blocks, and a potential source of Unintended Functions.

Stateflow charts are hierarchical because states can contain any of the above elements. A superstate can be composed of mutually exclusive substates (just one substate is active) or parallel substates (all states at the same level are active at the same time). Processing actions are contained within states as flowcharts, described as transitions between junctions. In contrast to states, junctions are exited instantaneously when entered, and the flowchart executes until a terminal junction (one without outgoing transitions) is reached, or all paths have failed. Flowcharts can also be guarded by event, and backtracking occurs if a wrong path is tried.

Some other Stateflow characteristics are a barrier for the adoption of this notation in the aviation industry. One surprising Stateflow feature is that the activation order is determined by the position of the components on the diagram, where states are ordered top to bottom and then left to right. This is an error-prone feature, because graphically reordering a Stateflow model can change its behaviour. Also, a Stateflow program can fail with a runtime exception for any of several reasons [RD.14]. A way to avoid these exceptions is to adhere to a subset of the language, using modelling guidelines [RD.10][RD.12] to restrict the notation to a safe subset. But these guidelines have no more formal basis than the language itself and are based on experience.

Simulink also provides a truth table block, which can be created with the Truth Table editor. Stateflow diagrams have many other advanced features, like handling vectors, matrices, and variable-size data. However, Stateflow is less widely used than Simulink, and the code generator is less mature.

14.3.2.4. SCADE Safe State Machines

Besides the data flow diagrams, the SCADE Suite also supports the **Safe State Machines** (**SSM**) notation for discrete control of safety-critical systems. Classic state machines are not adequate even for medium-size applications, since their size and complexity tend to explode very rapidly. However, SSM are hierarchical state machines (like Statecharts), evolved from the Esterel programming language [RD.32] and the SyncCharts state machine model [RD.33]. SSM have proved to be scalable in large avionics systems.

As other state machines, the mode transitions in a SSM are activated by an event, and have actions associated. SSMs are hierarchical and concurrent. States can be either simple states or macro states, themselves recursively containing a full SSM or a concurrent product of SSMs. When a macro state is active, so are the SSMs it contains. When a macro state is exited by taking a transition out of its boundary, the macro state is exited and all the active SSMs it contains are pre-empted whichever state they were in. Concurrent state machines communicate by exchanging signals, which may be scoped to the macro state that contains them.

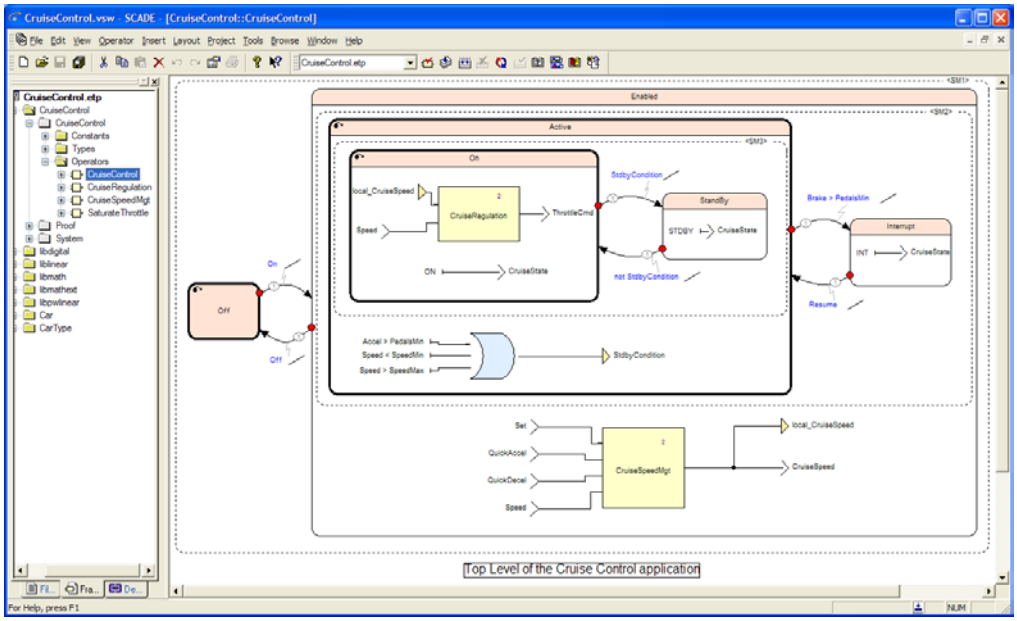


Figure 14-6: SCADE Safe State Machine

The definition of SSMs carefully forbids dubious constructs found in other hierarchical state machine formalisms: transitions crossing macro state boundaries, transitions that can be taken halfway and then backtracked, etc. These are non-modular, semantically ill-defined, and very hard to figure out, hence inappropriate for safety-critical designs. Their use is usually not recommended by methodological guidelines.

