

Re-use of Tests and Arguments for Assessing Dependable Mixed-criticality Systems

Carlos Fernando Nicolás Ramírez

Supervisors:

Dr. Goiuria Sagardui Mendieta

Mondragon Unibertsitatea

and

Dr. Peter Puschner

Vienna University of Technology



***A thesis submitted to Mondragon Unibertsitatea
for the degree of Doctor of Philosophy***

*Department of Electronics and Computer Science
Mondragon Goi Eskola Politeknikoa
Mondragon Unibertsitatea
Arrasate, June 2017*

Declaration

Hereby I declare, that this work is my original authorial work, which I have worked out by my own. All sources, figures and literature used during the elaboration of this work are properly cited and listed in complete reference to the due source.

Advisers: Dr. Goiuria Sagardui Mendieta, PhD and Dr. Peter Puschner, PhD.

Abstract

The safety assessment of mixed-criticality systems (MCS) is a challenging activity due to system heterogeneity, design constraints and increasing complexity. The foundation for MCSs is the integrated architecture paradigm, where a compact hardware comprises multiple execution platforms and communication interfaces to implement concurrent functions with different safety requirements. Besides a computing platform providing adequate isolation and fault tolerance mechanism, the development of an MCS application shall also comply with the guidelines defined by the safety standards. A way to lower the overall MCS certification cost is to adopt a platform-based design (PBD) development approach. PBD is a model-based development (MBD) approach, where separate models of logic, hardware and deployment support the analysis of the resulting system properties and behaviour. The PBD development of MCSs benefits from a composition of modular safety properties (e.g. modular safety cases), which support the derivation of mixed-criticality product lines.

The validation and verification (V&V) activities claim a substantial effort during the development of programmable electronics for safety-critical applications. As for the MCS dependability assessment, the purpose of the V&V is to provide evidences supporting the safety claims. The model-based development of MCSs adds more V&V tasks, because additional analysis (e.g., simulations) need to be carried out during the design phase. During the MCS integration phase, typically hardware-in-the-loop (HiL) plant simulators support the V&V campaigns, where test automation and fault-injection are the key to test repeatability and thorough exercise of the safety mechanisms.

This dissertation proposes several V&V artefacts re-use strategies to perform an early verification at system level for a distributed MCS, artefacts that later would be reused up to the final stages in the development process: a test code re-use to verify the fault-tolerance mechanisms on a functional model of the system combined with a non-intrusive software fault-injection, a model to X-in-the-loop (XiL) and code-to-XiL re-use to provide models of the plant and distributed embedded nodes suited to the HiL simulator, and finally, an argumentation framework to support the automated composition and staged completion of modular safety-cases for dependability assessment, in the context of the platform-based development of mixed-criticality systems relying on the DREAMS harmonized platform.



Resumen

La dificultad para evaluar la seguridad de los sistemas de criticidad mixta (SCM) aumenta con la heterogeneidad del sistema, las restricciones de diseño y una complejidad creciente. Los SCM adoptan el paradigma de arquitectura integrada, donde un hardware embebido compacto comprende múltiples plataformas de ejecución e interfaces de comunicación para implementar funciones concurrentes y con diferentes requisitos de seguridad. Además de una plataforma de computación que provea un aislamiento y mecanismos de tolerancia a fallos adecuados, el desarrollo de una aplicación SCM además debe cumplir con las directrices definidas por las normas de seguridad. Una forma de reducir el coste global de la certificación de un SCM es adoptar un enfoque de desarrollo basado en plataforma (DBP). DBP es un enfoque de desarrollo basado en modelos (DBM), en el que modelos separados de lógica, hardware y despliegue soportan el análisis de las propiedades y el comportamiento emergente del sistema diseñado. El desarrollo DBP de SCMs se beneficia de una composición modular de propiedades de seguridad (por ejemplo, casos de seguridad modulares), que facilitan la definición de líneas de productos de criticidad mixta.

Las actividades de verificación y validación (V&V) representan un esfuerzo sustancial durante el desarrollo de aplicaciones basadas en electrónica confiable. En la evaluación de la seguridad de un SCM el propósito de las actividades de V&V es obtener las evidencias que apoyen las aseveraciones de seguridad. El desarrollo basado en modelos de un SCM incrementa las tareas de V&V, porque permite realizar análisis adicionales (por ejemplo, simulaciones) durante la fase de diseño. En las campañas de pruebas de integración de un SCM habitualmente se emplean simuladores de planta *hardware-in-the-loop* (HiL), en donde la automatización de pruebas y la inyección de faltas son la clave para la repetitividad de las pruebas y para ejercitar completamente los mecanismos de tolerancia a fallos.

Esta tesis propone diversas estrategias de reutilización de artefactos de V&V para la verificación temprana de un MCS distribuido, artefactos que se emplearán en ulteriores fases del desarrollo: la reutilización de código de prueba para verificar los mecanismos de tolerancia a fallos sobre un modelo funcional del sistema combinado con una inyección de fallos de software no intrusiva, la reutilización de modelo a *X-in-the-loop* (XiL) y código a XiL para obtener modelos de planta y nodos distribuidos aptos para el simulador HiL y, finalmente, un marco de argumentación para la composición automatizada y la compleción escalonada de casos de seguridad modulares, en el contexto del desarrollo basado en plataformas de sistemas de criticidad mixta empleando la plataforma armonizada DREAMS.



Laburpena

Kritikotasun nahastuko sistemen segurtasun ebaluazioa jardura neketsua da beraien heterogeneotasuna dela eta. Sistema hauen oinarria arkitektura integratuen paradigmak datza, non hardware konpaktu batek exekuzio plataforma eta komunikazio interfaze ugari integratu ahal dituen segurtasun baldintza desberdineko funtzio konkurrenteak inplementatzeko. Konputazio plataformek isolamendu eta akatsen aurkako mekanismo egokiak emateaz gain, segurtasun arauak definituriko jarraibideak jarraitu behar dituzte kritikotasun mistodun aplikazioen garapenean. Sistema hauen zertifikazio prozesuaren kostua murrizteko aukera bat plataformetan oinarritutako garapenean (PBD) datza. Garapen planteamendu hau modeloetan oinarrituriko garapena da (MBD) non modeloaren logika, hardware eta garapen desberdinak sistemaren propietateen eta portaeraren aurka aztertzen diren. Kritikotasun mistodun sistemen PBD garapenak etekina ateratzen dio moduluetan oinarrituriko segurtasun propietateei, adibidez: segurtasun kasu modularrak (MSC). Modulu hauek kritikotasun mistodun produktu-lerroak ere hartzen dituzte kontutan.

Berifikazio eta balioztatze (V&V) jarduerak esfortzu kontsideragarria eskatzen dute segurtasun-kritikokoetarako elektronika programagarrien garapenean. Kritikotasun mistodun sistemen konfiantzaren ebaluazioaren eta V&V jardueren helburua segurtasun eskariak jasotzen dituzten frogak proportzionatzea da. Kritikotasun mistodun sistemen modelo bidezko garapenek zeregin gehigarriak atxikitzen dizkio V&V jarduerari, fase honetan analisi gehigarriak (hots, simulazioak) zehazten direlako. Bestalde, kritikotasun mistodun sistemen integrazio fasean, *hardware-in-the-loop* (HiL) simulazio plantek V&V inizatibak sostengatzen dituzte non testen automatizazioan eta akatsen txertaketan funtsezko jarduerak diren. Jardura hauek frogan errepikapena eta segurtasun mekanismoak egiaztatzea ahalbidetzen dute.

Tesi honek V&V artefaktuen berrerabilpenerako estrategiak proposatzen ditu, kritikotasun mistodun sistemen egiaztatze azkarrerako sistema mailan eta garapen prozesuko azken faseetarako erabili daitezkeenak. Esate baterako, test kodearen berrerabilpena akats aurkako mekanismoak egiaztatzeke, modelotik *X-in-the-loop* (XiL)-ra eta kodetik XiL-rako konbertsioa HiL simulaziorako eta argumentazio egitura bat DREAMS Europear proiektuan definituriko arkitektura estiloan oinarrituriko segurtasun kasu modularrak automatikoki eta gradualki sortzeko.



Acknowledgements

This thesis was carried out during my employment at IK4-Ikerlan Research Center, starting within the former Electronics Department that later became the Dependable Embedded Systems Department. There are many people who contributed to the success of this thesis.

First and foremost, I would like to express my sincere gratitude to my advisor Dr. Goiria Sagardui for the continuous support and guidance of my PhD, as well as for her patience and motivation. My sincere thanks also go to Dr. Peter Puschner from Vienna University of Technology for supervising this work, and to Dr. Christian El Salloum for his advice, seminal ideas and fruitful discussions during the kick-off stage.

Many thanks to IK4-Ikerlan, and specially to Dr. Jon Perez, Dr. Mikel Azkarate-askasua and Mr. Antonio Perez for offering me the opportunity, resources and invaluable support to develop the thesis. I have also to render thanks to all the workmates who provided me excellent help for this thesis, particularly I would like to thank those with whom I had a hand-to-hand collaboration: Iban Ayestaran, Asier Larrucea, Fernando Eizagirre, Gaizka Bellido and Irune Agirre.

I specially owe my grateful to the people of Orona Elevator Innovation Centre, who trusted my ideas and provided financial support to bring them to reality. I also want to thank to the partners with who I have worked in the European research project DREAMS for their support and cooperation.

This thesis is dedicated to my wife Toñi, and our daughters Raquel and Silvia, for their understanding and warm encouragement shown during this long process. Their optimism and continuous support helped me to overcome my moments of frustration and ultimately made this thesis possible.

Preface

During my professional career I teamed in several projects related to industrial computing. After graduating, I initially dealt with computer-based applications, then at some point, I turned to the challenge of developing embedded devices: programming processors with limited resources, constrained budgets, almost always increasing functional requirements, scarce prototypes, while racing against the clock (learning the meaning of *time-to-market*) and, at the same time, trying to achieve the quality objectives, was not a task for the heart-fainted. I soon realized the usefulness of model-based approaches and virtualizations to alleviate some of the development shortcomings, and thus I began to use it routinely.

The motivation for this research raised when I joined a project team to develop an embedded over-speed governor with a simple safety function: to brake an elevator car in case of unintended movements. This device was amenable to safety-certification, and the project manager -who shortly afterwards would become my boss- assigned me to the verification, validation and test activities. He asked me three direct questions: what infrastructure would I request, how much budget had he to allocate, and how much time should be devoted to complete the tasks. I felt in dismay: the preliminary specifications were far from complete, contained many ambiguous statements, and some validation criteria stated that the system should work in a context of continuous ranges for operational parameters. To make matters worse, there was no indication as how the system would be de-composed, and only partial interface specifications and rough overall system analysis were available. Even if I had conceived a validation plan that included some test specifications, it would have been impossible for me to define a sensible integration test, let alone estimate the cost of VVT in terms of time, as well as material and human resources. I was unable to answer any of the questions, yet I reflected about the obstacles shading the answers.

This work pretends to bring some light to the crucial task of assessing a safety embedded device in a cost-effective way, by means of a re-use strategy involving an interchangeable set of artefacts. Let's hope the reader finds it useful.

Contents

1	Introduction	1
1.1	Motivation	4
1.2	Contribution	8
1.2.1	Technical Contributions	9
1.2.2	Collaborations	10
1.2.3	Publications	10
1.3	Outline	12
1.4	Support	13
I	State of the Art	15
2	Background	17
2.1	The Time-Triggered Architecture (TTA)	17
2.2	Modelling Languages	19
2.2.1	Simulink	20
2.2.2	LabVIEW	21
2.2.3	SysML	21
2.2.4	Architecture Analysis and Design Language	21
2.2.5	Ptolemy	22
2.2.6	SystemC	24
2.2.7	Languages for Modelling Arguments	29
2.3	Safety Systems	30
2.3.1	Safety Integrity Level (SIL)	30
2.3.2	Systematic Capability	31
2.3.3	Fail-Safe and Fail-Operational Systems	33
2.4	Virtual Devices	33
2.4.1	Virtual ECU (V-ECU)	33
2.4.2	Restbus Simulators	33
2.5	Motion Sensors	34
2.5.1	Encoder Interfaces	35

3	State of the Art	39
3.1	Testing Objectives	39
3.2	Interrelationship between the Development Process Models and Testing . .	40
3.2.1	Product Development Process Models	41
3.3	Test Automation for Safety Critical Systems	43
3.3.1	Fault Injection	46
3.3.2	Model-based SW development vs. code-based development	48
3.4	Model-Based Testing	49
3.4.1	Methodological Issues in Model-Based Testing	49
3.4.2	Testing Effort for Model-Based Testing (MBT)	51
3.5	Test Re-usability	51
3.5.1	Testing Re-usability with Evolving Functionality	53
3.5.2	Testing Re-usability in Model-Based Development	53
3.6	Outlook for Testing in Software Engineering	54
3.7	Standardization of XIL Tests	55
3.8	Restbus Simulators	56
3.9	Projects on Certifying Mixed-criticality Systems	57
3.9.1	CESAR project	57
3.9.2	VERDE project	58
3.9.3	VARIES project	58
3.9.4	OPENCOSS project	59
3.9.5	AMASS project	60
3.9.6	ASCOS Project	60
3.9.7	PROXIMA project	60
3.9.8	MultiPARTES project	61
3.10	Discussion	61
II	Contribution	63
4	Theoretical Framework	65
4.1	Methodology	65
4.2	Hypotheses	67
4.3	Goals	67
4.3.1	Operative Goals	68
4.4	Case Studies	69
5	Model-based Testing with ATE Co-simulators for Re-using Tests	71
5.1	Why a Python ATE?	72
5.2	Test re-use on PS-TTM simulations	73
5.2.1	The PS-TTM Modeling and Simulation platform	74

5.2.2	The PS-TTM Automated Test Executor	75
5.2.3	The PS-TTM Synchronous Python Interpreter	79
5.3	Test re-use on Simulink simulations	84
5.3.1	Validation	85
5.3.2	Limitations	85
5.4	Discussion	86
6	Models and Code Re-use for COTS X-in-the-Loop Testbenches	89
6.1	Problem Statement	90
6.2	XiL Testbenches for Dependable Control Systems	91
6.2.1	HiL System with cRIO COTS Heterogeneous Platforms	92
6.2.2	Model-based workflow	94
6.3	Re-using Test Artefacts from MiL to HiL	96
6.3.1	Requirements for Real-time Simulation of Position Sensors	97
6.3.2	A Synthesizable Quadrature Encoder Simulator	97
6.3.3	Simulating Sensors with Bus Interfaces	100
6.3.4	Re-usable Instrumentation Models	108
6.4	Re-using Code Artefacts from SiL to HiL	109
6.4.1	CAN Distributed Control systems	111
6.4.2	Para-virtualization of I/O CAN Devices	112
6.5	Discussion	118
7	Framework of Re-usable Safety Arguments for Mixed-Criticality Product Lines	119
7.1	What is DREAMS?	120
7.1.1	DREAMS Platform-based Design for MCPLs	121
7.1.2	Certification	123
7.2	Certification Support in DREAMS	127
7.2.1	Certification Arguments	128
7.2.2	DREAMS Design Space Exploration (DSE)	130
7.2.3	Safety-compliant use of the DREAMS tool inventory	131
7.3	Automated Composition with Modular Safety Cases for Re-usability	132
7.3.1	Implementing GSN in Enterprise Architect model databases	132
7.3.2	Model-based Certification Workflow	134
7.3.3	Adapting MSCs for Reuse as Argument Models	137
7.3.4	Supporting Certification for Diverse Safety Standards	137
7.3.5	Integration of DB within Variability Management in MCPL	139
7.3.6	Integrating post-DSE VVT evidences	140
7.4	Summary	140
7.5	Discussion	142
7.5.1	Limitations	142

7.5.2	Improvements	143
7.6	Conclusion	146
III	Validation	147
8	Validation on Case Studies	149
8.1	Development of a Railway Controller	149
8.1.1	Context	149
8.1.2	System Description	150
8.1.3	Modelling the ETCS in PS-TTM	153
8.1.4	Reliability Assessment	158
8.1.5	Discussion	167
8.1.6	Conclusion	169
8.2	Model and Code Re-use for a HiL Elevator Simulator	170
8.2.1	Context	170
8.2.2	System Description	171
8.2.3	Problem Statement	177
8.2.4	Heterogeneous COTS Platform for HiL Elevator Simulator	178
8.2.5	The CAN Restbus Simulator	182
8.2.6	Discussion	189
8.2.7	Conclusion	190
8.3	Safety Cases for Wind Turbine Controllers	192
8.3.1	Context	192
8.3.2	System Description	192
8.3.3	Problem Statement	194
8.3.4	DREAMS Architecture for the Wind Turbine Controller	195
8.3.5	Safety Case Argumentation	201
8.3.6	Preliminary Safety-Case Generation	208
8.3.7	Discussion	210
8.3.8	Conclusion	212
IV	Conclusion	215
9	Conclusion	217
9.1	Review	217
9.1.1	Hypotheses Validation	218
9.1.2	Limitations	222
9.2	Lessons Learned	223
9.3	Future Work	225

V Bibliography and Appendix	229
A The PS-TTM Modelling Framework	247
A.1 Fault Injection Libraries	247
A.1.1 Fault Library for Platform-Independent Models	247
A.1.2 Fault Library for Platform Specific Models	249
A.2 Symmetric and Asymmetric Fault Injection	251
A.3 The Simulated Fault Injection Unit (FIU)	253
A.3.1 The PS-TTM toolset and workflow	255
B Re-using Virtual Devices for Virtual Testing	259
C A Primer on GSN Notation	265
C.1 GSN Basic Notation	265
C.2 GSN Modular Extensions	265
C.3 GSN Pattern Extensions	268
D Safety Cases	269
D.1 Example Safety-Case Structures	269
D.1.1 IEC 62741:2015 Dependability Cases	269
D.1.2 Safety Case Structure for EN 50129	270
D.1.3 Eurocontrol Safety Case Structure for EATM	272
D.2 Issues in Automated Verification of Safety Cases	274
E DREAMS Preliminary Safety Case Report	277

List of Figures

1.1	Juran's quality cost models for safety-critical and non-safety products . . .	5
1.2	Structure of the thesis	12
2.1	Structure of a TTA cluster with five nodes	18
2.2	The four interfaces of a TTA component	19
2.3	Safety architectures.	32
2.4	Quadrature encoder input (position) and outputs (A, B, A', B')	36
3.1	Waterfall Development Process (V-model)	41
3.2	MDD Process at Daimler	42
3.3	Test objects and test levels for model- and code-based control development	43
3.4	Incremental Development Process (Spiral Model)	44
3.5	Software testing research road-map	48
3.6	Model-based testing taxonomy	49
3.7	Model-based testing processes	52
3.8	Testing Evolving Software	53
3.9	Top-down reuse for multi-level testing	54
4.1	Research methodology followed in this thesis	66
5.1	PS-TTM Automatic Test Executor	76
5.2	Integration of the Python interpreter in the PS-TTM simulator	81
5.3	The virtual elevator door testbench in Simulink	86
6.1	Tool-set and workflow for the COTS HiL cRIO simulator	95
6.2	Simulink model of a synthesizable quadrature encoder	98
6.3	Simulink testharnesss for MiL validation of the quadrature encoder model .	99
6.4	Quadrature encoder and chronometer IPs in LabVIEW FPGA project for cRIO	101
6.5	Synthesizable Simulink model of a CANopen absolute encoder simulator (top-system view)	103
6.6	Simulink model of a CiA DS301 event timer	104
6.7	Simulink model of a fixed-point CiA DS406 encoder scaler	106

6.8	Simulink test-harness model for the validation of the CANopen absolute encoder simulator.	107
6.9	Simulink model for a synthesizable speed-triggered chronometer	110
6.10	Remote LabVIEW interface for the speed-triggered chronometer	111
7.1	Meet-in-the-middle methodology	121
7.2	Overview of the DREAMS platform architecture	122
7.3	UML stereotypes for the Goal Structuring Notation (GSN) meta-model	135
7.4	DREAMS workflow to support the certification of mixed-criticality product lines	136
7.5	Layered argumentation models for DREAMS re-usable components	138
7.6	DREAMS argumentation database for diverse safety certifications	139
7.7	Handling variability and safety in the DREAMS process	141
7.8	Mapping OPENCROSS argument templates to DREAMS certification arguments	144
8.1	ETCS on-board reference architecture	151
8.2	Mapping the functional structure of the ETCS to the PI-TTM model	154
8.3	Mapping the TMR structure of the ETCS to the PS-TTM model	156
8.4	Connection between ATE (Python) and SUT (ETCS)	157
8.5	PI-TTM system model for ATE-controlled fault injection simulation of ETCS	158
8.6	Results of the PIM simulation and fault injection	162
8.7	PS-TTM system model for ATE-controlled fault injection simulation of ETCS	163
8.8	Simulation results of the PS-TTM model	165
8.9	Results of the PSM simulation with fault configuration #5	166
8.10	Schematic view of the elevator group distributed control system, integrating multiple controllers and networked remote I/O devices	173
8.11	Elevator controller with absolute encoder and Controller Area Network (CAN) link	176
8.12	NI cRIO heterogeneous controllers to realize the HiLES simulators	178
8.13	Integration of the encoder simulator in the cRIO-9082 controller	180
8.14	Importing the HDL-coded CANopen encoder model in a LabVIEW/FPGA VI	181
8.15	LabVIEW block diagram of the Update Virtual Instrument (VI) from the CAN Restbus simulator for the dual-elevator HiLES	186
8.16	Deployment view of the cRIO HiL elevator simulator with CAN Restbus virtual devices	187
8.17	The simulator graphical user-interface (SGUI) front-end for virtual level-call devices (EXT1)	188
8.18	Application scope of the DREAMS platform in the Wind Turbine Controller	193
8.19	Possible deployments of the DREAMS Wind Turbine Control Unit	196
8.20	DREAMS libraries for HW / SW components and specification of safety compliance	197

8.21	'150 % model' architecture of the Wind Turbine Control Unit in AF3	200
8.22	Updated '150 % model' with abstract safety components	201
8.23	'150 %' safety compliance model of an abstract safety function	202
8.24	Generated '100 %' component model with a '1oo2' safety function architecture	202
8.25	Layered GSN modules for compiling the WTCU certification arguments . . .	204
8.26	Imported GSN patterns and generated GSN arguments in AutoFOCUS 3 (AF3)	207
8.27	Refinement of a GSN safety argument for the DREAMS Controller	209
8.28	Hand-crafted wind turbine GSN certification argument (modules & contracts)	211
A.1	Symmetric and Asymmetric Fault Injection	251
A.2	UML Meta-model of Fault Injection Configuration XML files	252
A.3	Integration of tools in the PS-TTM workflow	257
B.1	A distributed test architecture for automated functional verification	260
B.2	Partial virtualization of a test architecture using a RESTBUS simulator . . .	261
B.3	Virtual Testing: Full virtualization of a test architecture	262

List of Tables

2.1	Domain-specific Safety Integrity Levels	31
6.1	Software toolset to re-use sensor test models from MiL to HiL.	96
6.2	Input/output ports for the absolute encoder model	105
7.1	Evidences provided by the DREAMS toolset to support the safety certification	133
8.1	ETCS subsystems	152
8.2	Fault-injection campaign for the PIM model of the ETCS.	160
8.3	Fault-injection campaign for the PSM model of the ETCS.	166
8.4	Results of fault-injection campaigns in the PSM model of the ETCS.	167
8.5	Example PS-TTM simulator performance for PIM and PSM models.	168
8.6	Comparison between the PIM and PSM PS-TTM reliability analysis.	168
8.7	Remote I/O EXT device types.	174
8.8	NI cRIO FPGA specifications.	179
8.9	Measured NMT cycle times for the cRIO Restbus simulator.	190
A.1	Fault library for PI-TTM models	248
A.2	Fault library extensions for PS-TTM models	250
C.1	GSN Entities and Relationships	266
C.2	GSN modular extensions to support the development of safety cases.	267
C.3	GSN pattern extensions to support the abstraction of arguments.	268
D.1	EN 50129 Safety Case documentation structure for railway applications	271
D.2	Eurocontrol defined Safety Case template for ATM applications	272
D.3	Safety Case check rules and verification roles	275

List of Code Listings

5.1	Example of an XML test case configuration file for the TCI module	78
5.2	Example of test points specification file (XML file)	80
5.3	Example of a re-usable Python test script	87
6.1	HDL declaration of the generated IP for the quadrature encoder	100
6.2	Example of embedded application main C module	114
6.3	C++ CModule interface	115
6.4	C++ CMain wrapper with context switching	115
8.1	Example Python script to setup the PI / PS-TTM ATE and run a simulation	157
8.2	Fault config. #4 injected by the PI-TTM ATE on the ETCS PIM	161
8.3	Fault configuration #5 injected by the PS-TTM ATE on the ETCS PSM	164
A.1	Example of an XML fault configuration file for the FIU module	254
A.2	Example of a FIU configuration file for an 'Out of time' HW-fault	254

Acronyms

AADL	Architecture Analysis and Design Language.....	6
AF3	AutoFOCUS 3.....	222
API	Application Programming Interface.....	260
ARM	Argumentation Meta-model.....	126
ASAM	Association for Standardisation of Automation and Measuring Systems	
ATE	Automatic Test Executor.....	259
BTM	Balise Transmission Module.....	151
BVR	Base Variability Resolution.....	199
CAE	Claim Argument Evidence.....	126
CAS	Cycle Accurate Simulator.....	42
CAN	Controller Area Network.....	259
CCL	Common Certification Language.....	137
CCS	Concurrent and Comparative Simulation.....	46
CLFN	Call Library Function Node.....	184
CM	configuration management.....	135
CMT	clock management tile.....	179
COM	communication	
COTS	Commercial-Off-The-Shelf.....	270
CPU	central processing unit	
cRIO	Compact RIO.....	220
DAS	Distributed Application Subsystem	
DB	database.....	222
DBMS	Database Management System.....	224

DCM	Digital Clock Manager	179
DE	Discrete Events MoC	
DEM	Data Exchange Module	
DES	Dependable Embedded System	259
DHP	DREAMS harmonized platform	120
DMI	Driver Machine Interface	152
DREAMS	Distributed Real-time Architecture for Mixed-criticality Systems.....	120
DSE	Design Space Exploration.....	277
DSP	Digital Signal Processor	227
DUT	Device Under Test	259
EA	Enterprise Architect	206
EATM	European Air Traffic Management	269
ECS	Elevator Control System.....	171
ECU	Electronic Control Unit.....	92
ETCS	European Train Control System	150
ERTMS	European Railway Traffic Management System	150
EVC	European Vital Computer	150
E-TTM	Executable Time-Triggered Model.....	219
FIU	Fault-Injection Unit	247
FMEA	Failure Mode and Effects Analysis	159
FMEDA	Failure Mode, Effects and Diagnosis Analysis	
FMU	Functional Mock-up Unit.....	262
FPGA	Field-Programmable Gate Array.....	220
FSM	Functional Safety Management	224
FTA	Fault Tree Analysis	
GASC	Generic Application Safety Case	270
GPSC	Generic Product Safety Case.....	270
GSM-R	Global System for Mobile Communications-Railway.....	150
GSN	Goal Structuring Notation	272
HB	Heartbeat (CANopen message service).....	175
HCP	Heterogeneous Computing Platform.....	221

HDL	hardware description language	222
HFT	hardware fault tolerance	194
HiLES	Hardware-in-the-Loop Elevator Simulator	261
HiL	hardware-in-the-loop	261
HMI	human-machine interface	170
HW	hardware	270
I/O	Input/Output	259
ICE	In-Circuit Emulator	223
IEC	International Electrotechnical Commission	269
IP	Intellectual Property	222
ISS	Instruction-Set Simulator	225
IUT	implementation under test	159
JRU	Juridical Recorder Unit	153
LET	Logical Execution Time	219
LSS	Layer Setting Services	176
LV	LabVIEW	
LVRT	LabVIEW/Real-time	184
MBD	Model-Based Development	218
MBE	Model-Based Engineering	217
MBT	Model-Based Testing	217
MCP	Multi-core Processor	2
MCPL	Mixed-Criticality Product Line	277
MCU	Micro-controller Unit	223
MCS	Mixed-Criticality System	270
MDA	Model Driven Architecture	74
MDD	Model Driven Development	39
MDG	Model Driven Generation	132
MiL	model-in-the-loop	227
MMCM	mixed-mode clock manager	179
MoC	Model of Computation	220
MOGENTES	Model-Based Generation of Test-Cases for Embedded Systems	46

MSC	Modular Safety Case	277
NoC	Network-on-Chip	195
NMT	Network Management	183
NVRAM	Non-Volatile RAM	
OO	object oriented	
OS	operating system	
OSLC	Open Services for Life-cycle Collaboration	
OVP	Open Virtual Platform	
PBD	platform-based design	221
PCB	Printed Circuit Board	113
PCI	Peripheral Component Interconnect	
PCIe	PCI Express	108
PD	Product Development	41
PDO	Process Data Object	183
PIM	Platform-Independent Model	219
PI-TTM	Platform Independent Time-Triggered Model	225
PL	Programmable Logic	220
PSM	Platform-Specific Model	219
PS-TTM	Platform-Specific Time-Triggered Model	219
PE	Programmable Electronics	198
QEMU	Quick EMUlator	
RAM	Random Access Memory	
RT	real-time	225
RTL	Register-Transfer Level	168
RTOS	Real-time Operating System	179
R/W	Read / Write	
SACM	Structured Assurance Case Meta-model	126
SAEM	Software Assurance Evidence Meta-model	29
SASC	Specific Application Safety Case	270
SCCRC	Safety Compliance Constraints & Rules Checker	277
SCDM	Safety-Case Development Manual	269

SCR	Safety-Case Report.....	277
SC	Safety Case.....	272
SCL	Safety Communication Layer	195
SDO	Service Data Object.....	175
SDF	synchronous data-flow MoC	20
SECE	Sistemas embebidos robustos para el ascensor/Robust Elevator Embedded Systems	13
SEU	Single Event Upset.....	249
SFI	simulated fault-injection	167
SGUI	simulator graphical user-interface	188
SIL	Safety Integrity Level.....	194
SiL	software-in-the-loop	90
SLS	source-level simulator.....	57
SL	Simulink.....	226
SoC	System-on-Chip.....	120
SQL	Structured Query Language	
STNoC	STmicroelectronics' NoC	195
SUT	System Under Test.....	259
SW	software	270
TCI	Test-Case Interpreter.....	255
TIU	Train Interface Unit.....	153
TLM	Transaction-Level Modeling.....	168
TMR	Triple Modular Redundancy	155
TPDO	Transmit PDO	
TPM	Test Points Manager.....	157
TT	Time-Triggered	218
TTA	Time-Triggered Architecture	167
TTNoC	Time-Triggered Network-on-Chip	
UDP	User Datagram Protocol	
UML	Unified Modelling Language	224
URL	Uniform Resource Locator.....	135
USB	Universal Serial Bus.....	180

VALMOD	VALidación avanzada y MODular de sistemas críticos en transporte ferroviario	
		13
V-ECU	Virtual ECU	184
VHDL	Very High-Speed Integrated Hardware Description Language	190
VI	Virtual Instrument	184
VnV	Verification and Validation	126
VVT	verification, validation and testing	195
WCET	Worst-Case Execution Time	
WTCU	Wind Turbine Control Unit	192
WTCCS	Wind Turbine Controller Case Study	192
XiL	X-in-the-loop	91
XML	eXtensible Markup Language	161

1

Introduction

Embedded systems are programmable devices that integrate processors and dedicated hardware to implement control functions that interact with real-life environments [TLMP93, Zan08]. The improvement of semiconductor technology sustained a progressive reduction of the size of electronics, leading to increasingly powerful computing platforms with a higher density of components per silicon area. Nowadays many embedded systems integrate several execution platforms and communication interfaces to implement concurrent functionality – e.g., heterogeneous embedded processors integrating multiple cores and programmable logic. These embedded platforms foster the transition from federated to integrated architectures, seeking to lower the overall cost [Obe04, Ham03, NS10], but also prompting a complexity growth. As a consequence, testing modern embedded systems becomes a challenging activity due to system heterogeneity, design constraints and the increasing architectural complexity. At the same time, manufactures ask for a reduction of the time-to-market for the product development process. Therefore, current industrial practice brings forward the verification activities to earlier stages in the development of embedded systems, aiming at the contention of design-and-verify iterations.

Dependable embedded systems (DES) are embedded systems that could lead to loss of life, significant property damages or damages to the environment in case of failure. Such safety-critical systems must exhibit a degree of fault-tolerance as required by the safety standards for the identified potential risks. The development of a **DES** shall comply with the recommendations from safety standards for the specification, design and implementation of safety-critical applications, following state-of-the-art standardized procedures to ease the review and verification activities. Safety standards demand a predictable behaviour of the safety system under all foreseeable scenarios. Although formal reasoning could provide a

mathematical foundation for the correctness of a safety function, the applicability of formal methods is limited to cases where simplification assumptions hold. Whenever a rigorous proof of correctness is infeasible, simplified safety assessment activities still rely on simulation and testing [NS10]. As a consequence, the verification, validation and testing activities for dependable embedded systems claim a substantial preparation and realization effort.

Certification is a third-party attestation related to products, processes, systems or persons [ISO04]. An *attestation* is the issue of a statement, based on a decision following reviews, where the fulfilment of specified requirements has been demonstrated. *Safety certification* is an attestation where an authorized organization or a certification body assesses the fulfilment of the safety-related requirements of a system regarding specified safety requirements or a safety standard. In the scope of the IEC 61508 safety standard and its derivatives, safety is a systemic property, resulting from the composition of the properties of constituents, their integration and the interaction of the system with its environment. Thus, the IEC 61508 standard requires the certification of the system as a whole, where a change of a single aspect of the system may require the re-certification of the entire system.

With the introduction of the newest dependable computing platforms for safety applications the complexity of safety systems also increased correspondingly, bringing with it a growing number of potential defects [EJ09]. System engineers pursue the integration of multiple functions with different safety-, security- and real-time requirements on the same embedded computing platform, i.e., a *mixed-criticality system* [Bau11]. Multi-core Processors (MCPs) enable reductions in the cost, size and weight of the hardware, while improving the system scalability. However, MCPs pose certification challenges (such as the assessment of the temporal independence) which may potentially increase the engineering and certification cost to unacceptable levels [DAN+13, RGG+12].

The challenge of certifying a mixed-criticality system (MCS) may be alleviated by following a platform-based design (PBD) paradigm, tailoring a pre-built safety solution that would be eventually certified, e.g., the DREAMS harmonized platform [Lar17]. Typically PBD adopts the Y-chart development process [BCG+97, KDV+97], refining separately a platform model and a functional model of the system, then combining both by means of a mapping model to obtain a model of the complete system. Combined use of Model-Based Development (MBD) and Model-Based Testing (MBT) provide a methodology to tackle the complexity of an MCS, enabling concurrent engineering at both the system design and its verification. Interleaved MBD/MBT enables early validation in a platform-based design, but generates analysis predictions that yet shall be contrasted with experiments on the real system. MBT provides test vectors for: (i) the stimuli of the implementation under test (IUT), and (ii) the expected outputs from the IUT. For complex system environments a *reactive MBT* approach is more convenient, i.e., the stimuli fed to the IUT are computed from previous IUT output values using a mathematical representation of the environment. Improving the re-usability of artefacts from modelling to testing would enhance the test repeatability and recovers the investment from previous MBT tasks. In MBT models for platform hardware (HW), sensors and custom instrumentation would be developed. These

models become valuable assets for reproducing the test scenarios in a hardware-in-the-loop (HiL) test configuration to exercise the real system.

The platform-based design of Mixed-Criticality Systems (MCSs) also supports the development of whole product families. A *Mixed-Criticality Product Line (MCPL)* is a set of MCSs that share a number of HW/SW components, while differing in the features, functionality, and possibly in the safety requirements. In a MCPL each specific product configuration is referred to as a “*product sample*”. Model-Based Development (MBD) helps at optimizing an MCPL: variant modelling supports an automated Design Space Exploration of possible configurations for the Dependable Embedded Systems. In the context of the IEC 61508 standard, safety is an emergent property of the system, and thus a safety assessment could be required for each feasible product sample in order to certify the MCPL. This per-product safety assessment would also require a specific safety argument, but it could yet re-use the proof artefacts common to other product configurations in the MCPL.

The certification approval ultimately depends on the collection of analysis results, verification evidences and a sensible rationale supporting the safety claims. Several safety standards also accept a structured modular approach to demonstrate the validity of the safety claims: the Safety Cases. A *Safety Case (SC)* is a documented body of evidences that provides convincing and valid arguments that a system is adequately safe for a given application in a given environment. The scope of a Safety Case can be the whole system, a subsystem, or a component. Also an SC may depend on other SCs. A benefit of applying platform-based design to the development of Mixed-Criticality Systems is that the safety properties of the components in the base platform can be described by a number of Modular Safety Cases (MSCs). MSCs support a modular certification process that parallels the compositional approach in platform-based design.

A way to improve the re-usability of safety arguments to develop Safety Cases for Mixed-Criticality Product Lines is to provide a framework for the semi-automated composition of the safety arguments, starting from the MSCs: the process of assembling the safety argument for a given product sample would begin at the Design Space Exploration (DSE) phase, when each composition of product features requires an instantiation of the related MSCs¹. Such a framework for composing safety arguments would also enable a staged completion of the arguments: as development progresses, evidences gathered from verification and testing phases can be linked to the argument chain. Once there are no pending evidences, the proposed safety argument is ready to be reported –and eventually reviewed and challenged. To this end, the re-usable MCSs should be arranged as to ease the composition at the DSE, enable a phased completion as verification, validation and testing progresses, and support automated reporting according to a documentation structure. All in all this provides the logical framework to interrelate all the results from a safety-system development process.

¹ A Modular Safety Case (MSC) may contain alternate argument paths to support different safety claims, i.e., the higher the Safety Integrity Level (SIL) level, the more comprehensive the argument. The instantiation of an MSC consists of selecting a chain of arguments to the demonstrable safety property, although the links may end in evidences to be provided in later development phases.

1.1 Motivation

Current industrial practice relies primarily on testing for hardware and software verification. A key point is that testing always presumes some model of the system behaviour, although this model could be implicit (i.e., a mental model) or explicit (i.e., a formal model). Testing is a methodology to obtain better systems, therefore it is widely used in systems development. But typically, it is ad hoc, error prone, and costly.

We focus on testing methods for embedded systems, that constitute a form of reactive systems. A reactive system denotes software and hardware systems with a (usually) non-terminating behaviour that interact with a physical environment through observable events. As embedded systems became widespread in many application fields, research sought for more formal verification methodologies that also would scale-up to cope the rising complexity of the systems under test.

Test case generation can be considered the heart of testing. Research on this topic has provided a variety of successful test case design methodologies. But improving the testing process as a whole is more complicated. The test cases have to be executed on the system under test. In several application domains, test suites are used to show conformance to a standard. For this, test cases have to be interchangeable among developers. Furthermore, testing should be included in the overall development process.

Analogously to the evolution of design methodologies, testing has embraced the so-called model-based approach to tackle complexity. The model-based approach assumes the existence of a precise formal model of the system being developed. Design engineers can use this model to study the system to build. Test engineers can use this model either to generate complete test suites to show conformance of the model and the actual implementation, or, just to design purpose-specific test cases to check the developed system. In a model-based development approach the testing shall start at the system modelling phase. For testing reactive systems we usually also have to model the application domain. This environment model is the core of hardware-in-the-loop (HiL) testing techniques.

Embedded systems for safety critical applications (also known as dependable systems) implement safety-related functionality in a predefined environment. The environment for dependable systems usually presents some kind of risk of occurrence of hazardous events. Legal regulations require a product approval for the deployment of dependable systems, requiring that the whole life-cycle for safety critical products must comply with generic safety standards as IEC 61508, or domain specific standards like DO-178B, amongst others.

Safety standards impose different risk reduction measures depending on the foreseeable faults, the hazard originating from this fault and the severity of the possible damage. The IEC 61508 standard defines the risk as a probabilistic measure of fault events, and quantifies the required resilience of a system by its Safety Integrity Level (SIL). The operation of a dependable system must be as predictable as possible, in order to ascertain the required safety envelop. The compliance is checked by strict certification. Verification, validation and testing (VVT) are key activities that either assess the compliance of the development with

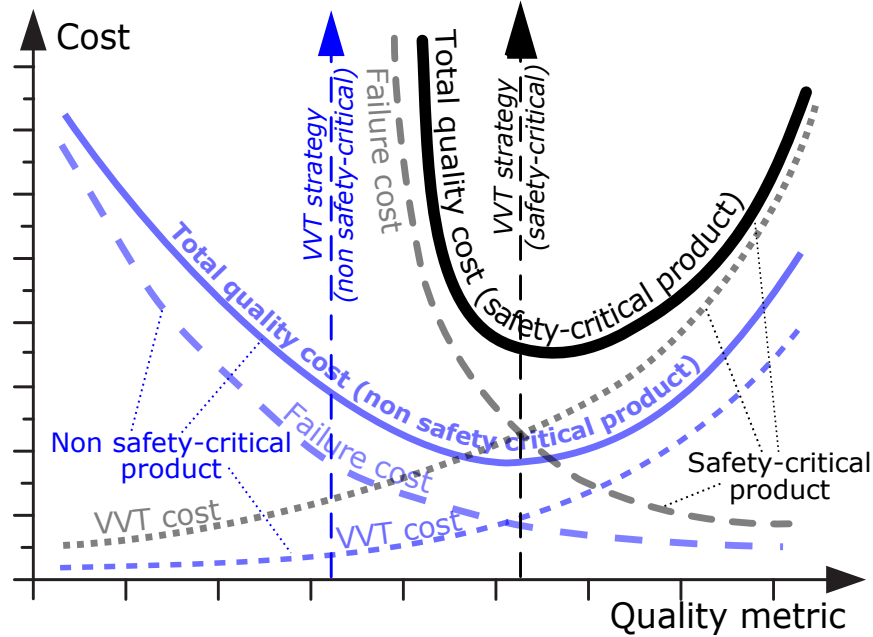


Figure 1.1: Juran's quality cost models for safety-critical and non-safety products

regard to the standards or detect deviations. The verification, validation and testing (VVT) activities impose a significant cost on the realization and certification of dependable systems, as the objective *quality metric* is usually much higher than that for non-safety programmable devices. Figure 1.1 informally depicts the *total quality cost* for both kinds of product developments, where *total quality cost* results from adding the cost of VVT activities and the cost of failures. For safety-critical products the cost of failures raises as a consequence of legal liability and the damage caused to the people or the environment.

Model-based development and the related testing methodologies have been in use for years in application domains like automotive, railway or aerospace. This long-run of model-based development in industrial practice provides a long enough time scope to review the actual achievements with regard to the efficiency of model-based process development, and also to detect the needs for improvement. A retrospective study [KZ10, BKZ+11] on the economical impact of model-based development reported significant potential for cost reductions in product development for the car industry. But a successful model-based development (including testing) requires a clear and structured process. Otherwise a development cost overrun can occur.

A related field of research is the modelling language itself. In Software Engineering the continuous development of abstraction mechanisms allows software developers to tackle the ever-increasing complexity. However, the commonly used description languages for

hardware design lag behind their software counterparts in terms of structural abstraction, encapsulation and even behavioural semantics [Nik11]. This motivates the need to define an adequate meta-model to describe the system structure and behaviour. To fill this gap, researchers have come up with languages like Giotto [HHK03], Architecture Analysis and Design Language (AADL) [AS5506A], BIP [BIP], DOL [DOL] or SysML [SysML] amongst others. A common driving force at defining these modelling languages is the quest for a concise underlying meta-model with precise semantics. AADL or SysML offer the advantage of being also embodied in Unified Modelling Language (UML) [UML] (an UML mapping to AADL is included as part of the MARTE profile [MARTE]), that is a semi-formal modelling language for Software Engineering. This enables the scaling-up of the architectural models to fine-grained software design using a common semantics. Although this embodiment in a more expressive language seems appealing, considerable research needs to be devoted to fixing the issues arising when combining the multiple underlying meta-models.

Some authors [Bro11, BKS+10a] hypothesize that in order to achieve a cost-effective model-based development we shall improve the seamlessness of the development tools, enabling model use throughout the whole development process. Currently available tools and methodologies limit model re-usability, nor is there an automated transformation into adequate models. This lack has a deep impact on the development cost of dependable systems, where the developer has to provide a complete tracing from requirements to the resulting system and safety standards impose a thorough conformance verification. Thereby an integrated model-based development process with tool support ranging from requirements analysis through code generation to validation would be tremendously beneficial.

The availability of models eases the examination of the system. In model-based verification we obtain a number of valuable test artefacts (e.g., abstract test suites) that have undergone a validation process. An optimized model-based testing process should pay-off for the effort invested in building such testing infrastructure. The improvement of the re-usability of testing artefacts could be a means to achieve this return-of-investment. On-going research from pioneering adopters (automotive) of model-based testing focuses on improving the the model-based testing process to enable re-usability [MK10a, MK10b, MK10c].

Another important issue is that verification (testing) by simulation is decreasingly practical as the system complexity rises. For mixed-criticality dependable systems the hardware is complex due to redundancy and may run heavy software loads such as full-featured operating systems and applications. Verification involves simulating all of these together, and it is not unusual for software simulations to run for days or weeks. In a model-based approach we can reconsider the classical question about when to stop testing, and re-state it as when to stop testing on software simulators. Some researchers suggest using hardware accelerators to support simulators for testing. The advantages of testing with hardware assisted simulations are multi-fold. A clear benefit is an increased simulation speed, that yields a reduction in overall testing time. But this technique also enhances test re-usability if we design a modular test architecture, as the test infrastructure subsumes or resembles the hardware-in-the-loop components needed for the verification of the real system

prototypes. Taken to the extreme, a whole test architecture and an executable simulator of the system under test would be deployed on synthetic hardware (FPGA) for accelerated simulations.

Whenever an application requires a high confidence level (i.e., a high **SIL**) and the selected implementation is based on programmable electronics (PE), the adoption of redundant structures necessarily involves the adoption of distributed architectures. For distributed embedded systems micro-kernels (namely hypervisors) and software architectural standards like AUTOSAR or ARINC-653 alleviate the complexity problem by decoupling the application software from the execution platform. The separation is achieved by abstracting the hardware as a virtual machine.

But still the problem of verifying distributed architectures remains, because the system behaviour becomes an emergent property that is only observable after assembling the components. This raises the question of how we can perform an early verification at system level for distributed real-time systems that could be reused up to the final stage in the development process. Regarding verification, virtualization eases the instrumentation of the software components in a cross-platform development. A key point is that the virtualization constitutes an enabling technology for an interleaved refinement of both the design and the test suite. This interleaved refinement requires a structured testing approach, by abstracting unneeded details in a given design phase. This could ease the maintainability of the testing framework.

Summing up, test re-use in a model-based development process is a hot topic for research due to its economical significance in product development. In the field of safety-critical applications, model-based testing is recommended as a suitable technique for the validation and verification of dependable systems, but standards usually do not provide complete guidelines for this. It is desirable to rely on a concise modelling language to describe the system and the test architectures, but rooted in a more expressive standard that would scale-up for more detailed design models while keeping the backwards compatibility with the initially sketched architectural and behavioural models. From the viewpoint of validation, testing has to deal with the examination of emergent behaviour. To perform the examination one needs to have the ability to build some dynamic artefacts (i.e., simulators). For that we need a tool-set of transformation tools and computational support for a variety of models-of-computation. The integration of open-source hardware simulation platforms like SystemC with other specialized modelling environments showed the feasibility of hardware-software co-simulation. Besides that, hardware defined by software **FPGAs** is blurring the gap between testing on simulators and testing the actual prototypes in real-time.

The **research topic** for this PhD is the *improvement of the testing re-usability in a model-driven development of real-time distributed embedded systems for safety-critical applications, comprising virtual processing platforms and mixed-criticality software components*. The **topic interest** is the *cost-effectiveness* achieved in a model based development by means of test automation and test reuse. This research looks for a reduction of the verification effort of mixed criticality dependable systems.

1.2 Contribution

This dissertation contributes with three complementary approaches to re-use tests and artefacts for the cost-effective development and certification of Mixed-Criticality Systems:

- (i) an integrated system/tester co-simulation framework to assess the fault-tolerance mechanisms in redundant system architectures –like those found in Dependable Embedded Systems–, through simulated fault injection and supporting the re-use of test specifications for real system testing;
- (ii) an **MBT** methodology to model sensors, instruments, as well as multiple nodes in distributed Dependable Embedded Systems (**DESs**), that are initially validated in Commercial-Off-The-Shelf (**COTS**) simulation environments and later re-used in heterogeneous **COTS** hardware-in-the-loop (**HiL**) platforms for real-time testing.
- (iii) a framework to model re-usable safety arguments for platform-based design (**PBD**) of Mixed-Criticality Product Lines (**MCPLs**) that enables a staged semi-automated composition of Safety Cases (**SCs**), supporting the integration of compliance evidences yielded either by analysis or testing activities at each phase in the **MCPL** development.

The **scientific interest** lies on the structured model based testing approach that mimic the abstraction boundaries into the testing framework, which promote the re-use of validated components to be deployed in diverse simulation environments: Commercial-Off-The-Shelf (**COTS**) and custom open-source simulators (either in model-in-the-loop (**MiL**) or software-in-the-loop (**SiL**) configurations), and heterogeneous execution contexts mixing Programmable Electronics (**PE**) or Programmable Logic (**PL**) (in a **HiL** configuration).

The proposed **main innovation** consists of a *verification methodology for dependable real-time embedded systems that enables the re-usability of system tests across different refinement phases of the design, from model to executable virtual prototypes.*

Each contribution is illustrated by its application to a case study on a demonstrator:

CS1 Train odometry system: The train odometry system case study focuses on the re-use of test specifications (e.g., test sequences, test configuration) generated during the safety analysis of a redundant **DES**. The train odometry system model is derived from a certifiable product concept for a **SIL4** safety requirement in the sense of EN50128 standard [CEN11]. This case study assumes a Triple Modular Redundancy functional structure, the set of random (**HW**) faults considered in the product Failure Mode and Effects Analysis for the given structure, and the source code implementation of the safety functions. Based on this, we use a modelling framework to obtain an executable Platform-Specific Time-Triggered Model (**PS-TTM**) model. The synchronous Automatic Test Executor (**ATE**) integrated in the **PS-TTM** simulator enables the automation of test campaigns, while a non-intrusive simulated fault-injection library [Aye15] stresses the system fault-tolerance. The train odometry system case study illustrates contribution (i).

- CS2** *Hardware-in-the-Loop Elevator Simulator (HiLES)*: The **HiLES** case study focuses on the model-to-**HiL** and code-to-**HiL** re-use of artefacts to enhance a real-time hardware-in-the-loop (**HiL**) simulator used for the functional verification of elevator controllers. This comprises two tasks: (i) developing a library of models for position sensors and custom instruments in a **COTS** modelling environment, and later transformed and deployed in a heterogeneous **COTS** computing platform. (ii) developing a para-virtualization framework to re-use the validated functionality of remote I/O nodes, recovering the **VVT** effort invested in building the real products when assembling the **HiLES** system. The **HiLES** case study demonstrates contribution (ii).
- CS3** *Mixed-criticality wind turbine controllers based on the DREAMS harmonized platform*: The **DREAMS** wind turbine case study focuses on the re-use of arguments for safety certification in a platform-based design (**PBD**) of Mixed-Criticality Product Lines (**MCPLs**). The **PBD** is supported by: (a) a harmonized heterogeneous computing platform for **MCPL** applications, the **DREAMS** harmonized platform, (b) a **DSE** tool-set to refine the design of the **MCPL** by using integrated analysis tools, and (c) a set of generic Modular Safety Cases (**MSCs**) for the safety components in the harmonized platform. The re-usable argument modelling framework provides a repository for **MSCs**, where modular Goal Structuring Notation extensions provide customizable place holders to complete the certification argument according to the safety claims, the **MCS** configuration and the supporting evidences. This case study shows contribution (iii).

1.2.1 Technical Contributions

This dissertation provides the following technical contributions:

- TC1** Adaptation of a Python-based Automatic Test Executor (**ATE**) engine for co-simulating fault-injection tests on Platform-Specific Time-Triggered Model of Dependable Embedded Systems.
- TC2** A library of synthesizable Simulink models for position sensor and custom instruments that improve the time coherency of outputs fed to a System Under Test (**SUT**) in a **HiL** test environment.
- TC3** A lightweight para-virtualization approach to simulate the **HW** and firmware of networked Input/Output (**I/O**) nodes in diverse test execution contexts (including **SiL** or **COTS HiL**), easing the re-configuration of the test architecture (by replacing devices with virtual replicas) while preserving the functionality of the devices.
- TC4** A Goal Structuring Notation meta-modelling extension and a **DBMS**-hosting of argument models and evidences to support the semi-automated composition of Safety

Cases for **MCS** by the Safety Compliance Constraints & Rules Checker (**SCCRC**) component integrated in the **DREAMS** Design Space Exploration tool.

1.2.2 Collaborations

The validation of the concepts developed in this thesis was done concurrently and in collaboration with other researchers, namely:

PhD thesis by I. Ayestaran [Aye15] I. Ayestaran contributed the **HW** models for simulated fault injection, developed the modelling tools using the Eclipse modelling framework, and worked on the the SystemC implementation of the Platform Independent Time-Triggered Model (**PI-TTM**)/**PS-TTM** Model of Computation (**MoC**) as well as the implementation of the simulated fault injectors (SFIs). The concept of **PI-TTM** and **PS-TTM** models was developed in collaboration with C.-F. Nicolas. C.-F. Nicolas contributed to the integration of the **ATE** in the SFI simulation framework to provide test repeatability and portability to real-time test environments.

PhD thesis by A. Larrucea [Lar17] A. Larrucea developed the Modular Safety Cases (**MSCs**) for the DREAMS platform. C.-F. Nicolas contributed the extended meta-model for modular Goal Structuring Notation (**GSN**) and **DBMS** infrastructure to deploy the **MSC** repository. The description of the **MSCs** as re-usable modular **GSN** models was developed in collaboration.

1.2.3 Publications

The contributions from this thesis were published in the bibliography listed below:

- Larrucea, A., Martinez, I., Nicolas, Carlos-F., Perez, J., and Obermaisser, R. (2017) **Modular Development and Certification of Dependable Mixed-Criticality Systems** In: *Digital System Design (DSD), 2017 20th Euromicro Conference on*, Vienna, Austria. 30 Aug.-1 Sept., 2017. [ACCEPTED PAPER]
- Nicolas, C.-F., Eizaguirre, F., Larrucea, A., Barner, S., Chauvel, F., Sagardui, G. and Perez, J. (2017). **GSN Support of Mixed-Criticality Systems Certification** In: *“Dependable Smart Embedded Cyber-physical Systems and Systems-of-Systems” at SAFECOMP 2017 (DECSoS '17), 17th ERCIM/EWICS/ARTEMIS Workshop on*, Trento, Italy. Sept. 12, 2017, [ACCEPTED PAPER]
- Nicolas, C.-F., Ayestaran, I., Poggi, T., Sagardui, G. and Martin, J.-M. (2017). **A CAN Restbus HiL Elevator Simulator based on Code Reuse and Device Para-virtualization**. In: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2017 IEEE 20th International Symposium on*, Toronto, Canada. May 16-18, 2017. Proceedings, pp. 117-124. IEEE. DOI: [10.1109/ISORC.2017.2](https://doi.org/10.1109/ISORC.2017.2)

- Sagardui, G., Agirre, J., Markiegi, U., Arrieta, A., Nicolas, C.-F. and Martin, J.-M. (2017). **Multiplex: A Co-Simulation Architecture for Elevators Validation** In: *Electronics, Control, Measurement, Signals and their application to Mechatronics (ECMSM), 2017 IEEE International Workshop of*, Donostia, Spain. May 24-26, 2017. Proceedings, pp. 1-6. IEEE. DOI: [10.1109/ECMSM.2017.7945883](https://doi.org/10.1109/ECMSM.2017.7945883)
- Nicolas, C. F., Ayestaran, I., Martinez, I. and Franco, P. (2016). **Model-Based Development of an FPGA Encoder Simulator for Real-Time Testing of Elevator Controllers.** In: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2016 IEEE 19th International Symposium on*, York, UK, 17-20 May 2016. Proceedings, pp. 53-60. IEEE. DOI: [10.1109/ISORC.2016.17](https://doi.org/10.1109/ISORC.2016.17)
- Larrucea, A., Perez, J., Nicolas, C. F., Ahmadian, H. and Obermaisser, R. (2016). **A Realistic Approach to a Network-on-Chip Cross-domain Pattern.** In: *Digital System Design (DSD), 2016 19th Euromicro Conference on*, Limassol, Cyprus, 31 Aug.-2 Sept. 2016. Proceedings, pp. 396-403. IEEE. DOI: [10.1109/DSD.2016.66](https://doi.org/10.1109/DSD.2016.66)
- Larrucea, A., Agirre, I., Nicolas, C.-F., Perez, J., Azkarate-Askasua, M. and Trapman, T. (2015) **Temporal Independence Validation of an IEC 61508 compliant Mixed-Criticality System based on Multi-core Partitioning.** In: *Specification and Design Languages (FDL), 2015 Forum on*, Barcelona, Spain. 14-16 Sept. 2015. Proceedings, pp. 1-8. IEEE. DOI: [10.1109/FDL.2015.7306359](https://doi.org/10.1109/FDL.2015.7306359)
- Ayestaran, I., Nicolas, C.-F., Perez, J., Larrucea, A. and Puschner, P. (2014) **A novel modeling framework for time-triggered safety-critical embedded systems.** In: *Specification, Design Languages (FDL), 2014 Forum on*, Munich, Germany. Proceedings, vol. 978, pp. 1-8. IEEE. DOI: [10.1109/FDL.2014.7119343](https://doi.org/10.1109/FDL.2014.7119343)
- Ayestaran, I., Nicolas, C.-F., Perez, J., Larrucea, A. and Puschner, P. (2014). **Modeling and Simulated Fault Injection for Time-Triggered Safety-Critical Embedded Systems.** In: *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2014 IEEE 17th International Symposium on*, Reno, NV, USA. Proceedings, pp. 180-187. IEEE. DOI: [10.1109/ISORC.2014.9](https://doi.org/10.1109/ISORC.2014.9)
- Ayestaran, I., Nicolas, C.-F., Perez, J. and Puschner, P. (2014) **Modeling Logical Execution Time based Safety-critical Embedded Systems in SystemC,** In: *2014 3rd Mediterranean Conference on Embedded Computing (MECO)*. Budva, Montenegro. Proceedings, pp. 77-80. IEEE. DOI: [10.1109/MECO.2014.6862662](https://doi.org/10.1109/MECO.2014.6862662)
- Ayestaran, I., Nicolas, C.-F., Perez, J., Larrucea, A. and Puschner, P. (2014) **A Simulated Fault Injection Framework for Time-Triggered Safety-Critical Embedded Systems.** In: Bondavalli A., Di Giandomenico F. (eds) *Computer Safety, Reliability and Security (SAFECOMP), 33rd International Conference on*, Florence (Italy). September 10-12, 2014. Lecture Notes in Computer Science (LNCS), vol. 8666, pp. 1-16. Springer. DOI: [10.1007/978-3-319-10506-2_1](https://doi.org/10.1007/978-3-319-10506-2_1)
- Perez, J., Gonzalez, D., Nicolas, C.-F., Trapman, T. and Garate, J. M. (2014). **A Safety Certification Strategy for IEC-61508 Compliant Industrial Mixed-Criticality Systems Based on Multicore Partitioning.** In: *Digital System Design (DSD), 2014 17th Euromicro Conference on*, pp. 394-400. IEEE. DOI: [10.1109/DSD.2014.38](https://doi.org/10.1109/DSD.2014.38)

- Perez, J., Nicolas, C.-F., Obermaisser, R. and El Salloum, C. (2010). **Modeling Time-Triggered Architecture Based Real-Time Systems Using SystemC** In: Kaźmierski T., Morawiec A. (eds) *Specification and Design Languages (FDL), 2010 Forum on*. Lecture Notes in Electrical Engineering (LNEE), vol. 106, pp. 123-141. Springer. DOI: [10.1007/978-1-4614-1427-8_8](https://doi.org/10.1007/978-1-4614-1427-8_8)

1.3 Outline

This thesis is structured as described below (see Fig. 1.2):

Chapter 2 presents the underlying concepts on which this thesis is founded.