Large applications contain cooperating continuous and discrete control parts. The SCADE Suite makes it possible to couple both data flow and state machine styles. Most often, one includes SSMs into block-diagrams design to compute and propagate functioning modes. Then, the discrete signals to which a SSM reacts and which it sends back are simply transformed back-and-forth into Boolean data flows in the block diagram. The computation models are fully compatible.

The SCADE Gateway for Simulink allows in the conversion of importing Stateflow charts into SCADE Safe State Machines.

14.3.2.5. Conclusions about state diagrams

State diagrams are a family of notations traditionally used in computer science, and its foundational theory has defined semantics. It is worth noting that these diagrams can be used also for requirements specification, and thus the use of the same notation reduces the possibility of introducing errors in the design process. But actually few of the notations have formalized semantics, and thus many of the toolsets are not adequate for the development of safety-critical code.

State diagrams are limited just to control logic, and not appropriate for numeric algorithms, thus this notation is usually mixed with block diagrams. Complex state diagrams can be usually very difficult to understand, at least the detailed behaviours and fine implications, especially when using more advances features like parallel states, history, aborted transitions… (even if those features were mainly added to reduce the complexity of the diagrams). Engineers sometimes feel that this notation is more unnatural and less powerful than block diagrams, and tend to use block notations to design everything in the system.

14.4. CONCLUSIONS

Many different notations for model-based development are available, including those originally designed for airborne software. These notations are focused on different stages of the software-development life cycle, and this report has focused on specialised notations for Formalized Requirements and Formalized Designs. These formalisms can potentially improve the safety of this kind of systems for different reasons:

- Formalized Designs and Formalized Requirements are more detailed than traditional ones, and the ambiguity is reduced.
- Dynamic verification allows a better understanding of the system, and the software can be developed based on calculation and prediction.
- Static consistency checks, dynamic simulations, or model-coverage analysis tools at model level allow finding much more errors at the requirements- and design-level, very early in the process.

More effort is devoted to the requirements and design phases, and this is especially important as experience shows that many software safety problems are related with errors in the requirements. Furthermore, these tools can also reduce the development time thanks to the existence of many already available blocks, time and costs due to the automatic code and test generation, and also due to the already mentioned property of reduction of bugs that needs to be fixed during validation.

However, even if these model formalisms can reduce some types of software flaws (like those caused by ambiguity in the specification), due to their nature it is also possible to introduce some kinds of Unintended Functions. For example, not all requirements can be modelled (due to the specific goals of these notations and the generic intent of some system requirements) nor all of them should be formalized (due to the high effort of writing these specifications), which could lead to the introduction of Unintended Functions due to the interactions of the Formalized Requirements with the not modelled ones, or due to a wrong selection of the set of requirements modelled. Or some of the toolsets are still not widely used, or not focused for the development of high-criticality levels.

In addition, some notations are adequate for describing discrete systems but not continuous ones, which sometimes lead to difficult-to-understand diagrams, and notations contain features with subtle semantics, which could result in surprising behaviours. Too much confidence could be placed into the executed simulations, but the results could be different when executed on the target platform due to different reasons, for example non-formal notation semantics or toolset's bugs.

Finally, these powerful and flexible toolsets are so highly configurable that this can be the source of Unintended Functions. Like with every development tool the user needs adequate training and previous experience, however the configuration of these modelling tools have much more implications on the obtained results than with other traditional development tools. Model simulations could be invalidated if wrong parameters are chosen (e.g. use of a configuration that must be later changed because is not adequate for later code generation). Or the configuration of the code generator could lead to sub-optimal sources, not adequate for the needed certification level or target platform.

The following table is a summary of the analysed notations, both for Formalized Requirements and Formalized Designs. The table is organized in six columns, described below:

- *Toolset*: Tool under study
- *Notation*: Each tool can support several different notation variants, and the same tool can support multiple formalisms.
- *Life Cycle*: The third table column indicates to which stage within the development life-cycle the notation focuses (either 'Requirements' or 'Development').
- Type: States the category of the formalism.
- *Formal Method*: Indicates whether the notation, or a subset of it, allows formal methods (i.e. formalized notation plus formalized analyses), specifying also the tool if separate translation is required. If this field indicates "Research" means that the tool doesn't support formal method by default, but There are some investigations that proves the support for formal methods for this tool (or a subset of it). These research information is described all along Section 14.3.