Chapter 3 analyses the state-of-the-art on model-based testing processes, its re-usability in test automation environments, its application to verify and validate dependable electronics, and the structured argumentation of safety properties and its relationship with the test evidences.

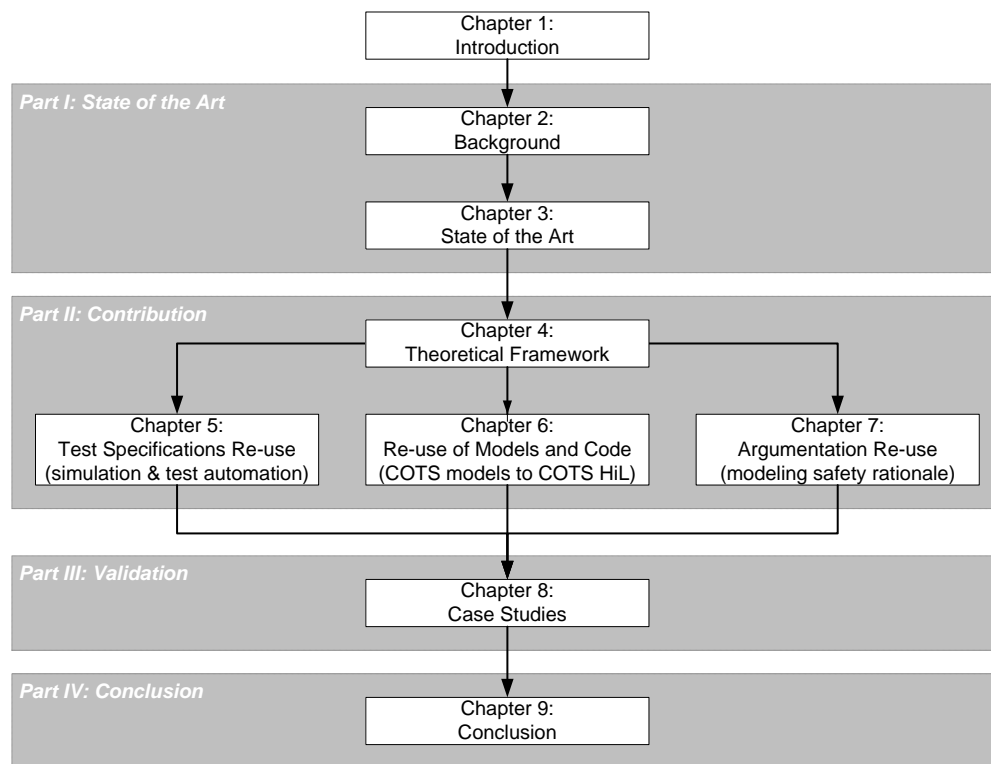


Figure 1.2: Structure of the thesis

Chapter 4 presents the theoretical aspects of this dissertation: research methodology, hypotheses, goals and case studies used to validate the implemented solutions.

Chapter 5 presents the adaptation of the Python interpreter to implement a sequential Automatic Test Executor (ATE) environment that enables both the model-to-system and system-to-model re-use of test scripts, to the Commercial-Off-The-Shelf Model-Based Development MATLAB/Simulink, as well as a novel Eclipse-based modelling and simulation tool for Time-Triggered Architectures, relying on a Time-Triggered extension of SystemC .

Chapter 6 presents the methodology to re-use model artefacts in test components for real-time test architectures to implement parallel stimulation and observation architectures in hardware-in-the-loop (HiL) simulators. It covers the code re-use strategy to integrate proven-in-use functionality in an X-in-the-loop sub-system of the test-bench, i.e., available for model-in-the-loop, software-in-the-loop or HiL configurations.

Chapter 7 presents the DREAMS modular certification approach, based on a framework to re-use certification arguments linked to test evidences, as to support the platform-based design of mixed-criticality product lines.

Chapter 8 describes three different case studies used to evaluate the approach and discusses the results: (a) application of the ATE integration to the validation of the fault-tolerance of a safety over-speed protection for trains, (b) application of the model and code reuse to build elevator simulators to verify the positioning controller, and (c) application of the argumentation re-use framework to a the platform-based design of a wind turbine controller product line, integrating safety protection functions and deployed on the heterogeneous DREAMS platform.

Chapter 9 sums up the conclusions and suggests possible future research work.

1.4 Support

This research has been partially funded by the programs listed below:

- The Spanish INNPACTO project VALidación avanzada y MODular de sistemas críticos en transporte ferroviario (VALMOD), under grant No. IPT-2011-1149-370000.
- The Spanish RETOS project Sistemas embebidos robustos para el ascensor/Robust Elevator Embedded Systems (SECE), under grant No. RTC-2016-5390-4, and
- The European FP7 project Distributed Real-time Architecture for Mixed-criticality Systems (DREAMS), under grant No. 610640.

Part I

State of the Art

2

Background

This chapter introduces terms and topics used throughout this dissertation.

2.1 The Time-Triggered Architecture (TTA)

The Time-Triggered Architecture (TTA) [Kop98, KB03, Kop11] provides a computing infrastructure for the design and implementation of dependable and safety-critical embedded systems. The TTA decomposes systems into nearly autonomous clusters and nodes that share a fault-tolerant global time base of known precision. The existence of this global time in all the components of the system enables to abstract the communication interfaces, guarantees the timeliness of real-time applications, and eases prompt error detection in communications. Therefore, the TTA is based on the time-triggered MoC [Kop98b], which relies on the sparse-time model of time. The TTA infrastructure guarantees the agreement between the time stamps at each node.

The interfaces and the predictable Time-Triggered Protocol (TTP) decouple the processing functions from communications among the distributed subsystems, thus simplifying the design of the internal application software of the nodes. In the TTA, systems are composed of one or several clusters, which are composed of one or several nodes interconnected by a replicated time-triggered network (Figure 2.1). Each node consists of a time-triggered Communication Controller (CC), a Communication Network Interface (CNI) and a host processor with memory that executes the operating system and the application software.

The communication system (composed of the communication network and controllers)

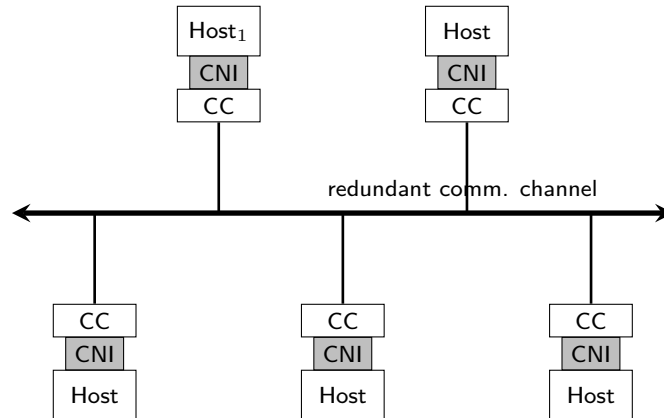


Figure 2.1: Structure of a TTA cluster with five nodes

executes periodically following an a-priori specified schedule, i.e., it reads a message from the CNI at the sending node at an a-priori known instant, and delivers it to the rest of the nodes at an a-priori known instant.

The dynamics of the real-time application are modelled by a set of relevant state variables, called real-time entities. RT entities have static attributes that do not change during their lifetime, such as their name, type or unit, and a set of dynamic attributes, such as their value at a given instant. The observation of an RT entity represents the information about its state at a particular instant, and can be captured in the following data structure:

$$Observation = \langle name, value, t_{obs} \rangle$$

In order to manage complexity, three different types of interfaces were defined in the very first specifications of the TTA [KS03, KB03]. Later versions specify four different interfaces [KOS+07, Kop11] (Figure 2.2):

- **Linking Interface (LIF):** The LIF is the interface that provides the timely information to the nodes during the operation of the system. This interface is used by the nodes to communicate among themselves, and it is therefore a time-critical interface that must meet the temporal specification of the application in all possible scenarios. This interface is also called *Real-time Service (RS) interface* in [KB03].
- **Configuration and Planning Interface (CP):** The CP is the interface used to configure the system, i.e., to connect a node to other nodes. It is used during the integration phase to generate the “glue” between the quasi-autonomous nodes. Hence, this interface is not time-critical. This interface is also called *Technology Independent Interface (TII)* in [Kop11].

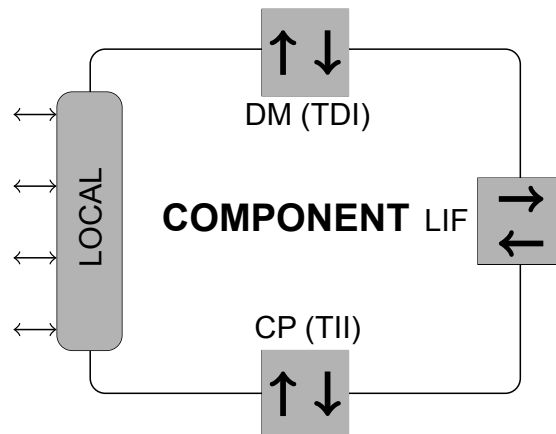


Figure 2.2: The four interfaces of a TTA component

- **Diagnostic and Maintenance Interface (DM):** The DM is the interface that enables maintenance engineers to observe the internal state of the nodes and set their internal parameters. This interface does not influence the temporal behaviour of the nodes, and it is usually not time-critical. This interface is also called *Technology Dependent Interface (TDI)* in [Kop11].
- **Local Interface:** The local interface connects the component to the external world. This interface is used to link the system with the environment.

The aim of the TTA is to design large real-time systems. Despite the fact that a large system will support many more functions than a small system, the complexity of each individual function must not increase with the growth of the system [KB03]. To that end, the only central element of the TTA is the global notion of time. The TTA supports the definition of subsystems by the inclusion of gateway nodes in the system. Gateway nodes are special nodes that contain two CNIs, and they are used to connect different subsystems. A node can be expanded into a gateway node when its computational limits are reached. This way, the interface to the original cluster remains unchanged in the value and time domains, whereas the functionality of the node is now distributed over a second cluster. Gateway nodes are also useful to integrate legacy systems.

2.2 Modelling Languages

The specification and design of safety-critical systems can be expressed in different modeling languages. As no single-language can precisely describe all the safety-related aspects of the system under design, in practice we need some mixture descriptions in multiple languages.

Research on testing mainly relies on 2 different formalisms:

Unified Modeling Language The Object Management Group released a language extension for testing modeling with Unified Modelling Language (UML) 2.0 namely the UML 2.0 Testing Profile, mainly used in the Software Engineering domain [BBL+05, BG09, BGL+07, CMK08, GD06, Gro05, HVF+05, MAM+06, UTP05, ZDS+05, ZSM06, ZXS08].

Matlab/Simulink/Stateflow This integrated development environment is widely used in the automotive domain. Due to its commercial base, there is intensive research and development for increasing the level of automation of design, coding and testing in the MathWorks environment. [Con07, MTest, SLDV, SLVV, Win09, Zan08, ZC05, ZC06, ZC08, ZMS07b]. The capabilities of the Matlab-Simulink-Stateflow for model-based development and testing can be extended to ESoC prototyping by co-simulation with SystemC or CatapultC and a FPGA-in-the-loop Simulink add-on. [BTZ05, HON08].

Other kinds of analysis require complimentary information and suitable modeling formalisms, i.e., probabilistic information contained in Markov chain models [The03].

2.2.1 Simulink

This subsection defines terms used to describe properties of Simulink model elements [SLUG].

Virtual blocks / subsystems: Virtual (i.e., non-atomic) blocks / subsystems are treated as all its internal blocks were at the same level in the model hierarchy as the subsystem when determining block method execution order. This can cause execution of methods of blocks in the subsystem to be interleaved with execution of methods of blocks outside the subsystem.

Virtual subsystems are hierarchical containers used to organize a model graphically. Simulink ignores virtual subsystem boundaries when determining block update order. Unconditionally executed subsystems are virtual by default.

Atomic blocks / subsystems: Simulink treats atomic blocks / subsystems as a unit when determining the execution order of block methods, executing the internal methods in a strict synchronous data-flow MoC (SDF) Model of Computation (MoC), i.e., all the inputs to the subsystem shall be updated at the same instant, and all the subsystem methods to update its outputs are invoked to update the output values at the same time.

Conditionally executed Simulink subsystems are atomic, e.g., event-triggered or enabled subsystems, function-calls, or Stateflow blocks (the latter is synchronous data-flow MoC container to execute a Discrete Events MoC MoC state machine).

An unconditionally executed subsystem can be designated as atomic to ensure that it is executed entirely before any other block is executed. This is useful to simulate certain types of execution, e.g., functions intended for a Time-Triggered (TT) runtime. However, changing the atomicity property modifies the behaviour of the model.

2.2.2 LabVIEW

Laboratory Virtual Instrument Engineering Workbench (LabVIEW) is a system-design platform and development environment for a visual programming language from National Instruments. The programming language used in LabVIEW, named G, is a dataflow programming language. Execution is determined by the structure of a graphical block diagram (the LabVIEW-source code) on which the programmer connects different function-nodes by drawing wires. These wires propagate variables and any node can execute as soon as all its input data become available. Since this might be the case for multiple nodes simultaneously, G can execute inherently in parallel. Multi-processing and multi-threading hardware is exploited automatically by the built-in scheduler, which multiplexes multiple OS threads over the nodes ready for execution.

2.2.3 SysML

SysML (Systems Modeling Language) [SysML] is a graphical modeling language for systems engineers based on a subset of UML2.0 with extensions such as new diagrams (e.g., parametric diagram) and modified diagrams (e.g., activity diagram). SysML supports the specification, analysis, design and validation of systems. It was motivated by the missing standard notation and semantic of UML. The newly introduced requirements diagram supports the specification of relationships between requirements using stereotypes (e.g., satisfy, derive, verify). The use case diagrams elaborate the interactions between external users and the system. They are expressed either from the point of view of the users or from the point of view of the system. Block definition diagrams describe the structure of a system in a hierarchical, tree-like fashion. In order to describe behaviour, sequence diagrams, state machines, and activity diagrams can be used. The parametric diagram is a new diagram that explains relationships between parameters (e.g., dependencies between variables).

The differences of SysML to the solution presented in this paper are similar to the ones for MARTE. SysML provides a language and does not constrain an implementation concerning a specific architecture.

2.2.4 Architecture Analysis and Design Language

The Architecture Analysis & Design Language (AADL) is an approved industry standard that has been developed under the guidance of the Society for Aerospace Engineers (SAE) [8]. Its core focus is modeling and model-based analysis of real-time embedded systems. Systems are

modelled in terms of components and their interactions, for which AADL distinguishes two classes of components: software components and execution platform components. Software components describe the software structure including the sequence of execution in the final system using threads, processes, sub-programs, and data.

The hardware of embedded systems is expressed in terms of execution platform components such as processors (execution of threads), memories (storage of code and data), buses (access among components), and devices (interaction with the environment). For specifying interactions between components, AADL provides ports (data, event, and event data port), which enable the directional exchange of data or events. Moreover, specialized connectors describing the access to a common shared resource such as a bus as well as for the interaction between sub-programs are defined in the AADL standard. In contrast to other modeling languages, e.g., such as UML, AADL specifies semantics for the standardized types, components and their interactions. This way, different tools have a common interpretation of AADL models, which eases the comparability of analysis results of different tools. AADL supports model interchange and tool chaining based on a standard XML/XMI definition. However, as explained for the MARTE UML profile, AADL provides the means for modeling and analysis of embedded systems but does not guide the system engineer in the way to design embedded systems. To our knowledge, AADL provides no mechanism to define meta-models for AADL that define what a valid model of an execution platform for a particular type of systems such as DECOS should look like.

2.2.5 Ptolemy

The Ptolemy project studies modeling, simulation, and design of concurrent, real-time, embedded systems. The focus is on assembly of concurrent components. The key underlying principle in the project is the use of well-defined models of computation that govern the interaction between components. A major problem area being addressed is the use of heterogeneous mixtures of models of computation. A software system called Ptolemy II is being constructed in Java. The work is conducted in the Center for Hybrid and Embedded Software Systems (CHESS) at the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley. The project is directed by Prof. Edward Lee.

The class of systems addressed by the project is sometimes called reactive systems. Reactive systems are those that interact with their environment at the speed of the environment. They are often embedded systems, and have been contrasted with interactive systems, which react with the environment at their own speed, and transformational systems, which process a body of input data to produce a body of output data. Reactive systems typically include elements of signal processing, communications, and real-time control. They are typically implemented with mixed technologies, possibly including embedded software, custom digital hardware, configurable hardware, analogue circuits, microwave circuits, and micro-electromechanical systems (MEMS).

A key principle in the Ptolemy project is the use of multiple models of computation in a hierarchical heterogeneous design environment.

A premise in this work is that no single general purpose model of computation is likely to emerge in the near future that will deliver what designers need. Modeling the diverse implementation technologies and their interaction is not reasonable within a homogeneous environment.

Ptolemy Project Objectives

The project aims to develop techniques supporting heterogeneous modeling, including both formal "meta-models" and a software laboratory for experimenting with heterogeneous modeling. In this context, it has explored methods based on data-flow and process networks, discrete-event systems, synchronous/reactive languages, finite-state machines, and communicating sequential processes. It has made contributions ranging from fundamental semantics to synthesis of embedded software and custom hardware.

The approach of the Ptolemy project at this time includes:

- Use of programming language concepts such as semantics, type theories, reflection, and concurrency theories in system-level design of electronic systems.
- Focus on domain-specific modeling and design problems so the designer can focus on the problem, not the tools.
- Emphasis on understanding of systems, which is promoted by visual representations, executable models, and verification.
- Use of Java, design patterns, **UML**, and a modern software engineering practice adapted to the realities of academic research.

Ptolemy software provides the laboratory for the experimental side of the project.

Ptolemy supports the interaction of diverse models of computation by using the object-oriented principles of polymorphism and information hiding.

For example, using Ptolemy software, a high-level data-flow model of a signal processing system can be connected to a hardware simulator that in turn may be connected to a discrete-event model of a communication network.

Since the Ptolemy Project began, numerous advances in semantics, design, simulation, and code generation have occurred. Results of the Ptolemy Project have included:

- adaptation of modern type theories to block-diagram-level specification of systems.
- timed extensions of communicating sequential processes (CSP) and process networks.
- a modular approach to synchronous/reactive design.

- a mathematical framework for comparing models of computation and studying their interaction.
- a bounded execution policy for process networks.
- formalization of computational models for data-flow.
- managing regularity in data-flow graphs using higher-order functions.
- automating the design of multidimensional multi-rate systems.
- simulating and scheduling multidimensional multi-rate systems.
- simulating heterogeneous systems.
- modular hierarchical finite state machines with various concurrency models.
- synthesizing embedded software from data-flow graphs.
- parallel scheduling techniques.
- optimizing interprocessor communication in parallel implementations.
- deriving fast algorithms for hardware/software partitioning of data-flow graphs.
- managing tool invocations and data dependencies in design processes.
- integrated heterogeneous design visualization.

The Ptolemy software environment has been used for a broad range of applications including signal processing, telecommunications, parallel processing, wireless communications, network design, investment management, modeling of optical communication systems, real-time systems, and hardware / software co-design.

Ptolemy software has also been used as a laboratory for signal processing and communications courses. Currently, Ptolemy software has hundreds of active users at various sites worldwide in industry, academia and government.

2.2.6 SystemC

SystemC [[IEEE1666-2011](#)], IEEE Std. 1666 is an ANSI standard C++ class library for system and hardware design, developed by the Accellera Systems Initiative [[Accellera](#)]. SystemC supports electronic system level design, by simulating of hardware-software systems.

Transaction level (TL)

TL denotes the abstraction level at which communication between concurrent processes is abstracted away from pin wiggling to transactions. This term does not imply any particular level of granularity with respect to the abstraction of time, structure, or behaviour.

Transaction level model (TLM):

TLM represents a model at the transaction level. Transaction level models typically communicate using function calls, as opposed to the style of setting events on individual pins or nets as used by RTL models.

This standard defines SystemC®¹ with Transaction Level Modeling (TLM) as an ANSI standard C++ class library for system and hardware design. The general purpose of this standard is to provide a C++-based standard for designers and architects who need to address complex systems that are a hybrid between hardware and software. The specific purpose of this standard is to provide a precise and complete definition of the SystemC class library including a TLM library so that a SystemC implementation can be developed with reference to this standard alone. This standard is not intended to serve as a user's guide or to provide an introduction to SystemC, but it contains useful information for end users.

SystemC models are hierarchical and executable. The models are composed of *modules* (called `SC_MODULE`) consisting of input/output ports, internal signals, concurrently running imperative processes and instances of other blocks. The execution of models is governed by the discrete-event (DE) driven simulation engine provided by SystemC.

In order to perform simulations, SystemC relies on the discrete notion of time with a configurable time granularity (from femtoseconds to seconds). This time might be used to provide a global notion of time among the components of SystemC models. Events are instantaneous in SystemC, and simulation time of processes is zero. Therefore, the processes triggered by a given event are executed sequentially, based on the concept of the 'delta-cycle'. The delta-cycle is basically an infinitesimal physical duration that does not advance simulation time, which is typically used to perform a sequential simulation of simultaneous tasks in zero simulation time, giving the illusion of a concurrent simulation of simultaneous processes.

One of the main strengths of SystemC is that hardware and software components can be described in a common language. What is more, both HW and SW components can be indistinguishable at the beginning of the design, since they are assigned to abstract modules that are only later do refined as HW or SW components. Due to its capability to describe both HW and SW components in a unique language, and its ability to simulate concurrent processes, SystemC has nowadays become the de-facto standard in HW/SW system development.

¹SystemC is a registered trademark of the Accellera Systems Initiative.

Each module of SystemC can express a set of processes, via the `SC_METHOD()`, `SC_THREAD()` and `SC_CTHREAD()` commands. These processes are distinguished as explained in the following:

- `SC_METHOD`: Every time the simulation calls one of these processes, they run from the beginning to the end and they return. They are not allowed to suspend or to be interrupted, so they are considered as atomic functions.
- `SC_THREAD`: Processes of this type are started only once by the simulator. Once the thread starts to execute, it controls the simulation until a `wait()` statement is found, which suspends the simulation and gives the control back to the simulator. Hence, `SC_THREAD` processes usually contain an infinite loop containing one or more `wait()` statements. It is also possible to terminate the thread by using a `return` statement.
- `SC_CTHREAD`: This process type is a variant of the `SC_THREAD` process, with the variation that imposes the process to be sensitive to clock edges. In addition to `wait()` statements, `SC_CTHREAD` processes can use `wait_until()`, which is equivalent to repeat `wait()` functions in a loop until a certain condition holds.

In industry, it is infrequent that all modules within a system are modelled at the same level of abstraction simultaneously. Instead, commonly, different models of abstraction are required within a given system during its development, for example:

- A designer may use a very detailed model for a design under test but a very abstract model for the generation of the stimuli for the system.
- With a very detailed model as a starting point, the designer might create a more abstract model in order to increase simulation speed when testing another part of the system.

In order to deal with this issue, SystemC allows modeling systems at diverse levels of abstraction, and even enables designers to model subsystems of a given system at different abstraction levels, such as the register transfer level (RTL) or the more abstract transaction-level model (TLM) [Ghe06]. Hence, SystemC does not impose a top-down or a bottom-up design flow. Besides abstraction, SystemC also provides the other two model simplification strategies:

- **Abstraction**: SystemC enables the designers to define hierarchical modules in order to hide or show the internal details of each component as desired.
- **Partitioning**: Communication and computation concerns are strictly decoupled in SystemC. Furthermore, partitioning may also be applied in the models by defining independent sub-modules for different aspects of the functionality of systems. The overall functionality is then achieved by the interaction of such sub-modules.

- **Segmentation:** SystemC enables simulating the resulting functionality of systems containing simultaneous sub-modules, by running them sequentially and making use of the concept of the delta-cycle.

Although the simulation engine of SystemC is natively event-triggered, different extensions have been made to it in order to support different MoCs. Some of them are briefly introduced in the following:

- **SystemC-AMS** [[IEEE1666.1-2016](#)]: SystemC-AMS (SystemC - Analog Mixed Signal) is an extension to SystemC that enables the developers to model and simulate analog and mixed (analog and digital) signals, so that continuous-time models and discrete-continuous heterogeneous models can be simulated and verified.
- **SystemC-H** [[PS04](#), [PS06](#)]: This extension to the SystemC language focuses on the design of heterogeneous models. It provides a simulation kernel that is capable of modeling and simulating parts of the system based on the concurrent sequential processes (CPS), finite state machine (FSM) and synchronous data-flow (SDF) MoCs.
- **HetSC** [[HSV05](#), [HV06](#), [HV08](#), [HetSC](#)]: HetSC is another approach for the modeling of heterogeneous systems in SystemC. It is essentially a library of MoC-specific communication channels that can be introduced in SystemC models to simulate the behaviour of systems based on different MoCs. Among others, the library includes communication channels for Kahn process networks, synchronous-reactive models or synchronous data-flow models.
- **HetMoC** [[ZSJ10](#)]: HetMoC is a formal heterogeneous framework for the development of multi-MoC systems in SystemC developed at the KTH Royal institute of Technology. This framework enables designers to design a network of heterogeneous processes that communicate among themselves by the exchange of signals via FIFO channels. Different domains are integrated by the use of domain-specific interfaces. The HetMoC framework provides interfaces for SR, DE, continuous and untimed models of computation.
- **SystemC-MDVP** [[FWI+14](#)]: The SystemC Multi Domain Virtual Prototypes (SystemC-MDVP) is a language extension to SystemC-AMS, developed in the CATENE CA701 H-INCEPTION project [[HINCEP](#)]. Therefore, analog-mixed signal simulation will be handled by SystemC-AMS, and other physical domains such as microfluidic systems will be supported by defining and integrating the necessary MoCs in the language.

Assertion Verification on SystemC TLM models

Assertion Based Verification (ABV) is a functional verification technique to detect design bugs, widely used in electronic systems verification [[TSP+09](#)]. An assertion is a directive to

check a property on a model of the system. Although ABV is applied on SystemC executable models at RTL, recently ABV abstraction is being raised to the Transaction Level (TL).

Aim The objective is to reduce the development cost in design flows with team redundancy from the initial specification (independent design and verification teams).

Method The approach consists of an automated verification framework that integrates with the SystemC executable model without embedding the verification code into the design model. The same stimuli inputs are reused across different abstraction levels.

Contribution A TL verification framework supported by a library built on top of SystemC.

Applications Simulation of an elderly people care system (CONFIDENCE). An Impulse-Radio Ultra-Wide Band (IR-UWB) is modelled in Matlab, the system TL models are based on the SAYY library, and assertion-based verification is applied on the executable models. The CONFIDENCE system is modelled hierarchically by means of nested models of different levels of detail.

Verification by embedding Matlab in SystemC TLM

SystemC is a standard open source class library written in ANSI C++ language for creating executable models of electronic systems [TSA+09a]. High abstraction models for the early system definition are addressed by means of un-timed or approximate-timed transaction level modeling techniques (TLM). TLM models allow high simulation speed.

Matlab is used for the design of executable specifications of algorithms.

Aim The objective is to reduce the development cost by using a cooperative software framework for early system architecture exploration by different design teams using diverse tools.

Method The approach consists of the functional verification of algorithms expressed in the MATLAB language by co-simulation with SystemC executable models of the system architecture.

Contribution A C++ abstraction class library (MatlabEngine++) that wraps Matlab's C API as a singleton for embedding a Matlab Engine in SystemC executable models.

Applications Simulation of an elderly people care system (CONFIDENCE) (see also ABV).

System Behaviour Capture: from UML to SystemC

While SystemC is the standard for creating executable models of electronic systems, Unified Modeling Language (UML) is widely used for software engineering. For abstract system conception and definition an UML subset that consists of use case diagrams and state machine diagrams suffices [PSU+08].

Aim The objective is to provide a smooth transition from **UML** system functional description to a SystemC executable model.

Method The approach consists of a basic transaction class library compliant to the SystemC TLM 1.0 standard, implementing a set of architectural patterns for Actors and Entities.

Contribution A SystemC transaction class library SAVY, with a **UML** description.

Applications Not described.

2.2.7 Languages for Modelling Arguments

A *safety case* specifies the interpretation and implementation of safety requirements in a system, including the engineering decisions and rationale to show the safety achievements. This information is mainly expressed in textual language, which suffices for simple safety cases. However, for complex safety requirements, pure textual language descriptions often lead to unclear, unstructured and ambiguous safety cases.

The problem of formalizing arguments was tackled in the philosophy essay from Toulmin [Tou58]. Toulmin made an analogy between logic and jurisprudence, focusing on the critical function of the reason, and took as a model the structure and procedures of law-suits. Toulmin introduced a graphical notation and also provided a generic argument pattern which supported the construction of hierarchies of arguments. Graphical representations are important in managing complexity, as these facilitate the transmission of complex ideas among individuals.

The idea of structuring logical arguments to demonstrate the actual achievement of a claim suits the intent of safety cases, i.e., the justification through evidences that a system is acceptably safe for a specific application in a specific operating environment. Following the Toulmin approach, several other argument notations were developed. There exist several languages that have been used to model arguments for safety cases, for instance:

1. Argumentation Meta-model (**ARM**) [ARM], nowadays subsumed in Structured Assurance Case Meta-model (**SACM**).
2. Goal Structuring Notation (**GSN**) [GSN], that found a widespread application in aerospace safety cases, with a number of software tools supporting **GSN**.
3. Claim Argument Evidence (**CAE**) [CAE], already used in **DREAMS** modular Safety Cases [DRE511, DRE512, DRE513].
4. Structured Assurance Case Meta-model (**SACM**), where the formal version of **SACM** is a combination of Argumentation Meta-model (**ARM**) and Software Assurance Evidence Meta-model (**SAEM**) documents [SACM].

At this point, we reflected on the best language to describe, store and compose argument models as intended, bringing these conclusions:

1. **ARM** is deprecated, and **SACM** should be used instead of ARM.
2. **GSN** has been in use for a while, and it has been enhanced and extended to support the certification arguments in many ways, e.g.:
 - Habli and Kelly introduced **GSN** extensions to support a safety case approach [HK10];
 - Hutchesson and McDermid [HM13] applied **GSN** for developing trusted product lines.
 - Denney et al. [DPP12] developed AdvuCATE, a tool to automate building certification arguments.
 - Denney et al. [DPH11] extended the **GSN** with annotations for confidence measurement, to handle also possible subjective criteria w.r.t. the strength or credibility of the solutions supporting the argument. This enables the integration of complimentary information, e.g., the ‘strength’ credited to the evidences by a reviewer to support an argument (e.g., the position paper CAP 760 [CAP760] states the supporting strength associated to evidences generated by specific arrangements of Verification and Validation (**VnV**) techniques).
 - **GSN** usage is recommended to structure the arguments in the aerospace domain [CAP760], with **SC** development guidelines [SCDM06].
3. **CAE** is a proprietary language, with limited tool support [ASCE, CERTWARE].
4. **SACM** provides more expressiveness than **GSN** or **CAE**, which eases the automation and integration of information; on the other hand, it does not define a graphical depiction of the arguments, where it is a useful feature of **GSN** or **CAE** to present the argument in a reader-friendly format. Another drawback of **SACM** is the novelty of the specification, which makes **SACM** unstable and could bring forward maintainability issues for the models or tools.

2.3 Safety Systems

This section summarizes safety-related terms used in this document.

2.3.1 Safety Integrity Level (SIL)

The standard IEC-61508:2010 [IEC61508] defines a Safety Integrity Level as “*a discrete level (one out of a possible four), corresponding to a range of safety integrity values, where safety*

integrity level 4 has the highest level of safety integrity and safety integrity level 1 has the lowest.”

Table 2.1: Domain-specific Safety Integrity Levels

Domain	Standard	Domain-Specific Safety Levels				
Generic	IEC-61508	–	SIL1	SIL2	SIL3	SIL4
Automotive	ISO 26262	QM	ASIL-A	ASIL-B/C	ASIL-D	–
Machinery	ISO 13849	PLa	PLb/c	PLd	PLe	–
Railway	EN 50126/128/129	–	SIL1	SIL2	SIL3	SIL4
Aviation	DO-178/254	DAL-E	DAL-D	DAL-C	DAL-B	DAL-A

ASIL: Automotive Safety Integrity Level / DAL: Design Assurance Level / PL: Performance Level / QM: Quality Management / SIL: Safety Integrity Level

2.3.2 Systematic Capability

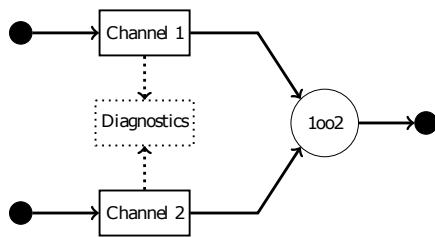
The standard IEC-61508:2010 [IEC61508] defines a systematic capability as a “*measure (expressed on a scale of SC 1 to SC 4) of the confidence that the systematic safety integrity of an element meets the requirements of the specified SIL, in respect of the specified element safety function, when the element is applied in accordance with the instructions specified in the compliant item safety manual for the element.*”

Architectures for low demand mode of operation

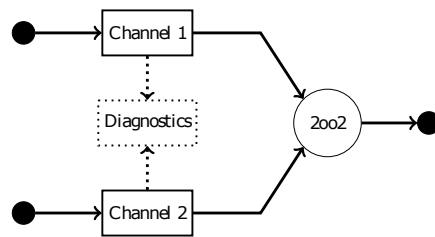
- *1oo1*: This architecture consists of a single channel, where any dangerous failure leads to a failure of the safety function when a demand arises.
- *1oo2*: This architecture consists of two channels connected in parallel, such that either channel can process the safety function (see Figure 2.3a). Thus there would have to be a dangerous failure in both channels before a safety function fails when demanded. It is assumed that any diagnostic testing would only report the faults found and would not change any output states or change the output voting.
- *1oo2D*: This architecture consists of two channels connected in parallel. During normal operation, both channels need to demand the safety function before it can be activated. In addition, if the diagnostic tests in either channel detect a fault then the output voting is adapted so that the overall output state then follows that given by the other channel. If the diagnostic tests find faults in both channels or a discrepancy that cannot be allocated to either channel, then the output goes to the safe state. In

order to detect a discrepancy between the channels, either channel can determine the state of the other channel via a means independent of the other channel. The channel comparison / switch over mechanism may not be 100 % correct in its operation therefore a parameter 'K' represents the efficiency of this inter-channel comparison / switch mechanism, i.e., the output may remain on the 2oo2 voting even with one channel detected as faulty.

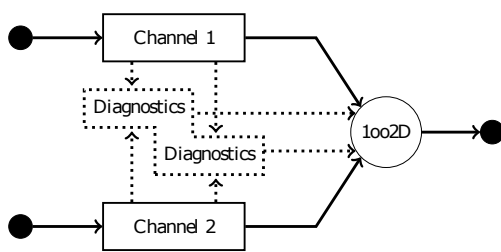
- 2oo2: This architecture consists of two channels connected in parallel so that both channels need to demand the safety function before it can take place (see Figure 2.3b). It is assumed that any diagnostic testing would only report the faults found and would not change any output states or change the output voting.
- 2oo3: This architecture consists of three channels connected in parallel with a majority voting arrangement for the output signals, such that the output state is not changed if only one channel gives a different result which disagrees with the other two channels. It is assumed that any diagnostic testing would only report the faults found and would not change any output states or change the output voting.



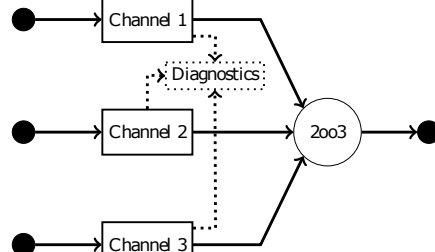
(a) Block diagram for a 1oo2 architecture.



(b) Block diagram for a 2oo2 architecture.



(c) Block diagram for a 1oo2D architecture.



(d) Block diagram for a 2oo3 architecture.

Figure 2.3: Safety architectures.

(source: [IEC61508])

2.3.3 Fail-Safe and Fail-Operational Systems

Safety-critical systems are classified as *fail-safe* and *fail-operational*, depending on the system actions upon failures:

Fail-safe system: A *fail-safe* system is a dependable system with a set of pre-defined safe-states that the system should reach when a component fails. Once in the safe state, the system operation remains disabled to prevent accidents.

Fail-operational system: A *fail-operational system* is a dependable system that shall remain operational in the event of failures, although it would provide its services in a degraded-mode. A typical example of a fail-operational system is an aircraft flight control: there is no reachable safe-state during flight, therefore the system has to keep the control of the airplane until landed.

2.4 Virtual Devices

This section introduces commercially used terms to designate virtual embedded devices.

2.4.1 Virtual ECU (V-ECU)

A Virtual **ECU** (**V-ECU**) [**V-ECUa**] is software that emulates a real Electronic Control Unit (**ECU**) in a simulation scenario. The **V-ECU** comprises components from the application and the basic software, and provides functionalities comparable to those of a real **ECU**. A **V-ECU** usually has the same software components as the finished **ECU**, representing the real **ECU** realistically.

The abstraction level of a V-ECU depends on its application case:

- **V-ECU** for developing a single **ECU** function (contains selected parts of the application software;
- **V-ECU** at application level (application software components, operating system).
- **V-ECU** including parts of basic software (application software components, operating system, hardware-independent basic software such as **DEM**, **NVRAM**, **ECU** state manager, **COM**, etc.).

2.4.2 Restbus Simulators

The term *Restbus simulator* denotes a test component that emulates the messaging behaviour of a group of devices communicating with a system under test.

2.5 Motion Sensors

This section describes a kind of sensors commonly used to control moving parts of a system. The case studies presented in this dissertation consist of systems that shall operate moving parts in a safely way: a train speed supervision system, an elevator simulator for elevator control systems, and a wind turbine over-speed protection system. The three considered applications implement safety functions that rely on measurements of the position or the speed of the moving part relative to some reference system. Among the diverse position sensing technologies, encoders are common to the three case studies. Regardless the encoder is included as a part of the system under study or is considered as an external device, it shall be also considered for the examination of the overall dependability.

An **encoder** is an electromechanical device that can measure motion or position. Most encoders use optical sensors to provide electrical signals in the form of pulse trains, which can, in turn, be translated into information about motion, direction, or position.

Rotary encoders are used to measure the rotational motion of a shaft. An **optical rotary encoder** consists of a light-emitting diode (LED), a disk, and a light detector on the opposite side of the disk. The disk has patterns of opaque and transparent sectors coded into the disk. The disk is mounted on the rotating shaft, so that as the disk rotates, the opaque segments block the light and, where the glass is clear, light is allowed to pass. This generates square-wave pulses, which can then be interpreted into position or motion.

Linear encoders work under the same principle as rotary encoders but using a stationary strip instead of the rotary disk. In linear encoders the position-detector assembly is usually attached to the moving body. Linear encoders can use different measuring technologies: optical (as optical rotary encoders), magnetic, etc.

As a single set of pulses does not indicate the motion direction, most commonly found encoders use two code tracks with sectors positioned 90 deg out of phase. These are called **quadrature encoders**. Therefore, by monitoring both the number of pulses and the relative phase of the output signals (usually named 'A' and 'B') from a quadrature encoder, both the position and direction of rotation can be tracked.

Encoders can be incremental or absolute. An **incremental encoder** only measures changes in position (from which you can determine velocity and acceleration), but it is not possible to determine the absolute position of an object. On the other hand, an **absolute encoder** is capable of determining the absolute position of an object. An absolute optical encoder has alternating opaque and transparent segments like the incremental encoder, but the absolute encoder uses multiple groups of segments that form concentric circles (in rotary encoders) or parallel tracks (in linear encoders). Each segment group has a different number of segments, one segment doubling the number of binary encoding from the prior. Absolute optical encoders have a separate light source and receiver for every segment group, so that the readings form a binary counting system. Regardless of the measuring technology, absolute encoders have to timely transmit a position value by some means.

Incremental encoders have some disadvantages. It is possible that the measurement

is continuously invalid due to signal glitches, unmeasured impulses or similar problems. Furthermore, after a loss of the supply voltage it is often necessary to return to a reference point, to reset the estimated position to a known value. For these reasons absolute encoders are often used in applications requiring high precision or where it is complicated or impossible to return to a reference point .

2.5.1 Encoder Interfaces

Incremental encoders are usually connected to a logic counting circuit, that may also decode the motion direction or even compute the speed of the moving part. Absolute encoders require parallel interface or a communication bus to transmit the position information.

Incremental Encoder Interfaces

According to the signal interface, incremental encoders can be single-ended, where the A and B signals are both referenced to ground requiring a single wire per signal, or differential, where there are two lines per each A (named A and A') and B signal (respectively called B and B'). In differential encoders all four lines are always supplying a known voltage (either 0 V or V_{cc}): when A is V_{cc}, A' is 0 V, and when A is 0 V, A' is V_{cc}. In the case of a single-ended encoder, A is either V_{cc} or it floats. Differential encoders are often used in electrically noisy environments because taking differential measurements protects the integrity of the signal.

Figure 2.4 depicts the pulse trains generated by a quadrature encoder with differential output interface. The top plot represents the position of the moving part to which the encoder is attached. The middle plot represent the scaled [A, B] output signals modulated by the encoder. As shown in the figure, the frequency of the pulses is proportional to the motion speed. When the direction of the motion reverses, then it also reverses the relative phase of signals A and B. The plot at the bottom represents signals A' and B', that are the logical negation of signals A and B.

The outputs from the quadrature encoder are connected to an edge counter circuit, which converts the pulse trains to a position value. For instance, signal A can be connected to the counter source terminal, making this the signal from which the pulses are counted. Signal B is then connected to the up/down terminal, to select whether a pulse edge increments or decrement the counter value. The current count is stored in a digital register that is instantly available for evaluation.

The process by which edge counts are converted to position depends on the type of encoding used. There are three basic types of encoding, X1, X2, and X4.

1. **X1 Encoding:** When channel A leads channel B, the increment occurs on the rising edge of channel A. When channel B leads channel A, the decrement occurs on the falling edge of channel A.

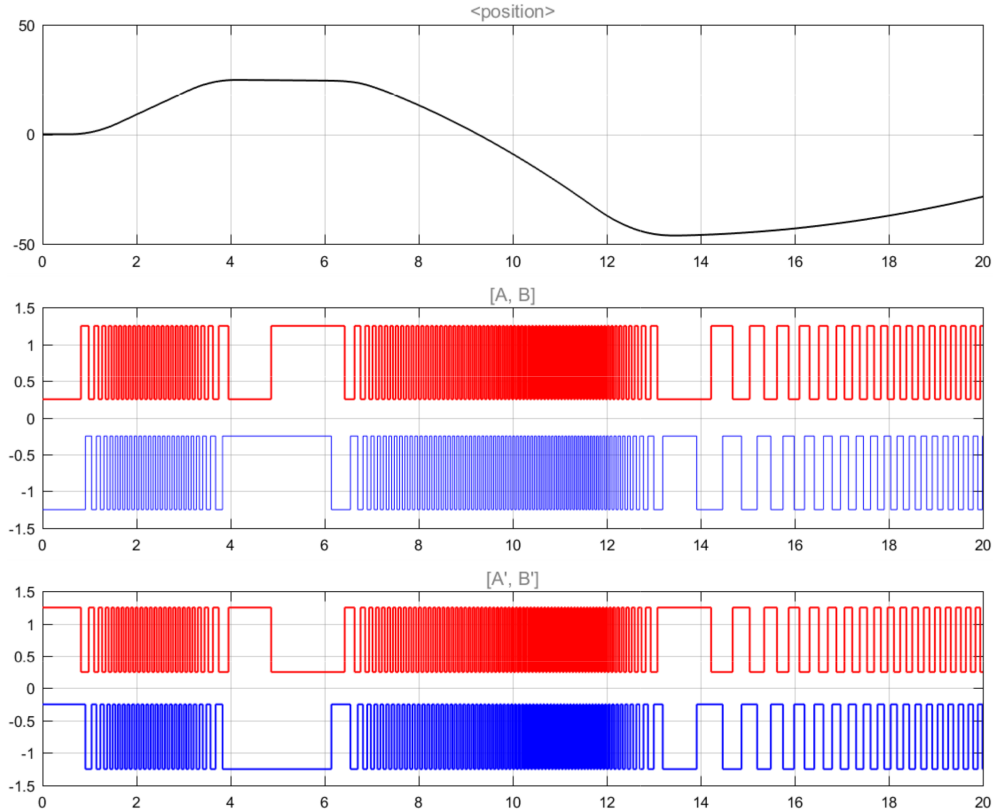


Figure 2.4: Quadrature encoder input (position) and outputs (A, B, A', B')

2. **X2 Encoding:** The same behaviour holds for X2 encoding except the counter increments or decrements on each edge of channel A, depending on which channel leads the other. Each cycle results in two increments or decrements.
3. **X4 Encoding:** The counter increments or decrements similarly on each edge of channels A and B for X4 encoding. Whether the counter increments or decrements depends on which channel leads the other. Each cycle results in four increments or decrements.

Absolute Encoder Interfaces

Absolute encoders read a position code that relates to an absolute numerical value. Thus, the position value is always directly available, and counters are not necessary. In addition it is not possible to get continuously invalid values caused by interferences or loss of the

supply voltage. Movements which are done while the system is turned off are immediately measured after the system is powered up. The absolute position value can be transmitted to the control system by different means:

1. **Bit parallel interface:** All bits of a position are transferred simultaneously using one line for each bit. This interface is fast and provides a suitable data transmission for low resolution applications. However, the cabling with bit-parallel interface becomes problematic for high resolutions, longer cable lengths, or multiple-axis machines. In those applications other methods of data transmission are more favourable.
2. **Synchronous-serial-interface (SSI):** The synchronous-serial-interface is based on the RS 485/RS 422 transmission standards, implementing the data transmission with one 6-wire cable: one twisted pair line for the data; one twisted pair line for the clock, and two wires for the power supply of the encoder. The balanced transmission provides high noise immunity; crosstalk on the line does not effect the signals. The twisted pair lines are sufficient for the transmission. Extremely high noise immunity is achieved when shielded twisted pair lines are used. There are also encoders with other interfaces similar to SSI, e.g., HIPERFACE or EnDat.
3. **Fieldbus Interfaces** Field buses enable the communication of automation components among themselves. Bus systems are increasingly used for automation, enabling the design of system concepts incorporating decentralized solutions, in particular open field bus systems such as Profibus, Interbus and **CAN** bus:
 - **Profibus:** A Profibus system consists of one or more masters and one or more slaves, which are connected by bus cables and bus plugs. The master is usually realized as a connecting module at the control system. Typical slave devices are sensors, actuators, transducers or display elements. Thus, an encoder operates on the Profibus as a slave. There are mandatory encoder profiles, which are called class 1 and class 2. The selected device class determines the specifications of the encoder, as well as the length of the input and output data. Absolute class 1 encoders cannot be parametrized, whereas class 2 encoders can be parametrized. Configuration is done by the means of a GSD file (electronic data sheet), that determines the address of the device (which identifies the encoder exactly) and the device class. Encoders can implement additional manufacturer-specific functions (e.g., velocity output) that can also be selected during the device configuration. Once the device configuration is set, the appropriate parameters (e.g., resolution, direction of rotation, software limit switch, etc..) are saved in a database and transferred to the encoder when starting the system. Data can be read from the encoder (e.g., position value) or written to the encoder (e.g., pre-set value) using the input and output addresses determined in the configuration.

- Interbus: INTERBUS is an open sensor/actuator bus system with one master and several slaves, called *users*. An INTERBUS system conforms to a ring structure. Beginning at the master (PLC or IPC), the bus system connects the respective control or computer systems to the peripheral input and output modules. The individual users are connected by an installation remote bus cable. The address of the individual users is given by their physical position on the bus. The absolute encoder is a *remote bus user*. There are three standardized INTERBUS encoder profiles: (a) K1 (not programmable; 16-bit process data); (b) K2 (not programmable; 32-bit process data); and, (c) K3 (programmable; 32-bit process data).
- CAN bus: Controller Area Network (**CAN**) is a multi-master bus system, i.e., all users can access the bus at any time as long as it is free. CAN operates with message identifiers. Access to the bus is performed according to the CSMA/CA principle (carrier sense multiple access with collision avoidance), i.e., each user listens if the bus is free, and if so, is allowed to send messages. If two users attempt to access the bus simultaneously, the one with the highest priority (lowest identifier) receives the permission to send. Users with lower priority interrupt their data transfer and will access the bus when it is free again. Messages can be received by every participant. Controlled by an acceptance filter the participant accepts only messages that are intended for it. There are encoders implementing different **CAN** protocols: DeviceNet, CANopen, CANopen Lift (specific variant for elevators), CANopen Safety (variant for safety-critical applications), etc.
- Ethernet: Besides the above mentioned buses, manufacturers provide also encoder variants with Ethernet interfaces, e.g., EtherCAT, Ethernet/IP, ProfiNet, Powerlink or Modbus/TCP.

3

State of the Art

Currently embedded software implements the bulk of functionality and features in programmable electronic products. The development and verification processes for dependable programmable electronics has become the main limiting factor for shortening the time to market [CFS05]. Embedded systems ubiquity in a broadening variety of products and applications, the increasing functional complexity of the systems and the pressure to shorten the time to market for embedded systems demand a highly efficient development processes. While aiming at lower development costs, sectors like automotive electronics or software engineering progressively introduced design tools with coding automation capabilities. This development paradigm is known as Model-Based Development (MBD) or Model Driven Development (MDD). The key of MBD / MDD is in achieving a better consistency of the built system with regard to the specifications, improve re-usability of knowledge and components, and better technical documentation about the system under design or under analysis by means of a set of models that comprise the relevant information about the system.

3.1 Testing Objectives

The ability to numerically express the testing objectives is a desirable prerequisite for the automation of the test design process. The rationale behind the research on testing for dependable embedded systems, and particularly for safety-related applications, is that the increasingly demanding certifications require objective and reproducible verification processes that should be assessed.

There are testing design approaches based on the quantification of the test objective, thus providing a scientific background for the problem of functionally expressing the fitness of a test suite by a number. One of them is the "evolutionary testing" technique that is based on the transformation of the test design aim to an optimization problem. A refinement of the test suites is carried out by applying meta-heuristic search algorithms, that improve the suitability of the test scenarios for the specific test objective. This motivates the analysis of the state of the art in evolutionary testing as a basis for further improvement of the metrics available for the test design.

Regarding the test objectives, this section recalls the test taxonomy used in [WSB01] as a common understanding of the test problems related to embedded systems verification. The test problems are classified in the following groups, depending on the test objective:

Functional testing: The test objective is to define a set of test scenarios for verifying the built system functions as specified in the requirements.

Robustness testing: The test objective is to exercise the fault tolerance mechanisms implemented in the system.

Safety testing: The test objective is to find operational scenarios where the system is working out of a predefined safety boundary. The safety boundary consists of a set of constraints on timing or variables derived from the safety requirements.

Structural testing: In structural software testing the aim is to find a set of test scenarios that exercise the maximum amount of code and cover all the possible control flows, in order to thoroughly verify the system in every possible state, according to some metric that refers to the code structure.

Time-behaviour testing: The test objective is to find operational scenarios where the system time constraints could be violated. A typical assumption is that the likelihood of misbehaviour is greater when the components of the system run faster or slower than the nominal reaction time, therefore the time-behaviour testing normally focuses on scenarios where the system components operate at their best and/or worst-case execution times (BCET/WCET).

3.2 Interrelationship between the Development Process Models and Testing

The overall testing process must be adapted to the development paradigm used. Figure 3.1 represents the conventional waterfall (also known as V-model) development paradigm. In the V-model, activities in earlier phases of the project define the verification procedures and plans concurrently with the specification and design. The main drawback is that there

is a significant lag between the System Level Testing Specification and the System Level Verification, as for the latter we require a completely assembled system.

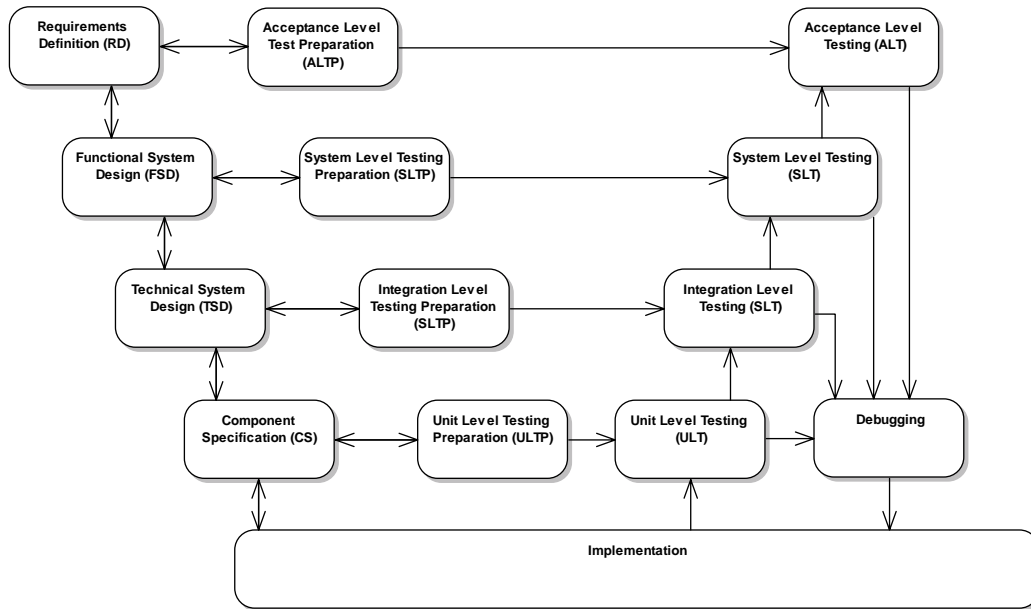


Figure 3.1: Waterfall Development Process (V-model)
(source: [CF09])

When the system is developed following a model-based approach, the availability of executable models enables testing at earlier stages, as the adapted V-model in Figure 3.2 shows.

Figure 3.3 outlines the differences in test objects and test levels between model- and code-based development processes.

Taken to the extreme, in a Spiral Development Process, Testing and Design are carried out almost simultaneously (see Figure 3.4). Although this Incremental Development differs substantially from the conventional V-model in Figure 3.1, this process suits the pure top-down approach initially devised in [Dij68].

3.2.1 Product Development Process Models

Projects are temporary allocations of resources commissioned to achieve a desired result. A Product Development (PD) project consists of many multi-functional activities working altogether to produce the information that will reduce the risk of the outcome being something other than the project stakeholders desire. The unclear path to a project's

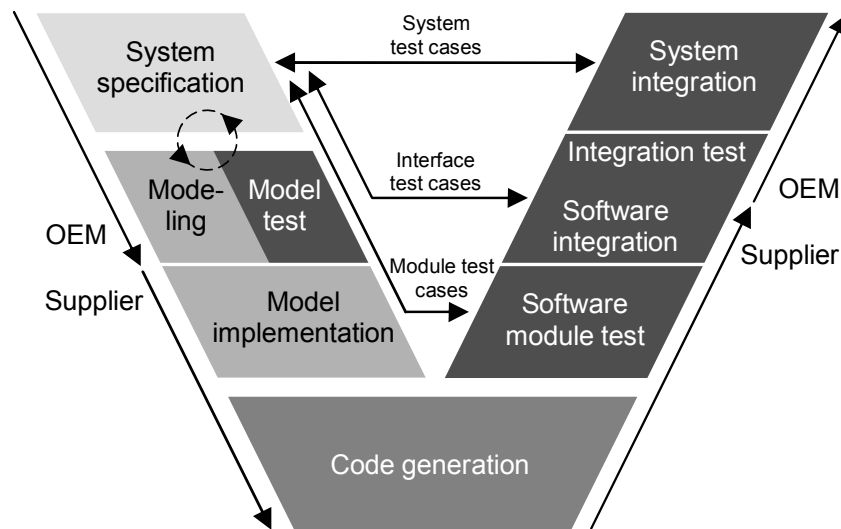


Figure 3.2: MDD Process at Daimler
(source: [MSR+10])

goal can be described in terms of process uncertainty and ambiguity. Ambiguity is a source of risk for PD projects. PD projects capable of co-evolving with their environments and stakeholder needs can profit from the accelerating pace of change in market needs.

PD process modeling can help a project manager to understand the feasible "design space" for the project, a set of process architectures called the process space.

Iteration occurs when the cumulative output deliverables of prior activities, plus the assumptions that can be reasonably made at the time, are insufficient to enable subsequent activities to add appropriate value to the project.

Iteration is a managerial control option and it will be exercised when it provides the path of greatest added value to a project.

Product performance can be represented as a vector of attributes, each measured by one or more technical performance measures (TPMs) that can be seen as one aspect of its overall performance value. The TPMs define the technical performance level of a project.

The execution of project activities: 1) uses resources; 2) creates deliverables that can revise one or more TPMs; and 3) thereby adjusts the state (and value) of a project. A PD process may be modelled as a Cycle Accurate Simulator (CAS), where activities are agents, deliverables imply their connections, and a process path emerges from the application of simple rules for activity selection and deliverable flow. The fitness of (value provided by) this process will depend on the dynamic state of the project (duration, cost, and technical performance) and its environment (represented by project goals). Adaptability in PD projects is facilitated by advance knowledge of the potential activities and their relationships (planning) and their rules for combination (work policy), because this enables

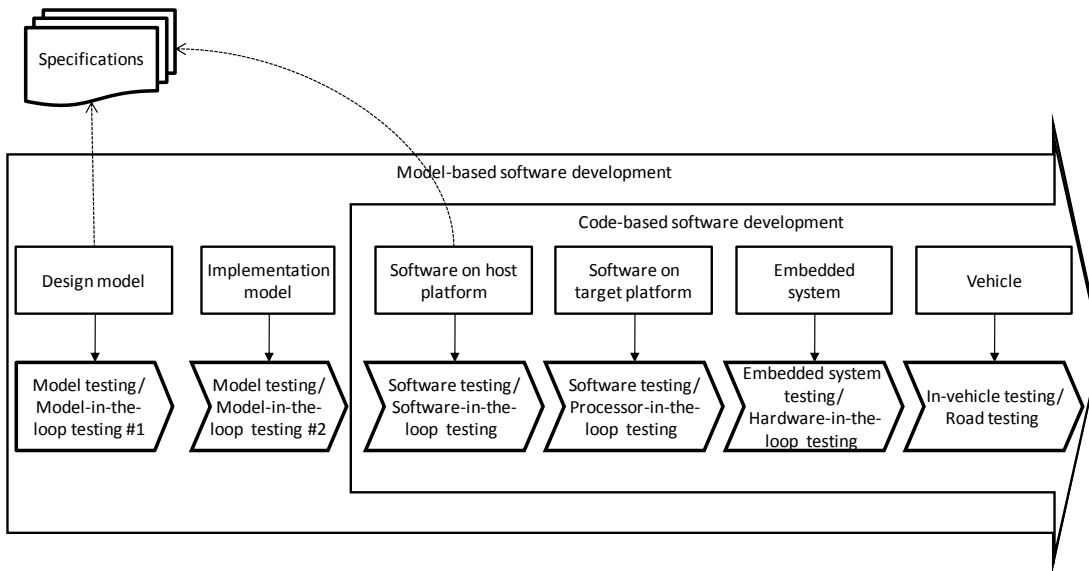


Figure 3.3: Test objects and test levels for model- and code-based control development

the activities to be quickly and effectively re-evaluated and reorganized over the course of a project.

Systematic project control entails: 1) the synchronization of internal and external data regarding the state of the project; and 2) the use of those data in making decisions on project changes.

3.3 Test Automation for Safety Critical Systems

In [Pel96] Peleska analyses the verification of dependable systems with emphasis on the safety aspect. Safety is a fairness property, thus detecting all the possible violations of the property will theoretically require an infinite number of testing executions. In contrast, the testing objective in practice is to find errors within a finite execution time. In practice, the conventional manual analysis of test outputs is not sufficiently trustworthy, as errors could be easily overlooked by humans. Also, as the system complexity grows, the test coverage required for safety systems would be unmanageable and unattainable with acceptable costs. A way to prevent this is to automate the test generation and test execution for safety critical reactive systems. This can be achieved by a translation of the test specifications to a formal language description upon which formal verification transformations can be applied. A key point is the trustworthiness of the test tool, and the verification suite required to certify such a framework.

Despite the introduction of formal methods, testing remains mandatory for verifying the

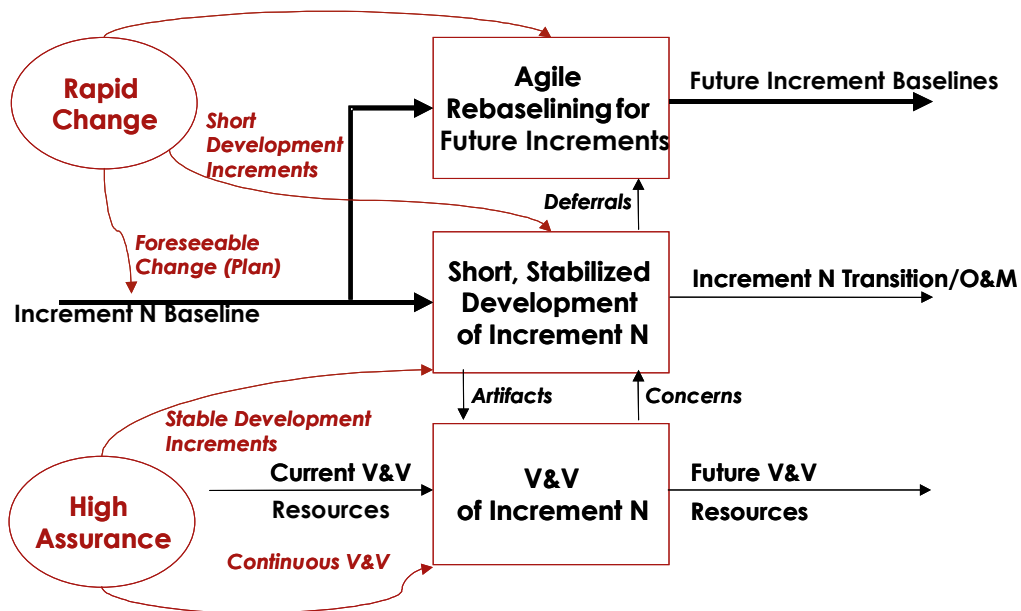


Figure 3.4: Incremental Development Process (Spiral Model)
(source: [Boe11])

behaviour on system level for safety-critical systems:

Formal development and verification usually cover partially the range of software and hardware components.

Black-box testing remains as the only means to check the correctness of system integrating third party components, because the vendors normally will not disclose the implementation details for external conformance assessment,

Safety-critical systems have to integrate several built-in self-test features to detect hardware wearing.

Acceptance testing is preferable to merely review of documents.

The general problem of testing against arbitrary types of specifications cannot be solved in a fully automated way. But it becomes tractable if limited to well-defined restricted classes of systems and test objectives, as the hardware-in-the-loop testing of reactive systems. The aim is to create "implementable instantiations" of theoretical results for the construction of dependable tools. One theoretical approach is Hennessy's testing methodology, based on un-timed CSP process algebra.

Logical Partition of a Test Automation System

In general, a Test Automation System integrates the following building blocks:

Test Generator: Creation of test cases from specifications

Test Driver: Interprets the test cases provided by the test generator and controls their execution on the target system. The test driver exerts stimuli on the input channels of the system under test and collects output data to analyse the behaviour.

Test Evaluator (Oracle): To verify the observed test execution against a specification and to decide if the execution is correct. The specification for test evaluation can differ from the specification used for test generation.

Test Monitor: To observe the test execution and decide whether all the relevant executions for a test case have been exercised on the target, and to decide if the so far executed test cases suffice for the required coverage. In many cases the black-box observations of the system interface alone are insufficient for the tasks of the test monitor. A Test Monitor usually requires additional channels for obtaining information about the internal state of the system.

Testing Terminology

Some common terms used to denote verification and validation activities are:

Validation denotes the process of determining the completeness of the requirements and their conformance to the customer's demands. This process is normally based on partial information, and thus cannot be fully automated.

Verification refers to the evaluation of development products in order to assure their consistency with the relevant reference documents. As opposed to validation, Verification relies on preceding information.

Formal Verification denotes verification done by mathematical reasoning, based on reference information described in a language with a precise semantics.

Testing is the execution of implemented system components providing data at their input interfaces while monitoring the component behaviour, to verify that the components satisfy the specified requirements for the sets of input data applied. Simulation denotes an abstract testing based on the symbolic execution of a specification with abstract input data.

3.3.1 Fault Injection

According to Avizienis et al. [ALR+04], a failure is an event that occurs when the delivered service deviates from the correct service. An error is a deviation from the correct service of an external state of the system, and a fault is the cause of an error. In other words, errors are the manifestations of faults whereas failures are the consequence of errors. Therefore, injecting faults into systems is a straightforward technique to verify that such faults do not cause failures in the system, i.e., the system is tolerant to that faults.

Thus, fault injection strategies and techniques have been very widely analysed [BP03, ZAV04] and several tools have been developed, most of them focusing on VHDL models [JAR+94, GBG+01, BGB+08, MPE09]. However, as previously stated, SystemC is nowadays the de-facto standard in industrial HW/SW system design and simulation. Therefore, fault injection design and simulation in SystemC models has been getting an increasing interest in the latest years [MVS07, SRH08, BMS08, LR11, RPV+13].

Misera et al. [MVS07] adapt fault injection techniques and strategies from VHDL models to SystemC models in order to analyse the limitations and possibilities of the SystemC kernel. They simulate systems including saboteurs and simulator commands, and they extend logic types of SystemC in order to perform a more realistic behaviour of logic components. Since the approach is based on strategies used in VHDL models, they focus on logic-level models. In [SRH08] Shafik et al. propose an alternative technique to the one presented in [MVS07], also focusing on logic-level models.

Bolchini et al. [BMS08] go one step further into multiple abstraction level fault injection. They present a fault injection environment for the ReSP simulation platform [BBF+08]. The approach enables injecting faults by using saboteurs and simulator commands, using a novel technique called reflective wrapper. It does not focus on a specific MoC, so simulation is paused and resumed whenever a fault is injected.

In [LR11] Lu and Radetzki use the Concurrent and Comparative Simulation (CCS) technique to inject faults into SystemC models. This approach makes it possible to perform more than one fault injection experiment in each execution. The developer must use a specific data-type in order to inject faults in variables, and fault libraries are not defined, so the tester must implement the fault models.

Reiter et al. [RPV+13] perform error injection in simulated HW models defined using the CHES modeling language by extending the HW models in order to inject errors. The approach provides a library of different error models, including data-corruption, timing-corruption, halt, and signal-loss. The framework does not rely on a concrete model of computation, and the paper does not describe how timing constraints of the System Under Test (SUT) are guaranteed.

Regarding fault models and their simulation, the Model-Based Generation of Test-Cases for Embedded Systems (MOGENTES) project [MOG31] specifies a number of HW and SW related fault and failure models and taxonomies. On the other hand, the international ASAM AE HIL [ASAMHiL] standard defines an interface to perform error simulation in Hardware

in the Loop testing.

Fault injection modeling in UML for software testing

[MSU+09] *Software Implemented Fault Injection* (SWIFI) is a verification technique to detect functional errors caused by hardware failures or systematic errors in the design. SWIFI consists of the insertion of predefined faults in any software accessible units like memories, registers, peripheral devices, etc. Software modeling is strongly recommended for the development of high integrity safety critical systems, and currently Unified Modelling Language (UML)2 is a widely used standard used for software modeling. Therefore a tool supporting Fault Injection (FI) at early design stages would extend the verification capabilities while modeling the system.

Aim To provide a UML Fault Injection framework for the verification of safety critical systems designs.

Method The approach consists of implementing two components the Executive Manager controlling the testing process and the Fault Injection Kernel (FIK), added to the model under test (MUT). An automated model transformer will add FI in the system model for standardized fault types.

Contribution A library of components for dynamic fault injection during simulation of a UML model in IBM Rhapsody Designer.

Applications 2oo3 control system for railways.

Fault injection modeling in HDL for system testing

[VMM+09] The verification of high integrity safety-critical systems redundant architectures (Moon) impose specific testing requirements regarding the inter-processor communication channels and the voter functionality.

Aim To provide a Fault Injection test architecture for the verification of Moon safety critical systems implementation.

Method The approach consists of implementing input/output and communication fault injectors named saboteurs. These test components provide an additional control port to trigger the fault behaviour from a test controller.

Contribution A MPC8280 Ethernet 100 communication saboteur implementation (FPGA as less intrusive alternative).

Applications Moon RBC/ERTMS (railway)

3.3.2 Model-based SW development vs. code-based development

Model-based Testing (MBT) is an approach for the quality assurance of embedded systems. MBT resembles Model-based Design (MBD) but applied to verification and validation activities. The idea behind MBT is that carrying out verification and validation earlier reduces the project risks. The expected benefit for early V&V is a shorter development time, and lower costs than those achieved with a late testing process. An MBT process works on the test specification, optimization and test planning concurrently with the design tasks:

The delay between current adoption of MBT and already deployed MBD motivates the research on many testing problems [Ber07] as illustrated for generic software testing in 3.5.

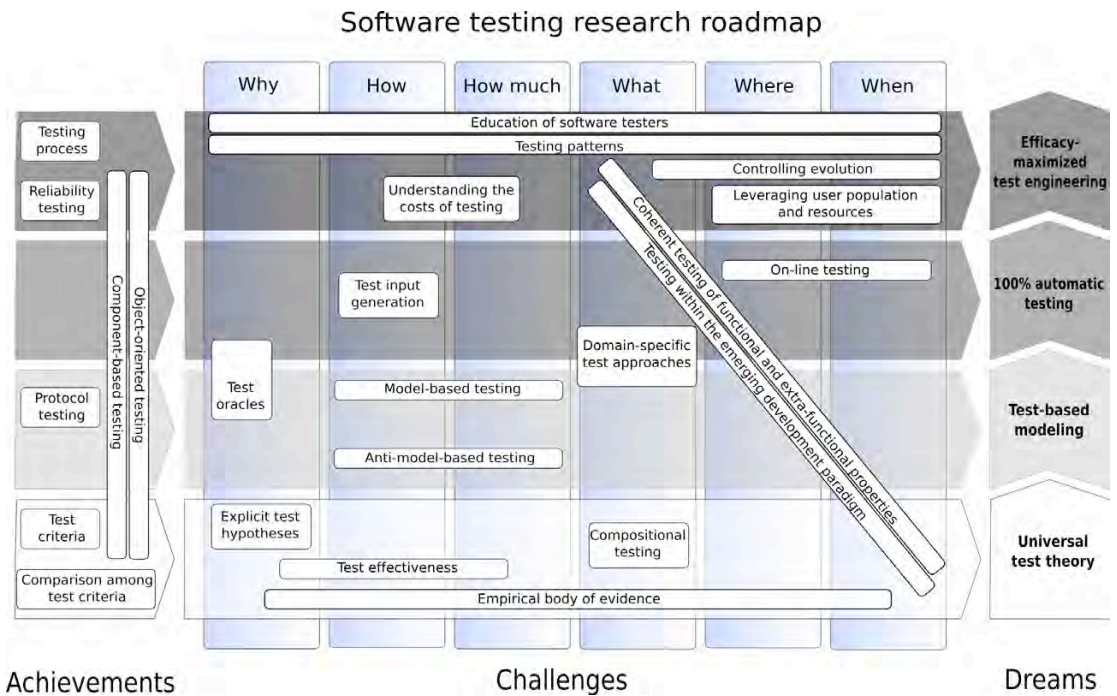


Figure 3.5: Software testing research road-map (source: [Ber07])

In the last years multiple MBT techniques for embedded systems have been proposed [Con08].

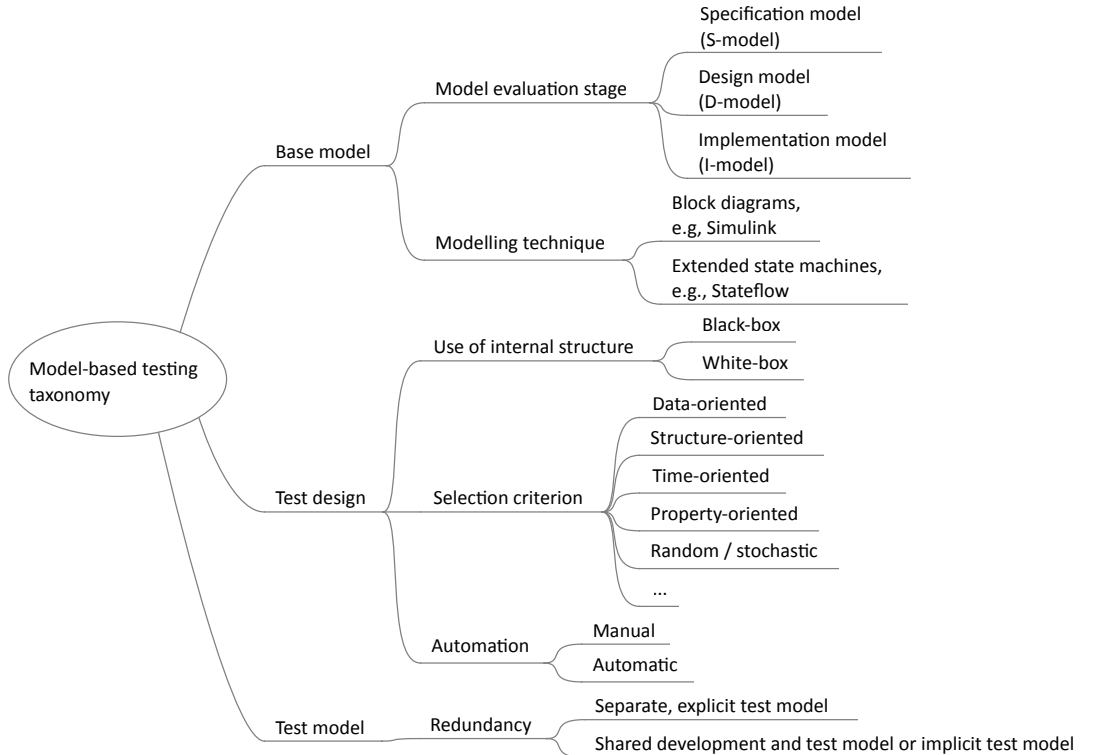


Figure 3.6: Model-based testing taxonomy
(source: [Con08])

3.4 Model-Based Testing

This section reviews research works on the applicability of Model-Based Testing (MBT).

3.4.1 Methodological Issues in Model-Based Testing

Pretschner and Philipps [PP05] analyse some methodological issues with regard to model-based testing. Testing denotes a set of activities that aim at showing that actual and intended behaviour of a system differ, or at increasing confidence that they do not differ. Often enough, the intended behaviour is defined by means of rather informal and incomplete requirement specifications. Test engineers use these specification documents to gain an approximate understanding of the intended behaviour.

That is to say, they build a mental model of the system. This mental model is then used to derive test cases for the implementation, or System Under Test (SUT): input and expected output. Obviously, this approach is implicit, unstructured, not motivated in its details and not reproducible.

The idea of model-based testing is to use explicit abstract behavioural models of the **SUT** to encode the intended behaviour of a system, based on the system specifications. The purpose of these models is to generate a finite set of traces that represent the stimuli and expected output for the test cases. In order to ascertain that the model represents the system specifications it is usual to carry out a validation of these test models. Finally the traces provided by the test models are used for the verification of the implemented system in order to increase confidence about the correctness of the realization.

With regard to model-based testing, two key questions arise:

1. If the test model has to faithfully represent the requirements, what are the benefits of building a test model, validating it, deriving the test cases and running them against the **SUT** instead of directly validating the **SUT**?
2. Taking into account time and cost constraints of engineering activities, is it possible to use a shared, single model for the automated synthesis of the **SUT** and for testing it?

The first question is answered by abstraction: the test model should be simpler than the actual **SUT**, so as to make it easier to understand, validate, and maintain and make it amenable for test case generation.

There are two basic approaches to simplification:

Omission of details Models are means to actually get rid of irrelevant details, and capture the fundamental ideas conveyed by the specifications, making them easier to understand and manipulate. This incurs an additional cost, as the different abstraction levels of the testing models and the **SUT** prevent direct usage of testing results from the model to verify the **SUT**. For testing the **SUT**, some specific test components need to be developed to reinsert the discarded information.

Encapsulation of details Encapsulation is a way to reduce complexity by bundling details of a system, which are packed in the form of libraries or language constructs. For model-based testing this usually leads to the problem of building and validating a model that is as complex as the system under test.

The answer to the second question is addressed by realizing that a testing activity requires some sort of redundancy, provided by the selected realization methodology. Four model-based testing scenarios can be distinguished:

Common model MBT The same model is used for both code and test-case generation. The lack of redundancy prevents using automatic verdicts for testing the **SUT** with automatic code generation, except for verifying the environment assumptions.

Automatic Model Extraction The test model is automatically extracted from an existing system. Low redundancy. The key is the abstraction purpose. Useful abstraction requires some domain and application specific knowledge, so fully automated abstraction is not achievable.

Manual Modeling The test model and the system models are built on top of a different specification. Depending on the degree of interaction between the teams responsible for specification and testing, for a low level of interaction a sufficient redundancy for automatic assigning verdicts is achievable.

Separate Models Two redundant and separated models are used for test case generation and code generation, thus allowing for automatic verdicts. This seems to be optimal although not yet empirically verified, i.e., in the sense that it combines model-based testing and model-based design, but is also most expensive approach.

In practice having a complete isolation between the development of the models and the development of the code is unrealistic. For iterative development processes with clear increment boundaries it is possible to interleave the development of both models, yet it is likely to incur some overhead. Despite the fact that it is accepted that modeling might be too expensive, modeling in itself helps in revealing errors, which is appealing for the embedded systems industry. The fundamental concern about model-based testing is whether it is more cost-effective than other testing approaches. Further studies with regard to the economics of model-based testing are needed in order to help managerial decisions about the development processes under given project constraints.

3.4.2 Testing Effort for Model-Based Testing (MBT)

Sinha et al. [SWS06] propose a measurement framework for the evaluation of four different model-based generation tools. Following the Goal Question Metric methodology they formulate metrics for complexity, ease of learning, effectiveness, efficiency and scalability. Later the authors compare four different MBTG tools using a selected case study.

Güldali et al. [GMS10] tackle the managerial point-of-view about testing, following a two-step approach in their research: first, provide an effort decomposition scheme suitable for MBT, and second, provide a systematic method for analysing and comparing different MBT scenarios. They identified six different model-based testing (MBT) scenarios described in the literature up to the date of their investigation, sketched in Figure 3.7.

3.5 Test Re-usability

This section reviews some approaches to test re-usability.

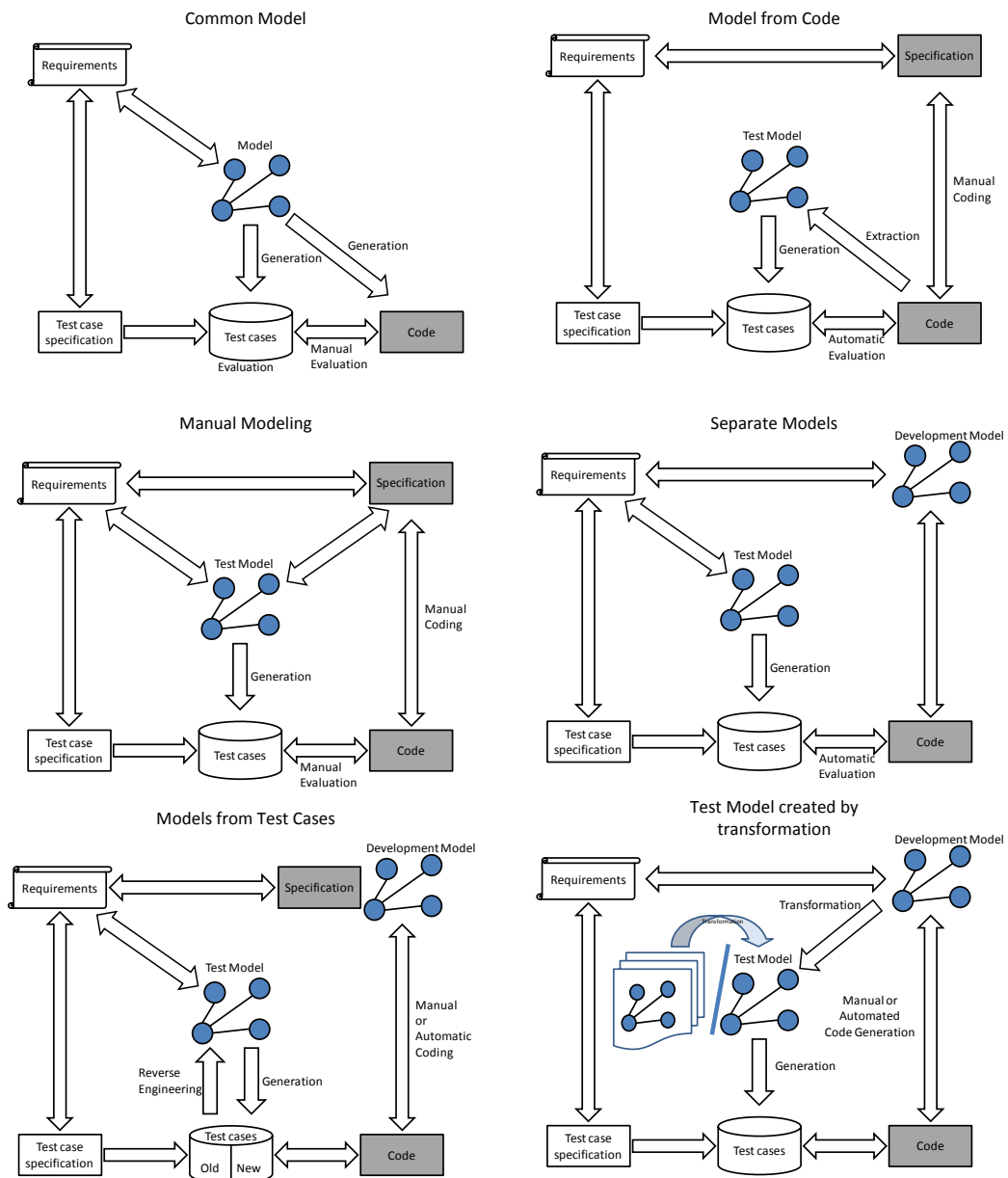


Figure 3.7: Model-based testing processes (source: [GMS10])

objective is to improve the test re-usability from tests exercised on abstract descriptions of the system. As shown in the figure, testing on models with mixed-levels of abstraction requires a number of abstraction/concretization adapters to provide interface compatibility among models. This topic was also analysed in [PP05], addressing some situations in which the test on models cannot be computed due to a causality reversal in such adapters in a bottom-up re-use approach.

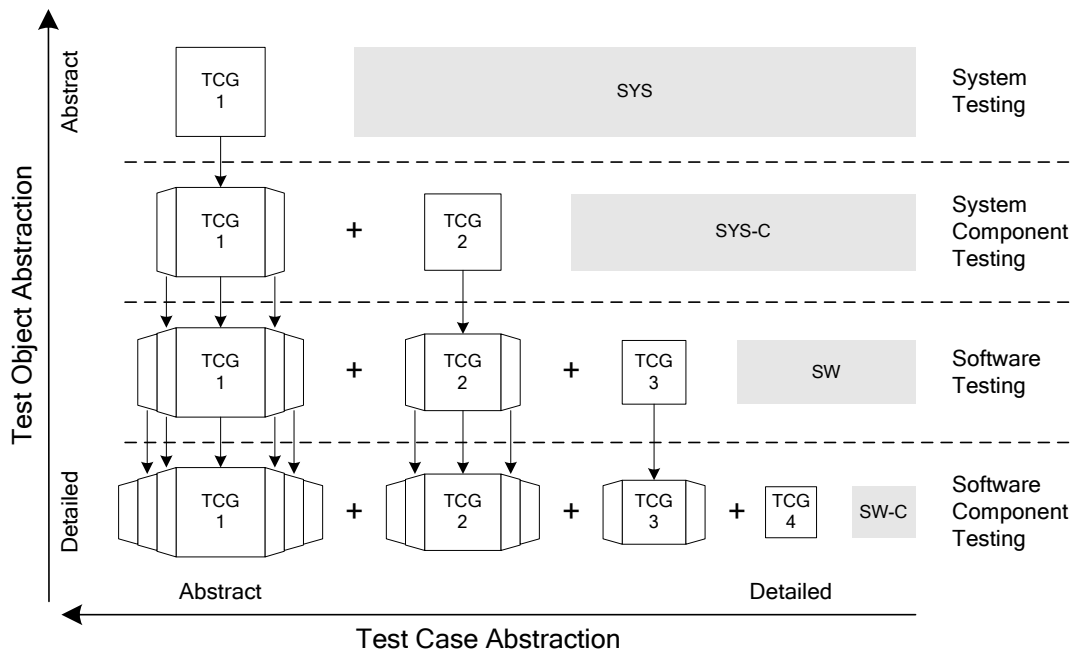


Figure 3.9: Top-down reuse for multi-level testing (source [MK10c])

3.6 Outlook for Testing in Software Engineering

Broy states that although formal methods help with constructive and analytical quality assurance, still an important part is the validation of the requirements, for which we need understandability and comprehensibility [Bro11]. A significant goal is the reduction of complexity and ease of understanding. This explains the acceptance of graphical description techniques. Often understanding is even more important than formality, specially for maintenance during the product life-cycle. Formal specification and verification can only prove a correct relationship between the formal specifications and implementations, but still cannot prove that the system meets valid requirements.

Model Based Engineering (MBE) is a software development methodology which focuses

on creating models or abstractions. It is closer to particular domain concepts than programming, computing and algorithmic concepts.

When early architecture verification is done accurately and the modules are verified properly, we usually get fewer defects during system integration. If as part of the architectural description of a system we provide an executable model of its constituent components, then we have an executable system description. With such a description we can generate test cases either at the system, integration and component levels.

For hierarchical systems the scheme of specification, design and iteration can be iterated for each sub-hierarchy in the architecture.

Cost reduction factors:

1. Avoiding and finding defects early in the process.
2. Applying proven methods that are standardized and ready for use.
3. Automation of the development tasks and steps whenever possible.
4. Reuse of implementations, architecture requirements and development patterns whenever possible.

The application domain should be modelled and integrated into the the development process, which should support modeling with suitable tools.

Rustan and Leino analyse the role of Behavioural Abstraction in Software Engineering [RL11]: Abstraction is a tool to cope with complexity by organizing a complex problem into manageable and understandable pieces. Besides common forms of abstractions like procedural abstraction, data abstraction and parametrization, we can use abstraction by occlusion to hide or reveal the inside part of components, depending on the scope. Hierarchical models fit for this purpose. Abstraction by occlusion is applicable to the structural model of the system, but we need a different abstraction to describe what the system does, namely the behavioural abstraction. Test suites constitute a form of behavioural abstraction, as the set of test cases show the intent of the system design.

Stepwise refinement: The process of describing the salient properties of the system to be constructed and then adding more details, each maintaining the properties described in previous stages.

3.7 Standardization of XIL Tests

ASAM XIL [ASAMXiL] is an API standard for the communication between test automation tools and test benches. The standard supports test benches at all stages of the development and testing process –most prominently model-in-the-loop (MiL), software-in-the-loop (SiL) and hardware-in-the-loop (HiL). The notation "XIL" indicates that the standard can be used for all "in-the-loop" systems. This has the advantage that it enables users to freely

choose testing products according to their requirements and integrate them with little effort. Using ASAM XIL-compliant products allows users of test systems to mix and match the best components from different suppliers without having costly integration efforts.

The standard furthermore decouples test-cases from real and virtual test systems. This allows users to transfer tests between different test systems with little to no migration effort. Consequently, tests can be easily re-used. Know-how is much easier transferred from one test bench to another, resulting in reduced training costs for development- and test engineers.

These advantages are the result of two major components of the ASAM XIL standard:

- the framework, which provides data measuring and mapping functionality independent from the used test bench hardware and software.
- the test bench, which provides port-based communication means to the simulation model, the ECU, the electrical subsystem and the ECU network.

Test automation applications which use the ASAM XiL API are used in all automotive E/E domains, e.g., drive-train, steering, electric lighting, etc. Cross tests for ASAM XIL implementations are carried out regularly to ensure compatibility between the test automation software and test benches.

The standard comes with a C# source-code library, which contains open source software that can be used by framework developers to implement the XIL API. It covers about 69% of the standard API methods. Provided methods are unit and data type conversion, reading mapping files, mathematical operations, and more. The source code has been professionally developed and constitutes a standard-compliant implementation of significant parts of the ASAM XIL API.

3.8 Restbus Simulators

Commercially the term *Restbus simulator* denotes a test component that emulates the messaging behaviour of a group of devices communicating with a system under test. Simulators as dSPACE VECU [V-ECUa] or ETAS ISOLAR-EVE [V-ECUc] feature Transaction-level Modelling (TLM) accuracy and connectivity to real buses, supporting the integration of platform-independent software components of a Virtual ECU (V-ECU) as part of the HiL infrastructure. Also bus-simulation applications can be linked to behavioural modules to emulate devices, e.g., the open-source *Busmaster* [BUSM] software tool, or Controller Area Network (CAN)oe [CANoe], both enabling the connection to real CAN buses. Similarly, Ferrari et al. developed DESYRE, a framework to support the *Platform-Based Design* [SM01] of distributed elevator control systems for large buildings [FCM+12]. DESYRE recreates the CAN network in a distributed elevator control system, simulating the behaviour of multiple node instances based on actual message timing data measured on a real system. A drawback of many of these tools is that they require specific real-time

simulation environments. As a consequence, tool restrictions on the capability to import user-code in the simulator force the test developers to modify the original code of the virtualized device.

Bringmann et al. [BEG+15] review the state-of-art in simulating *Virtual Prototypes* (VPs) to accelerate the SW development process. The CAN Restbus simulator described in this dissertation can be described as a source-level simulator (SLS), focused on emulating the functional behaviour of a standalone application by cross-compiling the source code and linking lightweight host-compiled OS and processor models. SLS VPs abstract the actual devices while being Transaction-level Modelling accurate. Unless the simulator provides strict memory separation between VP instances, an SLS running in a single execution context -e.g., a LabVIEW-RT process- has to update the VPs sequentially, which causes an overhead in simulation physical time due to context switching. Another limitation is the timing inaccuracy, that could hamper the overall system behaviour of the simulated nodes. The authors propose a *Timed Binary Control Flow*, where timing annotations alongside the equivalent binary code for the simulation host increase the timing-accuracy in the logical simulation time, yet enabling an ultra-fast offline simulation.

Similar to the VPs, the *Virtual Processing Components* [SGH+10] provide the basis for Glass et al. [GGR+14] to build a multiple-node simulator for Ethernet AVB buses. Our main contribution with respect to these works is that we implemented a Restbus simulator based on a CAN communication system, instead of Ethernet AVB.

An alternative to executing multiple VECUs in parallel are *instruction set simulators* ISS or full virtualization environments like QEMU [QEMU], OVPsim [OVP] or Simics [SIMICS]. Yet the integration of such tools with other real-time components is challenging, since they are not meant to work in a HiL environment.

3.9 Projects on Certifying Mixed-criticality Systems

This section recalls some recent EU projects related to safety certification of MCSs.

3.9.1 CESAR project

The CESAR project [CESAR]¹ was an ARTEMIS project: Cost-efficient methods and processes for safety relevant embedded systems. Product lines were introduced into the CESAR meta-model in an orthogonal way by the kind of technology that is used in the BVR tool which is applied in DREAMS. In CESAR it was demonstrated in the use case of a safety module for industrial use that several different models could be managed with the same variability model. Furthermore, we showed how the unit test cases could be managed in concert with variants. We demonstrated how the test suits were generated from the product line model making sure that the tests were directly relevant for the variant derived.

¹<http://www.cesarproject.eu>

We used this to automate the testing of the software over the optimal set of configurations. This proved surprisingly useful. This combined approach which makes it possible to tie tests to variants is similar to our DREAMS certification approach where arguments are tied to variants.

3.9.2 VERDE project

In the ITEA 3 project “VERification-oriented & component-based model Driven Engineering for real-time embedded systems” (VERDE) [VERDE], SINTEF through ICT-Norway developed an algorithm with a supporting tool to select an optimal set of configurations for testing a product line. The problem is theoretically intractable, but it was shown that in practice a near-optimal set can be found. The underlying approach is that of coverage arrays where combinations of features are guaranteed to appear in at least one configuration. It turns out that the needed set of configurations is significantly smaller than intuitively expected, and that the effect of testing on this systematically produced set of configurations is very high [Joh13]. This selection of a near-optimal set of configurations is also being used in DREAMS for design space exploration, and could be used as evidence to support testing arguments for a certification relative to (say) IEC 61508.

3.9.3 VARIES project

The ARTEMIS VARIES project (2012-2015) [VARIES] targets variability modeling and product line engineering. The VARIES project had some effort on certification of product lines which served as a baseline for this dissertation. VARIES [VAR47] includes a walk-through of the different analysis techniques brought forth and experimented within the project. These techniques can in many cases be useful for assessing DREAMS systems. Technically, the VARIES techniques could represent the means for fulfilling an argument in the argument chains of our DREAMS approach. VARIES showed which elements of the IEC 61508 functional safety standard could be assessed by which analysis approach. VARIES recognized that product lines were not a concept in certification while they could handle products with variability. In particular, we can mention the algorithm using coverage arrays for selecting a good set of configurations for testing. This technique is also used in DREAMS as an integral part of the domain space exploration, while for certification purposes it is well suited for assessing the integration of different components with variability. The coverage array approach implemented within the SINTEF BVR Tool [VHC+15] is also combined with managing the tests according to the variant selections. This is very similar to how the certification arguments are generated in our DREAMS certification approach.

3.9.4 OPENCROSS project

OPENCROSS is a European large scale integrated FP7 project dedicated to produce the first European-wide open safety certification platform: an Open Platform for Evolutionary Certification Of Safety-critical Systems for the railway, avionics and automotive markets. Certification of safety critical systems remains until now a manual and tedious endeavour. Certification encompasses various activities such as engineering, safety engineering, and V&V that span over multiple teams located in different places, sharing many artefacts (e.g., software components, documentation, test plans) of which there often exist multiple versions. The FP7 OPENCROSS Project [OPENCROSS] tackles this accidental complexity by providing an open platform for evolutionary certification of safety-critical systems. The OPENCROSS platform [OPE23] reconciles the development and certification activities during the development process. Amongst other features, the OPENCROSS platform (see Figure 1) supports the following activities:

Continuous compliance monitoring The tight integration of the OPENCROSS platform in the development process lets developers determine to which extent their contribution matches the safety requirements at stake.

Centralised management of certification assets The OPENCROSS platform keeps the certification assets up-to-date. It monitors changes and triggers or highlights required activities consequently.

The OPENCROSS approach relies on the common certification language (CCL) that unifies Goal Structuring Notation (GSN) and Structured Assurance Case Meta-model (SACM) notations [OPE53]. Common Certification Language (CCL) enables composition of evidence data, by distributing it accordingly to the components that form the system. The OPENCROSS platform hence automatically composes these fragments of evidence data as engineers assemble and reassemble the components, and highlights errors in the argumentation.

CCL – Common Certification Language is intended to generalize over certification in different domains, typically aerospace, automotive and rail. The motivation for CCL is to be able to reuse certification efforts. The CCL contains standardized conceptual models for key aspects of certification, and a vocabulary for harmonization and comparison of terminology from different domains [OPE47]. The CCL meta-models have a presentation scheme adopted from the OMG's Structured Assurance Case Model (SACM) [SACM]. Creating vocabularies is the foundation on which the commonalities between certification against different standards can be built. The vocabularies are ontologies where different terms are related. Safety argumentation is one central part of the OPENCROSS CCL method. The OPENCROSS project had a specific work package (WP5) that was dedicated to compositional certification [OPE53]. Actors of the OPENCROSS platform will not (directly) use the CCL itself (conceptual meta-models). Users of the platform will instead work on information that is based on CCL. Users of CCL are those handling the OPENCROSS platform as such.

3.9.5 AMASS project

The AMASS project [AMASS] builds up on OPENCROSS. AMASS is an ECSEL project that started in April 2016 on Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems. In their abstract found on CORDIS we see *“The ultimate aim is to lower certification costs in face of rapidly changing product features and market needs. This will be achieved by establishing a novel holistic and reuse-oriented approach for architecture-driven assurance (fully compatible with standards e.g. AUTOSAR and IMA), multi-concern assurance (compliance demonstration, impact analyses, and compositional assurance of security and safety aspects), and for seamless interoperability between assurance/certification and engineering activities along with third-party activities (external assessments, supplier assurance).”*

3.9.6 ASCOS Project

The EU FP7 ASCOS project [ASCOS] was initiated by the aviation industry and funded by the FP7 framework program. ASCOS developed novel certification process adaptations and supporting safety driven design methods and tools to ease the certification of safety enhancement systems and operations. The approach is to build a logical argument for the certification of any change to the total aviation system, supporting the top-level claim that changes are acceptably safe. The ASCOS project was motivated by the problems found to incorporate innovative new ideas while still needing to have the product certified. Therefore, they saw the need to formulate adaptation of the certification processes. The ASCOS Method uses modular safety arguments to provide a framework to integrate existing approval approaches while also providing the flexibility to adapt the approaches where necessary to enable the smooth approval of advances in aviation technology [Bul16]. The ASCOS approach had the following objectives:

- to be more flexible with regard to the introduction of new products and operations;
- to be more efficient, in terms of cost and time, than the current certification processes;
- to consider the impact of all elements of the aviation system and the entire system life-cycle on safety in a complete and integrated way.

Central to the final ASCOS approach was the use of modular safety arguments precisely described in GSN (Goal Structured Notation) and this coincides with our DREAMS approach.

3.9.7 PROXIMA project

In the Mixed-Criticality Forum we also find the EU FP7 project PROXIMA [PROXIMA] where one industrial benefit is declared to be “certification arguments for DO-178B and

safety standards". PROXIMA is an Integrated Project (IP) of the Seventh framework programme for research and technological development (FP7). The PROXIMA project provides industry-ready software timing analysis using probabilistic analysis for many-core and multi-core critical real-time embedded systems and will enable cost-effective verification of software timing analysis including worst-case execution time. Certification arguments for DO-178B and safety standards will also be provided in PROXIMA.

3.9.8 MultiPARTES project

The MULTIPARTES project [MULTIPARTES] provides support for mixed-criticality trusted embedded systems based on virtualization techniques for heterogeneous multi-core processors [PGJ+13, TCA+14]. For that purpose, the main challenge of this project is the support for mixed-criticality integration for embedded systems based on virtualization techniques for heterogeneous multi-core processors. The starting point for this is the XtratuM hypervisor, an open source hypervisor for real-time embedded systems. In addition to the multi-core platform virtualization layer, MultiPARTES devised an engineering methodology for the rapid model-driven development and production, taking advantage of partitioning of multi-core systems. The research results of the project have been validated in four case studies including energy, industrial control, video surveillance and space, and complemented by an application study for the automotive sector. The key results of this project are the reduction of the effort and cost on the engineering of mixed-criticality embedded systems, the increase of the reliability by decreasing the number of components (wires and connectors) that can fail, the reduction of resources required (e.g., energy consumption, weight and volume), the reduction of time-to-market by quickly integrating 3rd party applications into a partitioned system architecture and the reduction of the certification cost by enabling modular-based independent verification of the components of the system.

3.10 Discussion

A safety-critical system development requires thorough knowledge of the application. This knowledge can be elicited from experts in the field, who are required for the validation of the proposed design. In a pure model-based process we need models for both the system to build and the environment in which the system performs its function. Moreover the models can be exploited for system testing in hardware-in-the-loop configurations.

Testing re-usability in model-based testing is a topic of ongoing research. To our knowledge there is no clear guideline about how to re-use the test artefacts outside the modeling environment, in a mixed model-code simulator. Moreover, when going to actual system prototypes, verification re-usability is usually limited to the test vectors, despite the effort spent before in defining, scripting and running test cases against the system models. Availability of open source simulation solutions and extensible modeling languages pave the

way for a better and more comprehensive integration, in order to achieve the ideal seamless process that could finally provide the required product quality with an affordable effort.

Part II

Contribution

4

Theoretical Framework

This chapter presents the research methodology, hypotheses, and goals for the thesis, and introduces the case studies used to validate the implemented solutions.

4.1 Methodology

This research work has been developed following the methodology depicted in Figure 4.1. The main stages are described below:

1. **Thesis Outline:** The start phase consisted of writing a research proposal that was submitted to the University for review and approval. The contents of the Thesis outline included the preliminary outcomes of the following tasks:
 - *Problem Statement.* Analyse the problem and motivate it by answering questions such as: Why is it important? What does it solve? The context of the problem was introduced in Chapter 1 (see §1.1).
 - *State of The Art.* Identify existing approaches and solutions related to the research problem. Perform a critical analysis of the state of the art and identify: what has been done, existing limitations and weak points, points to contribute, etc. An update on the critical analysis of the existing work is presented in Chapter 3.
 - *Research Approach.* Define the research methodology to follow: tasks and phases, collaborations, work plan, etc. Formulate the hypotheses of the thesis

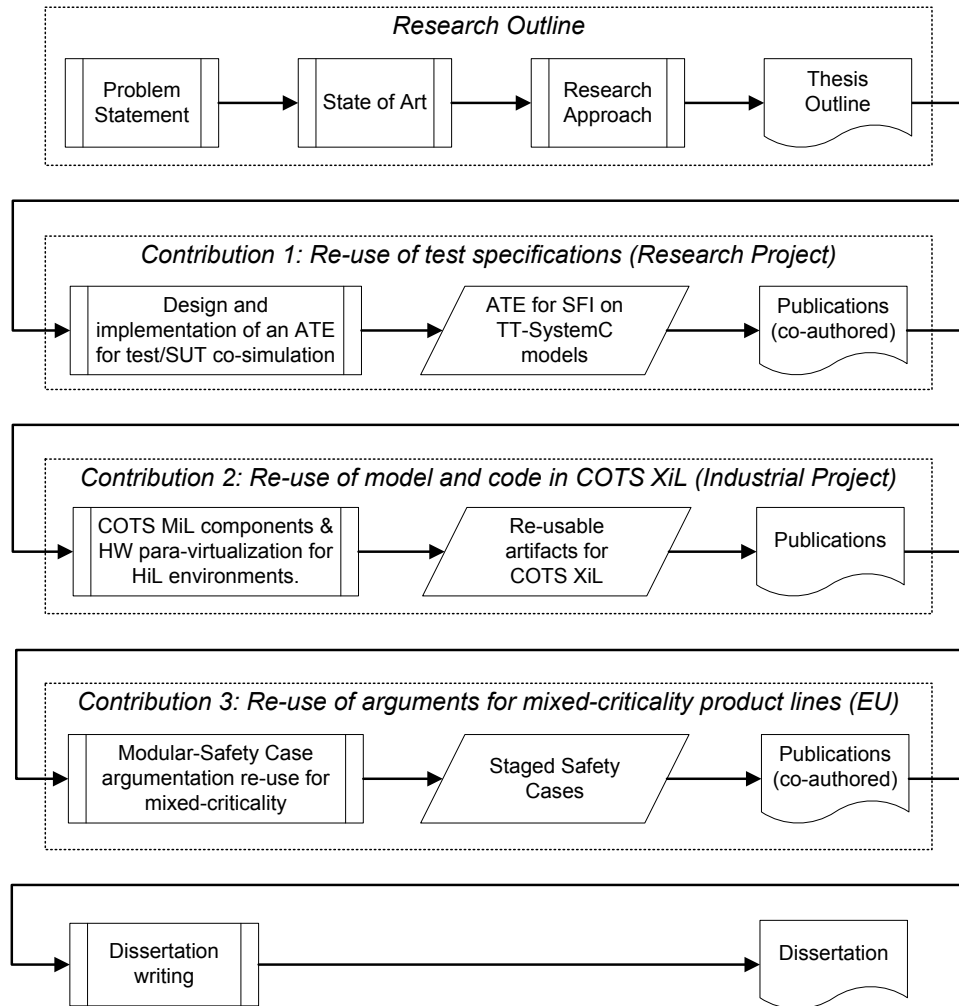


Figure 4.1: Research methodology followed in this thesis

and set goals to be performed. These goals will be used to validate or reject the hypotheses. They are described in Sections §4.2, §4.3 and §4.3.1 of this chapter.

2. **Goal Achievement:** Goal achievement implies (for each setted goal): (i) analysing the goal problem; (ii) implementing a solution; (iii) evaluating the solution; and (iv) publishing results. This phase is covered by different chapters of this dissertation: Chapters 5, 6 and 7 describe the design and implementation of the re-use approaches. The approaches have been evaluated experimentally with three case studies. The

results of these evaluations are presented and reviewed in Chapter 8. Finally, Chapter 9 presents the conclusion and discusses the validation of the hypotheses (see §9.1.1).

3. **Thesis Dissertation.** Write the thesis and perform the thesis defence.

4.2 Hypotheses

The *hypotheses* of this research and their links to the research objectives are:

- **Hypothesis A:** *An Automatic Test Executor (ATE) can be integrated in simulation frameworks so that it: (i) enables the early analysis of fault-tolerance mechanisms in dependable systems with redundant structures at various levels of abstraction, (ii) automates the simulation of fault-injection experiments, and (iii) facilitates test re-use across system models at different levels of abstraction, and even up to a real test-bench.*
- **Hypothesis B:** *Generic models of sensors and instruments can be designed and validated in a Commercial-Off-The-Shelf (COTS) modelling environment, such that these could be re-used to build real-time test components for a COTS heterogeneous hardware-in-the-loop (HiL) system to verify dependable systems, in which the obtained test components may execute in parallel, keeping a strict timing synchronisation.*
- **Hypothesis C:** *The capabilities of current COTS HiL platforms support the deployment of virtual replicas of nodes in a distributed dependable system, yielding a versatile and cost-effective test architecture through re-using code from actual devices.*
- **Hypothesis D:** *A model-based argumentation framework enables the compilation of interrelated sets of information to assist safety engineers in developing safety cases for Mixed-Criticality Product Lines (MCPLs), following a platform-based design (PBD) supported by a set of pre-built Modular Safety Cases (MSCs), and where results from analysis, verification and testing activities can be incorporated to the set of supporting evidences required to argue about the safety of a system.*

4.3 Goals

The *global objective* of this research is to develop a top-down verification and certification framework comprising models, techniques and tools to assess the safety of an embedded system composed of hardware and software subsystems by means of testing on simulators at different refinement phases of the development cycle.

The *contribution* consists of a model-based framework to improve the re-usability of tests and safety arguments across different refinement stages in a model-based development of dependable embedded systems implemented on programmable electronics. The approach

should enable traversing from testing on abstract functional design exploration models to the verification of physical prototypes by means of intermediate simulators at multiple levels of abstraction. The proposed approaches should be applicable to the verification of individual subsystems and components, for the case in which a complex system is refined either by partitioning or decomposition, while the collection of evidences generated during the development process is managed in a way that is consistent, rational and amenable to review.

4.3.1 Operative Goals

The *global objective* is realized by the operational objectives listed below:

Operative Goal A: Define an extended Executable Time-Triggered Model (**E-TTM**) meta-model (**PI-TTM** / **PS-TTM**) to describe: (i) the structure and behaviour specification of the dependable system under test, (ii) the effects of random faults on the system, (iii) the test cases, and (iv) the test outputs. *This goal links to Hypothesis A.*

Operative Goal B: Implement the **PI-TTM/PS-TTM** simulation engines on top of the **E-TTM** framework, integrating an Automatic Test Executor (**ATE**) with its operational environment, i.e., timing and fault injection component abstractions for test automation on the simulators. The resulting simulation framework will support the definition of a dependable system, first as a functional **PI-TTM** model, then as a **PS-TTM** that represents the redundant structure usually required for safety-related applications. Unmodified **PI-TTM** / **PS-TTM** models will undergo simulations of fault scenarios. The integrated **ATE** will support the automated execution of test campaigns with simulated fault injection to assess the fault tolerance on the system models. *This goal links to Hypothesis A.*

Operative Goal C: Analyse the requirements to model re-usable sensors and instruments in the Simulink language that could be transformed to **FPGA-IPs** and integrated in a real-time **COTS HiL** platform to test dependable systems. Develop a library of component models and validate the models in a case study. *This goal links to Hypothesis B.*

Operative Goal D: Design and implement a para-virtualization approach to build virtual devices by re-using product code that could replace real nodes in a real-time test bench for verifying a distributed dependable system. Implement and validate in a case study. *This goal links to Hypothesis C.*

Operative Goal E: Develop a model-based framework to assemble the evidences for a safety certification, supporting the partial automation of the system documentation. The argument models will be scalable, either to be completed in a top-down approach, or composed from partial argument fragments. Implement and validate in a case study. *This goal links to Hypothesis D.*

4.4 Case Studies

The experimental validation of the approaches proposed in this dissertation was carried out on three different case studies. Each case study considers a different application scenario. The application scenarios were either drawn from actual industrial projects, or developed for collaborative research demonstrators:

- Railway Controller Case Study:** The subject of this case study is the verification of a safety odometry system for railways, where the safety requirements, context and system structure were drawn from an actual product development carried out at IK4-Ikerlan. The safety function of the odometry system is to prevent a train from surpassing the track speed limits. According to the applicable railway safety standards the risk analysis for the odometry system yields a requirement of Safety Integrity Level (SIL) of 4. To achieve this requirement, the odometry system combines diverse information from multiple sensors, including encoders, accelerometers, balises, or GPS information. The fault tolerance relies on a Triple Modular Redundancy (TMR) functional structure, while Time-Triggered (TT) communication between the safety components provides time-determinism. The Railway Controller will be modelled using the PI-TTM / PS-TTM meta-model, and then its fault tolerance will be exercised in a campaign of automated simulations involving different fault scenarios.

This case study validates the achievement of operative goals A and B.

- Hardware-in-the-Loop Elevator Simulator (HiLES) Case Study:** This case study focuses on an X-in-the-loop (XiL) test framework for verifying elevator control systems. It is intended to migrate from a prototype-based development to a Model-Based Development (MBD) / Model-Based Testing (MBT) process for an evolving family of elevator control systems. Nowadays, the elevator control system is a distributed system composed of up-to hundreds of nodes. The core is the position control system, designed as a "universal product" to operate in varying configurations. The complexity added to the control SW brings with it an increasingly longer verification process, where the limited test-bench configurations and the operations required to modify the setup makes a pure HiL insufficient for a comprehensive validation. Moreover, the introduction of new positioning sensors for safety applications, as much as the adaptation to the new safety standards for elevators demand a framework for carrying out proof-of-concept demonstrators before deciding about the product design.

A staged model-in-the-loop (MiL)-software-in-the-loop (SiL)-HiL development provides a more economical assessment process. But the problem lies in the unavailability of a suitable model base. To start with, one should consider the number of models to develop and ultimately, maintain. Ideally, a manufacturer would benefit from relying on a limited set of model libraries, with each model thoroughly validated and ready to use in whichever configuration, e.g., MiL, HiL. HiLES will integrate

sensor and instrumentation models contributed in this thesis, that can be used in either **MiL** or **HiL** configurations. Moreover, a para-virtualization approach will be applied to integrate virtual nodes in the **HiL** infrastructure, reducing the number of test components external to the **HiL** platform, and enabling a more versatile, **SW**-defined test architecture.

This case study validates the achievement of operative goals C and D.

- **Mixed-Criticality Product Line (MCPL) for Wind Turbine Controls Case Study:** The third case study tackles the problem of assembling the safety-related evidences generated during the planning, design, development, and verification, validation and testing (**VVT**) activities of families of dependable Mixed-Criticality Systems (**MCSs**). The *Distributed Real-time Architecture for Mixed-criticality Systems (DREAMS)* provides an approach to certify a plurality of products by tailoring a pre-built safety analysis: the Modular Safety Cases (**MSCs**). A **MSC** can be viewed as a generic template covering all the possible uses of a component in a safe way, while addressing its applicability limits. To tackle the complexity of developing an **MCS** product line, **DREAMS** provides a Design Space Exploration (**DSE**) tool. **DREAMS DSE** feeds with the functional and safety-requirements, a database of platform components, a variability model and an automated combinatorial engine to resolve and optimize the feasible product configurations, e.g., meeting the safety requirements. During the certification process, the **DREAMS DSE** has to document the rationale sustaining the validity of the design configuration.

The Wind Turbine Controller case study will exemplify the application of the **DREAMS** solutions to develop a product line of controllers for wind turbines with safety requirements according to the generic functional safety standard IEC 61508 (the latter is used for the sake of generality, although in machinery the applicable safety standard is the IEC 61508 derivative ISO 13849). The wind turbine controller product line will be defined through a variability model with selectable features, amongst which are the target safety **SIL** levels. The wind turbine controller is composed of several subsystems with different criticality requirements. The subsystems will in turn integrate **SW** or Programmable Logic (**PL**) components, that previously should be characterized with regard to their Systematic Capability (**SC**) and other properties. Once a **DREAMS** user allocates the safety requirements in the **AF3** modelling environment, an optimization algorithm will perform a **DSE**, seeking for Pareto-optimal product configurations, i.e., the *product samples* in the product line. The safety of each product sample is evaluated and assessed by the **DREAMS SCCRC** component. For each safety-compliant product sample, a post-processing round in the **SCCRC** will traverse a database of argumentation models, to generate the documented rationale, i.e., the *product safety-case* justifying the validity of the safety claims.

This case study validates the achievement of operational objective E.

5

Model-based Testing with ATE Co-simulators for Re-using Tests

This chapter presents a test approach to re-use test descriptions from model simulations for testing the real system. The idea is to exploit the analysis phases in the development process to concurrently validate the correctness of the test sequences. Assuming that the test description is written in a programming language supported in the real-time test environment, the *return of investment* results from the re-utilization of previously exercised and validated test sequences, even in testing the physical prototypes. We propose two complementary approaches:

- *Test re-use on PS-TTM simulations:* This research focuses on a simulator framework to specifically analyse the design of safety systems with a redundant HW architecture and Time-Triggered (TT) computation. To that purpose we provide the test engineer with a customized simulation environment, the Platform-Specific Time-Triggered Model (PS-TTM) framework, that integrates: (a) SystemC TLM models; (b) the Executable Time-Triggered Model SystemC extension to simulate the Time-Triggered Model of Computation, contributed by J. Perez [Per11]; (c) a library for simulated fault injection, contributed by I. Aystaran [Aye15]; and, (d) the *contribution of this thesis*: an embeddable Python Automatic Test Executor to automate the simulation runs through Python scripts, while providing compatibility with real test systems and test re-use support.
- *Test re-use on Simulink simulations:* We aim at executing the same test specification in both simulation modes, on an offline simulator and a real testbench. From a

test perspective, the System Under Test (SUT) is considered as a black-box, so that we will only consider a model of the functionality of the system, abstracting the hardware properties. We analyse the feasibility and limitations of integrating our *synchronous* Python Automatic Test Executor with the simulation engine of the Simulink Commercial-Off-The-Shelf (COTS) modeling environment¹.

5.1 Why a Python ATE?

Python is a high-level programming language supporting multiple programming paradigms (e.g., object-oriented, imperative, functional, etc.). The CPython reference implementation of the Python interpreter [Python] is programmed in C/C++, and is open source software. For the programmer, Python has a set of standard libraries, named *modules*, to abstract the application from platform particularities. Python offers many potential benefits for test automation and test re-use:

1. *Python eases cross-platform portability*: There are Python interpreters for many hardware (HW) and operating system variants, even for low-end embedded processors [uPY]. Thus, the portability of a Python application is only limited by the availability of the external modules, and the dependencies on platform-specific features. The latter can be alleviated by adopting an object oriented approach, hiding the platform dependencies with replaceable abstractions for non-portable components.
2. *Python eases scaling up the applications*: The standard Python libraries are comprehensive, so that Python programs can be enhanced easily, adding more functionality as required. Currently Python applications range from quite simple programs to scientific computing or big-data analysis applications.
3. *The Python language has a compact syntax that fosters productivity*: The Python language syntax enables developers to express concepts with fewer lines of code. This makes Python more amenable to learn for non-programmers.
4. *Interactive Python accelerates learning*: As an interpreted language, Python allows for interactive execution of Python statements in a *Python shell*. This supports a program-and-go, where developers can try new Python code in shorter time than the conventional *edit-compile-link-execute* process required by other languages.
5. *The Python interpreter is embeddable*: The Python interpreter can be embedded in an application.

¹This implementation differs from executing our synchronous Python function in a given time step, that is a function nowadays included in Simulink. Our approach instead, simulates the concurrent execution of the test script and the system model, resembling a real-world automated test scenario.

6. *The CPython interpreter is extendible:* The Python interpreter can import user-defined *extension modules*, so that Python can invoke custom native functions programmed in C/C++. These extension modules do not work in other Python implementations, unless an equivalent module exists.
7. *There is a basic Python test support available:* The `unittest` Python library provides a basic framework for unit testing. A test engineer may re-use the `unittest` classes as templates for derived implementations, thus homogenizing the structure and execution of the test cases.
8. *There are standardized Python Application Programming Interfaces (APIs) for controlling XiL testbenches:* The ASAM association developed a number of standards to improve the interoperability of test infrastructures provided by different vendors. The goal is to ensure that test suppliers implement common APIs such that a test specification would run unmodified with test components from alternative brands. This enhances the test re-usability, avoiding costly re-programming activities and prevents tight dependencies with test equipment suppliers. In particular, the ASAM XIL 2.0 standard [ASAMXiL] evolved from a specification initially focused on hardware-in-the-loop (HiL) test systems [ASAMHiL], and now extends to more abstract test configurations, like model-in-the-loop (MiL) or software-in-the-loop (SiL). ASAM XIL 2.0 defines an API for programming test applications in C# or Python. The API provides classes to control diverse test components, like models, the SUT itself or electric error injectors. Therefore, it seems straightforward to scale-up Python test scripts to control Commercial-Off-The-Shelf test components compliant with the ASAM XIL specification, ranging from offline model-simulations of the systems to real-time test execution on a HiL test infrastructure.

5.2 Test re-use on PS-TTM simulations

This section introduces the Python-based PS-TTM Automatic Test Executor (ATE), a test automation component to provide repeatable test runs on PS-TTM simulators. The PS-TTM simulation framework consists of a set of tools and libraries of fault models intended to exercise the fault tolerance mechanisms implemented in the logical components of a dependable embedded system. The faults are injected on executable models, simulated with an E-TTM computation engine implemented with the SystemC library. The PS-TTM Automatic Test Executor (ATE) loads the definition of faults, sequences of input values and test points in standardized XML formats, based on the ASAM AE HIL standard. The values at the test points are recorded in data files with a standard structure. The test automation is achieved through Python scripts, executed by the ATE, whereas the low-level control of each simulator component is provided by Python APIs from dynamically loadable Python extensions.