- *Advantages / disadvantages*: Highlights the important considerations about its use for airborne software.

Tool	Notation	Life cycle	Type	Formal method	Advantages / disadvantages
BEACON	Signal Flow	Design	Block diagram	No	Advantages: ✓ Used in aviation projects ✓ Code generator for embedded systems ✓ Mix with state diagrams ✓ Conversion to Simulink Disadvantages: ⊗ Out of the market
BEACON	Control Flow	Design	State diagram	No	Advantages: ✓ Used in aviation projects ✓ Code generator for embedded systems ✓ Mix with block diagrams ✓ Conversion to Stateflow Disadvantages: ⊗ Out of the market
LabVIEW	G language	Design	Block diagram	No	Advantages: ✓ Wide user community ✓ Solid tool support ✓ Libraries available Disadvantages: ⊗ For automotive market
Matlab	Simulink	Design	Block diagram	Research	Advantages: ✓ Industry standard ✓ Very solid tool support ✓ Academia support ✓ Specialised code generators ✓ Translators to other notations Disadvantages: ⊗ Not formalized semantics

Tool	Notation	Life cycle	Type	Formal method	Advantages / disadvantages
Matlab	Stateflow	Design	State diagram	Research (SAL[1])	Advantages: ✓ Mix with Simulink ✓ Academia support Disadvantages: ⊗ Not formalized semantics
Nimbus	RSML	Requirements	Synch language	Yes	Advantages: ✓ Used in avionics projects ✓ Model execution Disadvantages: ⊗ Research tool ⊗ Overcome by RSML^{-e}
Nimbus	RSML^{-e}	Requirements	Synch language	Yes (NuSMV[2] & PVS[3])	Advantages: ✓ Used in aviation projects ✓ Consistency and completeness tools ✓ Safety & liveness checking ✓ Model execution ✓ Tests generation Disadvantages: ⊗ Research tool
Rhapsody	UML state machine	Design	State diagram	No	Advantages: ✓ UML is a standard notation ✓ Model executions Disadvantages: ⊗ Rhapsody state machines are not complete.
SCADE	Data Flow	Design	Block diagram	Yes	Advantages: ✓ Designed for airborne software ✓ Formalized semantics ✓ Deterministic behaviour ✓ Code generator qualified to ED-12 Level-A

[1] The Symbolic Analysis Laboratory (**SAL**) is an integrated formal verification environment.

[2] **NuSMV** is a software tool for the formal verification of finite state systems. It has been developed jointly by ITC-IRST and by Carnegie Mellon University.

[3] **PVS** (Prototype Verification System) is a mechanized environment for formal specification and verification.

Tool	Notation	Life cycle	Type	Formal method	Advantages / disadvantages
SCADE	Safe State Machines	Design	State diagram	Yes	Advantages: ✓ Mix with block diagrams ✓ Code generator qualified to ED-12 Level-A ⊗
SCRtool	SCR	Requirements	Tabular	Yes	Advantages: ✓ Used in aviation projects ✓ Dependency browser ✓ Consistency checker and model checker ✓ Simulator Disadvantages: ⊗ Medium tool support
SpecTRM	SpecTRM-RL	Requirements	Documented oriented	Yes	Advantages: ✓ Emphasis on safety-critical systems ✓ Simple to learn ✓ Ensures completeness ✓ Allows simulations Disadvantages: ⊗ Medium tool support ⊗ Difficult performing simulations
Statemate	ActivityCharts	Design	Block diagram	Research	Advantages: ✓ Used in airborne projects ✓ Mix with state diagrams ⊗
Statemate	StateCharts	Design	State diagram	Research	Advantages: ✓ Used in airborne projects ✓ Mix with block diagrams ⊗

Table 14-1: Summary of model formalisms

END OF DOCUMENT

EUROPEAN AVIATION SAFETY AGENCY
AGENCE EUROPÉENNE DE LA SÉCURITÉ AÉRIENNE
EUROPÄISCHE AGENTUR FÜR FLUGSICHERHEIT

Postal address
Postfach 101253
50452 Cologne
Germany

Visiting address
Ottoplatz 1
50679 Cologne
Germany

Tel +49_221_89990-000
Fax +49_221_89990-999
Mail info@easa.europa.eu
Web easa.europa.eu