Besides supporting the automated execution of comprehensive simulated fault-injection campaigns, the combination of standardized interchange formats for input/output values, the platform abstractions provided by the extension APIs, and the adaptability of the CPython interpreter brings an automated simulation environment parallel to real testbenches. This eases the re-use of test artefacts, from input stimuli even to test sequences, provided that suitable fault-injection units / saboteurs are available to re-create the simulated fault scenarios.

5.2.1 The PS-TTM Modeling and Simulation platform

The Platform-Specific Time-Triggered Model (PS-TTM) [ANP+14a] is a modeling and simulation platform for time-triggered safety-critical embedded systems, based on the Y-chart approach [BCG+97, KDV+97] and Model Driven Architecture (MDA) models. The goal of the PS-TTM framework is to give the designer an environment based on SystemC for the development and testing of time-triggered safety-critical embedded systems following the MDA.

In compliance to the Model Driven Architecture (MDA), the development of a system in the PS-TTM framework starts with the definition of a functional model, called Platform-Independent Model (PIM). Since the approach focuses on time-triggered systems, PIMs rely on the Logical Execution Time (LET) Model of Computation (MoC) [KS12]. The LET MoC defines the functionality of systems by specifying a logical duration for each computational job of the system, regardless of its physical duration. This permits the software engineers to communicate more effectively with the control engineers, since the properties of the system are closely aligned with the mathematical model of the control design.

Thus, a LET computation engine called Platform Independent Time-Triggered Model (PI-TTM) has been developed for the simulation of the PIMs [ANP+14c]. This engine has been built by providing an extension that imposes LET MoC constraints to the E-TTM simulation platform [PNO+10].

The design framework includes a library of HW components that can be assembled to generate a model of the target platform. In accordance to the MDA, once the PIM has been validated, it is deployed into the platform description model in order to generate the final PSM. The simulation of Platform-Specific Models (PSMs) is handled by the Executable Time-Triggered Model (E-TTM) engine. This gives the designers a higher freedom regarding the definition of the temporal behaviour of their systems, as the restrictions previously imposed by the LET MoC disappear.

This approach eases the modeling and validation of time-triggered systems, since it starts with the design of a purely functional LET-based model and enables a straightforward transformation into a platform specific E-TTM-based model, which guarantees that time-properties are intrinsically preserved in the final implementation when the platform is based on the Time-Triggered Architecture (TTA).

Appendix A expands the information about the features of the PS-TTM environment.

5.2.2 The PS-TTM Automated Test Executor

Herein we describe the properties and specifications of the **PS-TTM ATE**. The **PS-TTM** simulation platform includes a time-triggered simulated fault-injection framework. Testing teams would use this framework to assess the fault-tolerance mechanisms implemented in their **PS-TTM** models. The testing and simulated fault-injection tool is called **PS-TTM Automatic Test Executor (ATE)**.

From a testing perspective, the **PS-TTM ATE** features the following properties:

1. *Non-intrusiveness*: Non-intrusive fault-injection techniques are the ones that completely mask their presence, so that they have no effect on the system behaviour apart from the faults they inject. Therefore, the model is not modified to get the faults injected, which gives more reliable testing results.
2. *Availability for PIM and PSMs*: Testing and simulated fault injection can be applied to both platform independent and platform specific models, in order to facilitate the detection of design flaws at the earliest stages of the design.
3. *Repeatability*: The **PS-TTM ATE** provides repeatability to the testing and simulated fault-injection (**SFI**) activities, which enables to prove that the bugs found in previous versions of the models have been fixed in the newest versions.

The **PS-TTM ATE** is composed of the three modules shown in Figure 5.1:

1. The *Test-Case Interpreter (TCI)* sets the timed sequences of input signals and fault triggers from a test specification file. The **TCI** is composed of: (a) a Test Case Parser, (b) a Test Case Memory, and (c) a Test Case Data Generator.
2. The *Fault-Injection Unit (FIU)* sets the fault-injection points and configures the parameters for the fault behaviours. The **FIU** is composed of: (a) a Fault Injection Parser, (b) a Fault Set Memory, (c) the Fault Injector and (d) the Fault Libraries.
3. The *Test Points Manager (TPM)* sets the test points to capture signals as test outputs for later analysis. The **TPM** is composed of: (a) a Test Points Parser, (b) a Test Point Memory, and (c) a Test Point Data Recorder.

Test Case Interpreter (TCI)

The Test-Case Interpreter (**TCI**) is the module responsible for exercising the desired test cases. The **TCI** supports test automation, enabling the autonomous execution of different test cases defined by the test engineers. During the initialization phase of the tests, the **TCI** test case parser reads the test case specified by the test engineers and stores it in the test case memory. When simulation starts, signal-generators feed the SUT at each time tick with the signals corresponding to the test case stored in memory.

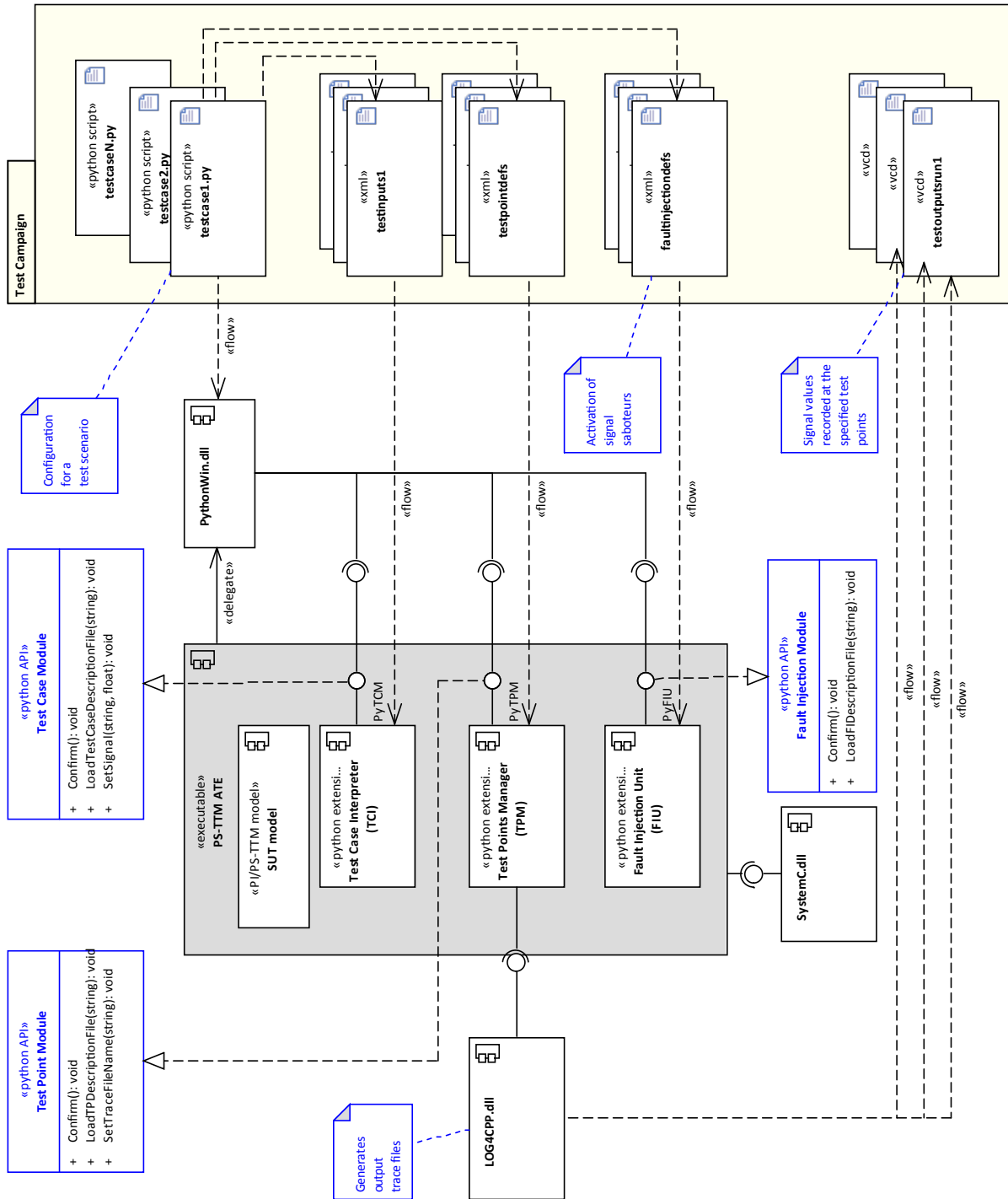


Figure 5.1: PS-TTM Automatic Test Executor

The test engineer defines test cases by timed sequences of input values for the test time span. A timed sequence is a set of tuples of time and signal values, the time value representing the instant at which the signal values will be set at the inputs of the **SUT**. The **TCI** holds the last value between each two consecutive signal change instants, feeding the **SUT** with the previous inputs. Listing 5.1 shows an example of the syntax of these files.

Fault Injection Unit (FIU)

The **FIU** provides simulated fault injection capabilities to the **ATE**. Specifically, the **FIU** enables the test engineers injecting different types of faults in the internal signals of the **SUT**. The Fault Injection Parser reads the fault configuration files (eXtensible Markup Language (**XML**) files) in the initialization phase of the simulation, and stores the fault injection campaign defined by the test engineers in its memory. Once the simulation starts, the **PS-TTM** engine automatically diverts the signals inside the **SUT** to the FI Set Interpreter & Fault Injector, which compares the properties of the received signals and the current time-stamp of the simulation with the Fault Injection Set stored in the memory. When a fault is triggered, the fault injector sabotages the signal according to the fault effect specified in the fault-injection campaign. To that end, the **FIU** uses a set of fault libraries for both platform independent and platform specific models. Once the signal has been corrupted, it is sent back to the **SUT** in zero simulation time, which guarantees that the temporal properties of the system are not affected by the fault injection process and remain intact when faults are applied. This way, the fault-injection process follows a non-intrusive approach, since the model of the system does not suffer any modification from the fault injection activities.

The **FIU** supports two different fault modes: transient and permanent. Permanent faults will remain active from their activation specified by the trigger time until the end of the simulation. However, *transient faults* are temporary misbehaviours, so their configuration requires to specify a duration in addition to the triggering instant. Once the injection of a *transient fault* is finished, the signal affected by the fault returns back to a non-faulty state.

The **XML** schema for the definition of fault-injection campaigns was derived from the ASAM AE HIL 1.0.0 standard for **HiL** testing. Although the aim of this work is to perform fault-injection experiments in model/software-in-the-loop configurations, close adherence to the standard eases the forward reuse of the fault-injection campaigns in the final prototyping phase, provided that fault injecting equipment is available. Section A.3 in the Appendix provides some examples of **FIU** configurations for **PIM** and **PSM** models of abstraction. Fault campaigns are optional: if omitted, the **ATE** will run a fault-free simulation.

Test Point Manager (TPM)

The Test Point Manager (**TPM**) is a module to observe the evolution of the internal signals of the **SUT** as the time flows.

As for the **TCI** and **FIU**, the configuration of the test points is read by the Test Point

Parser and saved in the Test Point Set during the initialization phase. During simulation the TP Set Interpreter identifies each signal of the SUT and sends the data of the signals specified in the TP Set to the Data Recorder, which stores the values of each signal along with its time stamp. When the simulation finishes, the TPM creates a value-change-dump

Listing 5.1: Example of an XML test case configuration file for the TCI module

```

1  <!-- TEST CASE -->
2  <TestCase>
3  <Setup instant="0.000">
4    <Set Variable="Encoder1" Value="0"/>
5    <Set Variable="Encoder2" Value="0"/>
6    <Set Variable="Acceleration" Value="0"/>
7    <Set Variable="NewBalise" Value="0"/>
8    <Set Variable="BalisePosition" Value="0"/>
9    <Set Variable="GroundInclination" Value="0"/>
10   <Set Variable="NextBalisePosition" Value="0"/>
11 </Setup>
12 <!-- More inputs might be defined between these time stamps -->
13 <Setup instant="95.750">
14   <Set Variable="Encoder1" Value="361"/>
15   <Set Variable="Encoder2" Value="361"/>
16   <Set Variable="Acceleration" Value="1.97327"/>
17 </Setup>
18 <Setup instant="96.000">
19   <Set Variable="Encoder1" Value="365"/>
20   <Set Variable="Encoder2" Value="365"/>
21   <Set Variable="Acceleration" Value="1.95508"/>
22   <Set Variable="NewBalise" Value="1"/>
23   <Set Variable="BalisePosition" Value="2000"/>
24   <Set Variable="GroundInclination" Value="0"/>
25   <Set Variable="NextBalisePosition" Value="3500"/>
26 </Setup>
27 <Setup instant="96.250">
28   <Set Variable="Encoder1" Value="369"/>
29   <Set Variable="Encoder2" Value="369"/>
30   <Set Variable="Acceleration" Value="1.93706"/>
31   <Set Variable="NewBalise" Value="0"/>
32   <Set Variable="BalisePosition" Value="0"/>
33   <Set Variable="GroundInclination" Value="0"/>
34   <Set Variable="NextBalisePosition" Value="0"/>
35 </Setup>
36 <Setup instant="96.500">
37   <Set Variable="Encoder1" Value="373"/>
38   <Set Variable="Encoder2" Value="373"/>
39   <Set Variable="Acceleration" Value="1.91921"/>
40 </Setup>
41 <Setup instant="96.750">
42   <Set Variable="Encoder1" Value="377"/>
43   <Set Variable="Encoder2" Value="377"/>
44   <Set Variable="Acceleration" Value="1.90151"/>
45 </Setup>
46 <!-- Test Case continues... -->
47 </TestCase>

```

file (*.vcd)² with all the data collected during simulation, in order to enable the test engineers to assess the emerging behaviour of the system.

Analogously to the FIU, the TPM reads the values of the internal signals of the SUT every time a job reads or writes a signal. The specification of the test point locations is similarly defined in XML files, using the following scheme: First, the file must specify the hierarchical location of the job that will read or write the signal. For each job, a set of entities (signal names) distributed in groups of inputs and outputs might be specified. The explicit specification of the job that will read or write the signal avoids ambiguities in the case in which a given name is repeated between ports of different jobs; moreover, specifying whether the signal is treated as an input or output by the job enables distinguishing between the reading and writing activities of a job, which might be useful in systems where a given job reads a variable at the beginning of its execution and re-writes the same variable at the end.

Listing 5.2 shows an example of the syntax of these files.

5.2.3 The PS-TTM Synchronous Python Interpreter

This section describes the internal adaptations carried out to integrate the Python ATE into the PS-TTM simulator. The Python ATE was embedded as an extension to the PS-TTM simulation framework, with the goal of enabling an accurate evaluation of the functional and non-functional properties of the models in PS-TTM, with a special focus on their fault-tolerance properties. This Python shell interpreter helps test engineers at either running a fully automated simulation, or manually modifying inputs or conditions in the test environment on-the-fly. The interactive experimentation eases the validation of the test suites, thus accelerating the preparation of test artefacts. Figure 5.2 represents the integration between the components introduced in Section §5.2:

- *Automated Test Executor (ATE) System*: This is the core *Modified Python Interpreter* in the Automatic Test Executor (ATE) System. We opted for integrating a Python console to load the test scripts, which enables interactive user command execution, enhancing the PS-TTM simulator usability. The PS-TTM simulator is based on the cross-platform E-TTM SystemC extension [Per11], that can be compiled for different hosts and operating systems (e.g., Linux, Windows). Similarly, the Python IDLE is a cross-platform Python console, and thus was considered for the interactive ATE implementation. However, Python IDLE detaches the execution of scripts to a new Python interpreter instance, which complicates the synchronization with the human-machine interface (HMI) IDLE instance. Due to this, we switched to the Windows-specific PythonWin [PYW] console. PythonWin is an open-source application, so that

²The value-change-dump file format is an industry-standard format to capture signal traces and waveforms. *.vcd files can be later imported to a number of SW applications, e.g., GTKWave [GTKW] (open source), EZWave (by Mentor Graphics) [EZW], or SimVision (by Cadence) [SimVision].

Listing 5.2: Example of test points specification file (XML file)

```

1  <!-- TEST POINT LOCATIONS -->
2  <Locations>
3    <!-- NODE_EVC_A -> PROC -> CORE1 -> JOB_ODOMETRY -->
4    <Job hierarchy="systemTSS.clusterEVC.nodeEVCA.procA.core1.jobOD0">
5      <Inputs>
6        <Entity>DAS_EVC_IN_ENC1</Entity>
7        <Entity>DAS_EVC_IN_ENC2</Entity>
8        <Entity>DAS_EVC_IN_ACCEL</Entity>
9        <Entity>DAS_EVC_IN_BAL_NEWBAL</Entity>
10       <Entity>DAS_EVC_IN_BAL_POS</Entity>
11       <Entity>DAS_EVC_IN_BAL_INCL</Entity>
12       <Entity>DAS_EVC_IN_BAL_NEXTPOS</Entity>
13     </Inputs>
14     <Outputs>
15       <Entity>DAS_EVC_ST_S</Entity>
16       <Entity>DAS_EVC_ST_V</Entity>
17     </Outputs>
18   </Job>
19   <!-- NODE_VOT_A -> PROC -> CORE -> JOB_VOTER_A -->
20   <Job hierarchy="systemTSS.clusterEVC.nodeVotA.procVotA.coreVotA.jobVotA">
21     <Inputs>
22       <Entity>CORE_VOTERA_IN_EMERG_A</Entity>
23       <Entity>CORE_VOTERA_IN_EMERG_B</Entity>
24       <Entity>CORE_VOTERA_IN_EMERG_C</Entity>
25       <Entity>CORE_VOTERA_IN_SERV_A</Entity>
26       <Entity>CORE_VOTERA_IN_SERV_B</Entity>
27       <Entity>CORE_VOTERA_IN_SERV_C</Entity>
28       <Entity>CORE_VOTERA_IN_WARN_A</Entity>
29       <Entity>CORE_VOTERA_IN_WARN_B</Entity>
30       <Entity>CORE_VOTERA_IN_WARN_C</Entity>
31     </Inputs>
32     <Outputs>
33       <Entity>CORE_VOTERA_OUT_EMERG</Entity>
34       <Entity>CORE_VOTERA_OUT_SERV</Entity>
35       <Entity>CORE_VOTERA_OUT_WARN</Entity>
36       <Entity>CORE_VOTERA_OUT_FAILURE</Entity>
37       <Entity>CORE_VOTERA_OUT_SYSTEMFAILURE</Entity>
38     </Outputs>
39   </Job>
40   <!-- NODE_VOT_B -> PROC -> CORE -> JOB_VOTER_B -->
41   <Job hierarchy="systemTSS.clusterEVC.nodeVotB.procVotB.coreVotB.jobVotB">
42     <Inputs>
43       <Entity>CORE_VOTERB_IN_EMERG_A</Entity>
44       <Entity>CORE_VOTERB_IN_EMERG_B</Entity>
45       <Entity>CORE_VOTERB_IN_EMERG_C</Entity>
46       <Entity>CORE_VOTERB_IN_SERV_A</Entity>
47       <Entity>CORE_VOTERB_IN_SERV_B</Entity>
48       <Entity>CORE_VOTERB_IN_SERV_C</Entity>
49       <Entity>CORE_VOTERB_IN_WARN_A</Entity>
50       <Entity>CORE_VOTERB_IN_WARN_B</Entity>
51       <Entity>CORE_VOTERB_IN_WARN_C</Entity>
52     </Inputs>
53     <Outputs>
54       <Entity>CORE_VOTERB_OUT_EMERG</Entity>
55       <Entity>CORE_VOTERB_OUT_SERV</Entity>
56       <Entity>CORE_VOTERB_OUT_WARN</Entity>
57       <Entity>CORE_VOTERB_OUT_FAILURE</Entity>
58       <Entity>CORE_VOTERB_OUT_SYSTEMFAILURE</Entity>
59     </Outputs>
60   </Job>
61 </Locations>

```

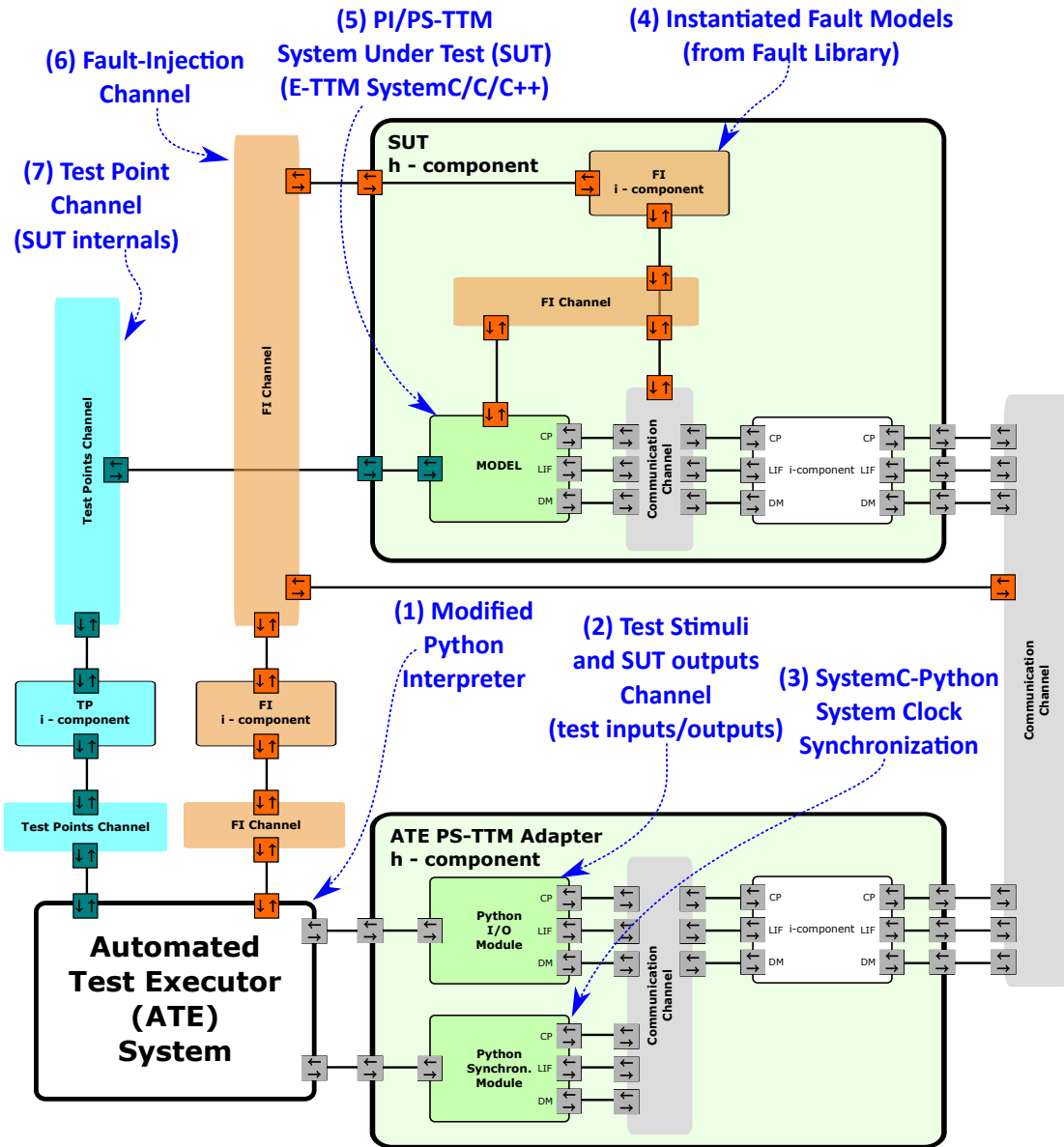


Figure 5.2: Integration of the Python interpreter in the PS-TTM simulator

we could re-write parts of the original Python interpreter, in particular, the access to the system (host) time. This way, we implemented a basic time virtualization by which an external simulation engine (i.e., SystemC) could update the virtual host time. The virtual Python clock supports the synchronous execution of test scripts in which time measurements rely on the host system time, with the time resolution offered by this timer.

- *ATE PS-TTM Adapter*: This is hierarchical **PS-TTM** component that acts as an interface to the *external* inputs and outputs of the **SUT**, as well as provides the virtual-clock source for the Python interpreter. The ATE PS-TTM Adapter has the following components:
 - *Test Stimuli and SUT Outputs Channel*: This **PS-TTM** component supports the Python **ATE** with signal generators to stimulate the **SUT** with timed sequences of input values. This component also captures the external outputs of the **SUT**, that are fed back to the **ATE** for evaluation or logging.
 - *Python Synchronization Module*: This component updates the virtual host time in the embedded **PS-TTM** Python interpreter. It is intended to intercept `wait(time)` Python statements. Upon such a statement the Python interpreter freezes until the specified amount of time elapses. This makes Python behave as executing in a **TT MoC**, as the underlying **PS-TTM** does.
- *SUT*: The **SUT** is the **PS-TTM** hierarchical component that contains the PI/PS-TTM model of the system under test. It consists of:
 - *Model*: The *Model* component wraps the functional or platform-specific models of the system in a **PS-TTM** executable component. The system model provides observation ports to access to internal signals. These signals are relevant to observe the system reaction to internal faults injected by the signal saboteurs. Their purpose is to give insight about the degraded behaviour of redundant safety systems, as those normally would mask the effect of faults in their external outputs (these are collected through the *Test Stimuli and SUT Outputs Channel* instead). This information is required to evaluate the actual fault tolerance of the **SUT** model.
 - *FI Component*: The *FI Component* contains the fault injectors (signal saboteurs) required for the fault-injection campaigns. Fault injectors are not fixed within the **SUT** model. Instead these are dynamically instantiated at runtime, based on the fault-configuration model, and relying on the SystemC signal introspection feature. This constitutes the non-intrusive approach to fault experiments.

Embedded Python ATE API

The Python Automatic Test Executor (ATE) embedded in the PS-TTM simulation framework provides an API of Python functions to configure and control the simulations. The user of the PS-TTM can either invoke the API interactively from the embedded PythonWin shell, or alternatively may run a Python script to automate the simulation process. The API consists of the functions listed below:

- `fimodule.LoadFIDescriptionFile(string filename)`: This function loads the fault-injection configuration from the file path specified by the `filename` argument. Then, the fault configuration is pre-processed by the FIU component that sets the parameters for the fault models that could be activated during the simulation. The test user may load multiple fault configuration files, in order to run several fault-injection campaigns during a single simulation. The fault configuration is optional: if omitted the ATE will run a fault-free simulation.
- `fimodule.Confirm()`: This function instructs the FIU that the setup of the fault-injection campaign is finished. When this command is sent to the FIU, the FIU notifies the ATE that it is ready to start. It is always required to call this function in order to start a simulation.
- `tcmodule.LoadTestCaseDescriptionFile(string filename)`: This function loads the input values for a test case from the file path specified by the `filename` argument. Test input values are timed sequences of values for multiple signals, which are fetched to the simulator by the TCI component. During the simulation Data Generator feeds to the SUT the input values corresponding to the simulation time.
- `tcmodule.SetSignal(string signal, float value)`: This function can be invoked at any instant during the simulation to modify input signals. The test user can invoke this command interactively to carry out experiments on the SUT model.
- `tcmodule.Confirm()`: This function notifies the TCI that the test-case configuration is completed, and the TCI is ready to start the simulation. The ATE waits until the rest of the required PS-TTM modules notify their readiness. Then the initialization phase finishes and the simulation begins. It is always required to call this function in order to start a simulation.
- `tpmodule.LoadTPDescriptionFile(string filename)`: This function selects the test output signals from the file path specified by the `filename` argument. This configuration is pre-processed by the Test Point Manager (TPM) component, to record traces of the signals to be observed. The user can select multiple sets of test points for a single simulation run; in such a case, all the specified signals are recorded. The test-point configuration is optional: when omitted, no information about internal signals is recorded.

- `tpmodule.SetTraceFileName(string filename)`: This function specifies the output file to store the signal traces recorded by the TPM over the simulation. The output format is the *value-change-dump*. This setup can be skipped if no test-point specification was previously given.
- `tpmodule.Confirm()`: This function notifies the TPM that the setup of the test-point configuration is finished. When this command is sent to the TPM, it notifies the ATE that it is ready to start. When the ATE receives the same notification from the other two modules, the initialization phase finishes and the simulation begins. It is always required to call this function in order to start a simulation.

5.3 Test re-use on Simulink simulations

The PS-TTM provides a non-intrusive fault simulation framework intended to exercise the fault tolerance mechanisms on the redundant structures used in safety systems. The PS-TTM simulation engine is provided by SystemC, which could be extended to support additional models of computation (MoCs) in order to simulate non-TT plant models, that should be coded in C/C++, then compiled and linked to build the simulator. This process could be less productive than using a COTS simulation environment with ready-to-use model libraries at the initial concept phase of a project. Therefore we tried a similar synchronization approach to integrate a Python ATE inside a Simulink model. The adaptation process consisted of these steps:

1. Define the ATE signal interfaces (i.e., Input/Output (I/O) ports) to the Simulink model of the SUT.
2. Implement the I/O port interfaces in a Simulink S-function, and also as a Python extension³ [MWSF17] to encapsulate the Python interpreter as an executable Simulink block. I/O ports in Simulink correspond to the I/O values read/written when the S-function is invoked. The I/O values are copied to internal memory that is accessible to Python I/O extensions. The Python I/O extension provides an abstraction to the Python test script so that it could be implemented with the actual I/O interfaces of a real testbench.
3. Instantiate a PythonWin console with the modified Python interpreter.
4. Update the Python system host time synchronization within the update method in the S-function.

³ A Simulink S-function is a dynamically loadable Simulink extension. A S-function can be written in C/C++ programming language, and compiled for the simulation host processor and operating system. It is also possible to link an S-function to external libraries, which provides great flexibility to customize the Simulink simulation.

5.3.1 Validation

The test re-usability with the Python **ATE** embedded in Simulink was validated using this scenario:

1. An elevator manufacturer developed a testbench for functional testing of cabin doors. The testbench was automated with a Python **ATE** extended with custom modules to: (i) command the door under test and (ii) read the limit switches and position sensors activated with the door motion. A Python test script defined a test suite with multiple test cases for opening/closing the door, verifying that the sensor signals reached the expected state within a time interval. Due to the dynamics of the door, the Python host/system time sufficed to evaluate the timeliness of the door response to the inputs.
2. A simplified Simulink model of the elevator door was also developed for the **HiLES** elevator simulator described in § 8.2.
3. We developed a Simulink model of the testbench (see Figure 5.3), re-using the Simulink door model, and connecting it to a virtual Python **ATE** embedded in an S-function.
4. We wrote an alternative implementation of the abstract Python class to access the door **I/O**, this time redirecting the read/write operations to the Python extensions operated from the Simulink S-function.
5. During simulation, the Python console launched the Python test scripts originally developed for the real elevator door, but instantiating the class implemented to access the now virtual testbench **I/Os**.

The simulation campaign with the Simulink environment successfully replicated the behaviour of the real testbench. This supports the development of test scripts that are first validated on the virtual door testbench (i.e., the Simulink environment) and then can be re-run on the real testbench. Listing 5.3 provides an example of a re-usable Python test script that could be executed either on the virtual Simulink testbench or on a real door testbench.

5.3.2 Limitations

The presented approach has the drawback that the S-function/Python **I/O** interface has to be re-programmed each time the **SUT** interface is modified. This impacts both parts of the program, the C code for the S-function wrapper, and the C/C++ code for the Python extensions required to read/write the **ATE** inputs and outputs. However, the inconvenience can be mitigated with a thorough **I/O** interface design before implementing the S-function and the Python extension modules, thus reducing later rework.

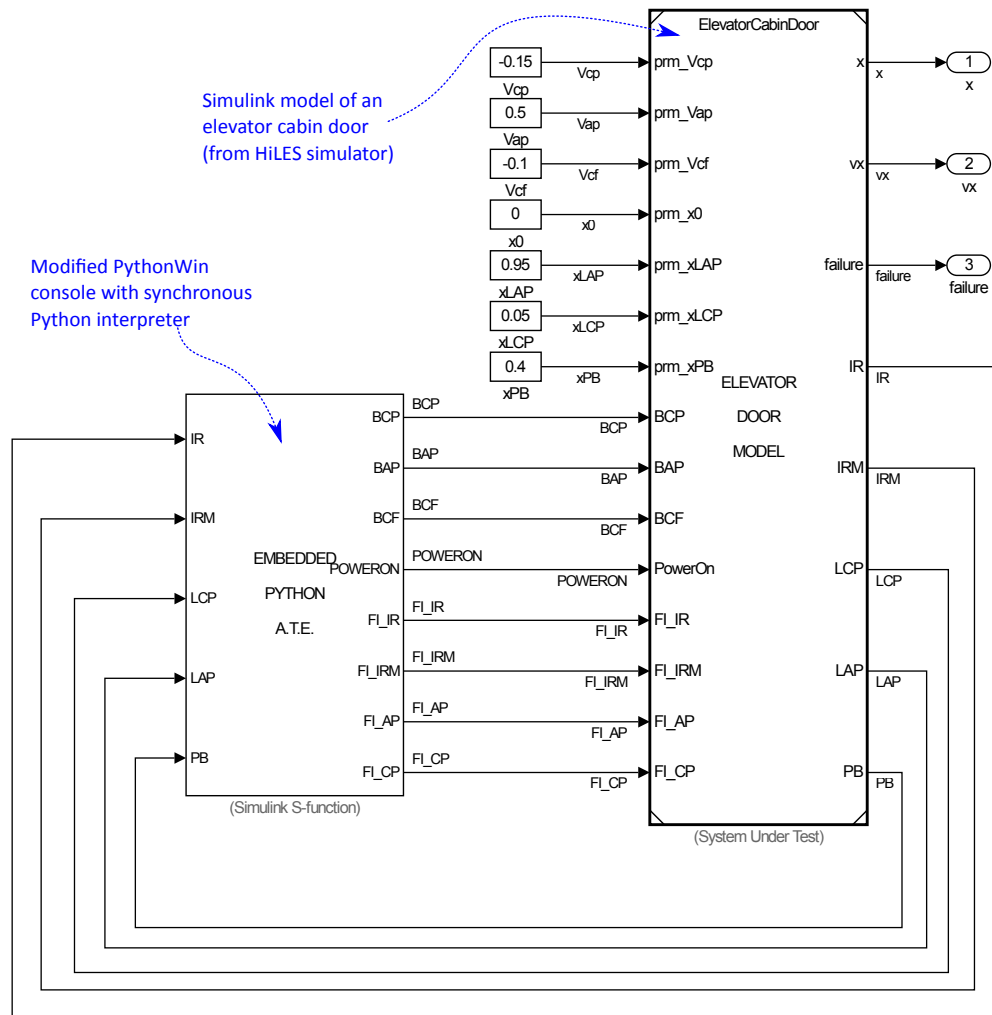


Figure 5.3: The virtual elevator door testbench in Simulink

5.4 Discussion

This chapter introduced a testing and simulated fault-injection framework for time-triggered dependable-systems based on the **PS-TTM** approach. The environment enables testing and injecting faults at different stages of the design, from platform independent models to platform specific models, which enables an early detection of design flaws in the system.

The Automatic Test Executor (**ATE**) presented herein is composed of three different modules for the design and simulation of test-cases, injection of faults during simulation and storage of simulation results for the evaluation of the behaviour of the system under

Listing 5.3: Example of a re-usable Python test script

```

1  import unittest          # basic testing framework
2  import ElevatorDoorProxy # wrapper to abstract from test I/Os
3  import time              # host timing
4
5  class TestLibrary(unittest.TestCase): # The test library
6      global theSUTproxy # Global object to access the tester I/O ports
7      t_max_to_open = 10 # Test parameter
8      ... # Other test attributes
9      def assertInputs(self, _IR, _IRM, _LAP, _LCP): # Input assertions
10         inputs = theSUTproxy.getDI() # Acquires the SUT outputs
11         # Verdict
12         if inputs.IR != _IR:
13             self.fail(errdescription)
14         ... # similar for all inputs
15     def testOpen(self): # Test door moves from closed to open
16         # 1. Sets to initial conditions (i.e., door closed)
17         ...
18         # 2. Commands the door to open
19         try:
20             theSUTproxy.open()
21         except:
22             self.fail(errdescription)
23         # 3. Waits
24         time.sleep( self.t_max_to_open ) # Synchronizes with
25             simulator
26         # 4. Reads and asserts inputs
27         self.assertInputs( self.OFF, self.OFF, self.ON, self.OFF )
28         ... # Restores the door state
29         return
30     def testClose(self):
31         ... # Similar to testOpen
32         ...
33         # More tests methods
34         ...
35
36 class TestSuite(unittest.TestSuite): # The test suite
37     testList = ("testopen", "testClose", ...)
38     def __init__(self):
39         unittest.TestSuite.__init__(self, map(TestLibrary, self.
40             testList))
41     def run(self, result):
42         unittest.TestSuite.run(self, result) # Runs the test cases
43
44 def testElevDoor(): # Runs the tests
45     global theSUTproxy
46     theSUTproxy = ElevatorDoorProxy.ElevatorDoorProxy()
47     testResult = unittest.TestResult()
48     testSuite = TestSuite() # instantiates the test suite
49     try:
50         testSuite.run( testResult ) #executes the test
51     finally:
52         del testSuite, testResult
53
54 if __name__ == '__main__':
55     import sys
56     ## print __doc__
57     sys.argv.append('-v')
58     testElevDoor()

```

such faults. The ATE is synchronized with the simulation time of the SUT in a way that functional tests and fault-injection experiments become reproducible.

The simulated fault-injection technique is non-intrusive, i.e., enables injecting faults in the system models during simulations without the need of performing any modifications to them. This is achieved by monitoring the signals of the SUT and modifying their values if required. The framework provides the user with a library of faults in order to configure the fault-injection experiments. The ATE imports these configurations for carrying out the simulations. As the mapping of LET and E-TTM-based models to time-triggered architectures is straightforward, this eventually facilitates the re-usability of tests even on real prototypes, provided that we could build a test harness with equivalent real-world Fault-Injection Units (FIUs).

We evaluated our framework in a case study consisting of a railway signalling system. We modelled the system at both PIM and PSM levels, and checked the behaviour of the system under different faults by means of the simulated fault-injection (SFI) capabilities provided by the framework. The results of the case study evaluation are described in §8.1

6

Models and Code Re-use for COTS X-in-the-Loop Testbenches

This chapter introduces a model and code re-use approach to build simulators for the functional verification of real-time controllers under multiple test configurations (X-in-the-loop). The Model-Based Development (MBD) relies on a Commercial-Off-The-Shelf (COTS) tool-set framework, that provides a suitable level of abstraction while separating the description of the functionality from the Model of Computation (MoC), i.e., the models are agnostic of the MoC. This separation of concerns enables the implementation of the MBD-generated test components into a COTS real-time Heterogeneous Computing Platform (HCP), even supporting the re-deployment of a component to alternative execution contexts (e.g., from processor to FPGA). To this purpose, the test designer has to refine an orthogonal MoC specification model suited to the selected platform, yet the possible platform constraints shall be considered beforehand, e.g., limitations regarding the components or signal types that would eventually undergo the synthesis process.

This chapter demonstrates how the proposed workflow enables a preliminary model-in-the-loop (MiL) validation of the test models, independently of the final test hardware (HW). Then current COTS automated coders translate the test component model into either hardware description language (HDL) or a conventional programming language, e.g., the C programming language. This chapter examines some of the pitfalls that could hinder the re-usability of the test component model at this stage, like the peculiarities of heterogeneous test platforms. The approach will be exemplified by application to the development of a hardware-in-the-loop elevator simulator for validating motion and safety functions deployed in an elevator control system.

6.1 Problem Statement

The development process for high-integrity safety-critical systems imposes a strict separation between the development and verification, validation and testing (VVT) artefacts. Ideally the development and evaluation phases should only share the specifications. The development of COTS computer simulators, and the generalization of co-simulation interfaces enable a staged simulation campaign, according to a previously established VVT plan. The VVT workflow may comprise:

1. An initial MiL approach, where abstract test models help at refining a library of test cases. The System Under Test (SUT) would eventually be replaced by a functional model of the expected behaviour, developed by the test engineers and based on the specifications, which are shared with the development team. In this phase, test engineers can perform simulation analysis to measure the test coverage and improve the test cases as needed. While MiL enables fast iterations to improve the quality of the test, the results may differ with those obtained in the subsequent tasks, when more detail is included in the simulation framework.
2. An intermediate software-in-the-loop (SiL) stage enables a cross-check of the software (SW) source code artefacts provided by the development team versus the test harnesses and test cases developed by the VVT team. In this phase, both teams gather feedback about their respective accomplishments: safety developers get test reports addressing potential defects, while they can use the simulation framework for SW debugging. The SiL iterations are slower than the MiL, as these require components provided by independent teams, i.e., the developers and the VVT teams.
3. The hardware-in-the-loop (HiL) test component provides a real-time simulation of the plant using the physical interfaces of the SUT. Test engineers use HiL verifications to examine the behaviour of the SUT under special situations that could be difficult or expensive to generate using a real context. To that purpose, the HiL simulators shall mimic the expected behaviour of the SUT operational context with sufficient accuracy. A problem with the HiL approach is that could require the development of specific models, e.g., sensor or actuator models with controlled fault-injection, or special instrumentation models that also require a validation process, in the sense that the model is capable of reproducing the abnormal behaviour as expected by an application expert.

The combined use of all these simulations in a development project introduces a number of new artefacts required in each phase. HiL simulations are the most demanding, as both an adequate signal fidelity and a deterministic real-time behaviour are needed to resemble a realistic environment to verify a system. Simulator modules updating at high sampling rates are usually targeted to programmable logic. Several HiL vendors provide specific IP libraries for FPGAs to simulate commonly used devices. The associated models are drawn from the

libraries to the **HiL** programming environment, then connected with the user-provided plant models, and finally, all of them get compiled and bundled in a bit-stream file for the **FPGA**. However, the related **IPs** are proprietary and cannot be customized if necessary.

Another issue arises when we require a testbench that integrates custom pieces of **SW** within the **HiL** system. This would be the case when the **SUT** communicates to custom protocols with a remote device. The **HiL** system setup could become a very labour-intensive task, risking a late delivery that would eventually compromise the completion of the development project on time.

Summing up: while the combination of several X-in-the-loop (**XiL**) approaches eases the early detection of errors it also requires a considerable preparation work. The problem is how to make a profit from the validation effort spent in **MiL/SiL** simulations to alleviate the setup cost of an equivalent **HiL** system. From **MiL** simulations we could benefit from an automated translation of validated utility models into programming artefacts suited to the **HiL** heterogeneity. From the **SiL** simulations we could benefit from portable **SW** artefacts that facilitate a cross-platform deployment from the host simulation platform to the **HiL** processor.

6.2 XiL Testbenches for Dependable Control Systems

The steadfast rise in computing power of embedded processors support the development of increasingly complex deployments of dependable applications. In many fields of application the expected life-time of functional safety systems exceeds a decade. At the same time, the fast innovation pace in electronics leads the manufacturers of embedded control systems to maintain different generations of electronic devices over the life span of the system/machinery, to provide retrofitting supplies for legacy electronics. One way to cope with this issue is to develop a 'universal' product concept, designing the control system hardware and software to keep interface- and functional compatibility with earlier product generations, while supporting the latest innovations. A key requirement to succeed in this approach is to provide suitable testbenches for regression and backward compatibility testing.

In particular this thesis tackles the testing of dependable control systems with safety functions related to motion supervision and control, where innovations in sensor components also foster the development of dependable systems with a higher technical performance. For instance, the introduction of state-of-art absolute position sensors improves the overall availability of an elevator system by shortening the start-up time.

Nevertheless this poses specific integration-testing challenges: building up testing facilities for motion controllers is costly, allows limited operational configurations, requires skilled staff to setup the full system, and the availability is low due to a high demand for regression testing of an evolving product. In this context a **HiL** simulator constitutes a cost-effective mean for testing embedded electronics, even though its maintenance and upgrading pose issues similar to those of the embedded systems. Moreover, the complexity of the **HiL**

simulators could exceed that of the devices or systems under test, thus requiring special computing platforms. An excessive dependency of the testing components on a particular deployment target becomes the main bottleneck as the testing platform gets outdated.

Several state-of-art **HiL** simulators offer the feature to simulate multiple communicating devices in real-time for testing distributed systems. In the automotive domain this capability is known as 'Restbus' simulation, meaning that the test system simulates multiple Electronic Control Units (**ECUs**) to represent the communication bus load (the 'rest of the bus') and provide appropriate signals to verify the functionality of the **ECU** or system under test. Whether this is done by running functional messaging models or by compiling platform-independent code, the building, integration and verification of these additional test components with current commercial-of-the-shelf test systems may require a substantial start-up effort. On the other hand, in a regression-testing scenario most of the networked embedded products in the distributed system would have previously undergone a thorough verification and validation. Reusing code from the actual devices to build virtual replicas of these devices could be a more cost-effective approach. In this context, the main technical challenge is the implementation of the software interface between different modules, also possibly written in different languages.

Therefore, it is crucial for manufacturers to preserve the knowledge about the product and the related test systems while minimizing all these inconveniences.

The adoption of a model-based development process for the test system provides a solid foundation for the future evolution of the product line. Furthermore, in order to shorten the time-to-market sometimes the manufacturer starts the integration of a new component concurrently relying on the specifications, even if the new part is not yet available. Current model-based development tools could be used to build functional mock-ups of those parts, which can then be deployed into the test system, effectively shortening the overall integration phase.

This chapter contributes a model-based workflow to ease the construction of components for a heterogeneous **COTS HiL** test system, by re-using **MiL** and **SiL** artefacts previously validated in **COTS** modelling environments. We address the capabilities and limitations of two mainstream **MBD** environments to support the development of such test components and the deployment into a real-time reconfigurable computing platform: MathWorks' Simulink product family [**SL**], and National Instruments' LabVIEW [**LV**]. We also describe the problems found when integrating the obtained test artefacts in a **HiL** system based on the **COTS** NI CompactRIO (cRIO) modular controllers, and possible the workarounds to solve them.

6.2.1 HiL System with cRIO COTS Heterogeneous Platforms

The IEC-61508 safety standard establishes requirements for the complete life-cycle of a functional safety system. As a rule, safety-related artefacts should be preserved until the complete retirement of all the product units. Under this assumption, and considering an

expected operating life of about 10 years, we preferred **COTS** computing platforms with modular **I/O** and interfaces to build the **HiL** test systems. Currently marketed **COTS HiL** systems for high-end test applications show a similar architecture combining multi-core processors and **FPGAs**. Interchangeable modules implement the tester interfaces (e.g., **I/Os**, bus communications), and a Real-time Operating System (**RTOS**) warrants the time-determinism of the simulations in real-time. Such an architecture is found in the SCALEXIO system by dSPACE [**SCALEXIO**], the Real-time Target Machines from Speedgoat [**RTTM**], the Hypersim simulator from Opal-RT [**OPALRT**] or the NI PXI solutions for **HiL** simulators [**NIPXIHIL**]. Most of these systems can be scaled-up by interconnecting multiple **HiL** units, resulting in a multi-core / multiple **CPU** distributed **HiL** system.

For the scope of this thesis we selected the Compact RIO (**cRIO**) controllers from National Instruments [**cRIO**]. The **cRIO** controllers are heterogeneous computing platforms intended for deploying real-time control applications that integrate a multi-core processor and a Xilinx Field-Programmable Gate Array (**FPGA**). The different **cRIO** variants offer scalability of the computing resources while sharing most of the modular parts and supplies, and a common LabVIEW programming environment. In the scope of this chapter contribution the **cRIO** platform was used to build a **HiL** elevator simulator (**HiLES**). **HiLES** replaced a preceding elevator simulator, where the substantial difference is that in the new deployment we targeted the **FPGA** to run the time-critical simulation models, including the elevator car dynamics and the associated sensor models. **HiLES** served for the evaluation of the **MiL/SiL** re-use approaches described herein.

The **HiL** elevator simulator consists of:

1. A real-time heterogeneous, reconfigurable computing platform for simulating the dynamic behaviour of electro-mechanical elevator subsystems.
2. Signal adapters to replicate the actual interfaces of sensors and actuators.
3. A graphical user interface to select the simulator settings, monitor the current state, record data, and trigger fault injections.

We used two **HiLES** system variants:

1. **HiLES** with **cRIO-9082** [**cRIO9082**], that integrates a dual-core Intel i7@1.33 Ghz processor, a Xilinx Spartan-6 LX150 **FPGA**, and uses the Phar Lap ETS **RTOS**.
2. **HiLES** with **cRIO-9039** controller [**cRIO9039a**], a heterogeneous computing platform featuring an Intel ATOM Quad-Core 1.91 GHz CPU processor, a Xilinx Kintex-7 325T **FPGA**, and NI Real-time Linux **RTOS**.

In the slots of the **cRIO** controllers we plugged C series modules to interface with the **SUT**: *digital input/output* for switches, command inputs and incremental encoders and

CAN interfaces for communications. An Ethernet port connects the **cRIO-9039** simulator to a remote test control application. The subsystems of the plant model requiring a fast sampling rate are computed in the FPGA to increase the throughput and timing coherence of the encoder signals. Lower sampling rate subsystems run in the **cRIO** processor, besides the communications server to the remote test controller. The processor runs a LabVIEW-RT application that handles the CAN port allocated to the Restbus simulator and orchestrates the overall synchronization.

6.2.2 Model-based workflow

We program the **HiLES** simulator in a mixture of languages, adopting model-based- or conventional- SW development processes when better suited. This is illustrated in Figure 6.1. The bitstream file for the FPGA is synthesized from a mixture of **HDL** files, some of which derive from Simulink models and are imported to the LabVIEW-FPGA project. Additional user-defined extensions can be imported as shared libraries in the **HCP** processor execution context. For instance, the test artefact obtained by re-using code-artefacts is the embedded CAN Restbus extension.

When specifying the new elevator simulator we faced the challenge of switching the computing platform from a conventional processor to an **FPGA** implementation. The preceding simulator was programmed in the C programming language, as a multi-thread/multi-rate application using two synchronisation sources: a fixed-cycle timed task read the elevator controller outputs and the operator commands, while a variable-cycle timer triggered the model computations, including the simulated position switches and the incremental encoder. This platform-specific timing control jeopardized the integration of new functions as the complexity of the simulator increased. The main driver to adopt an **MBD** approach for developing the new simulator was the need to reduce the burden of integration testing. **MBD** allows us to validate the new functions at the design phase, unveiling earlier potential problems and fixing them before programming the actual hardware.

We selected the MathWorks toolset to support our **MBD** starting the design from abstract functional descriptions. The functions can be described in a variety of languages: MATLAB code, Simulink models, Stateflow models, custom C-code. Complex functions can assemble parts expressed in different languages. Modeling with mixed-languages could actually shorten the development time, but also has some drawbacks: first, it requires more development tools, increasing both the tool acquisition and maintenance costs; second, the resulting model depends on more components, increasing the likelihood of maintainability issues; third, free selection of modeling language and modeling style may potentially bring project teams to develop different models with degraded readability, unless enforcing guidelines like MAAB [MAAB]. To minimize dependencies on MathWorks toolboxes, we limited ourselves to use those listed in Table 6.1.

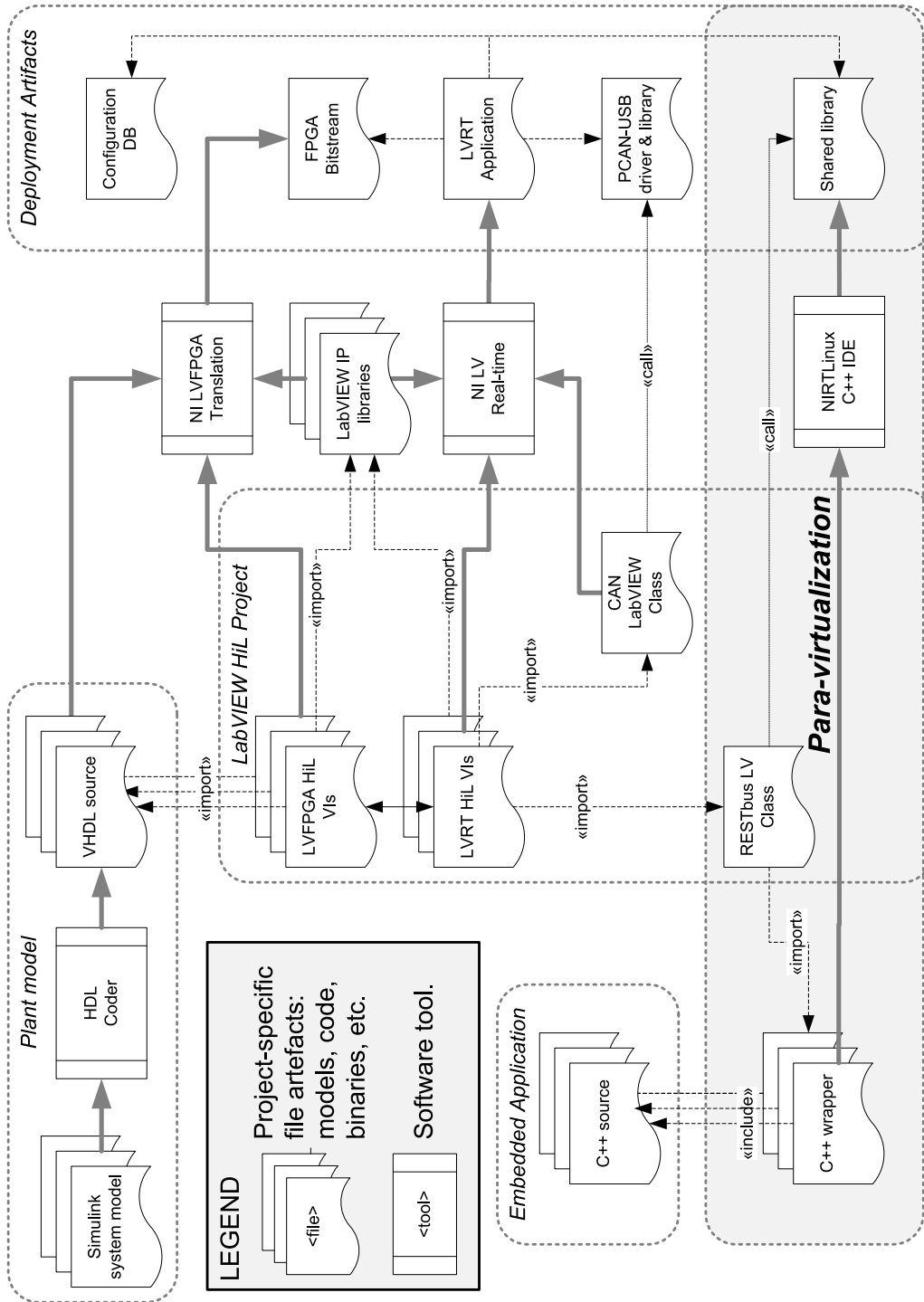


Figure 6.1: Tool-set and workflow for the COTS HiL cRIO simulator

Table 6.1: Software toolset to re-use sensor test models from MiL to HiL.

Vendor	Application	Purpose
MathWorks	MATLAB	Pre-requisite for Simulink.
	Simulink	Model authoring, simulation.
	Fixed-Point Designer	Fixed-point arithmetic support.
	HDL Coder	Model to HDL transformation.
	Simulink Verification & Validation	Model sanitization.
National Instruments	LabVIEW	Pre-requisite for LabVIEW/RT and LVFPGA.
	LabVIEW/Real-time	Programming the NI cRIO processor.
	LabVIEW/FPGA	Programming the NI cRIO FPGA.

We developed our new elevator simulator as a set of Simulink models compliant with the requirements of the HDL Coder transformer [HDLC]. We also developed a number of test-harnesses to simulate comprehensive validation test scenarios. This model base constitutes a valuable asset, as it captures the manufacturer's intellectual property in an executable form, thus fostering further design-space exploration by supporting simulated experiments to evaluate the performance of new designs. Once validated in simulation we transform the Simulink models to HDL [IEEE1076] files using HDL Coder.

At the final step we program the cRIO controllers using the National Instruments software listed in Table 6.1. We import the HDL files as LabVIEW IP Integration Nodes into the LabVIEW/FPGA Virtual Instruments (VIs) for the cRIO target.

6.3 Re-using Test Artefacts from MiL to HiL

This section presents the re-use of Simulink models, validated in a MiL configuration, to build test components which can be deployed in a COTS HiL heterogeneous test system. In particular we illustrate how to synthesize some types of position encoders as well as a customized synthetic instrument. These synthetic sensors and instruments ease the real-time functional verification of critical systems where safety functions depend on motion measurements, e.g., position or speed.

6.3.1 Requirements for Real-time Simulation of Position Sensors

A motion controller typically integrates position measurements from diverse sensors, exploiting the information redundancy for condition monitoring and diagnosis functions. The controller could detect a faulty device by analysing the temporal correlation of the signal values. A key requirement for a real-time plant simulator is the temporal coherence of the information: latencies in the simulation platform can foul the motion controller, bringing it into an emergency stop.

The required cycle times for the real-time simulation of incremental encoders depend on the speed of moving elements and the encoder resolution. To simulate fast systems with high resolutions, we would eventually exceed the minimum achievable cycle time of most **COTS HiL** processors. **FPGAs** prove advantageous for this purpose, as we can achieve short cycle times (e.g., below 100 nanoseconds), and the processing parallelism of an **FPGA** enhances the scalability of the **HiL** simulator without degrading the real-time performance. Besides, the Simulink language is inherently parallel, which eases the translation to **HDL** code required to implement the model in an **FPGA**.

On the other hand, implementing the simulator models in **FPGAs** has some drawbacks: the implementation of floating-point arithmetic operators requires additional IPs, that consume more area and resources than fixed-point implementations; fixed-point arithmetic requires specialized toolboxes; and, combined use of several model-to-HDL translators hinders the estimation of the critical timing path.

6.3.2 A Synthesizable Quadrature Encoder Simulator

Among the position sensors, encoders and limit switches are commonly used in motion control applications. Quadrature incremental encoders are relatively simple devices that transmit the position variations of a moving part attached to it as modulated output signals pulse trains (see §2.4.2). The instantaneous frequency of the pulse trains is proportional to the speed, while the pulse count is proportional to the position. A possible solution to simulate a quadrature encoder is to modulate the pulse trains using an input speed value. A special burst circuit would modulate the encoder outputs at the required frequency between every two consecutive updates of the speed value. This technique supports the computation of the plant dynamics in a conventional processor, at a lower sampling rate, while the fast rate subsystem deals with the generation of the signal edges. However, this method has a major drawback to attain a high fidelity simulation: while the speed is computed with a dynamic model, the position has to be either estimated, or updated by counting pulses. Differences may arise, and when the **HiL** system has to simulate redundant position sensors, then synchronization issues may lead to inconsistent simulator behaviour.

An alternative approach is to model the quadrature encoder as a *scale-and-modulate* system, as shown in the Simulink model from Figure 6.2. This encoder model takes as input the position of the moving object (which is calculated in the plant model). The position

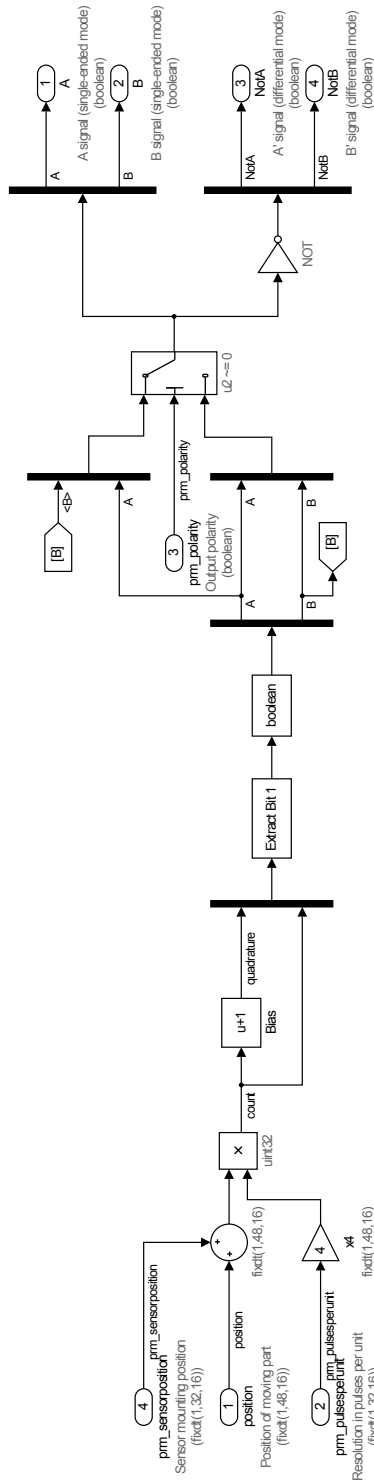


Figure 6.2: Simulink model of a synthesizable quadrature encoder

This figure depicts a Simulink model of a generic quadrature position encoder. The model input `position` is the position of the moving part, given in a `fixdt(Signed, WordLength, FractionLength)` fixed-point datatype. The encoder parameters are also modelled as inputs: `prm_position` defines the mounting reference position, `prm_pulsesperunit` defines the encoder resolution in pulses per unit of position, and `prm_polarity` selects direct or reversed phase of the A, B outputs, for values 0 or 1, respectively. There are three arithmetic operators whose internal datatypes shall be specified before converting to fixed-point: the *Adder*, *Multiplier* and *Gain* blocks. These should be re-scaled on changing the ranges for inputs or parameter values. Derivation of a floating-point variant is straightforward by replacing the fixed-point datatypes with `single` or `double` built-in Simulink types. This model is stateless and direct-feedthrough, i.e., it has no internal state memory, and the outputs at a given time instant are computed as a function of the inputs at the same instant.

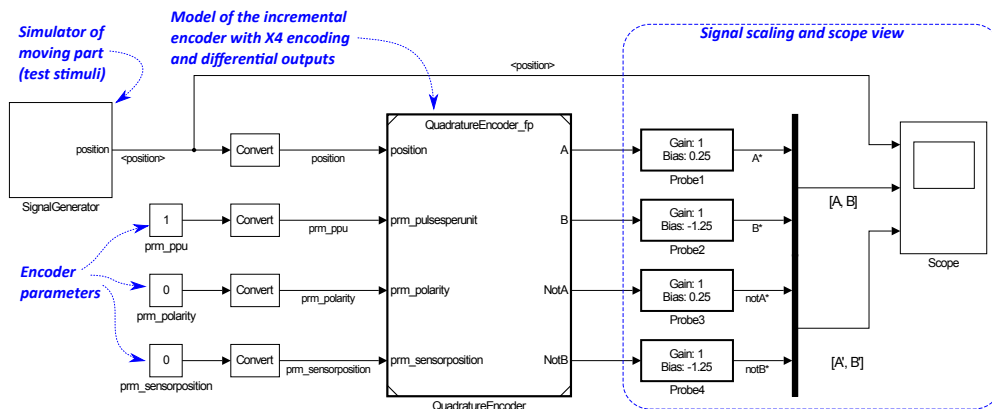


Figure 6.3: Simulink testharness for MiL validation of the quadrature encoder model

value is then scaled and biased according to the encoder resolution and a mounting reference position. The output signals A, B, A', B' are generated using an X4 encoding. In the output stage a polarity reversal can be applied to swap signals A, B and A', B' , which alters the relative phase of the outputs.

Model-in-the-Loop Validation

The quadrature encoder model can be first validated by means of instrumented Simulink models of varying complexity. Figure 6.3 shows a simple test-harness model, that was used to generate the example pulse trains shown in Figure 2.4. A number of different test-harnesses, including partial assemblies of models of plant subsystems were used for a comprehensive MiL validation of the sensor model before integration into the HiL real-time simulators.

Hardware-in-the-Loop Integration

We used the tool *HDL Coder* to translate this model to HDL. Listing 6.1 shows an excerpt from the generated HDL code, corresponding to the declaration of the quadrature encoder interface. The resulting code was imported to the LabVIEW/FPGA project for the HiLES simulator used in the case study presented in §8.2, on two different Xilinx FPGAs. Figure 6.4 shows the quadrature encoder IP, and the special-purpose chronometer instrument presented in section §6.3.4, integrated as LabVIEW Call Library Function Nodes (CLFNs) in the cRIO FPGA project. Both IPs are placed inside a timed LabVIEW loop that executes synchronously, at 2.5MHz in the cRIO-9082, or at 5 MHz in the cRIO-9039. The IPs read the same POSITION input value, which is computed at the plan model (not shown in figure).

The actual real-time behaviour of the simulated encoder was validated with laboratory instrumentation, checking the correctness of the pulse train frequencies with regard to the

Listing 6.1: HDL declaration of the generated IP for the quadrature encoder

```

1  LIBRARY IEEE;
2  USE IEEE.std_logic_1164.ALL;
3  USE IEEE.numeric_std.ALL;
4
5  ENTITY QuadratureEncoder_fp IS
6      PORT( clk
7            : IN      std_logic;
8            reset
9            : IN      std_logic;
10           clk_enable
11           : IN      std_logic;
12           -- Position of moving part
13           position
14           : IN      std_logic_vector(47 DOWNTO
15           0); -- sfix48_En16
16           -- Resolution in pulses per unit
17           prm_pulsesperunit
18           : IN      std_logic_vector(31 DOWNTO
19           0); -- sfix32_En16
20           -- Output polarity
21           prm_polarity
22           : IN      std_logic;
23           -- Sensor mounting position
24           prm_sensorposition
25           : IN      std_logic_vector(31 DOWNTO
26           0); -- sfix32_En16
27           ce_out
28           : OUT     std_logic;
29           -- A signal (single-ended mode)
30           A
31           : OUT     std_logic;
32           -- B signal (single-ended mode)
33           B
34           : OUT     std_logic;
35           -- A' signal (differential mode)
36           NotA
37           : OUT     std_logic;
38           -- B' signal (differential mode)
39           NotB
40           : OUT     std_logic;
41       );
42  END QuadratureEncoder_fp;

```

simulated speed of a moving elevator cabin.

6.3.3 Simulating Sensors with Bus Interfaces

Herein we present the Model-Based Development (MBD) of a real-time simulator of a CANopen absolute encoder for the functional verification of real-time motion controllers. We describe the step-by-step synthesis and deployment of the encoder simulator in the FPGA of a commercial-of-the-shelf NI cRIO real-time controller. We also discuss specific integration and portability issues that arose when migrating the simulator to a variant of the hardware platform.

CANopen Absolute Encoders

CAN networks suit the low-cost and performance requirements for implementing a variety of distributed control systems. Several encoder manufacturers offer devices with CAN

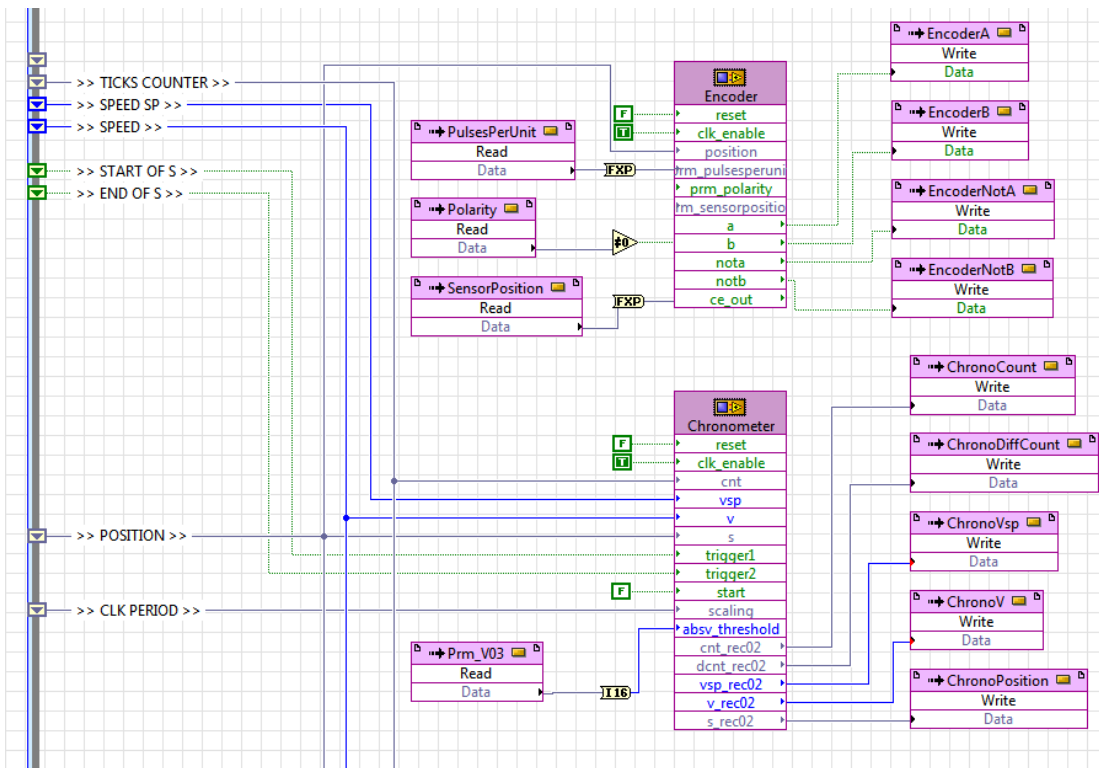


Figure 6.4: Quadrature encoder and chronometer IPs in LabVIEW FPGA project for cRIO

interfaces, offering alternative protocol implementations: custom, industry standard (e.g., CANopen CiA 406 [CiA406]), or domain-specific (e.g., CANopen CiA 417 [CiA417] for elevator applications).

We focus on building a functional model of an absolute encoder with a CAN interface and with the CANopen CiA 406 protocol, that could be used as a functional replacement for a real device, e.g., an ELGO LIMAX 02 encoder [Mat13, LIMAX]. CANopen encoders act as CANopen communication slaves and the controller devices act as CANopen masters. CANopen slaves react to master messages, except for cyclic messages that are transmitted upon the trigger of a programmable timer.

According to CANopen specifications, a CiA 406 encoder operating in normal mode publishes the position and speed data synchronously by sending a Process Data Object (PDO) (TPDO1). The basic node operation complies with the common CiA 301 [CiA301] services. The CANopen master can set the TPDO transmission cycle for sending a Service Data Object (SDO) message. The encoder also emits a Heartbeat (CANopen message service) (HB) periodically, triggered by a programmable timer. Other functions include

saving parameters, or resetting to factory configuration. The encoder implements Layer Setting Services (**LSS**) to setup configuration parameters like the node-ID and bit rate. The **LSS** configuration requires a point-to-point connection to the CANopen master and a reset of the sensor to become effective. The configuration is typically carried out at commissioning. For our purpose, we assumed that the sensor is pre-configured at start-up, and that the elevator controller will only overwrite the TPDO1 and **HB** transmission cycle times.

Modeling the CANopen absolute encoder in Simulink

The purpose of the encoder simulator is to verify the performance of the elevator-position controller. We first derived the requirements for the encoder model from the functional requirements for the position controller. As said in the preceding paragraph, the actual CANopen encoder implements **LSS** services intended to configure persistent parameters of the sensor at commissioning. We assumed that the sensor will be pre-configured prior to connecting it to the elevator controller. Therefore, we did not implement these setup functions. Instead we replace the non-volatile memory for communication parameters with a pre-set configuration file. These parameters will be fed into the model through additional input ports (e.g., the CANopen nodeid, see Table 6.2).

In normal operation the CANopen master, i.e., the position controller, would enable or disable the encoder messages using the **NMT** services, read the position and speed information contained in the TPDO1 datagrams, and re-configure the transmission cycles for heartbeat and TPDO1 messages. To keep the model simple, we implemented a subset of the CiA 406 object dictionary with only 3 read/write entries: the heartbeat cycle time (entry10700h), the TPDO1 transmission cycle time (entry18500h) and the Network Management (**NMT**) slave-mode state. The model provides inputs for an external tick counter and a scaling factor to control the timing for transmitting **HB** and TPDO1 messages. This external tick count is fed to the model *EVENTTIMER* shown in Figure 6.6, which implements a programmable event timer compliant with the standard CiA 301 [CiA301].

The interface to the elevator simulator consists of inputs for the elevator car position and the position reference of the encoder (expressed in mm, using a signed 48-bit fixed-point format) and the cabin speed (expressed in mm/s, in 16-bit signed integer format). We initially sought for a general model of an encoder compliant with standard CiA 406 [CiA406], so we designed the scaling function depicted in Figure 6.7. The figure shows the *CiADS406scaler* Simulink model that realizes a fixed-point arithmetic scaler for position and speed values, according to the CiA 406 CANopen standard for encoders. CiA 406 specifies that the measurement resolution could be dynamically re-defined by CANopen masters through the CANopen **SDO** service, and also defines the range of valid values for the resolution parameter. The resulting scaler model using fixed-point data-types requires non-standard fixed-point arithmetic with lengthy intermediate registers. Ultimately, the timing constraints for the **cRIO-9082** forced the removal of this function: once the encoder

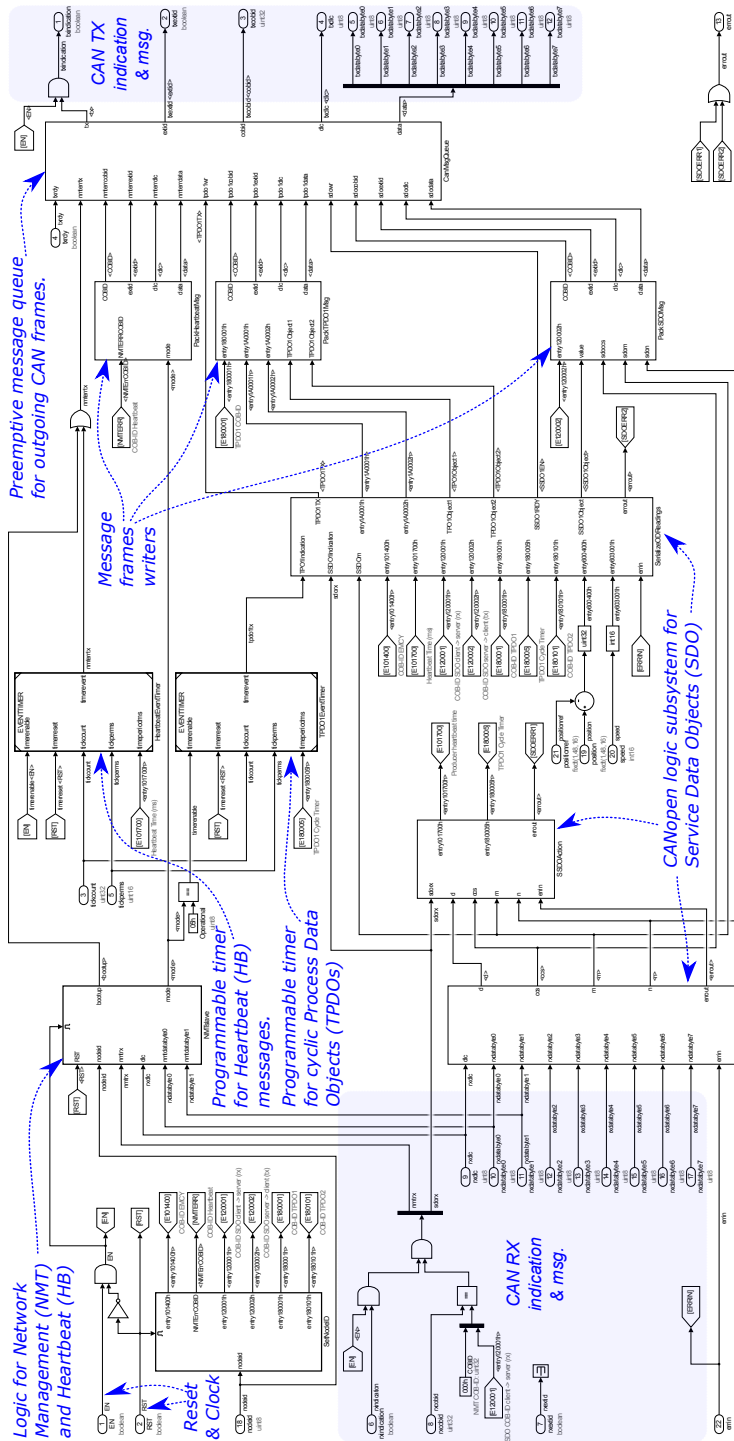


Figure 6.5: Synthesizable Simulink model of a CANopen absolute encoder simulator (top-system view)

This model only uses blocks from the standard Simulink block library, to minimize tool-dependencies. Every modeling element is translatable to VHDL by HDL Coder. The model is built on the assumption that the CAN interface will provide mailboxes or queues for storing the incoming messages. The model implements a pre-emptive queue for outgoing messages, to warrant the timely emission of frames with lower CAN IDs.

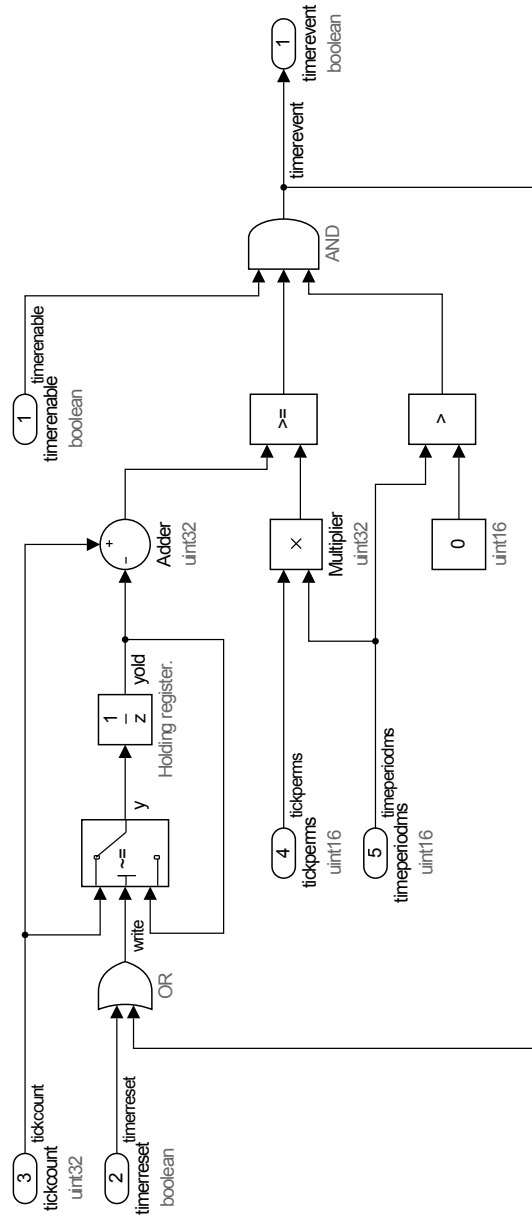


Figure 6.6: Simulink model of a CiA DS301 event timer

Figure 6.6 depicts the Simulink model EVENTTIMER, which realizes a CANopen CiA DS301 programmable event timer to trigger the transmission of heartbeat (HB) and cyclic TPDO CANopen messages. Input tickcount shall be connected to a timed counter which increments tickperms ticks per msec. Inputs timerenable and timerreset provide controls to hold the timer and clear the holding register (1/z). Input timeperiodms connects to the corresponding timer register in the CANopen Object Dictionary. When the timer is enabled, the boolean output timerevent holds ON for a cycle, clears the memory and then switches to OFF. This model only requires integer arithmetic operations, and is translatable to HDL for synthesis. Except for the duration of the output pulse, this model has no further dependencies with the sampling time, making it runnable in desktop simulations as well as integrable HiL platforms, even for different update rates.

Table 6.2: Input/output ports for the absolute encoder model

Port	I/O	Datatype	Description
EN	input(1)	boolean	Enable
RST	input(2)	boolean	Reset
tickcount	input(3)	boolean	External clock tick count
txrdy	input(4)	boolean	CAN port ready for TX
rxindication	input(6)	boolean	Signals a message RX
rxextid	input(7)	boolean	Indicates a 29-bit rxcobid
rxcobid	input(8)	uint32	CAN message ID (RX)
rxdlc	input(9)	uint8	RX msg. data byte length
rxdatabyten	input(10...17)	uint8	n -th RX databyte
nodeid	input(18)	uint8	CANopen encoder ID
position	input(19)	fixdt(1,48,16)	Cabin position in [mm]
speed	input(20)	fixdt(1,48,16)	Cabin speed in [mm/s]
positionref	input(21)	fixdt(1,48,16)	Position reference.
errin	input(22)	boolean	Signals a preceding error.
txindication	output(1)	boolean	Transmission trigger.
txextid	output(2)	boolean	Indicates a 29-bit txcobid.
txcobid	output(3)	uint32	CAN message ID (TX)
txdlc	output(4)	uint8	TX data byte length
txdatabyten	output(5...12)	uint8	n -th TX databyte
errout	output(13)	boolean	Signals an error.

model was coded in HDL, the bit-length of the holding registers for the arithmetic operations required cascading multiple DSP stages. As a consequence, the total delay resulting from the critical path was $245,00ns$ (Logic Delay: $91,82ns$ /Routing Delay: $153,14ns$) missing the timing constraint ($199,92ns@40MHz$) by $45,66ns$.

We assumed that the CAN port of the HiL simulator will provide queues for receiving and transmitting messages. We decided that the model will read a single incoming message at a time, and that only a single outgoing message will be written to the model outputs. In the worst case, in a single cycle the encoder could receive an SDO request while the event timers trigger the transmission of the HB and TPDO1 messages simultaneously. To solve this problem we modelled a triple-mailbox and a scheduler that outputs first the messages with higher priority, delaying the other until the CAN port is available. Besides that, as the different CANopen services run concurrently, we modelled a serializing/de-serializing access to the entries of the object dictionary to prevent read/write collisions.

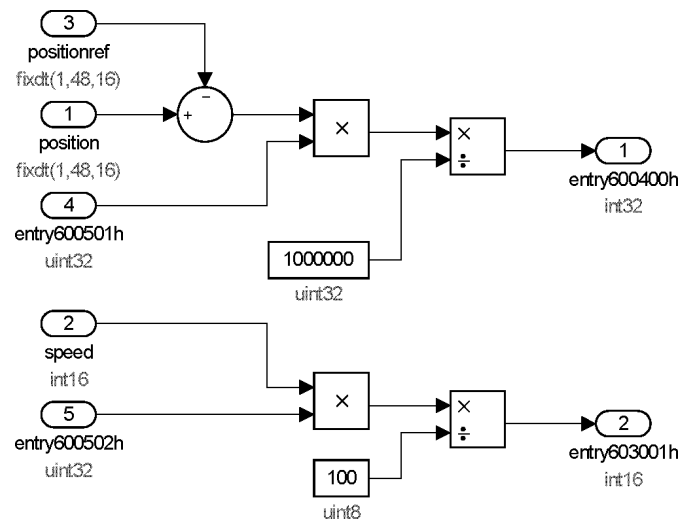


Figure 6.7: Simulink model of a fixed-point CiA DS406 encoder scaler

Validation of the Simulink model

To assess the functional equivalence of the Simulink encoder model with the manufacturer specifications we built a test-harness model that imports the encoder model as a Simulink' *Model Reference Block* (see Figure 6.8). For the CAN communication we developed a set of Simulink extensions (C-MEX S-functions) for controlling a CAN port from an IXXAT USB-to-CAN interface attached to the simulation host computer¹.

We also included a C-MEX S-function for simulation-to-host time synchronization that sleeps the computing thread to achieve soft-real-time behaviour².

This way the functional correctness of the sensor model can be validated using a standard computer, and a client application or device acting as a CANopen master. Although the timing of the functions is not deterministic (only a delay synchronization keeps the simulation time close to the elapsed host time), this desktop simulator reacts to the CANopen master as specified, with regard to the queuing of messages as well as the internal timing mechanisms to trigger the CAN transmissions work properly, even if the measured CAN message timestamps showed timing jitter.

¹Optionally a Simulink model can be connected to a CAN network using either the *Vehicle Network Toolbox* or *CANoe*.

²Instead of MathWorks Real-time Windows Target.

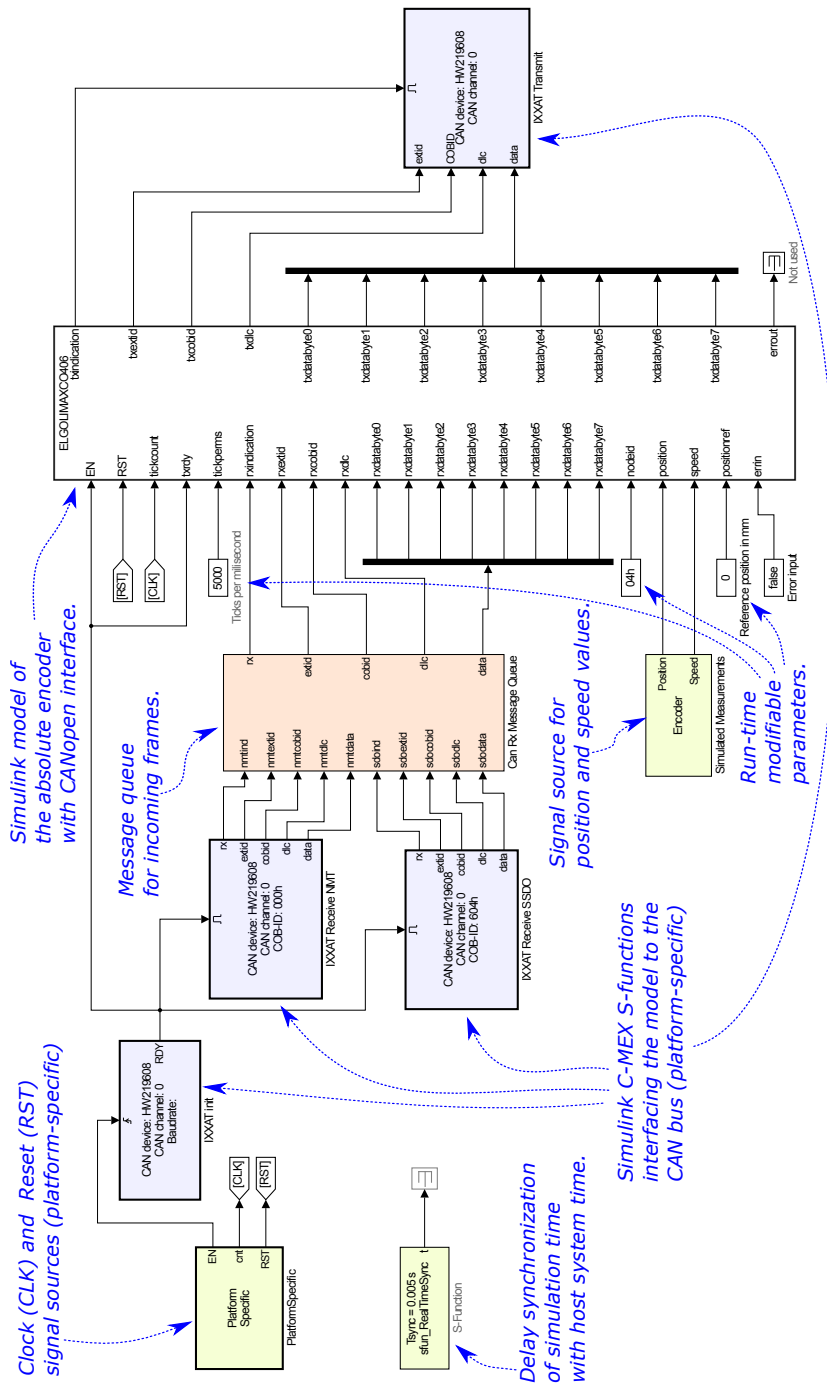


Figure 6.8: Simulink test-harness model for the validation of the CANopen absolute encoder simulator.

This test-harness enables the execution on the Simulink model of the encoder from the MATLAB/Simulink host. Custom C-MEX S-function blocks enable the use of an IXXAT USB-to-CAN interface to interface to a real CAN bus.

Transformation to HDL

We used HDL Coder [HDL] to generate a Very High-Speed Integrated Hardware Description Language (VHDL) description of the Simulink model, adding to the model VHDL profiling information. At first we used a generic FPGA coding profile, seeking for better portability across the cRIO variants. After some trials in LabVIEW/FPGA this approach turned up to be unfeasible, due to timing issues. Finally we had to set a different profile to generate HDL specific to each FPGA target. This way we exploited the features of the Xilinx Kintex-7 DSPs in the cRIO-9039 controller to actually shorten the critical execution path and match the timing constraints.

6.3.4 Re-usable Instrumentation Models

Current HiL computing platforms show a similar architecture, including: (a) a multi-core processor with multiple communication interfaces and (b) a number of internal buses (e.g., PCI Express (PCIe)) connecting the processor to reconfigurable computing resources (e.g., FPGAs). For complex plant models the HiL could be scaled-up to a distributed architecture, partitioning the models and deploying them on multiple interconnected processing elements. Test engineers must balance the computing overhead in the HiL platform, partitioning the model between the conventional processors and the Programmable Logic (PL) resources, while considering at the same time the potential performance bottlenecks at the internal communication buses.

The main processor in a COTS HiL platform usually deploys an RTOS to handle the communication buses and provide services to integrate the test within a quality management system. Due to this, the minimum cycle time achievable for simulations running in the main processor is constrained by the system RTOS/firmware. When the dynamic behaviour of the System Under Test (SUT) is faster than that minimum cycle time then the test engineer has to move the model to the PL subsystem. For a Model-Based Testing (MBT) approach this could bring the test engineer to develop an alternative set of models that would be coded to HDL.

Sometimes the HiL system is intended for the verification process, which requires a precise time instrumentation of the internal plant variables to evaluate the SUT, e.g., for benchmarking different variants or candidate designs of the SUT). On some occasions, these variables cannot be output to physical interfaces, thus preventing the use of instruments external to the HiL platform. Besides the increased cost of the required HW, there is the problem of potential data throughput bottlenecks (e.g., should the HiL transfer a big amount of data to the instrument/analyser in real-time, its functioning could be compromised by a low availability of bandwidth in a PCIe bus or a networking interface).

When the output from the instrument/analyser updates at a slower rate than the inputs required to evaluate the technical performance, then a feasible alternative would be to integrate the instrumentation components in the HiL platform. This also opens the door to

the development of custom instruments that benefit from a common timing source, which enhances the correlation of the measurements with the actual time sample taken when the interesting dataset was created.

This subsection exemplifies the application of the proposed **MBD/MBT** framework and workflow to develop such specialized components, by application to a synthesizable *speed-triggered chronometer*.

A Synthesizable Speed-triggered Chronometer Model

The verification of motion-control systems usually requires measurements of the time elapsed between motion-related events. For instance, in a **HiL** system like the **HiLES** elevator simulator we had to implement a speed-triggered chronometer to compare the relative performance of alternative motion control strategies.

To this aim we applied our **MBD** approach to design the special instrument shown in Figure 6.4. This Simulink model implements a speed-triggered chronometer to record the time elapsed between external activation events (triggered with the control inputs **TRIGGER1**, **TRIGGER2** or **START**) or when the speed-value reaches a pre-set threshold. We integrated this chronometer in the **HiLES** elevator simulator, such that the **START** event could be triggered by the user of the test system, while the **TRIGGER1** and **TRIGGER2** events are fired by the plant model upon changes in the speed set-point, or when the moving part starts/stops its motion. The time is measured in tick count increments, where the tick value depends on the sampling rate at which the model is executed. The model implements internal shifting registers to store the values of the speed, speed set-point and position inputs when a triggering event activates the `ChronometerMemory2` subsystem.

One can observe that the model from Figure 6.9 has scalar-type outputs instead of vectors or arrays. Although HDL Coder can translate Simulink models with vector outputs to HDL code, here we faced the limitation of the LabVIEW **CLFN** import capabilities, that only supported HDL modules with scalar I/Os. As a consequence, this cluttered the Simulink model.

The Simulink model of the instrument was coded to HDL and imported to the cRIO LabVIEW/FPGA project, as shown previously in Figure 6.4. The time measurements have a resolution related to the sampling rate in the **FPGA**, that is, $400ns$ for cRIO-9082, and $200ns$ for cRIO-9039. Therefore, the **FPGA** implementation of the synthetic instruments provides sufficiently precise measurements of time intervals. Finally, the recorded measurements are sent to the test control application. Figure 6.10 is a snapshot of the **HiLES** remote test interface showing the motion and time values recorded by the **FPGA** chronometer IP.

6.4 Re-using Code Artefacts from SiL to HiL

This section presents a para-virtualization approach for concurrently validating product lines of distributed dependable systems, even while the embedded software applications

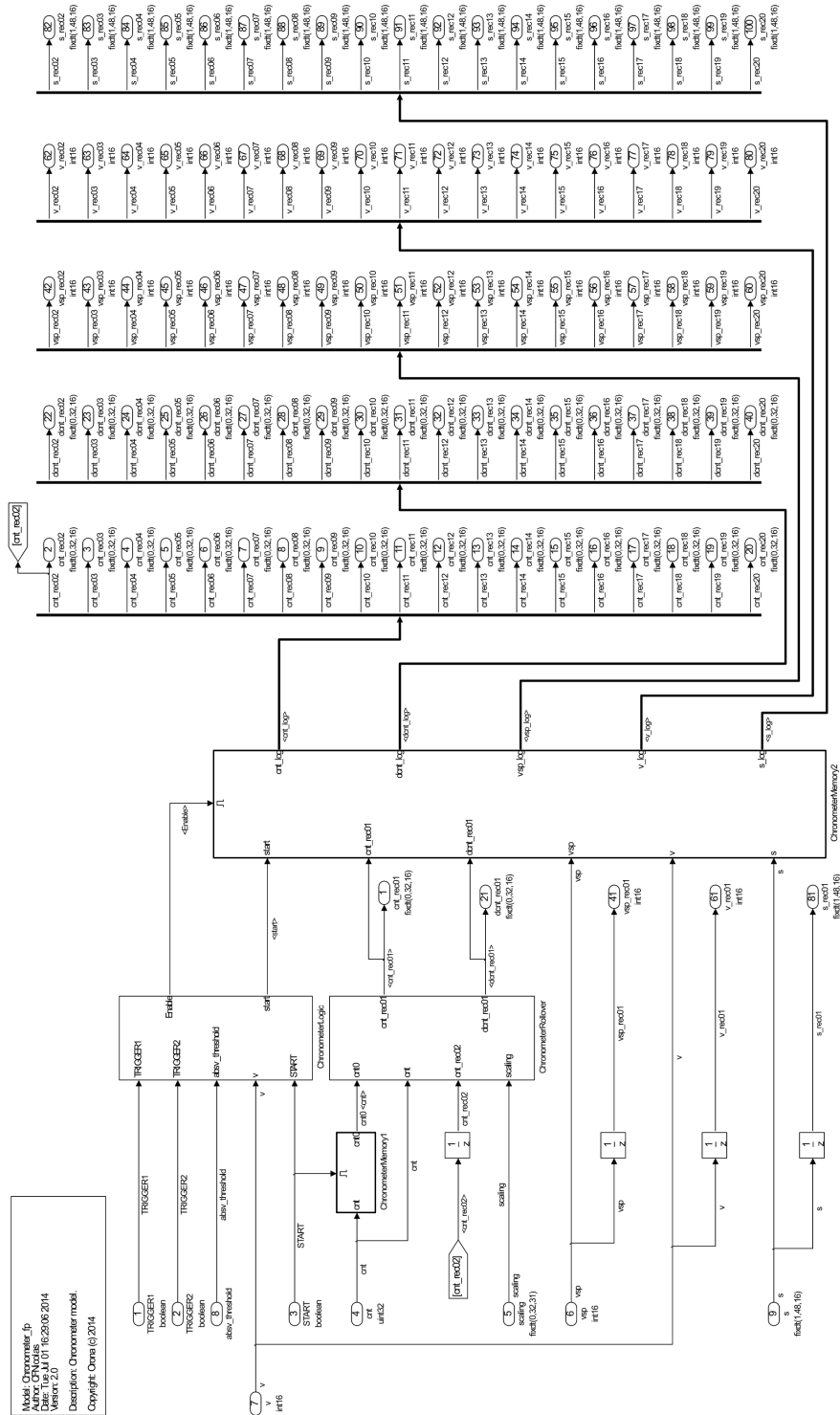


Figure 6.9: Simulink model for a synthesizable speed-triggered chronometer

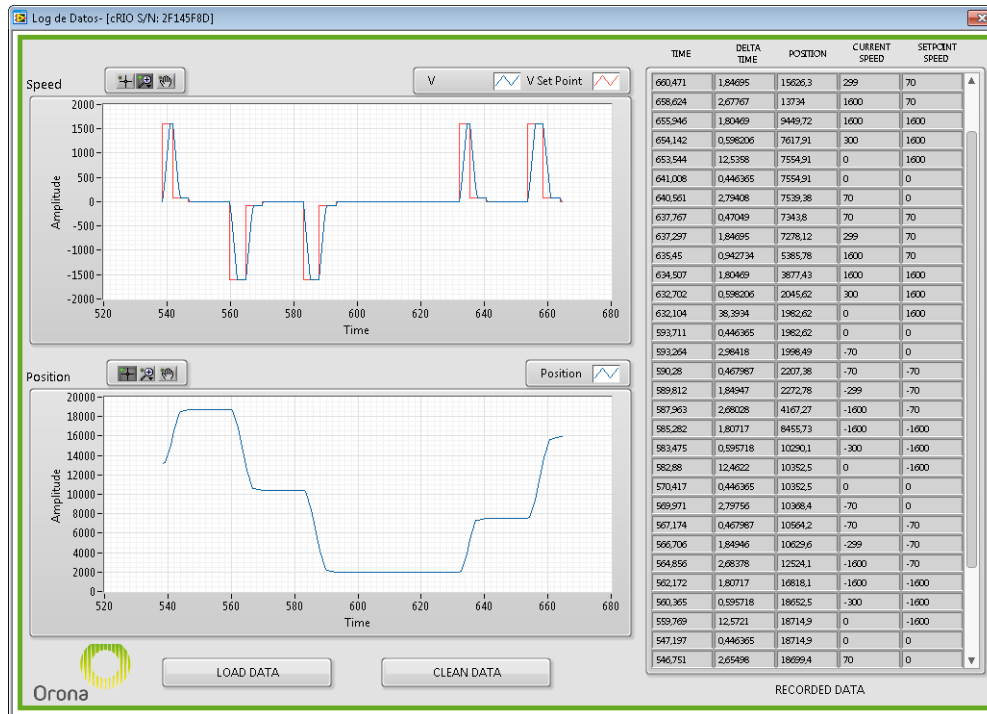


Figure 6.10: Remote LabVIEW interface for the speed-triggered chronometer

evolve incrementally. The para-virtualization relies on code re-use from the replaced products, which eases the upgrade of the test systems. The code re-use approach aims at: (i) virtualizing a number of networked embedded Input/Output (I/O) devices and (ii) integrating those virtual devices in a COTS HiL simulator.

6.4.1 CAN Distributed Control systems

Distributed embedded control systems may integrate diverse industrial networking technologies or buses, e.g., Ethernet, FlexRay, LIN, ARINC 429, Profibus, etc. Our work focuses on distributed systems with embedded I/O devices connected to a Controller Area Network (CAN) compliant with the CAN2.0A/B specifications.

The CAN bus was originally designed for automotive applications, but has also been adopted for industrial and building automation. Depending on the communication protocol, a single CAN node may implement diverse communication models, e.g., master/slave, producer/consumer, client/server, etc. For instance, in the CANopen protocol, a remote I/O node may act as a server (e.g., for accepting Service Data Object (SDO) request from a client device), as a slave (e.g., for configuring as the NMT request from the master), and

as producer (e.g., for transmitting cyclic PDOs messages).

In automation applications, CAN is used to interconnect the devices composing a distributed control system. Devices that in some configurations may amount to more than a hundred I/O nodes networked on a single CAN bus. As each real I/O device has at least one CAN port, and considering the constrained scalability of the HiL targets, one needs a virtualization approach that minimizes the number of CAN interfaces allocated to the virtual devices in the HiL system.

To that end, we first analyse the basic functionality of low-end I/O devices, as well as the CAN bus operation.

Assumptions on the I/O node embedded application

The para-virtualization approach is based on the following assumptions about the I/O application:

Assumption 1 *C/C++ Programming Language*: The embedded I/O node application shall be programmed in the C or C++ programming languages, so that the source code can be cross-compiled for the HiL processor targets. The platform-dependent functions are separated from the platform-independent ones.

Assumption 2 *Memory initialization*: The initial values of volatile and non-volatile memory are pre-determined by executing a memory setup routine (e.g., a C start-up procedure to initialize global or static C variables), and the source code for the initialization is available.

Assumption 3 *Peripheral initialization*: The peripherals used by the application are initialized at start-up, e.g., after powering-on or resetting the I/O device, and the source code for the initialization is available.

Assumption 4 *Interrupt service routines*: The source code of the interrupt service routines is available.

Assumption 5 *Non-blocking control flow*: The main execution thread in the application is based on an indefinite-loop that calls a non-blocking sub-routine.

6.4.2 Para-virtualization of I/O CAN Devices

As CAN is a half-duplex bus, a simple solution to simulate a group of networked nodes is to share a single CAN port between several virtual devices. A *CAN Restbus simulator* denotes a group of virtualized devices that share a single CAN port to connect to a CAN bus. The purpose of CAN Restbus simulators is to replace a multiplicity of embedded I/O nodes, using fewer HW resources to replicate a distributed control system.

As the simulated **CAN** interface of each virtual device increases the computing overhead, the challenge is to timely transmit the outgoing messages in a sequence indistinguishable from that generated by the real devices. **CAN** frames contain an 11- or 29-bit message identifier (CAN-ID) and a data field. **CAN** interfaces arbitrate the transmission of frames based on the CAN-ID: the lower the value of the identifier, the higher the priority to transmit the frame. Thus the range of CAN-IDs allocated to a message class in a **CAN** communication protocol, determines its precedence.

Virtualization of CAN Devices

In our case study we focused on the CAN I/O devices that share a common C source code baseline, used in two alternative deployments:

- *In standalone devices:* These are embedded networked devices with specialized interfaces. All the Printed Circuit Board (**PCB**) designs integrate the same Microcontroller Unit (**MCU**) architecture and share the application code. The behaviour is determined at run-time by means of a hardware identifier specific to each **PCB** variant.
- *As interfacing component:* In this configuration the remote I/O functionality combines with additional control functions on an **MCU** or **DSP** target. Parts of the source code are shared with the standalone embedded applications. This software is inherently portable across hardware platforms, easing the integration with the simulator.

Re-using the same source files of the actual nodes has a number of potential benefits:

- *Comprehensibility of behaviour:* The remote devices are reactive, i.e., their behaviour depends on the current inputs and messages and past action history. With a proper implementation the virtual device exhibits nearly the same behaviour as the actual component, getting similar I/O and messaging time series for an indefinite time period. This is advantageous to replay approaches, where the outputs (e.g., message sequences) of the simulator derive from a pre-defined sequence database.
- *Extensibility:* The number of simulated device instances to simulate can be modified at run-time. Thus we can reproduce scenarios were some nodes switch on or off dynamically without modifications in the physical layout.

The technique chosen to import the virtual devices in the real-time HiL platform is influenced by both the programming style of the legacy code and the models of computation available in the HiL controller. The main issues found were:

Issues related to the programming paradigm: For low-resource embedded devices it is common to deploy the application into a bare system, i.e., without an operating system. The source code of low-end embedded applications usually has many parts of

the program written in C. When developing an embedded application, programmers may use persistent global variables, allocated to fixed memory regions to access shared resources from different execution paths.

Listing 6.2 shows a simplified main C module for an embedded CAN device, where global variable `bState` and static variable `bDevId` would be allocated to fixed memory positions. The program control flow starts with a platform-specific initialization (e.g., a `__cstartup()` entry point) and then calls the `main()` function. The latter calls function `init()` that enables the CAN service routines as well as does other program initialization, like retrieving non-volatile device parameters. Variable `bDevId` is a *device identifier*, used to distinguish the multiple devices of the same kind that could communicate on the CAN bus. After completing the initialization, the `main()` function executes an indefinite loop that calls function `step()`. Function `step()` can use the system timing services to (almost) synchronously execute the device functions, e.g., acquire inputs, read incoming messages, update the internal state (e.g., by writing variable `bState`), modify the device outputs and send the outgoing messages.

The para-virtualization to simulate multiple instances of a program device like Listing 6.2 requires a proper context-switching to (i) retrieve the values for variables `bState` and `bDevId`, (ii) update the virtual time in the device, (iii) emulate the behaviour of the `main()` function, and (iv) store the updated state variables.

Listing 6.2: Example of embedded application main C module

```

1  char bState = 0;           /* global scope */
2  static char bDevId;       /* module scope */
3  /* program initialization */
4  static void init(void) {   /* module scope */
5      canInit();
6      ...                   /* setup */
7  }
8  /* synchronous task */
9  static void step(void) {   /* module scope */
10     if(elapsedTsamp()) {   /* check timer */
11         getInputs();       /* read inputs */
12         dequeueRxMsgs();   /* handles msgs */
13         ...                /* other processing */
14         setOutputs();      /* write outputs */
15         enqueueTxMsgs();   /* send msgs */
16     }
17 }
18 /* program entry point at startup */
19 void main(void) {          /* global scope */
20     init();                /* initialize */
21     for(;;)                /* endless loop */
22         step();            /* task */
23 }

```


Instead of calling the `main()` function, a multiple-instance virtual device simulator can conveniently call `init()` to simulate the power-on of a device instance and then recursively call the `step()` function for each active virtual instance. In case the platform-specific initialization routine also sets the initial values for global variables, then additional initialization instructions shall be added to the virtual device initialization.

The Restbus device simulator must handle these global objects, by switching to the memory values corresponding to each simulated device at the simulation time.

In order to handle multiple instances of the embedded application, we developed a

Listing 6.3: C++ CModule interface

```

1  class CModule {
2  public:
3      virtual void _get(void) = 0;
4      virtual void _init(void) = 0;
5      virtual void _step(void) = 0;
6      virtual void _set(void) = 0;
7  };

```

Listing 6.4: C++ CMain wrapper with context switching

```

1  #include "cmodule.hpp"
2  extern "C" {
3  #include "main.c" //< Wrapped C module
4  }
5  class CMain : public CModule {
6  private:
7      char          _bState; /* global scope */
8      static char  _bDevid; /* module scope */
9  public:
10     bool _dequeueTxMsg(canmsg_t *msg) { ... }
11     bool _enqueueRxMsg(canmsg_t *msg) { ... }
12     void _get(void) {
13         bState = _bState;
14         bDevid = _bDevid;
15     }
16     void _getOutputs(outputs_t *out) { ... }
17     void _init(void) { init(); }
18     void _set(void) {
19         _bState = bState;
20         _bDevid = bDevid;
21     }
22     void _setInputs(inputs_t *inp) { ... }
23     void _updateTimer(time_t elapsed) { ... }
24     void _step(void) { step(); }
25 };

```

set of C++ wrappers to implement the context switching for each C-file (to save and restore the internal state variables). Listing 6.3 shows the C++ interface `CModule`, realized by the C++ wrapper for module `main.c`, `CMain` (see Listing 6.4). Class `CMain` exports public methods to simulate basic embedded HW operations like filling or emptying CAN mailboxes, setting the inputs, reading the output values commanded by the application or updating the internal timers. These mirror the Application Programming Interface (API) as seen from the embedded application, that reads inputs and writes the outputs, while the para-virtualization wrappers do the opposite. Each C file containing global or static variables shall be wrapped in the same way. Note that this approach has a limited applicability, as static variables within function scope are not readable from the C++ wrapper methods.

Performance issues due to mixing languages: The motivation to use multiple languages to program the HiL simulator was to take advantage of the ease of integration provided with LabVIEW (see Fig. 6.1), while re-using as much code as possible to preserve the major part of the functionality from the simulated nodes. But the data-flow Model of Computation (MoC) of LabVIEW imposes a synchronized update of the inputs and outputs of a block. While beneficial for programming, this approach has the drawback of adding a computational burden due to memory transfers and data-type conversions that are critical for a multi-node simulator iterating over each virtual node instance.

Real-time synchronization issues: A heterogeneous platform like a cRIO-9039 suits the timing requirements for the execution of the models and the Restbus simulator. However, when the plant model involves redundant data-paths to the real-world devices it is hard to synchronize a signal going to an external actuator (e.g., a relay) with a correlated signal propagating through the elevator simulator to the Restbus simulator (e.g., the signal published as a CAN frame).

Other considerations for a CAN Restbus simulator are:

- In a CAN bus only one node may transmit a frame at a time. Thus it is feasible to share a single CAN port between multiple virtual devices, if proper queuing handles the in/outgoing messages for each instance.
- Not only the CAN bus-load varies with the number of virtual device instances, but also the frame delays, as the number of priority frames sent by the controller under test may change with the number of connected devices.
- When virtualizing retro-compatible devices, it is also possible to have allocations of CAN-IDs to protocols that are specific to a product line. In this case, then the worst-case transmission delays for a protocol change with the controller variant.
- There could be signals with redundant paths to the controller. Race conditions may arise between input changes notified by a message from a virtual device and the

alternative paths, e.g., the switching of a relay in the safety circuit. A wrong time correlation between them may trigger fault reactions from the System Under Test.

LabVIEW/RT API for virtual Device Simulators

Both run-times of LabVIEW and LabVIEW/Real-time support calling user-provided code components by means of a Call Library Function Node (**CLFN**). A **CLFN** in LabVIEW inserts a function call to a method provided by the shared library. **CLFN** is a LabVIEW façade that declares the function signature, including parameters and return type. The calling convention for the invoked function shall be declared as `stdcall` (only for **WINAPI**) or C. The library path can be specified as an input parameter to the **CLFN** in the caller **VI** (Virtual Instrument, LabVIEW program subroutine). When interfacing a **CLFN**, derived data types supported in LabVIEW (e.g., arrays and clusters) transform to C arrays and structures. These datatype conversions increase the execution overhead.

CAN implementation in **cRIO-903x**

In order to allocate the **CAN** Restbus simulator to the **cRIO** processor we require a **CAN** interface accessible from the LabVIEW RT execution context, for which we considered two alternatives: (a) a 2-port NI 9853 C-series **CAN** module as used in **HiLES** to simulate a **CANopen** absolute position encoder [NAM+16], or (b) a Linux-supported **CAN** interface. The C modules in a **cRIO** can be controlled from the **FPGA**, the processor or operate in hybrid mode, the latter implementing the shared access in the **FPGA**. If the virtual device simulator is intended to be deployed at the same target as the absolute encoder simulator described in §6.3.3, then the control of C-modules must be set to either **FPGA** or hybrid modes. After several tests the hybrid mode showed to be troublesome, and we were unable to get it going. Hence we finally preferred to integrate an external Peak PCAN interface connected to the Universal Serial Bus (**USB**) port of the **cRIO** controller. The NI Real-time Linux kernel (version 3.14.40) from the **cRIO-9039** provides support for **CAN** communications, yet only brings a limited set of pre-installed device drivers. The user can provide compatible device drivers for alternative **CAN** devices, either for real-time or non real-time operation. The latter also supports **CAN** devices attached to the **cRIO USB** ports. The real-time mode requires a time-deterministic handling of the communication ports, e.g., through PCI/PCIe or SPI buses, eventually limiting the RT-CAN devices to the NI modules.

Linux support for Peak **CAN** devices consists of source code for the device driver (for both, real-time and non-real-time operation) and a shared library. As we selected a PCAN device with **USB** interface, this is supported only for non real-time operation. Although this does not hinder most of our test applications, non-determinism makes the resulting HiL simulator unsuitable for the thorough verification of safety-critical functions involving **CAN** messages. We had to cross-compile the PEAK device-driver sources for the NI Real-Time Linux (NIRTL) kernel version, then download and configure the device driver. The API of

the Linux shared library differs from the Windows one. Therefore, we had to modify the LabVIEW VIs containing the CLFNs to the Windows Peak library. Alternatively we could have re-implemented an equivalent Windows-like API in the Linux variant, to facilitate the cross-platform porting of VIs between LabVIEW (Windows target) and LabVIEW-RT (NIRTL cRIO target) projects. This would eventually reduce the validation and future maintenance effort by avoiding variants.

6.5 Discussion

In this chapter we presented two complementary approaches aiming at reducing the cost for building time-deterministic and scalable HiL simulators for verifying real-time dependable systems implementing safety functions related to motion and having a distributed system of remote I/O nodes, connected through a CAN bus.

Our contribution is a workflow that aims at supporting cross-domain collaboration, so that experts in different fields could compose realistic models of the operating environment of the system to test. To some extent those models help at preserving the functional definition of test components at an abstract level, disregarding a particular HiL computing platform. An automatic model-to-code translator can be profiled and used to get the required programming artefacts for the specific HiL subsystem. In heterogeneous COTS HiL systems like the NI cRIO, the model subsystem requiring a faster sampling can be transformed to VHDL code and implemented in Programmable Logic (PL). The advantage of our proposed approach is that sophisticated models of devices peripheral to the SUT can be designed without needing a deep knowledge about FPGA programming. However, the suggested mixed-language programming showed some shortcomings, partly due to the novelty of the extensions for FPGAs from some MBD environments.

On the other hand, the para-virtualization approach pretends to enhance the scalability of the HiL system, by providing virtual replacement for low-complexity remote I/O networked to a CAN bus. This enables the simulation of many different configurations for a distributed control system. Although the functionality of each replaced device may be simple, the feasibility of preserving most of its internal behaviour supports the simulation of fairly complex overall emergent behaviours. For instance, multi-protocol virtual devices can be intentionally activated in a wrong configuration (i.e., acting as a *saboteur*) to examine the reactions of the SUT. This is of particular interest for verifying dependable systems relying on networked I/O devices. Even if the virtualization has been realized for CAN devices, the same approach could be extended to other networking technologies, with the exception of real-time protocols in which communication messages are marked with hardware-generated timestamps.

The model/code re-use approaches have been exemplified by their application to a HiL simulator, used for regression testing of elevator control systems (see §8.2). On this we evaluated the feasibility, benefits and limits of the re-use approaches proposed herein.

7

Framework of Re-usable Safety Arguments for Mixed-Criticality Product Lines

Preceding chapters 5 and 6 contributed techniques to improve the re-usability of artefacts used for the Model-Based Testing (MBT) of Dependable Embedded Systems (DESSs) on executable models, with the aim of recovering the validation effort spent in modeling or programming components that would interact with the DES at later system/validation testing phases. A consequence of adopting an MBT development process with executable specifications is an increasing amount of generated analysis results considering simulation amongst the analysis techniques recommended by safety standards. Due to possible model inaccuracies, several tests carried out on the models should be exercised again on the physical DES to ascertain the correctness of the implementation. These test runs try to trace the results forecast by simulations and may re-use test specifications, components or analysis tools formerly used in the simulation phase, which eases the comparison of the actual and the predicted system behaviour. A mandatory requirement for the safety certification is that all the test campaigns fit to an overall verification, validation and testing (VVT) plan, that should provide a logical chain of conclusive evidences gathered from different VVT tasks of the compliance of the DES to the safety requirements.

This chapter contributes a complementary model-based framework to iteratively assemble a logical argument for the safety certification of Mixed-Criticality Product Lines (MCPLs), developed with a platform-based design (PBD) approach based on the DREAMS platform and tools. The argumentation framework enables the re-use of pre-built arguments, the Modular Safety Cases (MSCs), to be later completed with the staged incorporation of VVT evidences into the overall logical chain to support the safety claims: the Safety Cases.

7.1 What is DREAMS?

Safety-critical applications could benefit from the standardization, cost reduction and cross-domain suitability of current Heterogeneous Computing Platforms (HCPs). These are of particular interest for Mixed-Criticality Product Lines (MCPLs), as both safety- and non-safety functions can be deployed on a single embedded device, provided that suitable isolation artefacts and development processes are used. The development of MCPLs can be facilitated by providing a reference architecture, model-based design and analysis tools and Modular Safety Cases (MSCs) to support the safety claims.

Modern HCPs enable architectural simplifications and standardization across multiple application fields to implement embedded systems with homogeneous hardware (HW) and software (SW). The research on bringing determinism and fault isolation to HCP platforms enable safety-critical applications for heterogeneous processors, while also deploying non-safety related applications. At the same time, the cost reduction in multi-purpose HW components fosters a common-platform development for multiple domains. In the scope of the Distributed Real-time Architecture for Mixed-criticality Systems (DREAMS) project [DREAMS], the safety-certification of MCPL according to the IEC 61508 standard is one of the objectives.

The target of DREAMS are families of embedded systems that embody varying sets of features, amongst which some relate to functional safety. Following the PBD paradigm, we developed a common HW/SW baseline to deploy the product functionalities of DREAMS. This platform, also referred to as the DREAMS harmonized platform (DHP), incorporates a multi-core heterogeneous System-on-Chip (SoC), where a reliable Network-on-Chip (NoC) is synthesized and deployed. Furthermore, the XtratuM hypervisor [XTR] is implemented in the DHP to prevent interference between SW components of different criticality. For each of these components, DREAMS developed separate MSCs¹ according to the IEC 61508 standard. These MSCs consider scenarios with different integrity requirements [LPA+15, LPO15, LPN+16].

Complementary to these MSCs, DREAMS provides a model-based design framework that is based on the open-source model-based development tool AutoFOCUS 3 (AF3) [AF3]. Under the DREAMS project this tool is enhanced with additional plug-ins to capture the MCPL requirements, define the variability model for MCPLs, sample the MCPL and instantiate the feasible product variant models, analyse and assess the safety properties on the variant models and build and refine the variant safety argumentation model in Goal Structuring Notation (GSN). From the latter we derive a variant-specific preliminary safety case. MCPL is an abstraction to represent all the possible product variants of a product family. As in the sense of IEC 61508 safety is an emergent property, the MCPL abstraction shall be resolved and each variant shall be instantiated before assessing its safety capability.

¹DREAMS HW/SW components are for experimental and demonstrative purposes only. The certification of HW, SW and tools is out of the scope of this thesis and the DREAMS project.

7.1.1 DREAMS Platform-based Design for MCPLs

The platform concept is an abstraction that covers several possible lower-level refinements [SM01]. PBD supports the meet-in-the-middle process [FKM11], where successive refinements of specifications meet with abstractions of potential implementations and the identification of precisely defined layers, i.e., the *platforms* [SCB+04]. The *meet-in-the-middle* approach aims at preventing the convergence to non-feasible solutions sometimes found with a *top-down* approach, as well as preserving abstractions to tackle the complexity arising in a *bottom-up* approach. A platform consists of a set of elements together within their constraints and rules. It can be thought of as a library of elements which can interconnect through communication components. Each element is characterised by its functionality and expected behaviour. Figure 7.1 shows the meet-in-the-middle methodology where it applies a top-down design (application design) for a high level of abstraction and implements a bottom-up design for a low level of abstraction (platform design). Both designs converge where the platform is ready to host an application which is ready to be hosted on a platform. From a HW perspective, the bottom-up approach is supported (low to high abstraction level). This enables the adaptation of the HW both at the design time and the run-time using dynamic and partial reconfiguration.

In the context of IEC 61508-2 the following requirements may be implemented to show the absence of systematic faults. The first requirement implies to *meet the requirements of Route 2_S or Route 3_S*. *Route 2_S "Proven-in-use approach"* establishes the compliance with the requirements of proven-in-use components. A component shall only be regarded as proven-in-use when it has a clearly restricted and specified functionality and when the absence of systematic faults is demonstrated (see Subsection 7.4.10 of IEC 61508-2). *Route 3_S "Pre-existing SW"* establishes the compliance with the requirements of IEC 61508-3, including the requirements for pre-existing and re-used SW. On the other hand, the

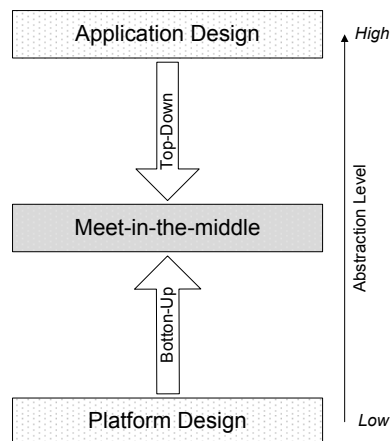


Figure 7.1: Meet-in-the-middle methodology

second requirement implies to *provide a safety manual* that includes a precise and complete description of the pre-existent components, enabling the assessment of the integrity of a specific safety function that depends wholly or partly on the pre-existing SW components (see Annex D of IEC 61508-2 and IEC 61508-3).

DREAMS aims at developing a cross-domain real-time (RT) architecture and design tools for complex networked systems where application subsystems of different criticality executing on networked multi-core chips are supported. This research project delivers virtualization technologies, model-driven development methods, software tools, adaptation strategies and validation, verification and assessment methods for the seamless integration of mixed-criticality systems to establish security, safety and real-time performance as well as data, energy and system integrity. The execution platform design is based on a cross-domain system architecture of a hierarchical distributed platform for mixed-criticality applications combining the logical and physical views (see Figure 7.2).

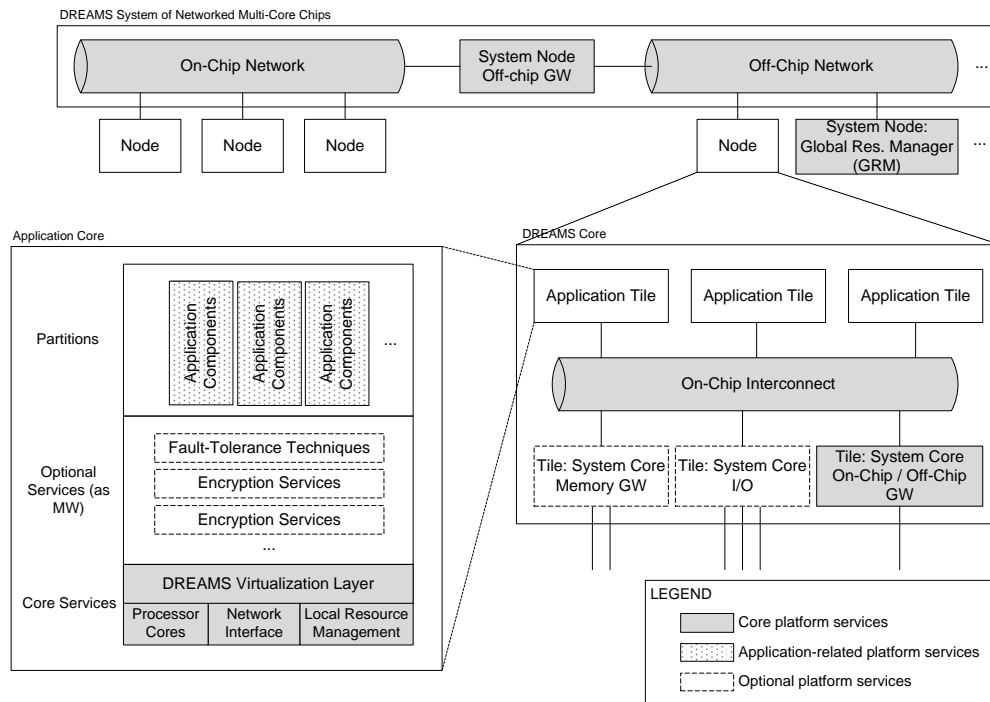


Figure 7.2: Overview of the DREAMS platform architecture

This architecture enables the implementation of heterogeneous application subsystems with different criticality levels (e.g., SIL1 to 4 according to IEC 61508), timing (e.g., firm, soft, hard, non-RT) and computation models such as Time-Triggered (TT) messages, data-flow and shared memory. The variability in the application subsystems can impose diverging

requirements, so that designers have to find different trade-offs between predictability, assessment and performance in processor cores (e.g., Zynq-7000 processor), hypervisors (e.g., XtratuM hypervisor), operating systems (e.g., Windows CE) and networks (e.g., on-chip and off-chip networks). This platform architecture provides global and local resource management units for executing multiple application subsystems, i.e., for managing their component and different execution environments and resources such as memories and I/Os.

7.1.2 Certification

This research focuses on the cost-efficient development and certification of Mixed-Criticality Systems (MCSs). MCSs have particular properties that pose special certification challenges [VAR48], requiring specific certification guidelines to overcome these.

What is certification?

Certification is a third party attestation confirming that defined requirements have been met. Certification may include products (components, devices, or technical systems e.g., tools), processes, organizational systems (like management systems) or persons. Certification and approvals refer to standards that define what requirements are to be met. Central to most certification of functional safety is the IEC 61508 [IEC61508] functional safety standard.

Certification is a process based on objective evidence for compliance with requirements. This evidence results from documentation review, audit or testing. Generally, certificates have an expiration date of validity and have to be renewed after a certain time. In the meantime, periodic verification may be necessary to ensure continuous validity. In case a certified product is either modified or used for a different application, a reassessment is necessary. The extent of such reassessment depends on various conditions, such as the certification scheme, applicable standards, intended application, or component certification, to name a few.

Safety certification usually requires two types of assessments:

1. *Process Assessments*: To show that the system development complies with the techniques and measures recommended by a safety standard. Sticking to a pre-defined development process would provide error-freeness, acting as a filter against systematic failures.
2. *Product/System Assessments*: To show that the system satisfies the safety requirements. From the perspective of process-based certification, safety is an emergent property of the whole system, including the operational environment.

Thus re-usability of process assessment is limited and re-certification is costly. On the contrary, product assessments enable the reuse of evidence artefacts, therefore supporting a compositional development. The certification of MCSs will require both types of

assessments. In a piecewise certification ideally both argumentations should be independent, while in practice preserving a strict separation between process and product assessment is challenging² [DGJ+12].

Safety Cases for Compositional Certification

A key tool to provide modularity in safety certification is the Safety Case (SC). A *Safety Case* is a structured argument supported by a body of evidence that provides a compelling, comprehensive and valid case that a system is safe for a given application in a given operating environment. Our approach to certification consists of providing a number of safety cases for foundational components (e.g., hypervisor, networking, etc.); when these components are integrated in a MCS configuration, then a specific global safety argument for this MCS can be also assembled to show the validity of the safety claims.

The Safety Case approach is already accepted in safety-application domains like railway applications (according to the requirements of EN50129 [TR50506-2]) or air traffic management systems. For the latter, EUROCONTROL published a safety-case development manual [SCDM06] for Air Traffic Management applications, based on the GSN notation [Kel98]. Other safety standards allow the use of SCs, even if there is no specific guidance on the SC structure, or the overall structure for cross-reference. Appendix D.1 recalls the Safety Case structures described in [TR50506-2, SCDM06].

Herein we will adopt a multi-layered SC structure, the underlying SCs corresponding to pre-built safety arguments for safety components and a number of derived SCs representing the safety argument for a particular MCS product from a mixed-criticality product line. In order to lower the cost of manual argumentation/documentation rework, we seek a computer-assisted production of SCs for derived MCS. It is worth recalling [NEFAB]: *“The Safety Case is not an alternative to carrying out a Safety Assessment; rather, it is a means of structuring and documenting a summary of the results of a Safety Assessment, and other activities (e.g., simulations, surveys, etc.), in a way that a reader can readily follow the logical reasoning as to why a change (or on-going service) can be considered safe.”* From the certification perspective this means that tool semi-automation alone does not guarantee the quality of the SC with regard to properties like readability, credibility or understandability of the arguments. The evaluation of such properties requires a semantic interpretation of the safety argument, involving many subjective terms that make automation extremely difficult (see the SC evaluation check-list in Appendix D.2, reproduced from [SCDM06]). In the scope of this research we assume that this task is would be carried out by a qualified reviewer.

²For example, the evidence for the execution of a test activity could be the evidence generated, i.e., the test results that would be also the supporting evidence for a product assessment, e.g., evidence of correct functionality.

Modeling Safety Arguments to Support Certification

Mixed criticality product lines consist of a number of complex systems implementing some functional safety feature. The safety-critical aspect of **MCSs** requires that the system suppliers engineer their products and services to prevent the risks posed by potential malfunction, in such a way that users and other stakeholders (e.g., a certification body) can rationally possess the needed confidence in them or at least judge their level of risk. Suppliers of **MCS** product lines must not only ensure their delivery of adequate systems, but customers and users require the explicit, valid, well-reasoned, and evidence-supported grounds for their confidence and decision making including related to engineering conclusions and their uncertainty. In the **DREAMS** piecewise certification approach we can associate the supplier role to the providers of the foundational components (and the corresponding safety argumentation), and the customer role to the integrated **MCS**.

Safety cases covering safety and security requirements for systems constitute a valuable tool for the interchange of assurance information. To make system assurance more practical, automation and meaningful exchange of this assurance-related information is needed. Argumentation models are flexible and extensible means for the representation and exchange of **SCs**. We shall describe the safety arguments using an argumentation meta-model for representing structured safety cases. A safety case is a set of auditable claims, arguments, and evidence created to support the claim that a defined system or service satisfies particular safety requirements. A safety case is a document that facilitates the information exchange between various system stakeholders such as suppliers and customers, and between the operator and regulator, where the knowledge related to the safety and security of the system is communicated in a clear and defensible way. Each safety case should communicate the scope of the system, the operational context, the claims, the safety and/or security arguments, along with the corresponding evidence.

Systems assurance is the process of building clear, comprehensive, and defensible arguments regarding the safety and security properties of systems. The vital element of systems assurance is that it makes clear and well-defined claims about the safety and security of systems. Certain claims are supported through reasoning. Reasoning is expressed by explicit annotated links between claims, where one or more claims (called sub-claims) when combined provide inferential support to a larger claim. Certain associations (recorded as assertions) between claims and sub-claims can require supporting arguments of their own (e.g., justification of an asserted inference). Claims are propositions which translate into statements in some natural language.

The precision in formulating the claims may contribute to the comprehensiveness of an assurance case. The context is important for the scope of the claim, and for clarifying the language used by the claim by providing necessary definition and explanations. Context involves assumptions made about the system and its environment. An explicit statement about the assumptions enhances the comprehensiveness of the argument. A well-structured argumentation flow between claims facilitates the communication of the entire **SC**.

A *safety case* specifies the interpretation and implementation of safety requirements in a system, including the engineering decisions and rationale to show the safety achievements. To achieve an effective certification method, we tackle the compilation of the whole set of safety information required by each variant of an **MCS** product line. Such a compilation is costly and time-consuming: even when two variants show minor differences, their safety argumentation may share only some small fragments. To improve cost-effectiveness, we aim at automating the construction of preliminary safety arguments, after a successful safety evaluation of a candidate **MCS**. A number of software components process the argumentation models from the **DREAMS** tool set (e.g., the **SCCRC** evaluator [DRE433], integrated in **AF3**). At this point, we reflected on the best language to describe, store and compose argument models as intended, bringing these conclusions:

1. Argumentation Meta-model (**ARM**) is deprecated, and Structured Assurance Case Meta-model (**SACM**) should be used instead of ARM.
2. The **GSN** has been in use for a while, and it has been enhanced and extended to support the certification arguments in many ways, e.g.:
 - Habli and Kelly introduced **GSN** extensions to support a safety case approach [HK10];
 - Hutchesson and McDermid [HM13] applied **GSN** for developing trusted product lines.
 - Denney et al. [DPP12] developed AdvoCATE, a tool to automate building certification arguments.
 - Denney et al. [DPH11] extended the **GSN** with annotations for confidence measurement, to handle also possible subjective criteria w.r.t. the strength or credibility of the solutions supporting an argument. This enables the integration of complimentary information, e.g., the 'strength' credited to the evidences by a reviewer to support an argument (e.g., position paper CAP 760 [CAP760] states the supporting strength associated with evidences generated by specific arrangements of Verification and Validation (**VnV**) techniques).
 - **GSN** usage is recommended to structure the arguments in the aerospace domain [CAP760], with **SC** development guidelines [SCDM06].

In addition, FORTISS developed a **GSN** editor, capable of handling and assembling argument models based on the **GSN** meta-model.

3. Claim Argument Evidence (**CAE**) is a proprietary language, with a limited tool support [ASCE, CERTWARE].
4. **SACM** provides more expressiveness than **GSN** or **CAE**, which eases the automation and integration of information; on the other hand, it does not define a graphical

depiction of the arguments, where it is a useful feature of **GSN** or **CAE** to present the argument in a reader-friendly format. Another drawback of **SACM** is the novelty of the specification, which makes **SACM** unstable and could bring about maintainability issues for the models or tools.

7.2 Certification Support in DREAMS

As pointed out in §7.1.2, the mixed-criticality systems considered in **DREAMS** have some special properties that affect certification and approval, namely:

1. The need for total separation between critical and non-critical applications.
2. The high degree of configurability and re-configurability.

The first of these properties, the need for total separation, has been extensively covered in [DRE551], where detailed argumentation chains are described both generically and in particular for the XtratuM hypervisor to assert how virtualization will keep applications fully separated.

To achieve the second property, the high variability and configurability, we shall demonstrate how explicit modelling product-lines of **MCS** makes it possible to generate detailed and product specific arguments for any individual product up for approval.

DREAMS aims at a cost-effective development process for **MCS** product lines, where product samples may share common features, subsystems or components. The overall **DREAMS** development approach relies on the composition of modules to design the products, thus enabling the final users to reuse safety assurance artefacts either at subsystem or component levels when analysing the safety properties of the samples (e.g., safety manuals). Likewise, the **DREAMS** certification approach also consists of a modular composition of arguments that links the safety claims, starting from the safety requirements, to the final evidences that the developers have to provide to the certification authorities. This composition of arguments shall reflect the design rationale, ending in a number of proofs such as test results analysis and validation, analysis results, formal proofs, referred documents or available certificates for pre-certified items.

The **DREAMS** toolset supports the semi-automated Design Space Exploration (**DSE**), seeking for the best possible product configuration for a given set of requirements, also including safety requirements. During the design stage, these tools automatically select alternative candidate deployments of the logical components on the target hardware, resolving the variability models and going through a number of verifications in a safety evaluator component: the Safety Compliance Constraints & Rules Checker (**SCCRC**). As for the certification, this toolset also automatically assembles an argumentation model for each product sample. The outcome after the completion of the **DSE** is a partial argumentation model, which will be mapped to a set of certification documents according to the Functional

Safety Management process chosen by the **DREAMS** user. These documents are generated in a semi-automated fashion, taking into account that part of the final evidences would come from Verification and Validation (**VnV**) activities to be carried out at later stages of the project –some of these evidences are not available in the **DSE** phase.

The modular certification approach in **DREAMS** is supported by a database of argument models for each pre-certified or certifiable component. The **DREAMS** argumentation database eases tackling with safety standards from different application domains requiring specific argumentation patterns, as it can handle multiple argumentation variants tailored to the domain certification requirements. For instance, for a part having an IEC 61508 argumentation template besides a DO-178C argumentation model the **DREAMS** toolset could assemble differentiated argumentations for certifying products containing this component according to either safety standards.

A substantial burden of the certification process comes from the compilation of final evidences resulting from **VnV**. Note that safety standards impose process redundancy in the safety product development to prevent systematic errors, stated as a requirement for strict separation between activities related to product design and implementation and the quality assurance activities, including **VnV** tasks. In order to comply with the isolation between development and verification, the **DREAMS** certification support shall support interoperability with **VnV** information repositories. The **DREAMS** database-based argumentation provides such an extensible framework to relate diverse information sources into a single argumentation model. In the event of development iterations, the **DREAMS** approach facilitates updating the whole set of information, enabling improvements at the coherence between arguments. In the near future, automated argumentation model checkers may be integrated in the **DREAMS** toolset and help the certification process by automatically unveiling the weak or missing points in an argumentation, therefore reducing iterations and lowering the overall certification costs. The **DREAMS** certification approach could also benefit during the other phases of the product lifecycle, as the same rework on the argumentation models is applicable to the product evolution (e.g., substitution of parts, retrofitting, enhancements to the safety product line), requiring a re-assessment of the resulting configuration.

7.2.1 Certification Arguments

When a candidate product fits the safety requirements, the **DREAMS** safety oracle outputs the external assessment of the evaluation rationale by an expert. A structured representation of claims, arguments and evidences in a standardized format is desirable, either by adopting **CAE**, **GSN** or **SACM** standards. This would also enable exporting the argumentation to other specialised tools for further formal analysis or even back-to-back validation of the **DREAMS** toolset with other safety-analysis **COTS** tools.

Composition of Safety Cases

Product line approaches can be applied to the different requirements imposed by different safety-standards, or even for different safety integrity requirements while claiming compliance with a selected standard. The compositional approach to argumentation shall support that different argumentation patterns can be applied.

Argumentation Patterns

Arranging the safety cases in a stereotyped structure eases the composition of the supporting arguments. The OPENCROSS project contributed a number of patterns (templates) to build a layered safety argument.. Whenever possible, DREAMS safety arguments for certification will adhere to OPENCROSS patterns. We remark that the application of patterns does not necessarily improve the readability of the resulting SC, as advised in [OPE53].

Composition of Evidence

Safety evidences are produced as a result of applying different analysis and testing techniques to a component to back up the contract compromises. Although some evidences are easy to compose (e.g., fault trees) others require a detailed examination of the validity of the argument in the new context of the composition, whereas some simply cannot be composed. A particular challenge is the assessment of the 'certifiability' of a system resulting from the integration of several pre-certified modules. The DREAMS certification can be classified in what OPENCROSS D5.3 names "*evolutionary chain of evidence*". This means that when the system is evolving, the chain of evidence changes. In particular, this scenario appears when there is an incomplete set of evidences. This corresponds to a development scenario in which evidence is gathered and structured for a new system, thus the evidence is progressively collected and structured. The safety argument is not valid until all the pieces of evidence in the argument link are available.

Certification Artefacts

The DREAMS toolset outputs the collection of available evidences supporting the safety claims (i.e., demonstration of compliance with regard to safety manuals). Whenever the claims require additional evidences, the toolset enumerates the required strategies or evidences. When the available information is too abstract to define the required strategies and evidences, these are annotated as "underdeveloped", requiring further development by an expert in order to realize them.

7.2.2 DREAMS Design Space Exploration (DSE)

Design-space exploration (DSE) investigates alternative solutions to system configuration at design time in order to obtain a balance between conflicting system properties such as functionality, cost, complexity or energy consumption to name a few. The proposed approach finds Pareto-optimal solutions based on the input goals configuration in DSE.

The DREAMS project therefore approaches design-space exploration at both the business and technical levels, using variability models and an evolutionary optimization, respectively [DRE412]. At the business level, the variability models capture the design decisions that govern what functionality will be offered by the system. At the technical level, the evolutionary DSE algorithm explores alternative engineering decisions resulting in contrasted trade-off between different extra-functional system properties. Although complementary, the final products yielded by both approaches must remain within the limits of the design problem.

Design problems are indeed constrained by the application or the application domain, which may require for instance compliance with laws or domain specific regulations and standards. In the context of MCS in particular, the compliance with safety standards influences both the functional and extra-functional concerns. In general, to be pertinent and support the designer's decision process, design space exploration must be integrated with specific analysis (e.g., analysis of safety, temporal properties, etc.), to ensure that obtained solutions remain within the problem scope. The process we propose in Figure 7.7 combines the exploration based on business variability specification with an evolutionary optimization based on a given set of goals (i.e., the defined design constraints and optimization objectives), while minimizing their overlap. In the following, we describe this combined process by splitting the common approach in product-line exploration into several steps (i.e., to derive from a product-line description a set of high-quality products defined as models in a given domain language).

- The description of a product line is a specification of the variability as well as all reusable assets from which final products can be constructed. Here, reusable assets are encoded as so-called "150 %" -model in the domain language that is used to model the final products. In the proposed approach, the DREAMS application and platform-meta-model is used as domain-specific language. Since this model contains all assets that can potentially be used in the final product, it is in general not a valid product model itself.
- The first step is to identify a set of resolutions that maximizes interaction coverage between the different features contained in the variability specification. Then, each resolution is realized by the BVR engine. Instead of directly constructing product models (as it would be the case in a traditional variability exploration process), the combined process yields a so-called "125 %" model.

- In the "125%-model", the variability has been only partially resolved. This partial variability resolution process focuses on the resolution of business decisions governing what features to include into a product. However, the model contains some remaining variability that mainly concerns technical decisions. This is because the resolution of these technical decisions requires information that is only available internally during the evaluation phase of the evolutionary optimization algorithm. Here, a typical example is the execution schedule that is computed during the decoding step of the algorithm and that is required to perform decisions regarding the application of fault-tolerance mechanisms (e.g., whether to replicate components or to use diverse implementations).
- As pointed out before, most of the features are directly "realized" using some reusable assets in the first step. However, some specific features (e.g., those concerning safety) require further technical investigations and hence cannot be resolved using solely the variability specification. The evolutionary optimization engine collects the remaining technical variability decisions and searches among alternative implementations to obtain a complete product model (i.e., in the present approach to compute an application to platform deployment and potentially a redundant version of the original application model). Here, the exploration ensures that the selection of all extra-functional constraints (e.g., safety, timing) are met and that the solutions w.r.t. the defined objectives are optimized (e.g., energy consumption). The results form a Pareto-optimal set of contrasted solutions, i.e., valid product models or "100 %" models.

The **DSE** provides an optimization of the partially resolved models yielded by BVR, from which it generates a set of Pareto-optimal solutions [DRE412]. For each of these solutions, the metrics computed during the evaluation phase of the evolutionary algorithm are provided. These metrics enable the designer to evaluate the quality of the selected "125 %" models, and hence to select the best strategy for product sampling.

7.2.3 Safety-compliant use of the DREAMS tool inventory

Current safety standards require a demonstrable compliance to development process paradigms. The IEC 61508:2010 safety standard defines the general requirements for each phase of the development process. Part IEC 61508-7 provides an application guideline and a collection of recommended techniques for the design, development and verification of the safety critical system. Unless specifically addressed, the project team shall define the actual selection and combination of techniques considering the required systematic capability level, but there is no guarantee of positive assessment from the certification body.

The **DREAMS** tool inventory [DRE541] provides a number of features and use cases intended to ease the development of Mixed-Criticality Systems (**MCSs**) product lines. Some of the tools implement analysis capabilities that help at assuring the safety of each feasible

system variant. Table 7.1 recalls the toolset, and summarizes the potential evidences generated by each tool. Once available, the argumentation model could be referred to these evidences in order to support the safety claims. Noticeably most of the tools work with models of the product variant, and therefore the satisfaction of the safety claims by the final system implementation shall be demonstrated by complementary means (additional verification, validation and testing).

7.3 Automated Composition with Modular Safety Cases for Re-usability

This section describes the contribution of a re-usable argument database based on the modular safety analyses for the main subsystems of the DREAMS platform: Hypervisor [LPA+15], cross-domain Network-on-Chip (NoC) [LPN+16], Commercial-Off-The-Shelf (COTS) multicore device [LPO15], and, partitioning [LAN+15].

Without loss of generality, for demonstration purposes we selected the GSN language to describe and build the safety argument models for certification of MCSs.

7.3.1 Implementing GSN in Enterprise Architect model databases

The GSN notation language has been adopted in the work presented in this thesis for representing a product line development process. The main reason for that is its acceptance in the safety domain, available guidelines, and tool support. On the other hand, we use the Unified Modelling Language (UML) modelling application Enterprise Architect (EA), from Sparx Systems [EA], to model the safety arguments for the modular safety-cases³. EA supports the user's extensions named Model Driven Generation (MDG) Technologies, to extend EA's modelling capabilities to specific domains and notations. Kuono [EAGSN] released a basic MDG GSN extension for EA, which we extended by adding the modular GSN extensions, as well as other GSN stereotypes found in safety cases (see Figure 7.3). EA supports a concurrent engineering workflow by storing the model in an external DBMS, supporting concurrent access by multiple users. The modular GSN models for DREAMS were stored in an SQL DBMS.

³The DREAMS tool-set is a derivative of the AF3 environment, for which there is a GSN extension to model arguments. However, AF3 relies on information file-storage, and the AF3 GSN extension underwent adaptations to integrate with other DREAMS analysis tools. EA Database Management System (DBMS) is implemented to support collaborative teamwork, enabling a concurrent development of the DREAMS baseline of MSCs. These tools also enable developing application-specific compliance and VVT-arguments.

7.3. Automated Composition with Modular Safety Cases for Re-usability

Table 7.1: Evidences provided by the DREAMS toolset to support the safety certification
(source [DRE553])

Phase	Tool	Tool output	Evidence provided
	<i>AutoFOCUS3</i>	<ul style="list-style-type: none"> Model of system logical architecture. Model of system platform architecture. Mapping of application components on execution units. 	
	<i>Timing Model Editor</i>	<ul style="list-style-type: none"> Model of the logical architecture, extended with timing constraints (repetition and end-to-end latencies). 	
	<i>Safety Model Editor</i>	<ul style="list-style-type: none"> Model of the safety functions of the system. Library of models for HW/SW components, annotated with safety properties. 	
	<i>BVR Variability Editor</i>		
	<i>BVR Product generator</i>		
HW/SW Detailed Requirement Specification	<i>RTaW-Timing Decomposition</i>	<ul style="list-style-type: none"> Analysis results for the sub-latency constraints for each scheduling domain. 	
HW/SW Detailed Design	<i>Xconcrete</i>	<ul style="list-style-type: none"> Partition/task scheduling parameters, for the nominal mode. 	
	<i>TTE-Plan</i>	<ul style="list-style-type: none"> Off-chip network scheduling parameters. 	<ol style="list-style-type: none"> Deployment model for the allocation of software to hardware. Redundancy model for software components.
	<i>RTaW-OnChip-TT-Sched</i>	<ul style="list-style-type: none"> On-chip network scheduling parameters. 	<ol style="list-style-type: none"> Deployment model for the allocation of software to hardware. Redundancy model for software components.
	<i>GRec</i>	<ul style="list-style-type: none"> Partition / task scheduling parameters, for a set of core failure related modes. Core failure related mode transitions for the configuration of the GRM. 	
	<i>MCOSF</i>	<ul style="list-style-type: none"> Partition/task scheduling 	

Table 7.1: Evidences provided by the DREAMS toolset to support the safety certification
(continued)

Phase	Tool	Tool output	Evidence provided
		<ul style="list-style-type: none"> parameters, for transition modes. Transition mode parameters for the configuration of the GRM. 	
Safety Model Analysis	<i>IKL Safety Constraint Checker</i>	<ul style="list-style-type: none"> Analysis results for the safety properties of the system architecture. Preliminary GSN argument model for preparation of safety case report. 	5. Documented assertion that deployment model (SW Components into SW Partitions, SW Hypervisors, and Tiles) satisfies the System Safety Requirements.
Timing Model Analysis	<i>RTaW-Timing Evaluation</i>	<ul style="list-style-type: none"> Analysis results for the timing properties of the system deployment and timing model. 	6. Assertion that deployment and timing models meet the timing constraints, i.e., maximum latency assumptions and recurrent execution/communication assumptions.
Timing Model Analysis	<i>TTVerify</i>	<ul style="list-style-type: none"> Analysis results for off-chip communication parameter set timing properties and constraints. 	7. Assertion that the model of off-chip TT communication scheduling meets the timing constraints assumptions.
Timing Model Analysis	<i>Virtual Platform</i>	<ul style="list-style-type: none"> Scheduling parameter sets for task/partition execution and on-chip/off-chip communication. Occurrence and repetition parameters for the injection of different communication related errors: omission failure, corruption, link failure, crash failure, delay failure, babbling idiot, masquerading. 	8. Assertion that the statistical analysis yields acceptable test results for the communication latencies observed on the simulator.

7.3.2 Model-based Certification Workflow

This thesis contributes an extended **DREAMS** modelling toolset to ease the certification of Mixed-Criticality Product Lines (**MCPLs**). Figure 7.4 delimits the scope of the contribution, that introduces a pivotal *Safety Models Database* repository to store the pre-defined **GSN** argument templates for the Modular Safety Cases (**MSCs**) accompanying the **DREAMS** solutions to deploy safety functions.

The *Safety Models Database* is deployed in an **SQL DBMS** that is initialized with the database schemas provided with the Enterprise Architect (**EA**) **UML** design environment.

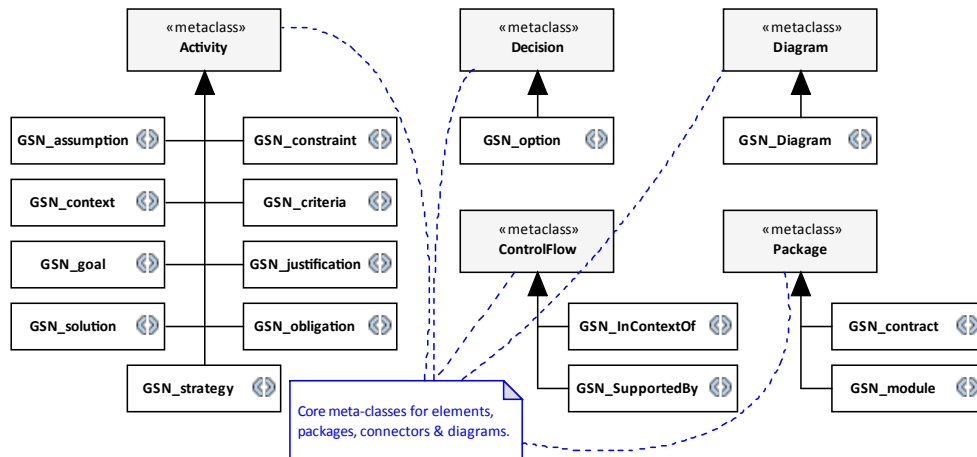


Figure 7.3: UML stereotypes for the GSN meta-model

This extension derives from the EA plug-in by Kuono [EAGSN]. The extended GSN toolbox is used to model re-usable safety-case argument fragments. These GSN models are stored in one of the SQL DBMS servers supported by EA, and later retrieved, traversed and instantiated by the DREAMS DSE engine.

Using EA we create an empty model in the DB, then we import the MDG profile extension to define the GSN meta-model, including the modular extensions. Developers of the MCSs can then concurrently build the GSN argumentation models for each MSC. The GSN arguments for MSCs may be parametrized, i.e., define place-holders to be replaced by specific data from a product safety case.

In case of incorporating COTS safety components to the DREAMS pool of components, the corresponding safety argumentation model should be added to the *Safety Models Database*. Other accompanying safety evidences in digital formats could be stored in a separate configuration management (CM) system. For instance, a safety-compliant product typically would have a safety certificate, and a safety manual. These could be referred to from within the GSN argument model using an Uniform Resource Locator (URL) to the CM system.

A user designing a DREAMS MCPL starts by creating a variability model, which defines all the feasible features, amongst them, the possible target safety requirements. Separate DREAMS models would be created to represent the properties of additional components to be considered at the automated DREAMS Design Space Exploration (DSE), e.g., WCET timing models for new application-specific SW parts.

Then the user selects the desired features in a MCPL configuration, where the Base Variability Resolution (BVR) component resolves the business variability, and generates a reduced variability model to start the DSE phase.

The DSE component generates a number of possible product configurations, the *Product*

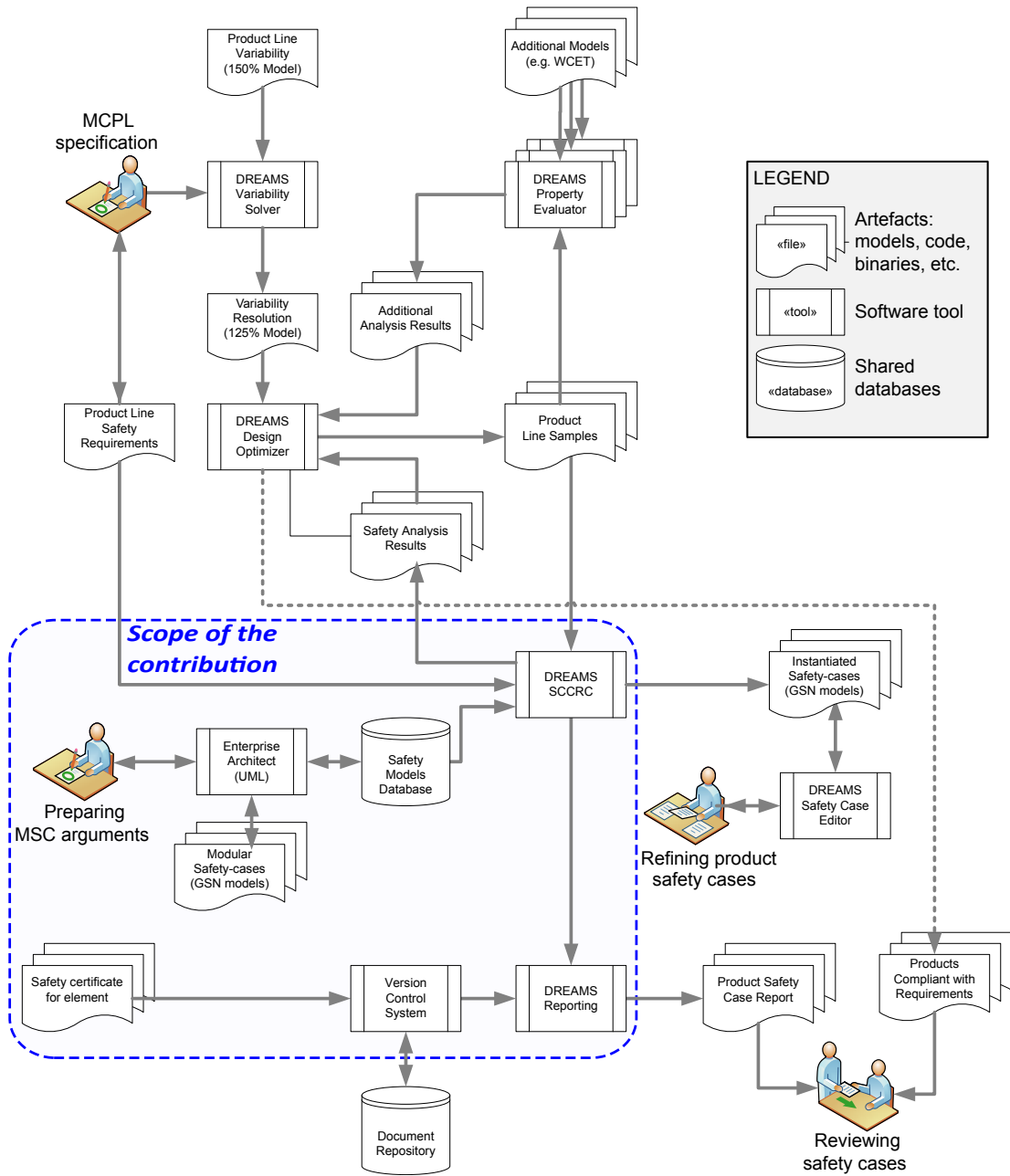


Figure 7.4: DREAMS workflow to support the certification of mixed-criticality product lines

Line Samples, which are evaluated by the DREAMS Property Evaluators, as well as the Safety Compliance Constraints & Rules Checker (SCCRC). The SCCRC checks the resulting safety properties for each particular product configuration, and feeds the verdict back to the DSE optimizer.

Once the valid product candidates are found, the user can invoke the SCCRC to automatically generate a preliminary Safety-Case Report (SCR). To create this document the Safety-Case Report (SCR) combines two information sources: (i) the internally generated GSN safety argument and (ii) the argument models for the MCSs stored in the database. The DREAMS reporting tool writes up the safety argument, detailing the rationale by which the SCCRC judged the product as compliant to the safety requirements. The resulting document also links the claim support to the evidences referred to in the MCSs.

The overall argumentation can be later completed by adding to the *Safety Models Database* additional GSN branches that compile the safety evidences gathered at later development stages (e.g., verification and validation results). The user can then invoke the SCCRC again, to check the completeness of the argument, and to automatically update the SCR.

The *Safety Models Database* eases tool integration into a collaborative framework by collecting the pre- and post-design information contributed by actors with different roles in the safety project.

7.3.3 Adapting MSCs for Reuse as Argument Models

The argument database contains inter-related GSN models corresponding to the safety analyses presented in [DRE511, DRE512, DRE513]. We should remark that each of these argumentation models consist of two layers: a generic safety case argumentation for an abstraction of the considered component, and a specific safety argumentation for a particular component choice. Figure 7.5 depicts this structure: the abstract argumentation modules (folders in second column from the right) is supported by a corresponding safety argument for a concrete component (folders at the rightmost part). The situation where several alternative components can be used as replacements is represented by the GSN Choice elements, and the justification about why the real component supports the abstract argument is represented by the contract arguments.

7.3.4 Supporting Certification for Diverse Safety Standards

Product line approaches can be applied to different requirements sets for different safety-standards, or even for different safety integrity requirements while claiming compliance with a selected standard. The compositional approach to argumentation shall support the application of different argumentation patterns. The OPENCROSS project defined a safety argumentation meta-model by combining the graphical depictions of GSN with the extended expressiveness of SACM, yielding the Common Certification Language (CCL). CCL

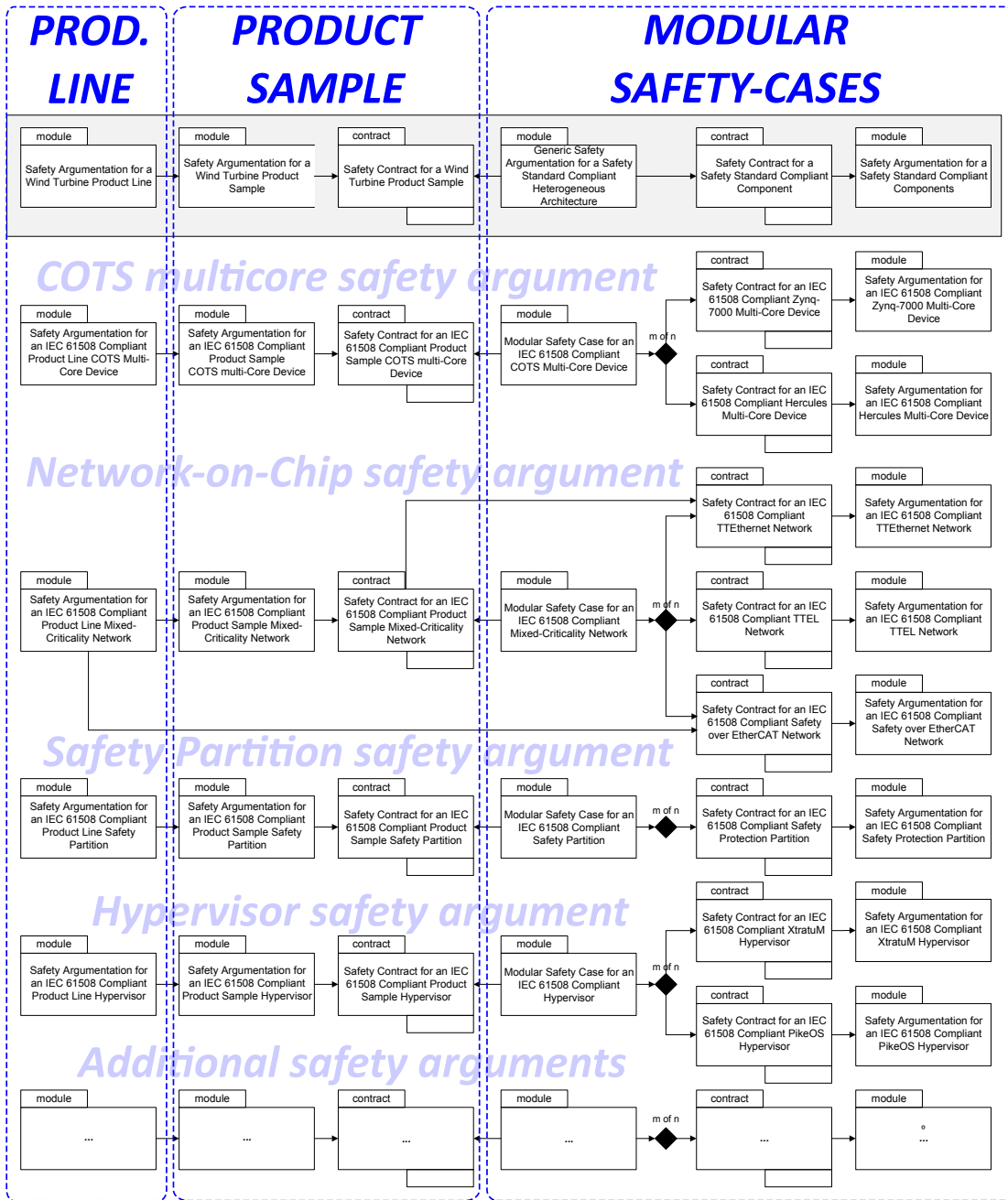


Figure 7.5: Layered argumentation models for DREAMS re-usable components

argument models can be agnostic to the specific safety standard, and by using mapping components, one can adapt an argumentation model to the argument structure required by a particular safety standard. OPENCOSS claims that this effectively provides cross-domain translatability of the safety arguments.

Unfortunately, CCL is not yet standardized, neither are the required tools available to the public. In the scope of DREAMS, a simple alternative is to include variants of the argument modules for cross-domain safety components, where each of these variants is tailored to the argumentation structure required by the applicable safety standard (see Figure 7.6). These variants would be stored in the argument database. These would be recalled by an extended SCCRC when needed.

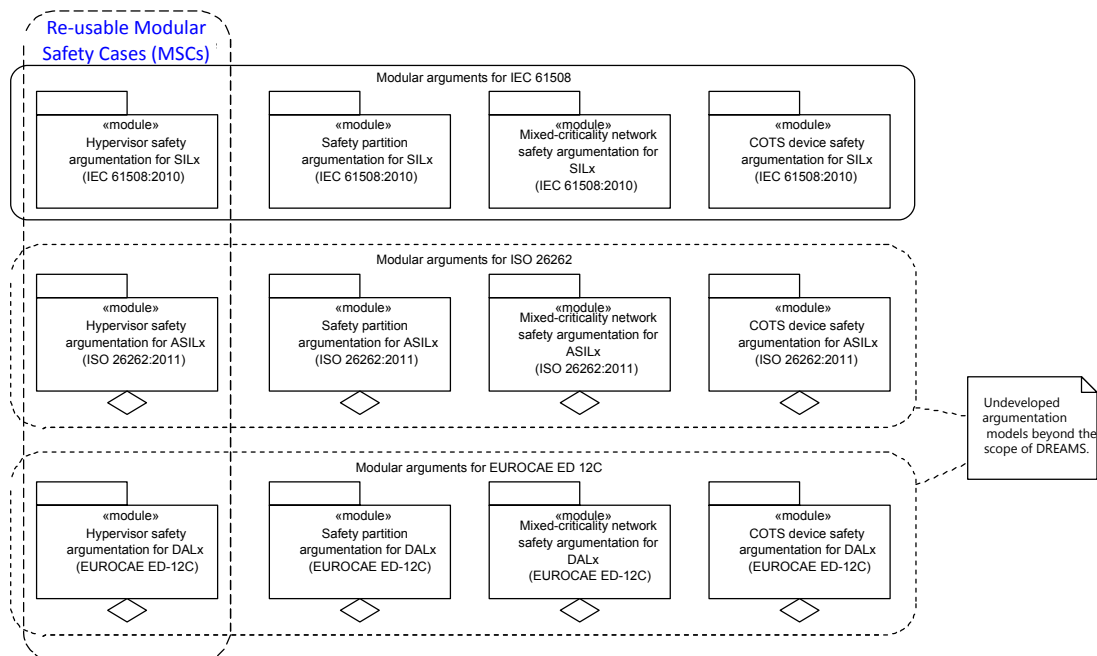


Figure 7.6: DREAMS argumentation database for diverse safety certifications

7.3.5 Integration of DB within Variability Management in MCPL

The DREAMS tool-set supports the development of Mixed-Criticality Product Lines (MCPLs) by handling variability in two layers (see Figure 7.7):

- *Business variability*, capturing **what** the system has to do.
- *Technical variability*, describing **how** the system does it.

In **DREAMS** the business variability is solved by the user when selecting the required features in the product line (e.g., functionality, **SIL** levels, etc.). On the other hand, technical variability is resolved automatically by the **AF3-DSE** design optimizer.

AF3 extensions compose pre-built **MSCs** according to the compliant product configurations, then document the preliminary safety cases with cross-references to either available or due documents. To attain this goal, the **DREAMS** workflow consists of the following steps:

1. Build the argumentation meta-models for the common components.
2. Set the design objectives into the design space explorer.
3. Run the optimizer.
4. When a product line configuration meets the safety requirements, a safety argumentation model is generated by the safety-validator.
5. The report generator translates the argumentation model for a given design solution into a set of documents with proper references to already available information (e.g. pre-built argumentations).

7.3.6 Integrating post-DSE VVT evidences

The **DREAMS** tool inventory allows the developer to generate a number of evidences supporting the safety claims. Nevertheless, to credit an **MCS** configuration for certification, the claiming organization shall elaborate many other complementary arguments and their corresponding evidences. Process redundancy and qualification of tools as required by safety standards imply that these additional pieces of information should be generated by an independent team, and possibly using different tools and formats. The integration of this information into a single argumentation is out of the scope of the **DREAMS** project. However, the underlying argument database constitutes a feasible integration infrastructure: it could be used to store and refer to the arguments and evidences generated in project tasks, executed concurrently with the design and analysis of the **MCSs** product line.

7.4 Summary

The certification of safety-critical systems requires a considerable effort to provide coherent information about the system properties and its development, in an understandable and re-viewable from. Preserving the coherence of the information with the information split across multiples sources is challenging. Argumentation models are valuable tools to integrate diverse information sources into a global overview of the safety claims and the rationale and evidences to support these claims.

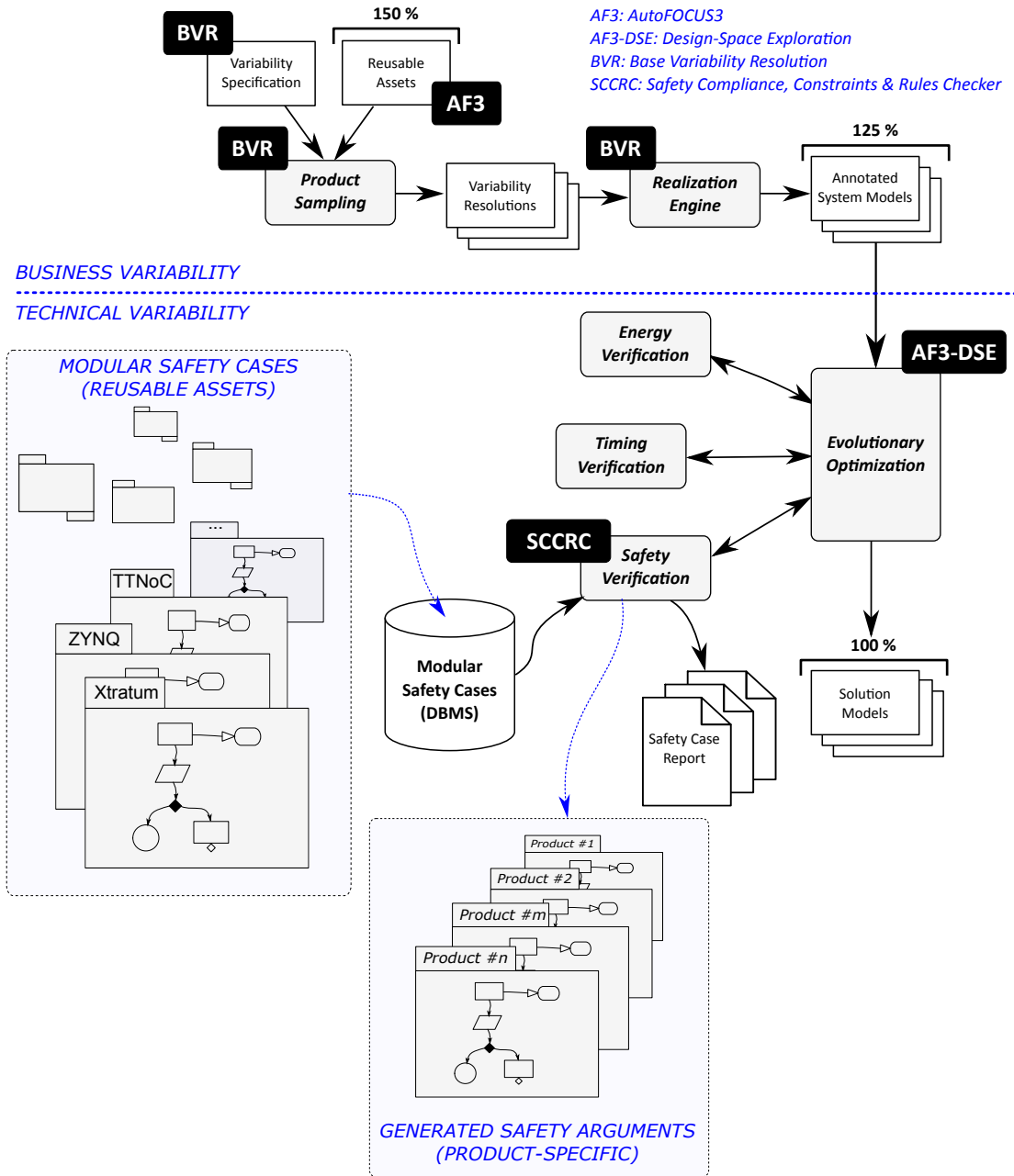


Figure 7.7: Handling variability and safety in the DREAMS process (adapted from [DRE431])

While **MCSs** product lines bring up families of alternative designs satisfying the safety requirements, safety remains an emergent property specific to each particular product configuration. Thus for certification purposes, one shall bundle the safety-relevant information on a per-product configuration.

In the **DREAMS** approach a number of pre-built safety components form the basis to build the different variants, using two levels of abstraction to ease the instantiation of the components. These foundation arguments are stored in an argument database, that could also support the development of complementary supporting arguments, e.g., collection of **VVT** arguments and evidences. This database will be accessed from the **SCCRC** component to assemble a preliminary **SC** for each **MCSs** configuration, generated in a semi-automated fashion, and containing the required document references to support the overall safety claim.

7.5 Discussion

This section describes some identified limitations as well as possible improvements to the contributed modular certification approach.

7.5.1 Limitations

The adoption of a model-based safety argumentation in **DREAMS** supports the automated safety assessment of candidate **MCS** configurations, as well as the transcription of the safety rationale to a set of cross-referenced documents for expert review. However, there are limitations in the prototype implementation of the system:

1. *Hard-coded model-to-document transformer:* The **SCCRC** module transforms a **GSN** safety argument model into a safety-case document structure calling an **SW** module programmed in JAVA. This module accesses the **DBMS**, traverses and retrieves the **GSN** graphs to assemble the safety argument for a given system configuration. The conversion of the argument graph to \LaTeX code is programmed in the module, that at the time does not support document templates or scripts to extract the information. As a consequence, the **SCR** has a fixed format. A new **SCR** writer module should be coded to adapt the **DREAMS** tool-set to other **SC** document structure.
2. *The arrangement of safety arguments is specific to a standard:* **DREAMS** focuses on safety certification with regard to IEC 61508 and its derivatives. The adaptation of the **MCSs** to other safety applications (e.g., DO-178C) may require the definition of alternative **GSN** graphs, as well as a modified **SCR** writer module. This hampers the applicability of **DREAMS** to derive cross-domain safety **MCPLs**. The **OPENCROSS** project tackled this problem differently by defining a more abstract safety meta-model. Safety arguments were specified at this abstract level, then these were converted to safety arguments specific to the safety standard applicable in the domain.

3. *Definition of the proper contents in SCRs*: A wholly connected safety argument would translate into a too detailed and complex document, that ultimately could be rejected from the review. The current workaround is to insert GSN contracts to be used as argument end points, but this is prone to user abuse, that would fragment the information in too many documents.

7.5.2 Improvements

The DREAMS GSN modular argument eases the composition of system and pre-built components safety arguments in a single GSN graph. GSN modules and GSN contracts are organizational constructs to manage the argument model, and do not affect the underlying ('flat') chain of argumentation. However, the supporting evidences for a given argument may be available at different stages, depending on the kind of evidence.

The OPENCROSS [OPE53] project contributed an argumentation template consisting of several module patterns, that target compositional certification. Arguments are classified in three groups, according to the kind of evidence(s) provided:

- *Direct evidences*, relying on a direct proof of the system correctness;
- *Backing evidences*, related to cross-verification of compatibility (e.g., holding of assumptions at integration); and
- *Compliance evidences*, that show how a component/subsystem was developed according to the accepted standard procedures.

Figure 7.8 depicts a generic GSN system safety argumentation model, structured in compliance with the pattern of modules and contracts. We remarked in the figure a suitable mapping between DREAMS argumentations and the generic argumentation templates defined by OPENCROSS:

- Module patterns `CompNDev`, `CompNAssump`, `CompNSpec` and `CompNCompliance` shall be instantiated by a safety engineer for each modular subsystem in the DREAMS platform. The group of instances capture the safety arguments from the Modular Safety Cases (MSCs) and defines the logical interface for integration (e.g., the assumptions to hold). This constitutes a library of re-usable arguments.
- Module patterns `SysSafe` and `SysSafeReq` would be instantiated in DREAMS when specifying the MCPL requirements. The instances shall be defined by the DREAMS user, and would be linked to modules and contracts from the MCSs.
- Contract patterns `CompNInSys` and `SysAssump` would be instantiated by the DREAMS DSE toolset, through the SCCRC safety oracle.

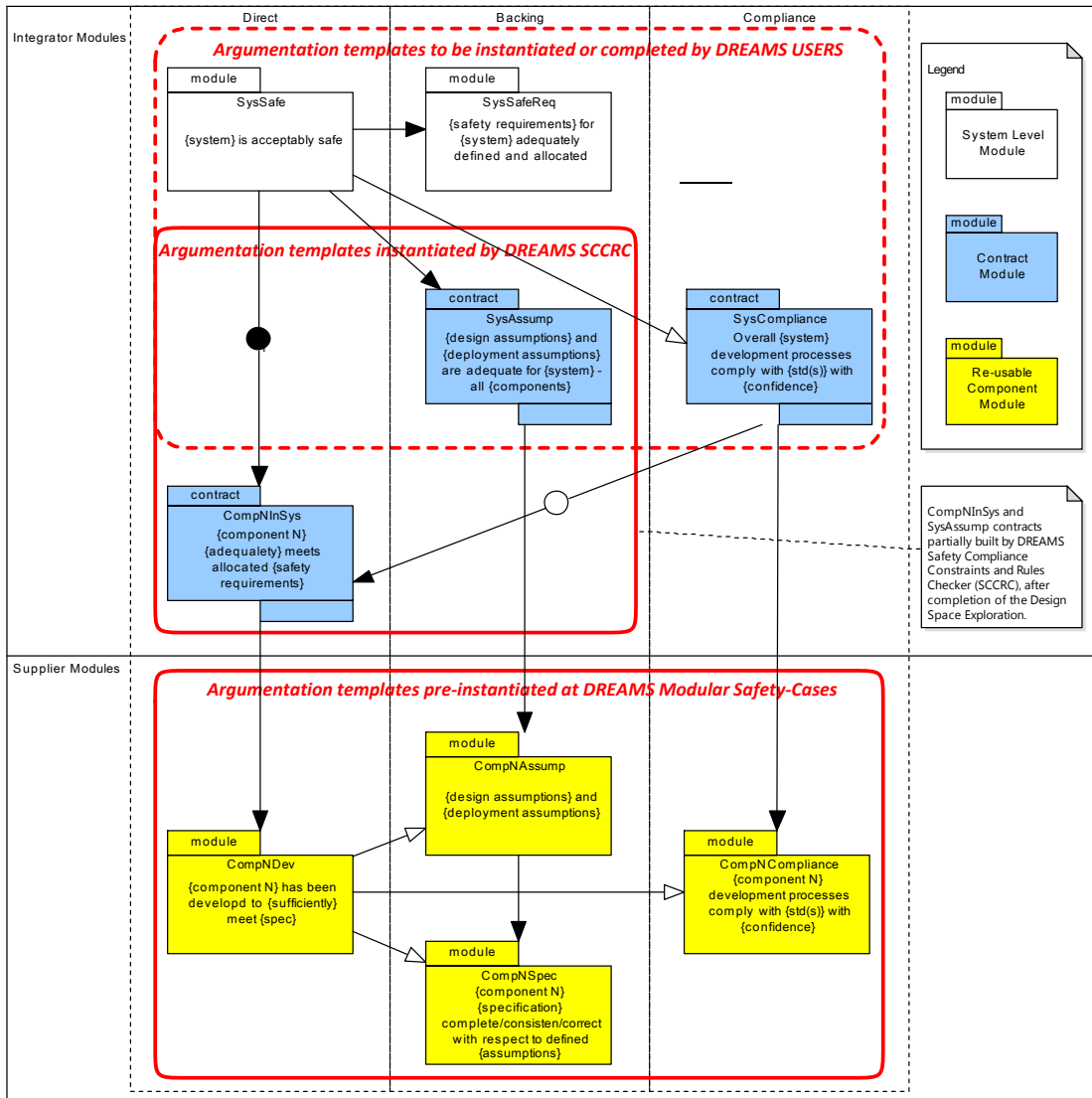


Figure 7.8: Mapping OPENCROSS argument templates to DREAMS certification arguments

The instantiation of the contract `SysCompliance` requires additional information about the process development, e.g., safety management, staff qualification, quality control process, etc., that should be provided by the organization.

The **DREAMS** support to the certification of **MCSs** is twofold:

1. *Product-oriented Certification Support:* The **DREAMS** harmonized platform provides safety components that are re-usable across different application domains. These

components can be combined in different deployments, as to satisfy the requirements specific to each product variant of an **MCS** product line.

2. *Process-oriented Certification Support*: The **DREAMS** toolset provides automated analysis and configuration features that would ease the staged development of the **MCS** product line, and eventually provide evidences to support the safety claims.

When the **DREAMS** toolset explores the design space (**DSE**) it relies on oracles that compute the performance of a candidate product for different properties, including safety. The safety oracle analyses the suitability of the product architecture and the safety manuals of the intended components and subsystems to establish which safety integrity level can be claimed. This evaluation is based on a rationale elicited from the safety standard, and the information provided from the safety manuals. A preliminary Safety Case (**SC**) is then generated, based on safety-relevant information from the **DREAMS** platform components and the system composition analysis, i.e., a given **MCS** product configuration. This preliminary **SC** can be either completed at a later stage with additional arguments and evidences (e.g., by integrating information gathered from **VnV**), or may refer to documents that shall be available prior to the certification to support the safety claim.

We identify the following challenging issues for assembling preliminary **SCs** for **MCSs**:

1. The integration of fragments of diverse information, described in diverse formats and possibly stored on distributed information sources (FTAs, HAZOP, requirements databases, natural language documents, system models, etc.);
2. The preservation of traceability and coherence between information cross-references;
3. The description of the safety claim arguments in a human-friendly format.

The integration of the safety-relevant information is one of the main concerns of the projects **OPENCOSS** [**OPENCOSS**] and **AMASS-ECSEL** [**AMASS**]. In **DREAMS**, we relate the information sources by using textual links (e.g., an **URL** or a bibliography reference entry) to tackle issue 1, and the toolset will partially address the issues 2 and 3 by relying on compositional argument models to build the **MCSs** safety-claim argument. To this end, we require re-usable and coherent information about the modular safety-cases, and this should be ideally arranged in some argument database⁴. In order to simplify the composition of arguments in the **AutoFOCUS3** environment, we selected the Goal Structuring Notation (**GSN**) (see Appendix C) to represent the arguments. Already developed modular **SCs** for **DREAMS** components are translated to **GSN** and stored in the argument database.

⁴The database for modular safety arguments would refer to information out of the scope of the **SCCRC** component (e.g., graphical depictions), required to ease the overall understanding of the preliminary **SC**.

7.6 Conclusion

The objective of this research is to provide a structured methodology to ease the certification of mixed-criticality product lines built upon the certified and/or certifiable components of the **DREAMS** harmonized platform. The cost-effectiveness of certification of mixed-criticality product lines is challenging, as safety is an emergent property specific of each particular system configuration (i.e., a product sampled from the product line). This implies that a specific safety argument shall be developed and provided for each valid product variant, and all the supporting evidences shall be collected and structured alongside this particular argumentation. To fulfil this requirement we propose a composition of safety arguments resembling the compositional approach adopted for the product development: we provide a library of safety-argument fragments for all the **DREAMS** re-usable components, expressed in a graph notation that enables the semi-automated composition of argument models and the production of evidence documents (i.e., preliminary safety-cases).

When developing a mixed-criticality product line, the integrated design toolset can partially assemble a root safety argument for each product configuration. **DREAMS** analysis tools help at assessing the satisfaction of the safety-related requirements for a candidate product model by static or dynamic analysis. **DREAMS** fault-injection tools provide additional verification and validation tools to test the actual system under some foreseen fault scenarios, bringing additional evidences to show that the system handles these faults adequately with regard to the safety requirements. However, safety cases will typically require additional evidences to credit for safety certification that are either not specifically targeted by **DREAMS** or shall be produced by other means or tools (e.g., test plans, test artefacts for the real system). The argumentation models proposed in the **DREAMS** certification approach address this situation, as evidence elements in the argumentation model (i.e., **GSN** Solutions) can link to other documents. This way, the argument model becomes the backbone of the overall argument for a successful product certification.

The proposed certification approach preserves the process redundancy required by safety standards: the strict separation between development and verification activities. While the project team concerned with the design and development brings safety argument models for the **MCPL** configuration, the **VVT** project team works on producing credible arguments and conclusive evidences or counter-evidences that should substantiate or rebut their safety claims for each particular system configuration. Both teams produce two argumentation models independently, and afterwards the **VVT** team will connect its argument fragments to the corresponding design and development argument model, linking to the argument nodes the required supportive evidences from analysis, testing or review. This yields a global safety-argument model of the system. Finally, this global argument model can be mapped to a number of standardized documents, as required by a Functional Safety Management (**FSM**) process specification.

Part III

Validation

8

Validation on Case Studies

This chapter presents three different case studies used to determine the applicability, achievements and limitations of the re-use approaches contributed in Chapter 5 (re-use of fault campaigns on Platform-Specific Time-Triggered Model (PS-TTM) models of safety systems), Chapter 6 (re-use of models and code for real-time HiL test systems), and Chapter 7 (re-use arguments for safety certification of mixed-criticality systems).

8.1 Development of a Railway Controller

This case study evaluates the feasibility of integrating an automated test executor to exercise the fault tolerance of a redundant safety controller architecture compliant with the Time-Triggered (TT) communication paradigm, by using a SystemC-based PS-TTM simulator for platform and functional models based on the TT model of computation.

8.1.1 Context

The system under consideration is a safety controller intended to prevent a train from exceeding the allowable speed limits. From the requirements and risk analysis it is stated that the safety device shall fulfil a Safety Integrity Level (SIL) of 4, according to the EN50129 [EN50129] railway standard, and it will integrate programmable electronics to deploy the safety functions. On the other hand, considering the safety properties of the hardware (HW), a redundant system structure is specified to achieve the required SIL level. The distributed functions of the controller will communicate through a TT network

that provides determinism. Besides, the safety application shall be developed seeking diversification in the programming tools and teams. As a consequence, the safety application would be composed of pieces of software that would be either programmed manually or coded automatically from a functional model.

It is intended to use the Platform-Specific Time-Triggered Model (**PS-TTM**) modelling and simulation framework presented in Chapter 5 to support the early analysis of the achievable fault tolerance based on the functional and deployment specifications. To that aim, we model the Time-Triggered execution of the safety functions and the target system structure at two abstraction levels: first we abstract from **HW**, which yields a platform-independent model; second we combine the functional model with a deployment model, mapping functions to **HW** components, which yields the platform-specific model. Concurrently, simulated fault-injection campaigns are planned for each model variant, configured with the **PS-TTM** and carried out with the integrated Automatic Test Executor (**ATE**). The analysis of the test results from the fault-injection campaign would unveil potential weaknesses in the safety-system concept.

8.1.2 System Description

The European Railway Traffic Management System (**ERTMS**) is a European-wide standardization initiative to enhance the safety, efficiency and cross-border interoperability of rail transport across Europe. **ERTMS** replaces signalling equipment with digitized, mostly wireless versions, and defines a single standard for train control and command systems. The two main components of **ERTMS** are European Train Control System (**ETCS**), i.e., the standard on-board train control [WG09], and the Global System for Mobile Communications-Railway (**GSM-R**), the GSM mobile communications standard for railway operations.

European Train Control System (ETCS)

The **ETCS** prevents over-speeding in high-speed trains by supervising the travelled distance and speed, and activating an emergency brake when the train exceeds the authorized values. The safety requirements for the **ETCS** state that it shall be designed as a safety-critical embedded system for **SIL-4**. In addition, the **ETCS** shall be a *fail-safe system* (see §2.3.3): in the event of a massive system failure, the **ETCS** should apply the emergency brakes to stop the train, so that the train reaches its safe state. Figure 8.1 sketches the structure of the **ETCS**, that is composed of the subsystems listed in Table 8.1.

Functionality of the ETCS

The on-board safety processing unit of the **ETCS** is the European Vital Computer (**EVC**), its functionality consisting of the following tasks, as shown in Figure 8.2a:

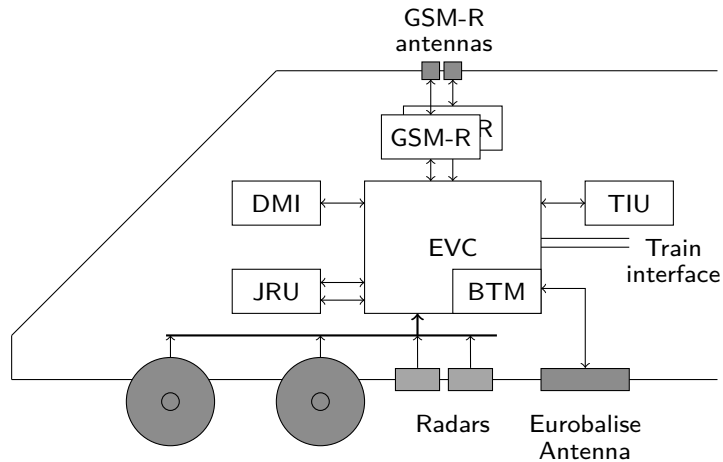


Figure 8.1: ETCS on-board reference architecture
(source: [WG09])

1. *Estimation of the speed and position of the train:* The **EVC** reads the information provided by a longitudinal accelerometer placed in the chassis of the train and two angular encoders located in two different wheels. With these measurements the **EVC odometry subsystem** estimates the current speed and position of the vehicle.
2. *Selection of the operation mode:* The driver can choose between two operation modes:
 - In *Standby* mode, the emergency brake of the train is activated and the service brake and warning signal are deactivated.
 - In *Supervision* mode, the **EVC** supervises the current speed and position of the train and activates the warning and brakes when the maximum permitted speed values are exceeded. Eurobalises provide absolute position references to correct the estimations. Upon detection of a balise in the railway by the Balise Transmission Module (**BTM**) the odometry system replaces the computed position estimate with the value read from the balise.
3. *Emergency brake control:* This task controls the activation and deactivation of the emergency brake of the train.
 - When the system is in *Standby* mode the emergency brake is activated.
 - When the train is in *Supervision* mode, the estimated position and speed are compared to a predefined braking curve that sets a maximum speed for each position on the track. The emergency brake is activated when the estimated speed exceeds the speed set by the braking curve.

The emergency brake is released only when the train is stopped and the driver sends a *reset* command through the Driver Machine Interface (DMI).

4. *Service brake control*: This task controls the activation and deactivation of the service brake and the warning signal.
 - In *Standby* mode, both the service brake and the warning are deactivated.
 - In *Supervision* mode the estimated position and speed are compared to the service-brake and warning braking-curves. The warning signal is activated when the speed of the train reaches the warning activation speed set by its braking-curve, and analogously, the service brake is activated when the speed exceeds the service brake activation speed.

Both the warning and the service brake are deactivated when the speed of the train falls below the warning activation speed.

Table 8.1: ETCS subsystems

(source: [ANP+14d])

ID	ETCS Component	Description
BTM	<i>Balise Transmission Module</i>	The BTM unit processes the information provided by the eurobalises as the train passes them, and transmits it to the EVC. The Loop Transmission Module (LTM) provides analogous functionality with the data received from Euroloops.
DMI	<i>Driver Machine Interface</i>	Interface for the train driver, which periodically updates state information like the speed and position of the train, and transmits user command events (e.g., button pressed).
EVC	<i>European Vital Computer</i>	EVC is the locomotive central safety processing unit that executes all the safety functions related to the supervision of the traveling speed and distance. The EVC executes the safety kernel, that includes the odometry subsystem, which fusions information from disparate sensors to estimate the vehicle speed and position.
GSM-R	<i>Global System for Mobile Communications - Railway</i>	The GSM-R is an interface for the management of bidirectional information exchange between the remote control centers and the train.
JRU	<i>Juridical Recorder Unit</i>	The JRU subsystem records a trace of all relevant external events (e.g., new eurobalise message) and internal events (e.g., activate emergency brake).
OS	<i>Odometry Sensors</i>	Is a group of sensors consisting on encoders, Doppler radars and longitudinal accelerometers that provide a set of measurements for angular speed and acceleration, and send the measurement data to the EVC.
TIU	<i>Train Interface Unit</i>	The TIU reads/writes a set of input/output digital values, such as the emergency brake digital output.

8.1.3 Modelling the ETCS in PS-TTM

This subsection describes how a simplified **ETCS** system is modelled applying the PI/PS-TTM approach. The subject of our case study is building a reduced **ETCS** system, omitting the **GSM-R** and Juridical Recorder Unit (**JRU**) subsystems. Besides, the boundary of the System Under Test (**SUT**) comprises the **EVC** and the **DMI**, while the **BTM**, the Train Interface Unit (**TIU**) and the odometry sensors are considered parts of the **SUT** environment.

Platform-Independent Model (**PIM**)

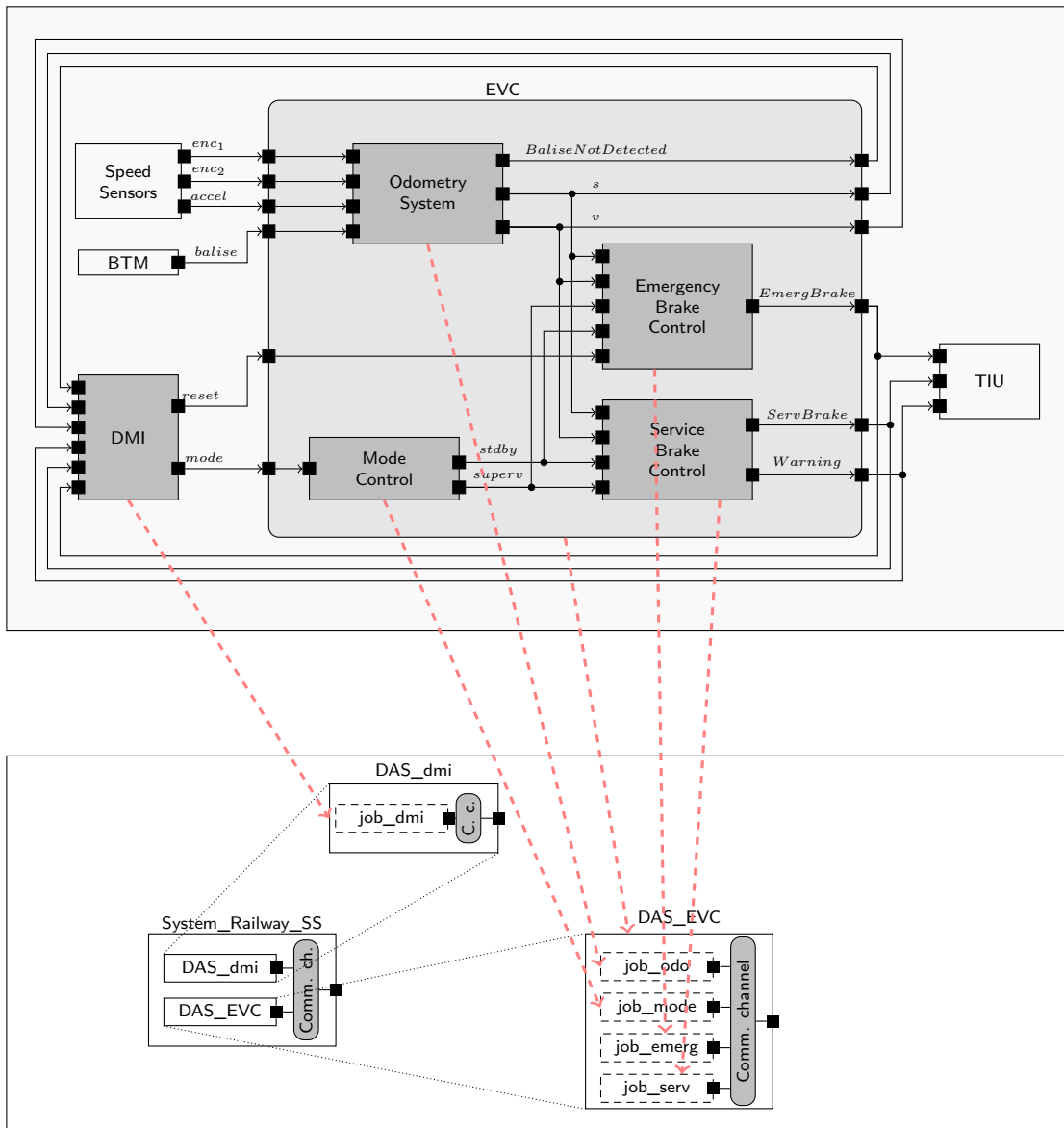
We design the **PIM** of the **ETCS** system relying on the **PS-TTM** modelling framework described in Chapter 5. We develop the platform-independent model of the **ETCS** in SystemC using the Platform Independent Time-Triggered Model (**PI-TTM**) library. As shown in Figure 8.2, we designed the **SUT** hierarchically, according to the spatial distribution of the functions. The **PIM** implementation consists of 5 *jobs* deployed in 2 main **DASes** (Distributed Application Subsystems): the 'DAS_EVC' and the 'DAS_DMI'. If preferred, the 'DAS_EVC' could be further decomposed by aggregating **DASes**, as shown in [ANP+14d].

Each function in the decomposition in Figure 8.2a maps to a *job* in 8.2b, where the thick red dashed arrows depict the mapping relationships. The 'DAS_DMI' contains a single *job*, 'job_dmi', that enables the driver to select the desired operation mode and release the emergency brake. Four jobs are mapped to the 'DAS_EVC', one for each of the previously defined tasks: The odometry estimation of speed and position is allocated to 'job_odo', the operational mode control goes to 'job_mode', and the emergency and service brake controls go to 'job_emerg' and 'job_serv' respectively.

The **ETCS** system activates an emergency brake when the values estimated by the odometry system exceed the authorized limits. Therefore, the odometry subsystem shall provide accurate and reliable measurements. Usually, an odometry algorithm is based on a fault-tolerant sensor-fusion approach. In this case, we design the algorithm following one of the approaches described by Malvezzi et al. in [MAR10]. The algorithm estimates the speed of the train and the travelled distance with the information provided by an accelerometer, that measures the acceleration of the train, and two encoders, each one of those measuring the speed of a different wheel.

We design the functions of the **SUT** in SCADE and we generate the C-code implementation automatically using KCG. Then we integrate the resulting components into the platform-independent model of the system for their verification. We also design a simplified model of the **ETCS** environment using SCADE. In this example we set the period of the system to $250ms$. All *jobs* in the **SUT** have *frequency* = 1, so the Logical Execution Time (**LET**) of the jobs is $250ms$.

(a) Functional structure of the ETCS.



(b) Platform-independent model of the ETCS in PI-TTM.

Figure 8.2: Mapping the functional structure of the ETCS to the PI-TTM model

Platform-Specific Model (PSM)

Once we have defined and verified the functional system with the **PI-TTM** library, the model is refined into a **PSM**. Following the recommendations from the IEC 61508 safety standard, we choose a Triple Modular Redundancy (**TMR**) configuration for the **ETCS** platform to be able to achieve the **SIL-4** safety requirement. Redundancy increases the robustness of the system against random faults, such as hardware faults.

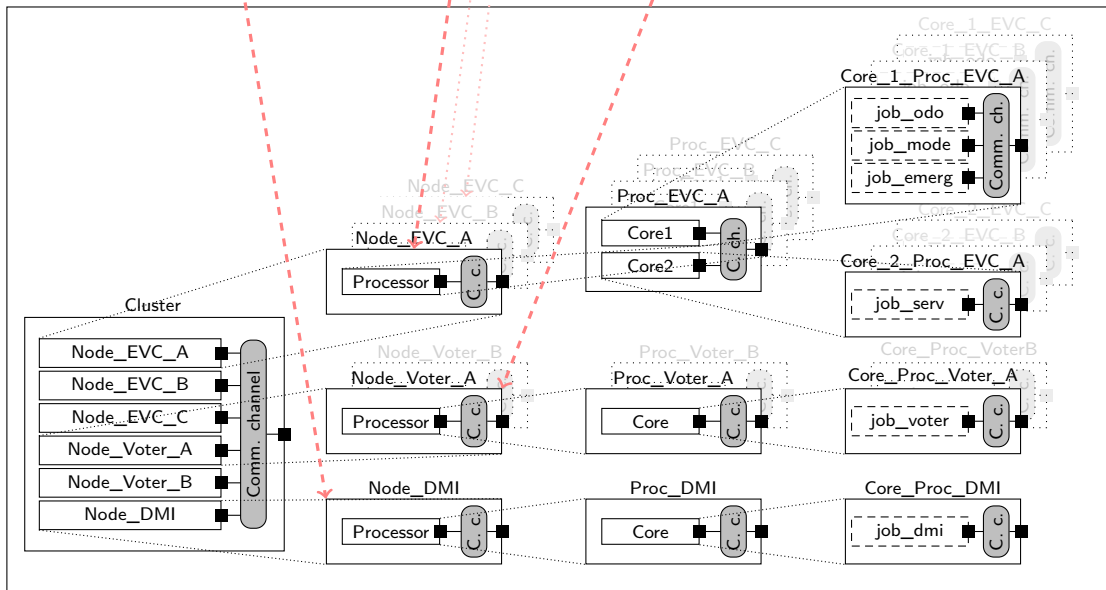
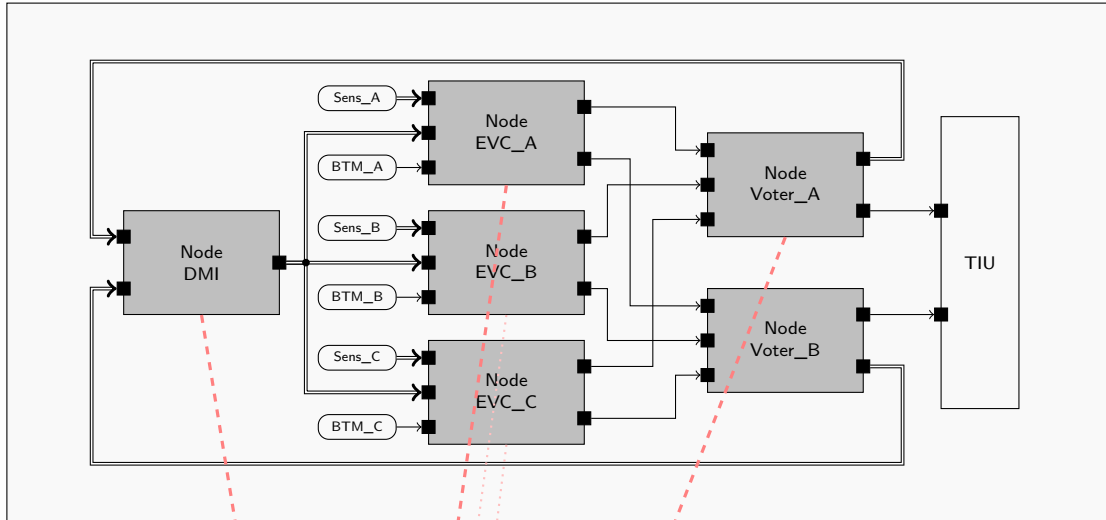
In this example the **PIM** is deployed into a Triple Modular Redundancy (**TMR**) platform, in compliance with the requirements from the EN-50126 international safety standard for railway applications. The **TMR** system is composed by three main *nodes*, each of them hosting a replica of the simplified **EVC** functionality. Each of the *nodes* is connected to its dedicated sensors and Balise Transmission Module (**BTM**).

The introduction of redundancy in the system requires the implementation of voters to handle the redundant output values. Two *2oo3* voters handle the replicated values of the **EVC** nodes. The voters receive 9 input signals (a warning, service brake and emergency brake from each **EVC** node) and they compare the replicated values to produce 3 output signals (voted warning, service brake, and emergency brake). Besides, the voters output two additional signals, the *failure warning* and the *system-failure warning signals* to inform the driver about the detection of failures in the system. The functionality of the voting system is the following:

- The voters start in *normal voting mode*. In this situation, if the three replicated input values are equal, the voter remains in *normal voting mode* and forwards the input values to the output value. No failure warning is sent to the **DMI**.
- If one of the replicated values received by the voter differs from the other two, the voter switches to *degraded voting mode*, behaving as a *1oo2* voter where the voting algorithm ignores the inputs coming from the faulty node. The result of the *1oo2* algorithm is forwarded to the output, and a *failure warning* is sent to the **DMI**.
- If there is a disagreement between the two active inputs when the voter is in *degraded voting mode*, the voter sends a *system-failure warning* to the **DMI** to notify a multiple failure in the system. The voting system is disconnected and the emergency brakes are applied to the train, whereas the service-brake and the warning are deactivated.

Figure 8.3 shows the **PSM** of the system in **PS-TTM**, modelled as a *6-Node Cluster*: The dual Voters (**Voter_A**, **Voter_B**) and the driver interface (**DMI**) are deployed on separate single-core processor nodes, each processor / core executing a single job. The redundant **EVCs** are hosted in three identical nodes, each node integrating a dual-core processor. One of the cores is the host for the safety-critical jobs, i.e., the *odometry* job, the *mode-control* job and the *emergency-brake-system* job, whereas the other core hosts the *service-brake system* job. We design the functions with SCADE and we generate C code automatically

(a) TMR structure of the ETCS.



(b) PS-TTM model of the ETCS.

Figure 8.3: Mapping the TMR structure of the ETCS to the PS-TTM model

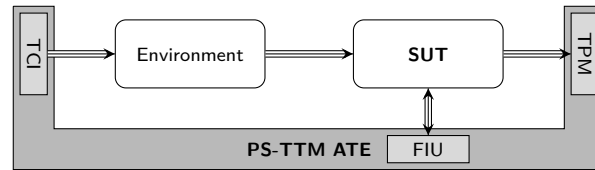


Figure 8.4: Connection between ATE (Python) and SUT (ETCS)
(source: [ANP+14d])

using the KCG tool. C code for the voters is also automatically generated from SCADE models.

The PS-TTM also integrates an ATE with the Test-Case Interpreter (TCI), Fault-Injection Unit (FIU) and Test Points Manager (TPM) modules mentioned before, which enables verifying the functionality of the system and validating the fault tolerance mechanisms. The PI / PS-TTM ATE is connected to the SUT and the environment model as shown in Figure 8.4 to assess the fault tolerance mechanisms introduced in the platform-independent / specific model of the system by injecting faults during simulation.

Listing 8.1: Example Python script to setup the PI / PS-TTM ATE and run a simulation

```

1 # File      : psttm_test.py
2 # Description: Python script to configure the test-case, test-points
3 #           and fault-injection for a single ATE-controlled
4 #           simulation
5 #           on the PS-TTM model.
6 import tcmodule as tc      # loads the test-case configuration
7 tc.LoadTestCaseDescriptionFile("./psmtestcase.xml")
8 tc.Confirm()              # configures the signal generators
9
10 import fimodule as fi     # loads the fault campaign configuration
11 fi.LoadDescriptionFile("./psmfaultinjection.xml")
12 fi.Confirm()              # configures the signal saboteurs
13
14 import tpmodule as tp     # loads the test-points configuration
15 tp.LoadTPDescriptionFile("./psmtestpoints.xml")
16 tp.SetTraceFileName("./psmtestresults")
17 tp.Confirm()              # configures the signal probes
18 # The simulation starts immediately upon setup completion,
19 # then the Python ATE exits automatically when finished.
20 # END OF FILE psttm_test.py

```

The PS-TTM ATE enables the test automation for massive fault-injection campaigns through its Python Application Programming Interfaces (APIs) to control the TCI, FIU and TPM components. For instance, listing 8.1 is a single-simulation Python script

that: (a) loads the timed sequences of **SUT** inputs from file 'psmtestcase.xml', (b) configures the fault mode and triggers the fault-injection saboteurs according to file 'psmfaultinjection.xml', and (c) specifies the virtual signal probes through the test-point configuration 'psmtestpoints.xml'. Output traces in *value-change-dump* format are recorded in file 'psmtestresults.vcd'. The statement `tp.Confirm()` runs the simulation automatically (if the test-case inputs are already configured) and then exits.

8.1.4 Reliability Assessment

This section describes the reliability assessment made to the railway-signalling system model described in §8.1.2. The fault tolerance mechanisms to be implemented in the system (e.g., **TMR**) are evaluated by means of simulated fault injection. Thus, the **PS-TTM ATE** framework is connected to the **SUT** and the environment model as in Figure 8.4.

Figure 8.5 depicts the extended **PI-TTM** model hierarchy for the **SUT** model from Figure 8.2b. To support test automation, the **PI-TTM** simulator model consists of a single **PS-TTM System** instance ('SYSTEM_testmdl'), that connects two **DASes** in closed-

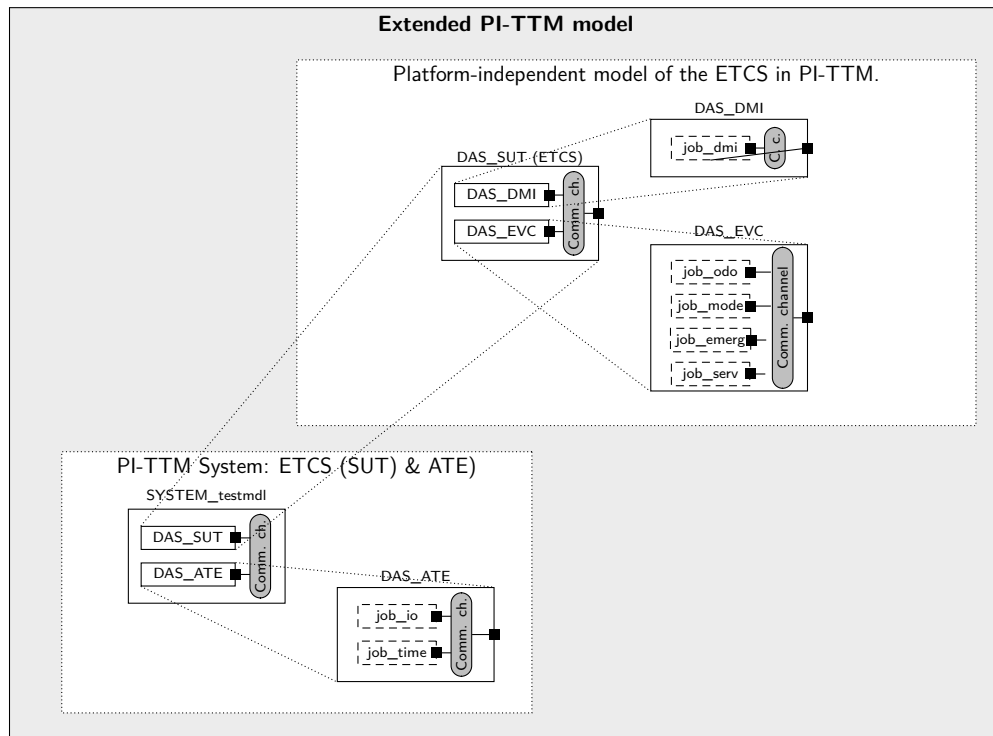


Figure 8.5: PI-TTM system model for ATE-controlled fault injection simulation of ETCS

loop: the **ETCS** model (i.e., the **SUT**, 'DAS_SUT'), and the Python **ATE** ('DAS_ATE'). 'SYSTEM_testmdl' is the top-level view of the test setup (including the **SUT**) and has no external communication interfaces. All the stimuli signals fed to the **SUT** are timely generated at the **TCI** component of the **ATE**, the effects of faults on the signals are injected by the *saboteurs* (i.e., executable fault-models) from the **FIU** component, and the data sinks to record output traces of selected signals are provided by the **TPM** component.

The virtual **ATE** 'DAS_SUT' includes two *jobs*:

- *job_io*: this task generates synchronously the test inputs from the test case description, pre-loaded through the **TCI** interface.
- *job_time*: this task controls the simulation time.

This way, the **TCI** parses the test cases defined in **XML** files and feeds the environment during simulation with the corresponding input signals. The environment model generates the sensor values and sends them to the **SUT**. The simulation engine sends all the communication signals inside the **SUT** to the **FIU**, which modifies their values according to the fault-injection campaigns defined by the testing team. The results of the simulations are sent to the **TPM**, which stores them for their off-line evaluation.

We designed the fault campaigns to exercise the fault tolerance of the **ETCS PIM/PSM** models against the faults identified in the Failure Mode and Effects Analysis (**FMEA**) that the implementation under test (**IUT**) should mask. Table 8.6 compares the complexity for both the PIM and PSM models in terms of number of modelling items, potential failure modes and error causes (more detailed information is provided in [Aye15]).

Reliability Testing on the Platform-Independent Model (**PIM**)

As mentioned before, the odometry algorithm should be tolerant to a fault in one of its sensors. Injecting faults at the functional model enables an early assessment of the robustness of the algorithm. Therefore, we first simulate the **PIM** by means of the **PI-TTM** engine with a pre-defined test case and we store the results provided by the odometry algorithm. We consider the results of this fault-free simulation the golden behaviour of the system. Then, we carry several simulations including the fault-injection campaigns shown in Table 8.2, and we compare their results with the golden behaviour.

For instance, Listing 8.2 shows the **XML** fault configuration #4 in Table 8.2 ('Wheel slipping during acceleration'). The integrated Python **ATE** loads the fault configuration in the **FIU** component of the **PS-TTM** framework, the **FIU** sets the trigger and behaviour configuration in the fault model. During simulation, the 'encoder 1' signal is sabotaged at simulation time $t = 91sec$ in the **EVC** input (DAS_EVC_IN_ENC1). Figure 8.6 graphs an extract of the results obtained in the simulation, showing the effect of fault #4 on the odometry estimates. The results show that the odometry algorithm designed for this system provides accurate results in the estimation of the travelled distance even in the

Table 8.2: Fault-injection campaign for the PIM model of the ETCS.

(source: [ANP+14d])

#	Fault Location			Fault			Fault Set		Description
	Job	Entity	Type	Effect	Attributes	Mode	Trigger time(s)	Duration (s)	
1	job_odo	enc ₁	input	I_C	0	p	130.0	-	Wheel stuck / Encoder broken
2	job_odo	enc ₁	input	I_S	-	p	180.0	-	Encoder broken (measuring a fix value)
3	job_odo	enc ₁	input	I_R	0,600	p	80.0	-	Encoder broken (measuring wrong values)
4	job_odo	enc ₁	input	I_C	600	t	91.0	7.0	Wheel slipping during acceleration
5	job_odo	enc ₂	input	I_C	0	t	150.0	8.0	Wheel skidding (blocked by brakes)
6	job_odo	accel	input	F_A	1.1	p	0.0	-	Accelerometer incorrectly installed
7	job_odo	accel	input	F_S	-	p	200.0	-	Accelerometer broken (measuring a fix value)
8	job_odo	accel	input	F_R	-2, 2	p	20.0	-	Accelerometer broken (measuring wrong values)
9	job_odo	accel	input	F_OR	-0.1, 0.1	t	35.0	50.0	Noise in the signal

presence of faults in the sensors. The maximum position-estimation error occurred during the 8th fault-injection campaign, where the maximum error raised up to 3.07m from a total of 6044.46m (0.05%, at 160.5sec). The maximum error in percentage took place during the 6th campaign, and reached 4.76%. Anyway, this happened at instant 8.250sec, where the travelled distance was still very low (0.21m travelled, 0.22m measured due to the fault).

Regarding the estimation of the speed, within the experiments made by means of our fault injection framework show the robustness of the algorithm. In this case we also get the maximum error in the 8th campaign, where at instant 151.75s we find a disagreement of 1.350m/s respect to the non-faulty simulation (60.23m/s, 2.24%).

All in all, estimation errors made by the algorithm are considered acceptable, since they always fall below the 5% of the travelled distance and speed, and never go further than $\pm 5m$ and $\pm 2m/s$. As a conclusion, we state that the algorithm has shown to be specially sensitive to faults in the accelerometer, so future work could focus on the improvement of this fact.

Reliability Testing on the Platform-Specific Model (PSM)

The PSM of the system introduces redundancy to tolerate hardware-related faults. A TMR architecture is chosen to guarantee the availability of the system even in the presence of

Listing 8.2: Fault config. #4 injected by the PI-TTM ATE on the ETCS PIM

```

1  <!-- FAULT CONFIGURATION -->
2  <FaultConfiguration>
3    <FaultSet>
4      <Name>FC4</Name>
5      <FaultMode>TRANSIENT</FaultMode>
6      <Duration>7.0</Duration>
7      <TriggerInstant>91.0</TriggerInstant>
8      <Fault ref="_p5LWtRbKEe0w28kCcXd1kQ" />
9    </FaultSet>
10 </FaultConfiguration>
11 <!-- FAULTS -->
12 <Faults>
13   <Fault id="_p5LWtRbKEe0w28kCcXd1kQ">
14     <Name>i4</Name>
15     <Location ref="_LeWBYBbJEe0Eg pzBm6gXmA" />
16     <FaultEffect>Integer_Constant</FaultEffect>
17     <ConstantValue>600</ConstantValue>
18   </Fault>
19 </Faults>
20 <!-- FAULT LOCATIONS -->
21 <Locations>
22   <Location id="_LeWBYBbJEe0Eg pzBm6gXmA">
23     <Job>system_inst.das_sut_inst.das_evc.odo_inst</Job>
24     <PortType>Input</PortType>
25     <Entity>DAS_EVC_IN_ENC1</Entity>
26   </Location>
27 </Locations>
28 </pim_fault_metamodel:FaultInjectionSettings>

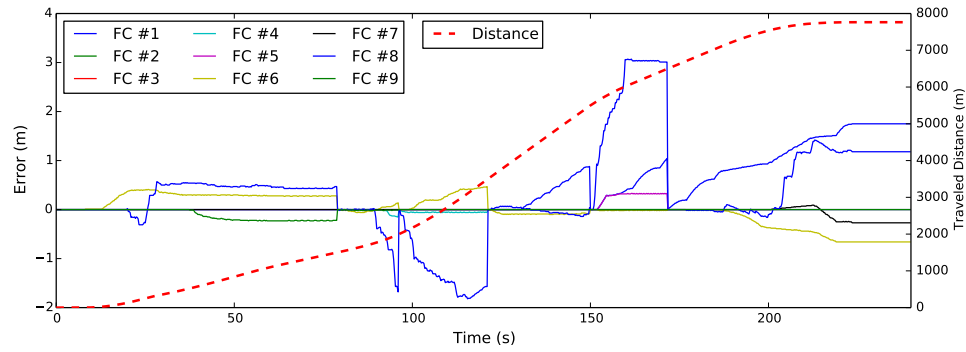
```

faults. As stated before, a voting algorithm should be able to identify any failure in the replicated nodes. In such cases, the voters send a failure-warning signal and ignore the results provided by the faulty node. If a second failure is detected in the system, the voters activate the emergency brake and inform about a failure in the system. We set the period of each job to the time-triggered macrotick, i.e., $250msec$.

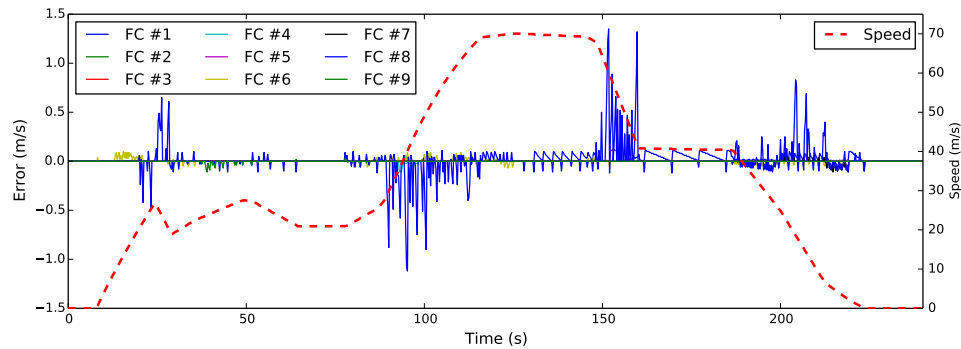
Listing 8.3 is an example eXtensible Markup Language (XML) fault configuration file used to simulate the fault campaigns on the PS-TTM model of the ETCS. This file is loaded from the PS-TTM ATE, besides the test case description and test points specification.

Table 8.3 shows the fault-injection campaigns made to the PSM in order to evaluate the voting algorithm.

Figure 8.8 shows the results captured by the TPM module when the system is simulated against a pre-defined test case. In the first simulation over the PI-TTM library (see Figure 8.8a), the TCI feeds the SUT with the test case and the system estimates the speed and activates the service brake and warning signal accordingly. In the second simulation (see Figure 8.8b) we repeat the same test case but we simulate a blocked wheel, by injecting a “stuck at 0” fault in the *encoder1* between $t = 24.0sec$ and $t = 27.0sec$ with the FIU. As the figure shows, the odometry system masks the fault and estimates the speed correctly,



(a) Traveled distance and estimation error due to faults



(b) Speed and estimation error due to faults

Figure 8.6: Results of the PIM simulation and fault injection
(source: [ANP+14d])

so the brakes and warning are activated as expected.

Figures 8.8c and 8.8d show the results of the simulations of the PS-TTM model. In the first simulation (see Figure 8.8c) we repeat the previous test case and we obtain similar results, with a small delay due to the execution time of the voters. In the second simulation we execute the same test case and we inject a permanent “no-execution” fault in the *core1* of the EVC node A at instant $t = 14.5sec$, which mimics the destruction of the core. As Figure 8.8d shows, the speed estimated by node A gets stuck at $t = 14.5sec$, whereas the other nodes still estimate the speed correctly. The output of the system is still correct, since the voters mask the faulty node.

During the fault-free simulation, the system activated the warning 4 times, and the service brake once. As expected, the failure and system-failure warnings were not activated. The results obtained for each fault-injection campaign are summarized in Table 8.4. All

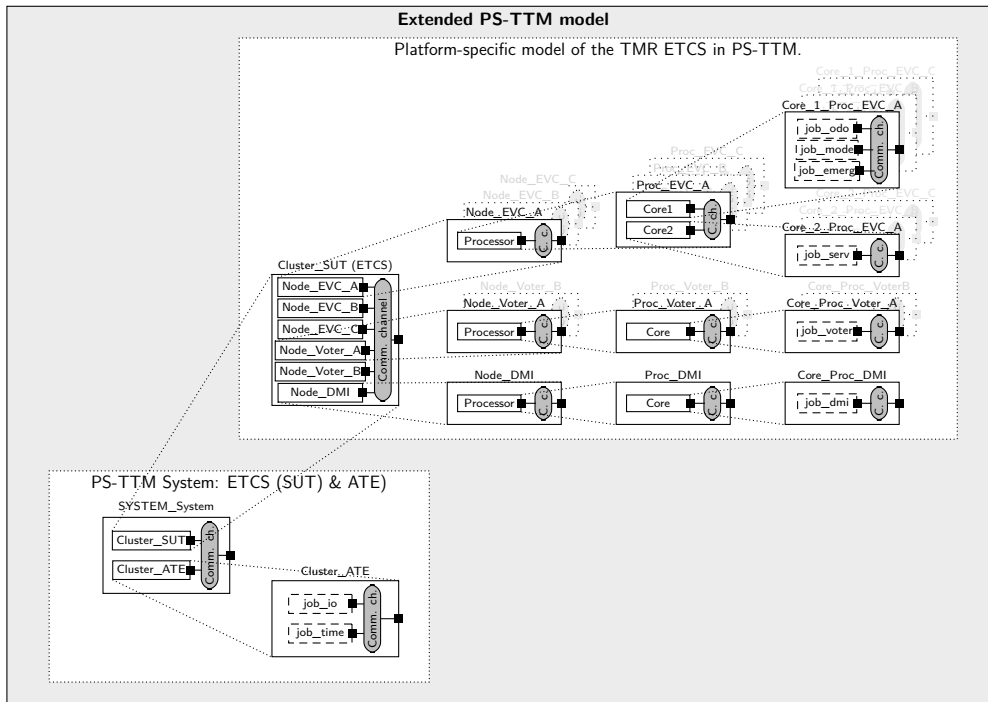


Figure 8.7: PS-TTM system model for ATE-controlled fault injection simulation of ETCS

the faults injected in the system during the simulations were detected by the voters. Since we configured all jobs in the system with a period of one macrotick ($250msec$), *corruption* and *babbling* faults were detected $250msec$ after their injection in the system, as expected. *Bit-flips* in signals and *open circuits* were also detected in the next macrotick.

However, *no-execution* and *out-of-time* faults, injected in the 1st, 3rd and 5th fault configurations, took longer to detect. This happened because, due to the state of the system at the moment of the injection, the faults were dormant. In fact, *no-execution* and *out-of-time* faults do not get active until the value of the signal changes, since they do not cause an alteration of the signal values by themselves.

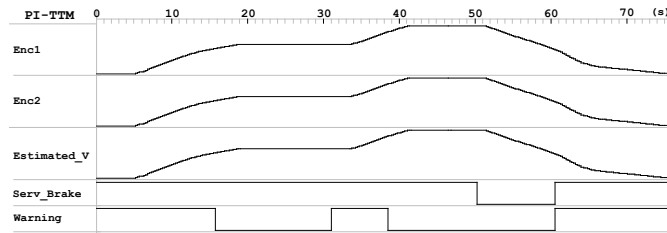
All in all, the faults were detected as expected. The voters also notified a multiple failure caused by the two faults injected in the system during the 5th simulation. That occurred at instant $150.25sec$, and also caused the activation of the emergency brakes of the train, as stated by the requirements. Figure 8.9 shows an extract of the results of the 5th simulation. For the sake of simplicity, we omit input signals of emergency and service brakes from the figure.

Listing 8.3: Fault configuration #5 injected by the PS-TTM ATE on the ETCS PSM

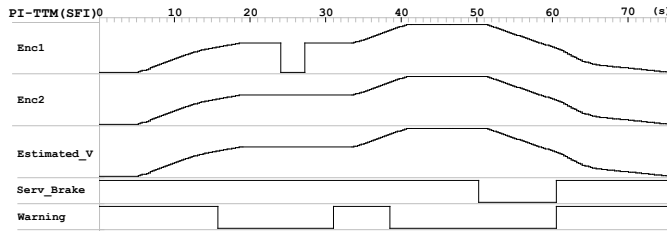
```

1 <pim_fault_metamodel:FaultInjectionSettings>
2   <!-- FAULT CONFIGURATION -->
3   <FaultConfiguration>
4     <FaultSet>
5       <Name>FC5A</Name>
6       <FaultMode>PERMANENT</FaultMode>
7       <Duration></Duration>
8       <TriggerInstant>60.0</TriggerInstant>
9       <Fault ref="_p5LWtRbKEe0w28kCcXd1kQ"/>
10    </FaultSet>
11    <FaultSet>
12      <Name>FC5B</Name>
13      <FaultMode>PERMANENT</FaultMode>
14      <Duration></Duration>
15      <TriggerInstant>150.0</TriggerInstant>
16      <Fault ref="_p5LWtRbKEe0w28kCcXd1kR"/>
17    </FaultSet>
18  </FaultConfiguration>
19
20  <!-- FAULTS -->
21  <Faults>
22    <Fault id="_p5LWtRbKEe0w28kCcXd1kQ">
23      <Name>b5a</Name>
24      <Location ref="_LeWBYBbJEe0EgpzBm6gXmA"/>
25      <FaultEffect>HW_NoExecution</FaultEffect>
26    </Fault>
27    <Fault id="_p5LWtRbKEe0w28kCcXd1kR">
28      <Name>b5b</Name>
29      <Location ref="_LeWBYBbJEe0EgpzBm6gXmC"/>
30      <FaultEffect>HW_Corruption</FaultEffect>
31    </Fault>
32  </Faults>
33
34  <!-- FAULT LOCATIONS -->
35  <Locations>
36    <Location id="_LeWBYBbJEe0EgpzBm6gXmA">
37      <Job>system_inst.cluster_sut_inst.node_evc_B</Job>
38      <PortType></PortType>
39      <Entity></Entity>
40    </Location>
41    <Location id="_LeWBYBbJEe0EgpzBm6gXmC">
42      <Job>system_inst.cluster_sut_inst.node_evc_C.proc_evc</Job>
43      <PortType></PortType>
44      <Entity></Entity>
45    </Location>
46  </Locations>
47 </pim_fault_metamodel:FaultInjectionSettings>

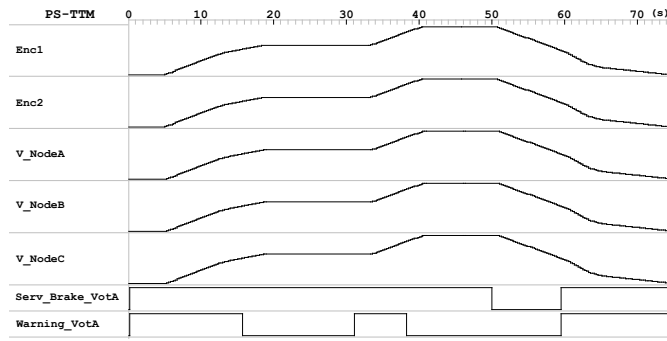
```



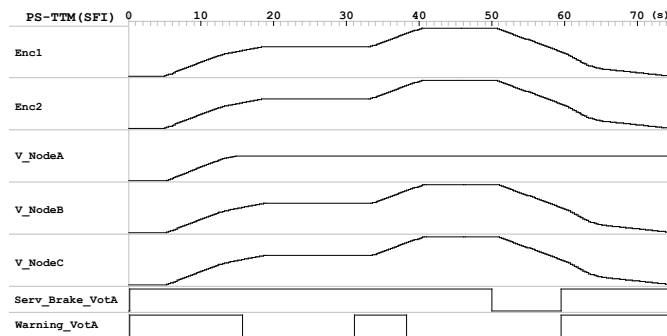
(a) PI-TTM test-case results



(b) PI-TTM test-case results with SFI in encoder 1.



(c) PS-TTM test-case results.



(d) PS-TTM test-case results with SFI on the core 1 of Node A.

Figure 8.8: Simulation results of the PS-TTM model
(source: [ANP+14b])

Table 8.3: Fault-injection campaign for the PSM model of the ETCS.
(source: [ANP+14d])

#	Fault Location	Fault		Fault Set			Description
		Effect	Attributes	Mode	Trig. time(s)	Duration (s)	
1	Node_EVC_A	NE	-	p	85.0	-	Node A stopped working
2	Proc_EVC_B	C	-	p	20.0	-	Processor B provides incorrect results
3	EVC_C_Core1	OoT	0.50	p	120.0	-	Core 1 of a processor C is out of time bounds
4	Node_EVC_A	B	-	t	40.0	25.0	Node A babbling, incorrect results
5	Node_EVC_B, Proc_EVC_C	NE, C	-	p, p	60.0, 150.0	-	Double failure (Node B stops, then processor C incorrect)
6	job_serv_A <i>serv</i> output	B_I	-	t	160.0	0.50	Bit-flip in Service. Brake signal sent by node A
7	job_emrg_C <i>emrg</i> output	B_OC	-	t	105.0	15.0	Emerg. Brake not received from Node C (noise)

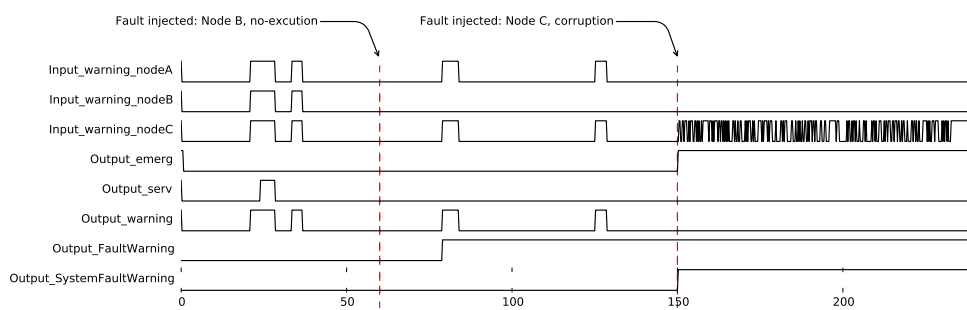


Figure 8.9: Results of the PSM simulation with fault configuration #5
(source: [ANP+14d])

Table 8.4: Results of fault-injection campaigns in the PSM model of the ETCS.
(source: [ANP+14d])

#	Fault trigger instant (s)	Fault warning activ. instant (s)	System fault activ. instant (s)
1	85.0	125.25	-
2	20.0	20.25	-
3	120.0	125.25	-
4	40.0	40.25	-
5	60.0, 150.0	79.00	150.25
6	160.0	160.25	-
7	105.0	105.25	-

8.1.5 Discussion

This case study uses our Time-Triggered (TT) modelling framework for safety-critical embedded systems. The modelling workflow follows the Y-chart paradigm over the Logical Execution Time (LET) and TT models of computation (MoC). We introduced the Platform Independent Time-Triggered Model (PI-TTM), which is a LET simulation engine built as an extension to the Executable Time-Triggered Model (E-TTM) that enables the simulation of platform-independent LET-models in SystemC. In addition, the Platform-Specific Time-Triggered Model (PS-TTM) extension enables the design of platform specific E-TTM models including abstractions of the HW components.

The deployment of LET and E-TTM based models into time-triggered architecture platforms becomes straightforward. Moreover, the fact of relying on time-triggered models of computation reduces the number of system failures that need to be considered since failures due to faulty synchronization or altered orders of execution are prevented by construction. Besides, the fact of implementing the LET Model of Computation (MoC) as an extension to the E-TTM engine enables merging subsystems described in both models of computation. This enables the simulation of mixed-abstraction level subsystems with fault injection, seamlessly combining descriptions of subsystems at PIM and PSM design stages.

Finally, the framework integrates a time-triggered ATE that enables non-intrusive simulated fault-injection (SFI) in the PI-TTM and PS-TTM models. The ATE is synchronized with the SUT to enable reproducibility in the tests and SFI experiments. The selected SFI configuration scheme is compliant with the international ASAM AE hardware-in-the-loop (HiL) standard, which would ease the forward reuse of the SFI campaigns in all development phases up to the deployment of the final Time-Triggered Architecture (TTA) prototypes.

Table 8.5: Example PS-TTM simulator performance for PIM and PSM models.

(source: [Aye15])

Simulator configuration	Simulation time	Computation time (min,max)	Acceleration factor
PI-TTM	240s	[1.01s, 1.18s]	x200
PI-TTM	240s	[6.19s, 6.72s]	x40

Results running the PS-TTM simulator on a Windows 7 SP1 host (in x86/32-bit mode) with an Intel i7 quad-core processor 2.60 GHz. PS-TTM applications built on SystemC version 2.3.0 and Python 3.2.

Limitations

Table 8.5 compares the simulated time to the execution time for both the PIM and PSM. The test scenario consisted of a train journey between two stations, covering 7.5km in 240s. The time acceleration is about $x200$ for the PIM and $x35$ for the PSM.

The computing time varies with the model complexity, as well as the activated fault injectors or signal probes. Table 8.6 summarizes the number of modelling artefacts and scenarios considered for the analysis at the PIM and PSM abstraction levels. It should be noted that these figures correspond to a Transaction-Level Modeling (TLM) abstraction, and that a more precise Register-Transfer Level (RTL) simulation would increase the computation time considerably, which eventually would yield long, impractical simulations.

Table 8.6: Comparison between the PIM and PSM PS-TTM reliability analysis.

Item	(Type)	PIM	PSM	Notes
		#	#	
Modeling elements	<i>Jobs</i>	5	15	PSM has triple redundancy in EVCs (with 4 <i>jobs</i> deployed on each node), and 2 Voters (with 1 <i>job</i> per node).
	<i>DASes</i>	6	-	Used in PIM only.
	<i>Nodes</i>	-	6	Used in PSM only.
	<i>Procesors</i>	-	6	Used in PSM only.
	<i>Cores</i>	-	9	Used in PSM only.
Failure modes	<i>Maskable</i>	36	261	
	<i>Non-maskable</i>		21	Due to systematic errors in SW.
	<i>Any (total)</i>	109	282	
Potential error causes		276	608	

8.1.6 Conclusion

This section presented the application of the testing and simulated fault-injection framework for Time-Triggered (TT) dependable-systems based on the PS-TTM approach. The environment enables testing and injecting faults at different stages of the design, from Platform-Independent Models (PIMs) to Platform-Specific Models (PSMs), which enables an early detection of design flaws in the system.

The Automatic Test Executor (ATE) used to assess the fault tolerance in this case study is composed by three different modules for the design and simulation of test cases, injection of faults during simulation and storage of simulation results for the evaluation of the behaviour of the system under such faults. The ATE is synchronized with the simulation time of the System Under Test (SUT), such that functional tests and fault injection experiments become reproducible.

The SFI technique is non-intrusive [Aye15], i.e., engineers can setup different fault configurations to carry out different fault tolerance simulation tests cases using the same system model: the PS-TTM simulator exploits SystemC introspection to instantiate the fault injectors at run-time, so that no system-model modifications are required. This is achieved by monitoring the signals of the SUT and modifying their values if required. The framework provides the user with a library of faults to configure the fault-injection experiments. The ATE imports these configurations for carrying out the simulations. The mapping of LET and E-TTM-based models to time-triggered architectures is straightforward. This eventually facilitates the re-usability of tests even on real prototypes, provided that we could build a test harness with equivalent real-world Fault-Injection Units (FIUs).

The PS-TTM ATE framework presented in Chapter 5 is evaluated using a railway signalling system case study. We modelled the system at both PIM and PSM levels, and checked the behaviour of the system under different faults by means of the simulated fault-injection (SFI) capabilities provided by the framework. Our non-intrusive SFI approach enabled an early assessment of a fault-tolerant odometry algorithm long before assembling a costly system prototype, and eased identifying its main weaknesses. We also evaluated the behaviour of the voters introduced in the Triple Modular Redundancy (TMR) system by means of our framework. The response of the voters under the presence of faults was considered successful. This case study evidences the suitability of the framework for simulated fault injection in TT safety-critical systems modelled with the PS-TTM platform at different stages of the development.

Related Publications

The results presented herein also appeared in [ANP+14a, ANP+14b, ANP+14c, ANP+14d].

8.2 Model and Code Re-use for a HiL Elevator Simulator

This case study evaluates the practicality of re-using models and actual production code of devices to build test components and special-purpose instruments that could be integrated in real-time testbenches for the validation of distributed safety control systems.

8.2.1 Context

The system under consideration is an automated testbench for the verification of distributed elevator-control systems. Elevator systems can adopt multiple configurations, depending on factors like the building, the sensor system, the human-machine interfaces or the drive technology, amongst others. In order to avoid a proliferation of variants, a manufacturer of elevator controllers could opt for an *universal product concept*, i.e., a highly configurable control system that could be parametrized to fit any possible elevator installation. Such configurability is achieved by embedding all the possibly required control functions in the controller application. The immediate consequence is a complex control flow in the controller software, with many optional routes that become active depending on the configuration.

In this scenario, the verification of the elevator controllers on real test elevators provides a too limited set of feasible configurations. Further, such a verification on real elevators becomes impractical for a thorough verification. However, the product maintenance for the elevator controller depends on a comprehensive regression testing, as modifications in the system software may have side-effects on the functionality for some configurations. As a consequence, it is required an equivalent test system that could be easily reconfigured and scaled-up to replicate different buildings, elevator sensors and elevator actuators: the Hardware-in-the-Loop Elevator Simulator (**HiLES**).

The **HiLES** testbench integrates a programmable simulator to replicate the mechanical behaviour of physical parts (e.g., the elevator car, the car doors, etc.), while providing the same signal formats as the actual elevator sensors (e.g., position limit switches, encoders, elevator user interfaces, etc.) and actuators (e.g., drive, safety-circuit, etc.). The **HiLES** simulator is a real-time hardware-in-the-loop (**HiL**) testbench that can reproduce several elevator installations by switching to a different model configuration. **HiLES** connects to a mock-up of an elevator-control system for elevator groups. The elevator-group controller is a distributed system consisting of multiple networked embedded devices, physically scattered along the shaft, e.g., the level-call pushbuttons or the human-machine interface (**HMI**) aboard the cabin.

The original **HiLES** incorporated several remote Input/Output (**I/O**) nodes from the elevator-control system, which have to be manually installed or removed/disabled. Besides the inconvenience of requiring more physical room for the elevator testbench, this drawback eventually constrains the simulator versatility and increases the cost of the test campaigns, as it limits the feasible test automation.

Another threat to scalability in the original **HiLES** test system came from the sequential

execution of the simulator models: the computation time depends on both, the model configuration, and the capabilities of HW/SW from the simulator controller platform. Pushed to the limits, the exploitation of every available computing resource to run higher complexity models demanded very platform-specific optimizations, which on the other hand, degraded the simulator portability to alternative execution contexts. The complexity of all the possible interactions made the feasibility of simulator improvements unpredictable, which caused maintenance problems.

To overcome the above-mentioned limitations, an alternative deployment was selected: instead of conventional processors, the new HiLES would use a Commercial-Off-The-Shelf (COTS) heterogeneous controller providing Programmable Logic (PL) resources, i.e., a Field-Programmable Gate Array (FPGA). Considering that the original HiLES was programmed in the C/C++ programming languages, that the coding tweaks applied over time made the source code almost unusable for other platforms, and last, but not least, that the deployment of simulator components on the FPGA require coding of those parts in hardware description language (HDL), it was decided that the new HiLES should be built from scratch, following a Model-Based Development (MBD) process.

The library of models for HiLES would be developed in the MATLAB/Simulink [ML,SL] COTS development environment. By using the transformations to C (Simulink Coder) or HDL (HDL Coder), alternative implementations could be obtained for either the conventional processor or the FPGA parts of the simulator platform. It is intended to apply the techniques presented in Chapter 6 to enhance the HiLES test system in several ways:

- Provide a strict timing synchronization in all the simulated position and speed sensors.
- Integrate virtual position sensors with fault injectors to support fault testing.
- Integrate virtual position sensors with communication interfaces.
- Provide scalability in the remote I/O nodes, by replacing the physical embedded devices with virtual replicas that could be instantiated on-demand.

8.2.2 System Description

The subject of this case study is the real-time verification of an Elevator Control System (ECS) with a federated architecture, i.e., a distributed embedded system with spatially scattered and networked nodes, each node having local computing resources. The main node in the ECS is the *elevator controller*.

The *elevator controller* is an embedded device implementing motion control, diagnostics, monitoring and safety functions. Among other tasks, an elevator controller processes and queues the calls from the users, controls the motion of the cabin and the elevator doors, and guards against the unsafe operation of the system. An elevator controller may operate

as part of an *elevator group*, where a dispatching algorithm optimizes and coordinates the allocation of elevators to serve the user calls. The controller application can be set up to operate with any possible elevator configuration: for each hoist system (hydraulic, AC or variable speed drive), arrangement of sensors (position switches, incremental or absolute position encoders) and layout of user interfaces (conventional button call boxes and/or destination floor selectors).

In modern elevator systems the users may call an elevator through a variety of **HMI**s, ranging from conventional pushbuttons and simple displays, to advanced kiosks or multi-functional touch displays. For the sake of simplicity, in this case study we consider an elevator system with a conventional **HMI**, i.e., level and car pushbuttons. In addition, an elevator-control system would integrate special-purpose **HMI**s for maintenance and rescue, that shall be operated in a safe manner by qualified staff. A safety circuit prevents the elevator from moving in unsafe conditions. For instance, the elevator doors shall always be closed before the cabin starts moving, in order to avoid trapping the passengers against the walls or leaving people in risk of falling through the well. To avoid this, an interlock device connected to a safety circuit disables the elevator drive until the doors are properly closed. For safety reasons, the safety circuit is usually redundant with the elevator controller, i.e., the safety circuit operates independently from the controller, although the controller can monitor the state of the circuit for diagnosis and maintenance purposes.

Figure 8.10 sketches a simplified control system for an elevator group. The group consists of a number of Elevator Controllers communicating through a dedicated Controller Area Network (**CAN**) bus ('**CANH BUS**'). Each controller has a secondary **CAN** bus ('**CANV BUS**') connecting it with the distributed slave nodes, i.e., car/level **HMI**, door control interfaces ('**EXT3**'), and safety-monitoring devices ('**EXT4**'). The controller commands the drive to move/stop the cabin, read a set of position switches installed along the shaft to update the current position estimate and determine the speed setup.

As mentioned in §8.2.1, the elevator-positioning system varies depending on the drive technology, e.g., hydraulic or electric, as well as the sensing system, e.g., limit switches or encoders, etc. The development of new position sensors for elevators enables further product improvements and yields benefits. In particular, absolute encoders are easier to install and calibrate, minimize the elevator start-up time (they do not require a homing movement after powering-on or resetting the elevator controller) and may even replace several safety elements with a single safety-encoder device, thus simplifying the safety circuit and easing the diagnostic of system malfunctions. However, an elevator-controller concept based on the universal-product approach may require a major overhauling of the control functions and interfaces to use the new device. Moreover, the novelty of many of these components may delay the integration in the elevator-control system till pre-series sensors become available. Hence, the availability of a **HiL**-enabled virtual sensor replacement could boost the development of an enhanced elevator-control strategy, by supporting a concurrent engineering process.

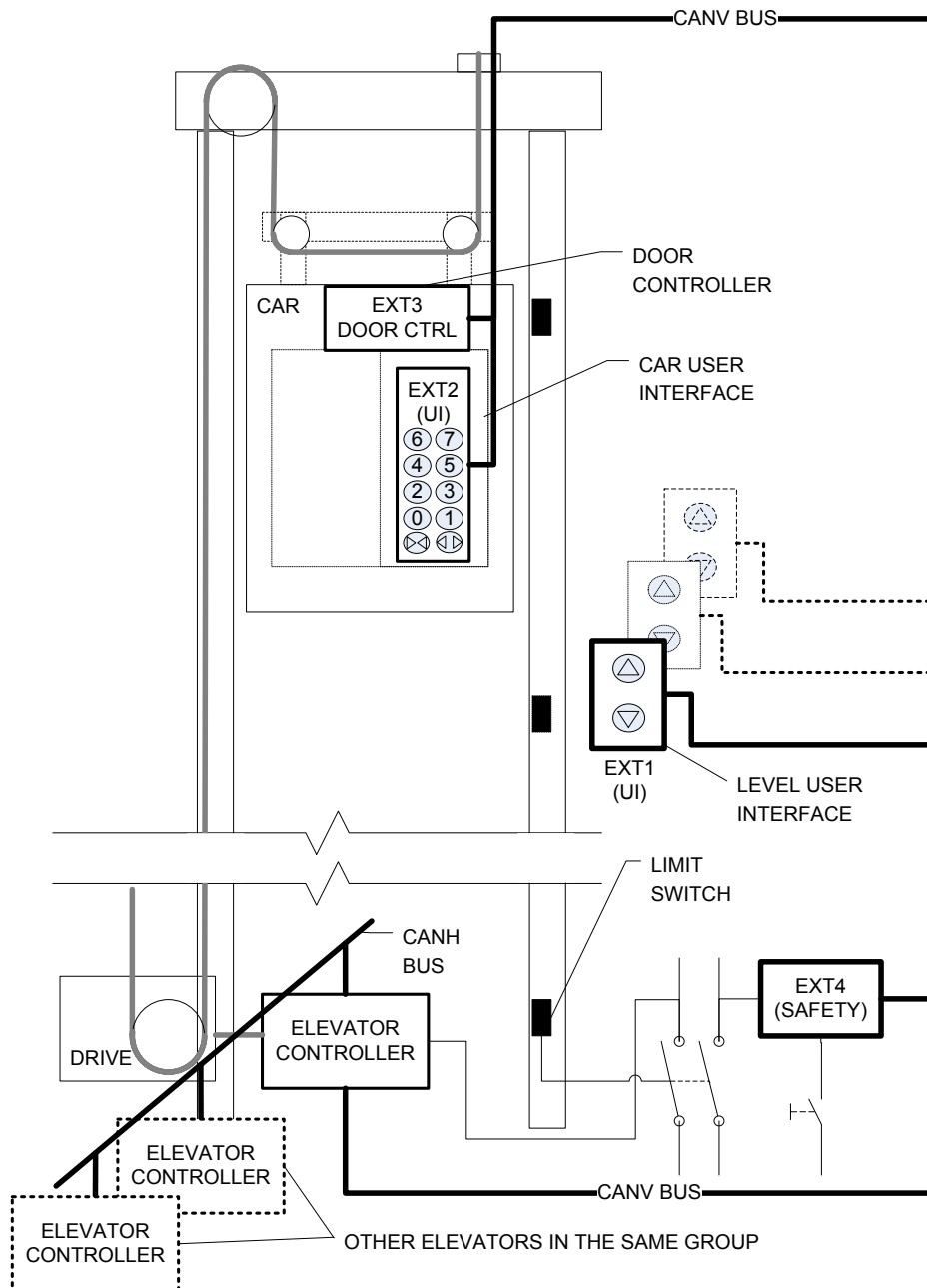


Figure 8.10: Schematic view of the elevator group distributed control system, integrating multiple controllers and networked remote I/O devices

(source: [NAP+17])

Distributed control systems for elevator groups

Our case study consists of a simulator for testing distributed control systems for elevator groups. The plant consists of 1 to 8 elevators ($1 \leq N^{elev} \leq 8$), with up to 3 doors per elevator ($1 \leq N_i^{door} \leq 3$), and up to 64 levels ($1 \leq N_i^{level} \leq 64$). The distributed control system comprises N^{elev} elevator controllers interconnected through a CAN bus, called the *horizontal bus* (CANH), as shown in Figure 8.10. When operating as a group, a dispatcher handles the traffic and allocates the user's calls to each controller.

Many peripheral elements of the elevator-control system are networked I/O embedded devices, named *Extensions* (EXT). Each controller handles a local CAN bus, termed the *vertical bus* (CANV), that links I/O Extensions of various sub-types: the N_i^{level} user interfaces at the levels (EXT1) and inside the elevator car (EXT2), the N_i^{door} door controllers (EXT3) and special-purpose sensors and devices (EXT4/EXT5) located in the shaft (see Table 8.7). EXT3 and EXT4 devices also monitor some safety-circuit contacts in order to diagnose eventual elevator malfunctions. The number of nodes on a CANV bus depends on building parameters like the number of levels. By design, most remote I/O-device types implement similar functionality while differing in the signal interfaces. These I/O-device variants integrate a common micro-controller architecture and share the application code to simplify the maintenance and evolution of the product line.

The expected product lifetime of the elevator system is about 20 years, during which the elevator manufacturer has to supply replacements for the electronic devices. A way to avoid stocks of obsolescent spare parts is to require the evolving embedded product lines to be retro-compatible, i.e., when updating the embedded product line, the manufacturer has to preserve the interoperability with former elevator controllers. To that end, a comprehensive regression testing phase follows each development iteration, to verify the behaviour of the system for each control configuration. When the control system evolves to a new product generation, new functional and interface specifications extend the application code for the I/O devices, while required to be retro-compatible with precedent product families.

Table 8.7: Remote I/O EXT device types.

Type	Purpose	Max. devices	NMT Cycle
EXT1	Level user interface.	64	3 s
EXT2	Elevator car user interface.	16	3 s
EXT3	Elevator door interface.	3	3 s
EXT4.0	Safety-related I/O.	1	200 ms
EXT4.1	Safety-related I/O.	1	3 s
EXT5	General purpose I/O.	64	3 s

Elevator Position Control System

An elevator controller operates the cabin movements by reading a set of sensor measurements and commanding the speed setpoint on the main drive. The conventional position control mode is the creep-to-floor travel, where the elevator drive operates at two speed values: at start the drive accelerates to the high-speed value. Then it keeps uniform motion until the controller detects a predefined position switch. When the latter happens, the controller commands the drive to the low-speed value (*creep speed*) and waits for activation of a second position switch (car levelling at landing position). When found, the controller triggers a timed stop command and brakes the cabin.

A typical arrangement of sensors combines encoders either incremental or absolute and position switches attached to the elevator frame. As incremental encoder measurements introduce uncertainty about the actual position of the car, the elevator controller corrects the eventual drifts with the absolute references provided by the position switches. The main drawback of incremental encoders is that the controller requires a homing move to find a reference position at power-on, lengthening the elevator start-up time, specially for high-rise elevators. Moreover, due to the uncertainty about the actual position and speed, the controller shall select at a conservative sub-optimum speed set-point to prevent overshooting the desired landing position.

Absolute encoders measure the elevator car position with regard to a fixed position reference, so that the controller skips the homing move. This shortens the start-up time and enables the optimisation of the speed set-point, increasing the overall availability and capacity of the vertical transportation system.

CANopen Absolute Encoders for Elevators

CAN networks suit the low-cost and performance requirements for implementing distributed elevator-control systems. Several elevator manufacturers adopted CAN solutions, and encoder vendors developed devices with CAN interfaces [Mat13,LIMAX], offering alternative protocol implementations: custom, industry standard (e.g., CANopen CiA 406 [CiA406]), or lift standard (i.e.,CANopen CiA 417 [CiA417]). Our work focuses on building a functional model of an ELGO LIMAX 02 absolute encoder with a CAN interface and CANopen CiA 406 protocol. Figure 8.11 shows an elevator-control system integrating an ELGO LIMAX device, where the encoder is a CANopen communication slave and the elevator controller is a CANopen master.

According to the manufacturer's specifications, an ELGO LIMAX 02 CiA 406 encoder operating in normal mode synchronously publishes the position and speed data by sending a Process Data Object (TPDO1) cyclically. The basic node operation complies with the common CiA 301 [CiA301] services. The CANopen master can set the TPDO transmission cycle by sending a Service Data Object (SDO) message. The encoder also emits a periodic Heartbeat (CANopen message service) (HB) message, triggered by a programmable timer.

Other functions include saving parameters or resetting to factory configuration. The encoder implements Layer Setting Services (*LSS*) to setup configuration parameters like the node-ID and bit rate. *LSS* configuration requires a point-to-point connection to the *CANopen* master and a reset of the sensor to become effective, and is typically carried out at commissioning. For our purpose, we assumed that the sensor is pre-configured at start-up and that the elevator controller will only overwrite the TPDO1 and HB transmission cycle times.

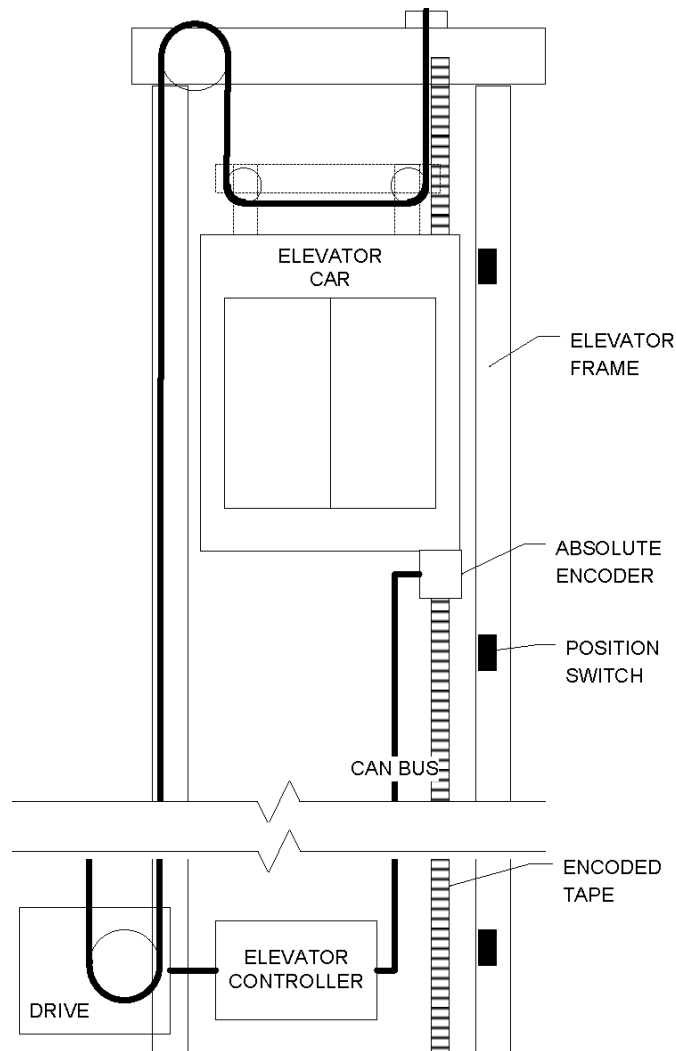


Figure 8.11: Elevator controller with absolute encoder and *CAN* link
(source: [NAM+16])

Development of the HiL Elevator Simulator (HiLES)

The introduction of elevator controllers based on programmable custom electronics raised the need for versatile test systems. Although the elevator controllers are eventually validated on special test facilities (e.g., test towers), the developers of the embedded applications have to debug the software thoroughly beforehand on a system mock-up.

The alternative to the test tower is the lower-cost real-time Hardware-in-the-Loop Elevator Simulator (HiLES). Since the development of the first programmable elevator controller, a real-time HiL elevator simulator was also concurrently developed and maintained. The real-time HiL-elevator simulator is intended for the verification of the elevator controllers, with different elevator configurations and under many operational conditions (e.g., simulated faults). The function of HiLES is to compute the kinematics of the elevator car and the car doors, then replicate the electronic interfaces from the position sensors, e.g., limit switches and encoders. The test operator can setup HiLES to reproduce multiple elevator variants by plugging in HW elements as required by the installation, loading the SW components and switching the building-parameter databases.

The HiLES testbench also has a number of I/O embedded devices connected to the CANV buses: EXT1 and EXT2 user interfaces connect to the elevator controller, and EXT3 and EXT4 nodes monitor the safety circuit. For the latter, the simulator outputs related to safety protection route through redundant signal paths, e.g., limit switches located at the bottom and top of the elevator shaft signal the arrival of the elevator car at a potentially dangerous height: when the elevator controller detects these signals, it slows down the drive and eventually brakes, landing the cabin at the lower/uppermost level. In the event of a delayed reaction, a second limit switch connected in series in the safety circuit would trigger an emergency stop. HiLES updates these signals according to the simulated car position and feeds them to the controller. An external HW interface replicates the signals to act simultaneously on the safety circuit and the inputs of the monitoring devices.

8.2.3 Problem Statement

A feature of the HiLES simulator is the emulation of incremental (pulse) encoder signals. Our first simulator was built on a computer with a customized real-time operating system. Over time, demanding requirements and the obsolescence of the platform forced a first migration to an up-to-date 'soft' real-time configuration. At that time the key requirement was to reduce the cost of the computing platform, although this constrained the achievable simulation performance with regard to the incremental encoder simulation. In the meantime, improvements in elevator machinery attained higher speeds. For the HiL elevator simulator, simulating taller buildings required lowering the computing cycle, in order to keep the temporal coherence of the position information fed back to the elevator controller. Considering the past experience with former simulators, the priority requirement was that the new simulator should simulate an incremental encoder with the maximum

resolution at a maximum car speed of $6m/s$. This proved unattainable with the preceding HiL architecture. As a consequence, the manufacturer requested a new simulator in 2012.

On the other hand, while using actual embedded devices from the elevator-control, the system provides a realistic communication behaviour in HiLES, this also limits the feasible test automation: EXT1 and EXT2 devices require manual actions from the test operator to stimulate/observe the control system, the reconfiguration of the testbench to simulate different buildings requires physically plugging and removing the devices and the simulation of abnormal operation scenarios require additional fault-injection hardware.

8.2.4 Heterogeneous COTS Platform for HiL Elevator Simulator

Our development targeted two different simulator controller platforms: the NI cRIO-9082 and the NI cRIO-9039 (see Figure 8.12). These controllers integrate different FPGAs: Xilinx Spartan-6 for NI cRIO-9082 or Xilinx Kintex-7 for NI cRIO-9039.

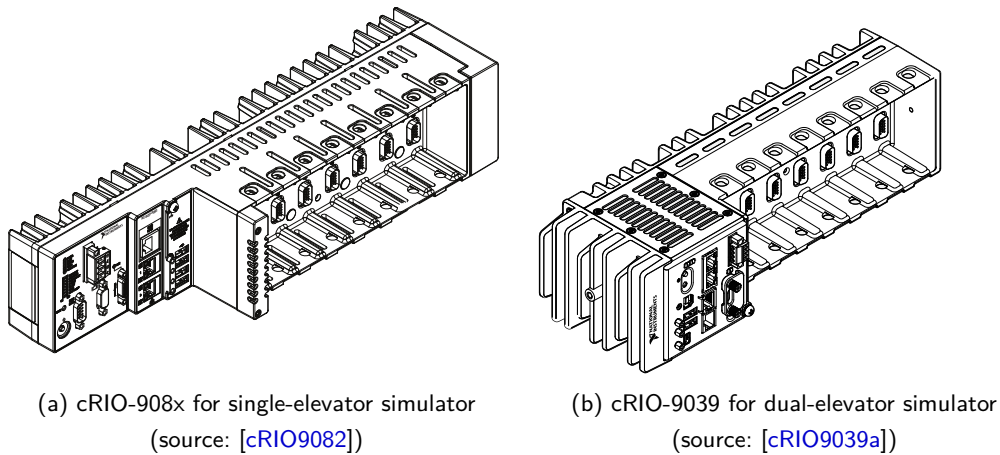


Figure 8.12: NI cRIO heterogeneous controllers to realize the HiLES simulators

Single-elevator simulator using NI cRIO-9082

We initially developed the HiL elevator simulator targeting the cRIO-9082 model [cRIO9082], that integrates a dual-core Intel i7@1.33 Ghz processor and a Xilinx Spartan-6 LX150 FPGA. The LX150 FPGA sufficed to deploy a single-elevator simulator with an absolute encoder.

Dual-elevator simulator using NI cRIO-9039

In mid-2015, National Instruments marketed the cRIO-9039 variant [cRIO9039b], featuring a quad-core Intel Atom E3845@1.91 GHz processor and Xilinx Kintex-7 7K325T FPGA. As

shown in Table 8.8, the cRIO-9039 FPGA doubles the resources available in the original cRIO-9082, enabling the deployment of a dual-elevator simulator in a single cRIO-9039 controller. A key portability issue is that the maximum clock divider is different, getting a minimum frequency of 2.5MHz for the cRIO-9082 and 5MHz for the cRIO-9039. As a consequence, the time constraint for the critical path in the cRIO-9039 is half than that for the cRIO-9082.

In 7 series FPGAs the clock management tile (CMT) includes a mixed-mode clock manager (MMCM). MMCM has some restrictions that we must be adhered to. In general, the major limitations are the VCO operation range, input frequency, programmable duty cycle, and phase shift. The minimum and maximum VCO operating frequencies are defined in the electrical specification of the 7 Series FPGA Data Sheets [Xil15]; it is possible to set it between 600MHz and 1200MHz. With this minimum VCO frequency, when the MMCM is configured in cascade mode, the CLKOUT6 divider can be cascaded with the CLKOUT4 divider. This provides a capability to have an output divider that is larger than 128. CLKOUT6 feeds the input of the CLKOUT4 divider creating a minimum output frequency for this pin of 36kHz.

When using the NI software, this last divider cannot be used, imposing a minimum operating frequency of $36\text{kHz} \times 128 = 4.6\text{MHz}$. This problem was not observed when using the cRIO-9082 platform (6 series FPGAs) because the clock management uses Digital Clock Managers (DCMs) and the tile architecture differs from that used in 7 series FPGAs.

CAN implementation in cRIO-90xx

In order to achieve time determinism and coherence in the generation of position information, all the encoder simulators would be deployed on the FPGA subsystem in the cRIO controller. Thus, we selected the NI 9853 C-series modules, with 2-port high-speed CAN [NI9853] and directly controllable from the FPGA. However, to allocate the CAN Restbus simulator to the cRIO processor, we required a CAN interface accessible from the LabVIEW RT execution context, for which we considered two alternatives: (a) share the NI 9853 module already used to simulate the CANopen absolute position encoder or (b) an NI cRIO Real-time Operating System (RTOS)-supported CAN interface.

The C modules in a cRIO can be controlled from the FPGA, the processor or operate in hybrid mode, the latter implementing the shared access in the FPGA. The HiLES simulator

Table 8.8: NI cRIO FPGA specifications.

Device	FPGA Type	Number of Flip-Flops	Number of 6-input LUTs
cRIO-9082	Xilinx Spartan-6 LX150	184,304	92,152
cRIO-9039	Xilinx Kintex-7 7K325T	407,600	203,800

was set to **FPGA-control** to generate encoder signals, so we tried the hybrid mode. After several tests, this mode showed up to be troublesome, and we were unable to get it going. Hence, we finally preferred to integrate an external Peak PCAN interface connected to the Universal Serial Bus (**USB**) port of the cRIO 9039 controllers, featuring an NI Real-time Linux **RTOS**¹. This enabled the installation of the Linux `pcan` driver, although the **USB** to **CAN** interface is not supported in real-time mode.

Integration

Figure 8.13 shows a simplified sketch of the single-elevator with absolute encoder **HiL** test system, deployed on the cRIO-9082 controller. This illustrates the real-time composability achieved by Programmable Logic (**PL**): there are two groups of IPs generated from Simulink models which correspond to the elevator and encoder models. These are computed side-by-side, resembling the parallel behaviour of the real-world.

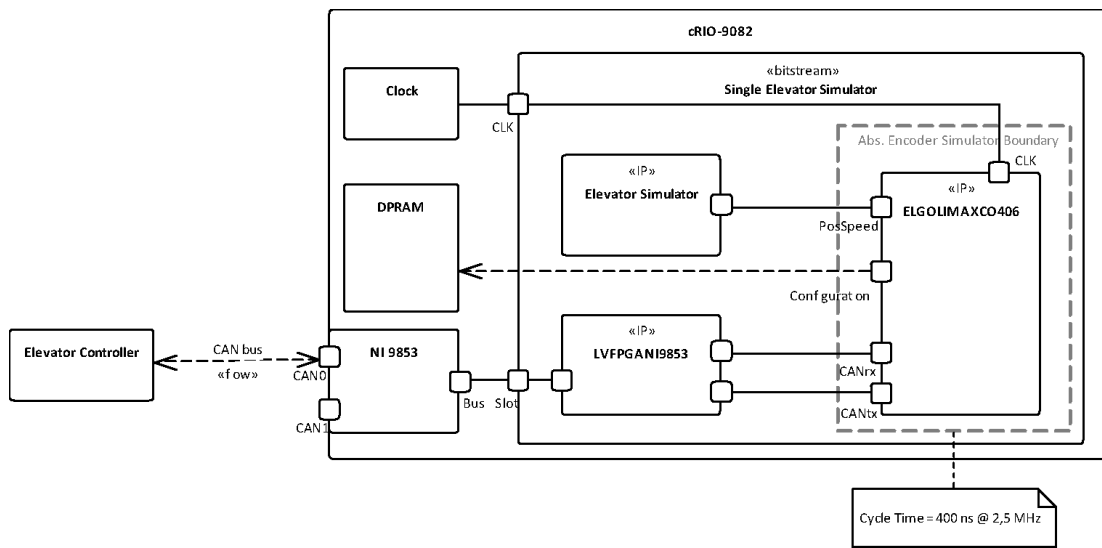


Figure 8.13: Integration of the encoder simulator in the cRIO-9082 controller
(source: [NAM+16])

Figure 8.14 illustrates how the LabVIEW/FPGA VI links the **HDL**-coded model with the IP libraries to control the hardware resources. The encoder simulator runs inside a LabVIEW timed loop, triggered by the **FPGA** clock. We designed two separate queues to decouple the synchronous execution of the elevator models from the event-triggered processing of incoming and outgoing messages. Each queue has a separate while-loop that reads from or writes to the **CAN** port from the NI 9853 interface.

¹The cRIO-9082 has a different **RTOS**, which a much more limited set of supported devices.

Issues with cRIO-9039 dual-elevator simulator

The cRIO-9039 **FPGA** doubles the resources allocated to a single-elevator simulator in a cRIO-9082, therefore our first attempt to get a dual-elevator simulator was to replicate the IP Integration Node in the LabVIEW/FPGA VI. We also duplicated the required dual-port RAM resources, communication queues and configuration parameters, and we allocated the secondary **CAN** port in the NI 9853 to the new simulator instance. During the development of the dual-elevator version, we found the issues enumerated below:

1. *Failure of LabVIEW/FPGA compilation:* When dealing with the **FPGA** project in the National Instruments 32-bit LabVIEW/FPGA environment (release 2015DS2) we found that the system was not able to successfully process the project files and some unexpected errors showed up when synthesizing the **FPGA** code. The problem arose when the NI software tried to create a project including the VHDL wrappers automatically generated by the environment. It seemed as if the LVFPGA environment was unable to cope with the resulting project size, which resulted in excessive memory use for the **HDL** compilation, resulting in a software failure. After reporting this problem to NI, we tried another approach, splitting the original design in several VIs that realize the same functionality but use a different project architecture. With this turnaround we got the whole **FPGA** design synthesized.
2. *Unexpected IP behaviour:* As both elevator simulators run synchronously in the **FPGA**, we first tried to implement the dual-encoder simulator using both **CAN** interfaces from a single NI 9853 module. Although the LVFPGA library provides IP blocks to control these ports separately, we did not manage to get both **CAN** ports working, without knowing the reasons. To prevent this issue, we chose to install 2 NI 9853 modules per cRIO-9039 controller, allocating the first **CAN** port from each module to an encoder simulator instance.

8.2.5 The CAN Restbus Simulator

A second objective of the Elevator-Simulator case study is to reduce the verification cost of the distributed elevator-control systems, through embedded code re-use to co-simulated networked devices in the **HiLES** simulator. This will demonstrate the feasibility and limitations of including actual production code from embedded devices as part of the **COTS HiL** test system, providing a rich-behaviour replacement of the original nodes for real-time testing.

The target for deployment of the CAN Restbus Simulator is the cRIO-9039 **HiLES**, featuring the NI Real-time Linux **RTOS**, which offers better compatibility with third-party drivers than the cRIO-9082. In the cRIO-9039 slots we plugged C-series modules to interface with the elevator-control system, e.g., *digital input/output* for limit switches, command inputs and incremental encoders, or **CAN** interfaces as those required for the **FPGA** absolute

position encoder simulator (see §8.2.4). Additionally, serial ports simulate the main drive interface and an Ethernet port connects the cRIO-9039 simulator to a remote test control application. The elevator model is computed in the FPGA to increase the throughput and timing coherence of the encoder signals, whilst the processor runs the communication server to the remote test controller. We program HiLES in a mixture of languages, adopting model-based or conventional SW development processes when better suited (see §6.2.2).

Queue design: Understanding the elevator CANV bus

The CAN Restbus simulator would replace a multiplicity of embedded I/O nodes connected to a CANV bus in the elevator-control system: the *extensions* (EXT). More than a hundred EXTs can be networked to a single CANV bus. Currently the embedded I/O application supports 3 controller product families, each requiring a particular CAN dialect which the application configures during start-up. There are some high-layer custom protocols common to all the controller variants, but differing in the frame specification:

- *Process data object (PDO)*: Transmitted synchronously by the controller to modify the outputs in the I/O devices.
- *Urgent service data object (USDO)*: Transmitted asynchronously by the I/O devices after input changes.
- *Non-urgent service data object (NUSDO)*: Sent by the controller to set and read values from the I/O device.
- *Network management protocol (NMT)*: Messages to identify the nodes available in the bus. I/O devices transmit Network Management (NMT) messages synchronously.

According to the range of CAN-IDs allocated to each protocol, in our case the message precedence is (ordered from the highest to the lowest priority): Process Data Object (PDO), USDO, NMT and NUSDO. The implications for the CAN Restbus simulator are:

- In a CAN bus only one node may transmit a frame at a time. Thus, it is feasible to share a single CAN port between multiple virtual devices, if proper queuing handles the in/outgoing messages for each instance.
- Not only the CANV bus-load varies with the shaft configuration (for instance, increasing the levels in a shaft model may require more EXT1 interfaces), but also the frame delays, as the number of PDOs sent by the controller increases with the number of connected devices.
- The allocation of CAN-IDs to the protocols is specific to each product line, then the worst-case transmission delay for a protocol changes with the controller variant.

- There are signals with redundant paths to the controller. Race conditions may arise between input changes notified by a message from a virtual device and the alternative paths, e.g., the switching of a relay in the safety circuit. A wrong time correlation between them may trigger fault reactions from the elevator controller.

In some elevator systems the EXT4 devices monitor the bottom and top limit switches in the shaft. When a so-configured EXT4 detects an input change, it immediately signals the new state to the controller by sending an USDO. EXT3 devices also handle safety inputs to prevent trapping people with the elevator doors, e.g., photoelectric sensors. Therefore the virtualization of EXT3 and EXT4 devices requires a minimum-delay timing to properly simulate the real-elevator behaviour.

Implementation: Restbus simulator LabVIEW API

To implement the Restbus simulator we first designed an API to deploy the device para-virtualization under a LabVIEW run-time as a shared library. This container holds both the hardware resource emulation and the embedded application. Second, we adapted and re-arranged the source code of the embedded application to isolate the platform-specific dependencies.

The Restbus simulator was packaged as a runtime-loadable component. This component is imported within the LabVIEW/Real-time (LVRT) project as a Call Library Function Node (CLFN) (see §6.4.2). At first, we designed an Application Programming Interface (API) to invoke single instances of Virtual ECUs (V-ECUs) from a LabVIEW Virtual Instrument (VI), that is, a LabVIEW V-ECU. A LabVIEW *Restbus* consists of a LabVIEW *class* including a private array of aggregated LabVIEW V-ECU nodes. The LabVIEW *Restbus* `update` function provides a single control for incoming messages, and a single indicator (i.e., output terminal) for outgoing frames. Additional I/O terminals connected the LabVIEW *Restbus* instance to the virtual I/Os, `DIP Switch`, `Power-on` and `Reset` signal arrays. When all possible V-ECUs were 'powered-on', in this configuration the execution overhead caused by data-type conversions between LabVIEW and the C-coded simulator exceeded the maximum allowable sampling time.

We required a trade-off analysis on the performance of memory transfer: LabVIEW or C provide similar programming statements (e.g., `for/while` loops), but for our case study we had to implement the loop statements in the C++ wrapper, encapsulating the whole set of virtual nodes of a Restbus group behind a single LabVIEW call node. This reduced the memory transfers which eventually showed to be faster than its LabVIEW loop counterpart, but we sacrificed the composability offered by LabVIEW.

Restbus simulator under LabVIEW/Real-time

To integrate the Restbus simulator with the elevator simulator we have to adapt the LabVIEW and native-libraries projects to the cRIO target. To realize the CAN ports, we

Algorithm 1 Pseudo-code from the Restbus Simulator LabVIEW-RT Loop.

```

1: Initialize CAN and UDP Ports
2: for all Buses do
3:   Read UDP messages      // Commands from SGUI
4:   Read CAN messages     // From control system
5:   Read DPRAM            // Read model & FPGA I/O
6:   Call Restbus API      // Sequential update
7:   Write DPRAM           // Write model & FPGA I/O
8:   Send CAN messages     // To control system
9:   Send UDP messages     // Results to SGUI
10: end for

```

chose an NIRTLL-compatible PEAK CAN interface. We selected a dual-CAN PEAK PCAN USB Pro that can be plugged into one of the 2 USB ports of the cRIO. We cross-compiled the Linux-driver and library provided by PEAK, using the NIRTLL kernel source code and the development environment from NI. As our implementation requires additional dynamic-loadable libraries, we cross-compiled the source code for the Restbus database and the virtual nodes, as we did for the PEAK library.

We adopted an object-oriented paradigm to write the LabVIEW code, using LabVIEW classes (see Figure 8.15) to abstract the implementation of the simulator from the chosen CAN device (Peak PCAN) and from the computing platform (desktop PC running Windows OS or cRIO with Linux RT), since each driver and low-level function can be implemented in subclasses with common interfaces. To implement the Restbus simulator in a cRIO-903x controller we branched the LabVIEW project from the Windows desktop simulator. First we included the cRIO-903x real-time target. LabVIEW inheritance relationship between classes is declared in the parent class. An abstract VI from the parent links to every alternative implementation from the derived classes. When LabVIEW links an application, the parent LabVIEW class requires the deployment of all the child variants, even if these are not used. As a consequence, we had to clone the shared source code from the parent classes to prevent LabVIEW-RT from deploying incompatible components into the cRIO RTOS (e.g., deploying the support for CAN ports in Windows OS).

The final deployment step was the integration of the Restbus simulator with the real-time HiL elevator simulator. The simulator in the cRIO' FPGA computes the position-related signals for the elevator car and the doors, including the state of the limit switches, encoder measurements, etc. It also publishes the instantaneous values computed for these signals through a dual-port RAM (DPRAM) interface. The cRIO processor reads the published values and retransmits them to the remote user interface. To integrate the Restbus simulator, we connected the signals to the inputs of the EXT3 and EXT4 devices. Besides, we monitor the execution time inside LabVIEW Timed Loops to check the timeliness and computation of periodic tasks.

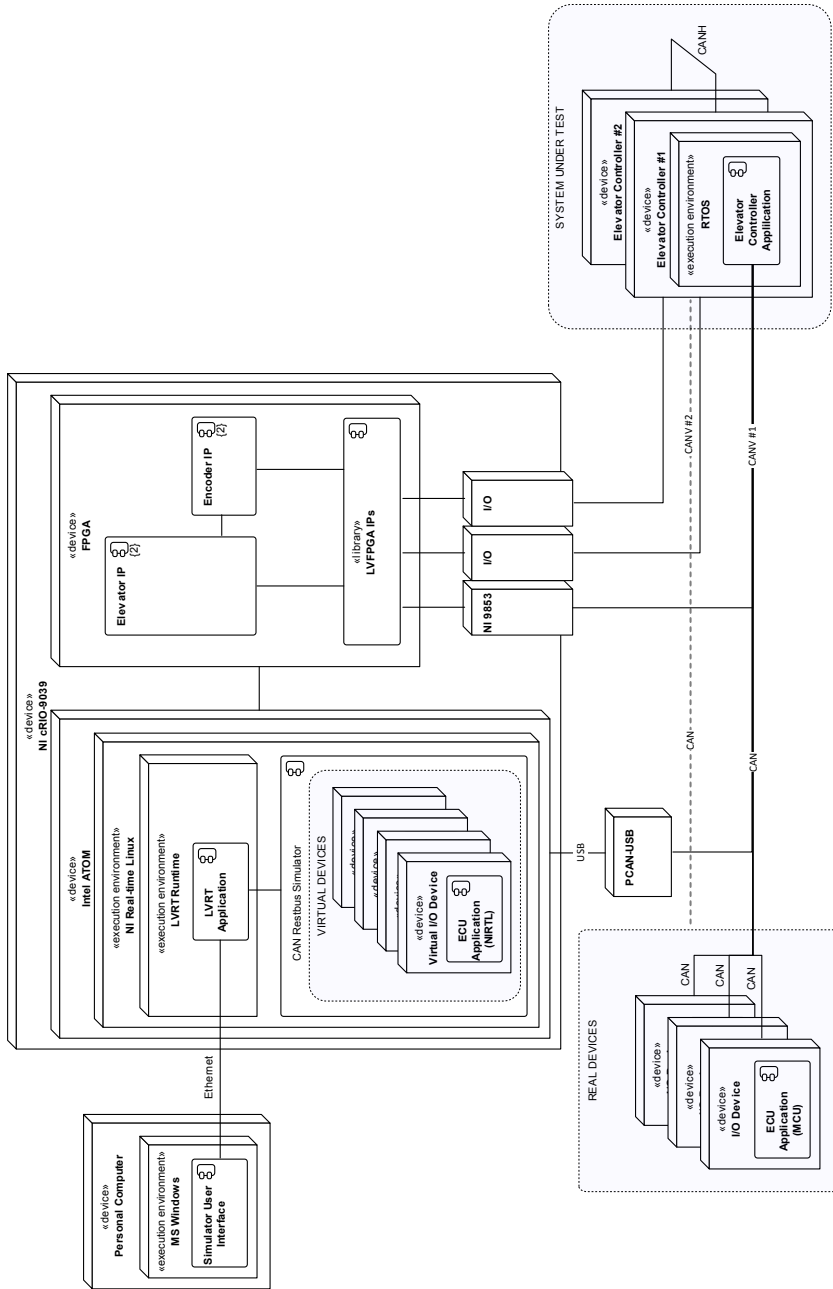


Figure 8.16: Deployment view of the cRIO HiL elevator simulator with CAN Restbus virtual devices

In this figure the Compact RIO (cRIO) FPGA realizes 2 instances of the elevator model and their I/O and control interfaces. The cRIO multi-core processor implements 2 instances of the CAN Restbus simulator, each of them connected to a separate CAN port (only one CANV bus shown in figure). Elevator-group simulators consist of multiple dual-simulators running concurrently, where a traffic dispatcher coordinates the elevator controllers to attend the users calls. (source: [NAP+17])

Remote simulator user interface

The Restbus simulator integrates a communication server to connect remote client applications over Ethernet. The communication server implements a custom UDP protocol that makes it possible to write test automation applications in other programming languages. The server also provides a discovery service by cyclic broadcasts. The server expedites the simultaneous writing of input values to the distributed system. This feature enhances the repeatability of tests, that was formerly unattainable with separate real I/O devices requiring user actions (EXT1 or EXT2 devices). We developed a remote simulator graphical user interface (SGUI) application in LabVIEW to validate the Restbus simulator. At start-up, the SGUI application scans the Ethernet network for available Restbus simulators. Upon successful connection, the SGUI retrieves and updates the current power-on status of each virtual device.

The SGUI emulates user-operated components to support automation and save physical space, allowing the user to set the inputs and observe the outputs of the virtual devices (see Figure 8.17). Push buttons and switches enable the test operator to simulate the power on/off of a virtual device running in HiLES, configure the device identifier or call an elevator from any level, while the LED outputs show the status of the call.

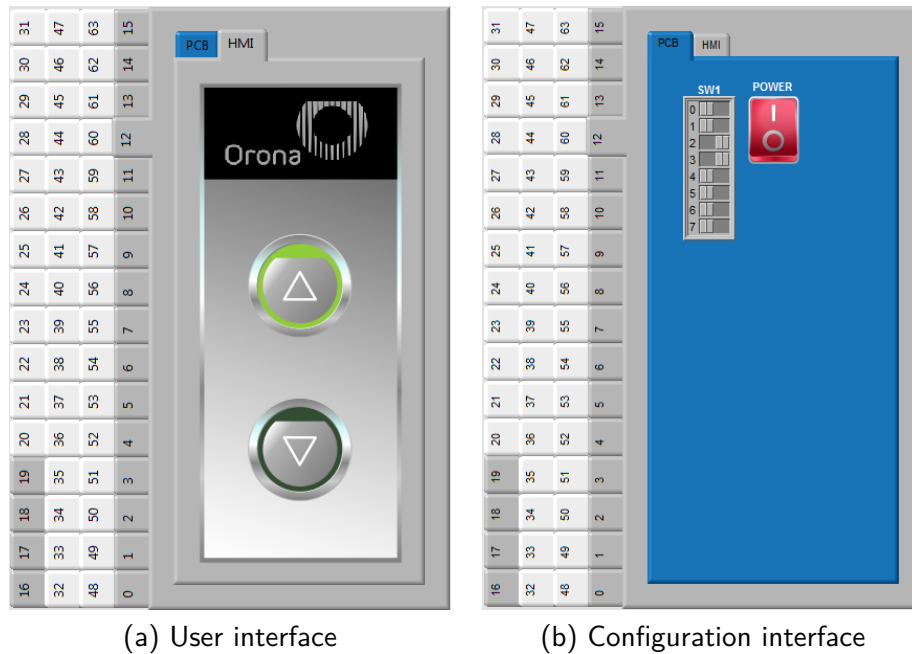


Figure 8.17: The SGUI front-end for virtual level-call devices (EXT1)

8.2.6 Discussion

For our test application we tried out a single-model approach instead of a dual-model development (i.e., a platform-independent specification model and a platform-specific implementation model). The project team consisted of 3 people: a seasoned Simulink user developed the bulk of the Simulink model, a developer with a mid-level background in both Simulink and LabVIEW programming integrated the single-encoder cRIO-9082 variant, and an FPGA expert with programming knowledge about cRIO controllers and LabVIEW synthesized the dual-encoder cRIO-9039 variant. In both developments we achieved the required real-time simulator performance. The FPGA provided the execution parallelism to add new functions with virtually no interference in the operation of the rest of the HiL simulator. The main advantages of preserving the function of the HiL test system as a set of models were an improved communication across the project team, easier maintenance of the test components and better portability throughout the heterogeneous computing platforms.

Currently we have 2 HiL simulator variants derived from the same Simulink model, cRIO-9082 (FPGA Xilinx Spartan-6 LX150) and cRIO-9039 (FPGA Xilinx Kintex-7 325T). Although the FPGA variants impose different timing constraints, by profiling the transformation settings we got both implementations showing the same real-time behaviour. Nowadays we can modify or enhance the features of the test system and initially validate them by relying on the models of computation supported by the modelling environment. Before, the maintenance and validation of former HiL elevator simulators was cumbersome, time-consuming and error-prone: the real-time behaviour of the simulator was confronted with the actual System Under Test (SUT), that was also under development and therefore its software (SW) was unstable. Even worse, the constraints of a given HiL computing platform limited the evolution of the test system, requiring at times a major overhauling. We expect that in the mid-term the new HiL simulator will be more cost-effective than the precedent HiL simulators.

First, we tested both the Windows and the cRIO CAN Restbus simulators by monitoring the CAN bus and checking the timing and data contents of the synchronous NMT messages. Table 8.9 shows the timing for NMT messages transmitted by the simulators when running all the Extensions (see Table 8.7) on the cRIO HiLES. We measured similar timing results for the Windows CAN Restbus simulator. We tested the Restbus simulator both in an open-loop (“No HiL”) and closed-loop (“HiL”) configuration. The mean value of the cycle time is shown together with the standard deviation (Std. Dev.). In the “No HiL” case, the Restbus simulator is tested without elevator controller; in the “HiL” case the elevator controller is used to close the control loop. The timing results obtained in the closed-loop configuration are slightly degraded, possibly due to the extra traffic on the CAN bus generated by the control system. The measured cycle time values in closed-loop configuration for the cRIO variant met the timing specifications from Table 8.7 with a precision of 6.6%. These results were considered satisfactory, since timing requirements are not tight, i.e., they are approximations of the expected timing for real devices.

Table 8.9: Measured NMT cycle times for the cRIO Restbus simulator.

Virtual Device	No HiL		HiL	
	Mean Value	Std. Dev.	Mean Value	Std. Dev
EXT1	3.098 s	0.006	3.222 s	0.70
EXT2	3.098 s	0.006	3.222 s	0.69
EXT3	3.098 s	0.006	3.222 s	0.73
EXT4.0	0.196 s	0.018	0.205 s	0.28
EXT4.1	3.098 s	0.006	3.222 s	0.74
EXT5	3.098 s	0.006	3.222 s	0.75

The Restbus simulator update cycle time is 50ms, although the measured computation time for updating 140 active virtual devices is 25ms, divided as follows: the **CAN** read and write operations take 3ms and 1ms respectively; the DPRAM accesses are negligible in terms of time; the call to the API that simulates the devices, i.e., the core of the Restbus simulator, take 8ms per channel. Thus it would be feasible to run 2 virtual Restbuses sequentially in the update loop.

After this preliminary test we included the elevator simulator and the elevator-control system, as in the configuration shown in Figure 8.16, to check the consistency of the Restbus simulator in a realistic scenario. The control system was able to communicate and recognize all the simulated devices from EXT1 to EXT5 as if they were physical. No problems were detected in extensive trials with the elevator simulator by employing simulated devices EXT1, EXT2, EXT3, and EXT5 together with a physical version of EXT4. On the other hand, the simulated version of EXT4 caused some problems, possibly due to the latency in the processing system of the Restbus simulator (50ms) and of the **USB** to **CAN** converter (which is non real-time). Indeed, when connected to a simulated EXT4, the control system is not able to brake and stop the elevator before reaching the limits of the shaft.

8.2.7 Conclusion

The Elevator-Simulator Case Study demonstrates the application of our contributed model-based development workflow to improve the re-usability of test artefacts from offline- to real-time **HiL** simulations. We discussed some issues related to the portability of the Very High-Speed Integrated Hardware Description Language (**VHDL**) description when deploying the simulator to a cRIO variant, as well as how we tackled these platform-related problems.

Our case study shows that state-of-art **COTS** model-based development tools enable designing and validating real-time test systems at an abstract level, while reliable automated

translators provide deterministic real-time implementations for the HiL test system. However precautions shall be observed from the beginning to avoid pitfalls in the model-to-code transition: in an MBD environment like MathWorks or LabVIEW we can model a specified function in different ways (e.g., using alternative modelling languages), but not necessarily every possible choice would be translatable for each supported target. In order to minimize development iterations we should constrain the modelling artefacts to certain options, although the compliance to a subset of the modelling language would compromise the readability of the models.

We expect future enhancements in MathWorks' modelling libraries that will enable a full-scale simulation of mixed system components while increasing the set of artefacts supported by the translators. As for NI LabVIEW, we expect an improved HDL import function (e.g., support for non-scalar signals) to improve the design readability and prevent errors while linking the components manually. Anyhow, before adapting our baseline models we first will re-assess the ability of model-to-code transformers (i.e., Simulink Coder, HDL Coder) to translate the new model libraries into the real-time implementations for the HiL simulators. Regarding the HiL encoder simulator, we plan to develop an extended encoder model with integrated safety-related functions.

We also used this case-study for improving the testability of distributed systems by re-using the embedded application code and by providing a para-virtualization of target hardware devices to build a scalable Restbus simulator. We integrated our Restbus simulator into an overall plant HiL simulator, where the virtual instances of the simulated nodes were stimulated by CAN messages, timing events, signals computed by the plant model and commands from a test-execution environment. The real-time Restbus HiL simulator enabled a more thorough validation of real-time embedded controllers, as it is software-reconfigurable, may reduce the overall cost of the testbench by replacing external hardware, eases the insertion of node mutants, allows integration into a HiL test infrastructure and enhances test automation. In the future we expect to complement the real-time testbench by developing a completely virtual platform simulator that provides additional analysis features.

A future enhancement in the virtualization would be the integration of a Instruction-Set Simulator (ISS) that would re-use the actual binary code of the simulated devices. As far as we know, current technology limits the feasibility of a generic real-time ISS simulator, yet we think this could be adapted for the simulation of multiple low-end devices in a HiL environment.

Related Publications

The results presented herein appeared in [NAM+16, NAP+17].

8.3 Safety Cases for Wind Turbine Controllers

This case study evaluates the relevance of the safety argument re-use approach to ease the certification of Mixed-Criticality Product Lines (MCPLs) developed with the DREAMS platform-based design. Certification of MCPLs is supported by an automated compilation of product-specific Safety Cases (SCs), backed up by a machine-readable database of pre-built modular safety cases, modelled as composable safety arguments and patterns in Goal Structuring Notation (GSN).

8.3.1 Context

The system under consideration is a product line of supervision and control systems with integrated safety-protection functions for off-shore wind turbines. Off-shore turbines operate in a harsh environment, posing technological and maintainability challenges, while requiring compliance with more stringent safety requirements than ashore turbines. Wind-turbine manufacturers face the problem of handling the multiple variants of control and protection systems required to each possible wind-turbine configuration.

A workaround to develop safe off-shore variants for the wind-turbine control system is to add an additional protection loop with external devices. However, a more robust, maintainable and flexible approach is desirable for a wind-turbine manufacturer. A platform-based design rooted in a certifiable modular solution would provide such a simplification: the DREAMS platform and toolset.

8.3.2 System Description

The Wind Turbine Controller Case Study (WTCCS) consists of a product line of mixed-criticality Wind Turbine Controllers based on the DREAMS harmonized platform that should be hypothetically certifiable according to the ISO 13849-1:2006 [ISO13849] safety standard for machinery, that is rooted in the IEC 61508 functional safety standard.

Figure 8.18 sketches the scope of application of the DREAMS platform in the WTCCS. A wind farm comprises a number of wind turbines. Each wind turbine is operated by a Wind Turbine Control Unit (WTCU). The WTCUs in a wind farm are interconnected through real-time and deterministic networks and are also linked to a central Wind Park Control Center, that is a gateway between the wind farm and the remote SCADA center from the utility company.

In the scenario represented in Figure 8.18 each WTCU manages a number of distributed I/O nodes connected to a real-time EtherCAT network. The main functions of the WTCU are executed in the GE Alstom GALILEO V4 platform. GALILEO is a real-time computing platform to deploy the supervision and control system, though it may support other real-time applications such as wind farm control. GALILEO V4 is based on commercial HW (APC910 industrial PC with a dual-core x86 processor) and customized SW and an RTOS.

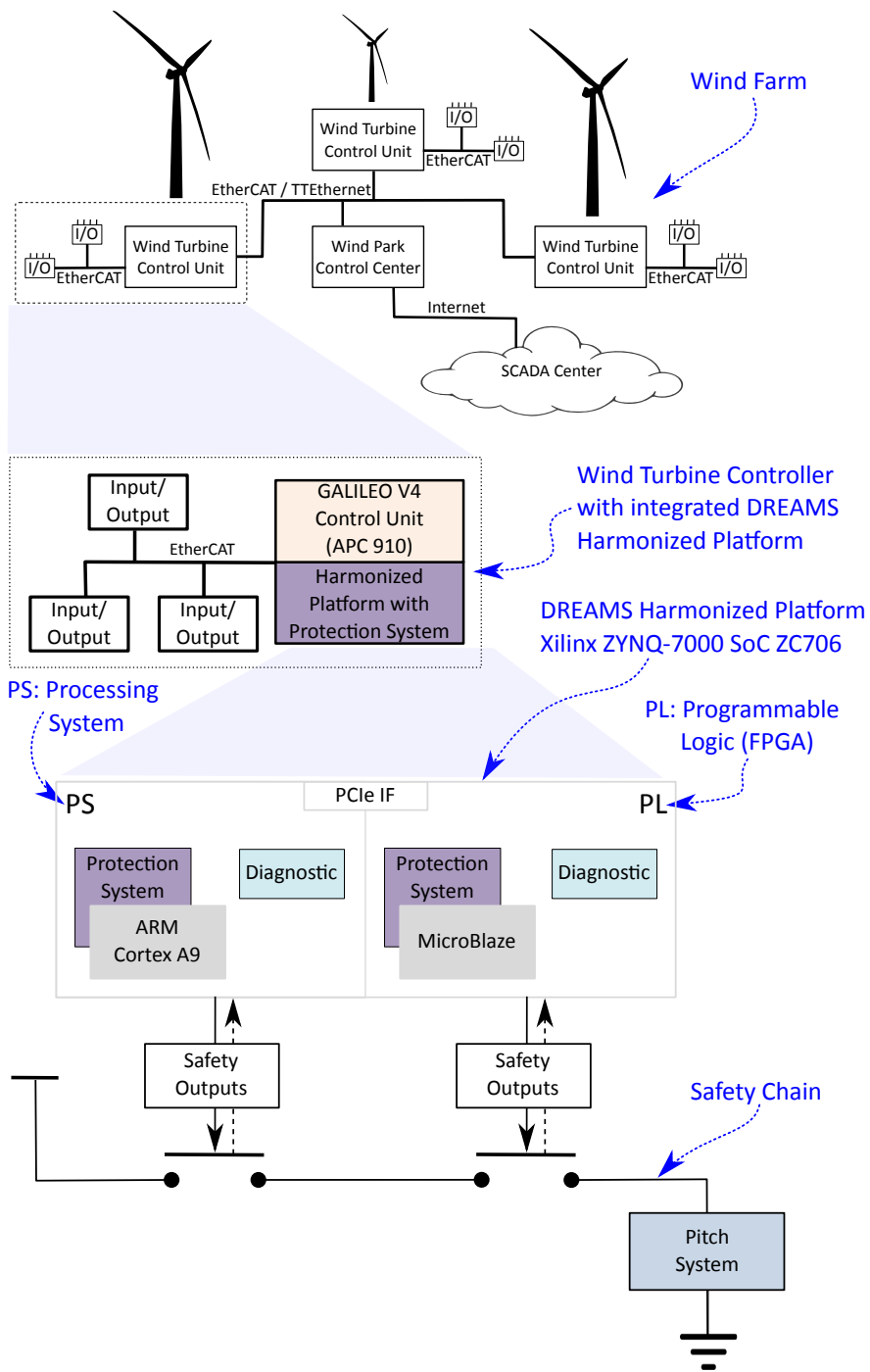


Figure 8.18: Application scope of the DREAMS platform in the Wind Turbine Controller (adapted from [DRE721])

The wind-turbine protection system is in charge of maintaining the wind turbine in a safe state. The main functionality of the protection system is to assure that the design limits of the wind turbine are not exceeded. The protection functions shall be activated as a result of an error of the control function (running in the supervisory system) or of the effects of an internal or external fault or dangerous event. It should be activated in cases such as over-speed, generator overload or fault, excessive vibration, or an abnormal cable twist due to nacelle rotation by yawing.

A possible deployment for the protection functions would be the **DREAMS** Harmonized Platform. This is a heterogeneous computing platform based on the Xilinx ZYNQ-7000 SoC, that integrates a Processing System (PS) featuring a dual-core ARM Cortex A9 processor and a Programmable Logic (PL) subsystem (Artix-7 **FPGA**) in a single chip, as represented in Figure 8.18. Synthetic components as processors (μ Blaze) and **TTNoCs** support a variety of alternative deployments for the protection functions. Complementary to the **HW / SW** solutions offered by **DREAMS**, a number of Modular Safety Cases (**MSCs**) ease the dependability assessment of the realized safety application.

8.3.3 Problem Statement

We aim at achieving a higher degree of integration between the supervisory system and the protection system, thus making the overall solution more robust, maintainable, and flexible, while satisfying the safety and non safety requirements.

The **WTCU** variability comes to the Safety Integrity Level (**SIL**) requirement: it is desirable to have a customizable protection system to achieve upto a Safety Integrity Level (**SIL**) of 3, which may require a varying safety structure to increase the hardware fault tolerance (**HFT**). So we seek to combine **GALILEO** with a protection system deployed on the **DREAMS** platform.

A feasible solution is to integrate the **GALILEO** platform and the harmonized platform via **PCIe**, while an EtherCAT field-bus connects the computing platforms to several **I/O** modules. This solution could realize heterogeneous redundancy to achieve the specified **HFT**, where the requirements for on-chip redundancy detailed in IEC-61508-2 Annex E [**IEC61508-2**] must be met in order to achieve certification. The resulting demonstrator should address the assessment of the dependability of the **WTCU** product line and reuse as many as **HW** and **SW** components from former controllers developed by GE Alstom. However, we identified the following challenges ahead:

1. Assuming that the requirements for the **MCPL** of Wind Turbine Control Unit (**WTCU**) are sufficiently specified, the **DREAMS** workflow runs an automated Design Space Exploration (**DSE**) that optimizes the Mixed-Criticality Product Line (**MCPL**) according to the requirements, building up a number of Mixed-Criticality Systems (**MCSs**) from a library of **DREAMS** components and another library of manufacturer-provided application components.

Could we similarly build up the safety cases for each mixed-criticality system upon the DREAMS modular safety-cases in an automated way?

2. Safety standards encourage development-process redundancy to reduce the likelihood of systematic errors. This means that verification, validation and testing (VVT) would be planned and developed concurrently and separately. However, due to the cost of VVT tasks, many of these activities would be postponed until the DSE had analysed, selected and assessed the set of presumably valid MCS product samples. Once built, these products should later undergo several VVT phases that would contribute the evidences to confirm the safety achievements. These incrementally gathered evidences complete the Safety Case (SC), but at a later stage.

Could we incrementally integrate the newly available evidences supporting the claims in the safety cases while avoiding extensive documentation rework?

3. The safety assessment for a product line of Wind Turbine Control Units (WTCUs) may consider these as standalone systems, excluding the wind turbine itself. However, the safety assessment of the final application would require a composed safety argument, where the safety claims for the wind turbine builds on the WTCU safety cases.

Can we scale up the safety arguments to compile a safety case up to a whole system application or even a system-of-systems?

8.3.4 DREAMS Architecture for the Wind Turbine Controller

The Wind Turbine Controller Case Study (WTCCS) consists of a product line of mixed-criticality Wind Turbine Control Units based on the DREAMS harmonized platform, that should be hypothetically certifiable according to the ISO 13849-1:2006 [ISO13849] safety standard for machinery, which is rooted in the IEC 61508 functional safety standard. The DREAMS hardware (HW) / software (SW) architecture for the WTCU is represented in Figure 8.19, showing the supervision, control and protection units. According to the IEC 61508 safety standard, these platforms can be considered as independent HW systems.

Furthermore, this system architecture supports the execution of functions with different criticality levels (such as SIL-1 to 4 according to IEC 61508). The XtratuM hypervisor [XTR] is used to split the CPUs of the PS and the soft-core(s) of the Programmable Logic (PL) into partitions where the functionalities with different criticality are executed. The protection unit shown in Figure 8.19 communicates with external sensors (e.g., wind speed sensor) and actuators (e.g., safety relay) through a safe fieldbus protocol composed of a non-safe fieldbus EtherCAT and a Safety Communication Layer (SCL) integrated on top of a Network-on-Chip (NoC). The combination of the NoC and the SCL enables temporal and spatial independences, depending on whether a shared memory is used to communicate between the partitions or not. The NoC implemented in this case study is the STmicroelectronics' NoC (STNoC), which is complemented with the NoC SCL cross-domain pattern. The SCL guarantees a safe communication between the partitions.

The AutoFOCUS 3 (AF3) toolset would carry out the preliminary safety assessment for each feasible deployment using model libraries of HW components, logical components, as well as safety compliance models (see Figure 8.20). ‘Logical components’ denote any kind of SW or synthesizable IP (e.g., hypervisors, operating systems, executable applications, bitstream files for programmable logic, etc.) that could be deployed in the system. In the WTCU, the SW items typically represent complete SW stacks specific to a target processor.

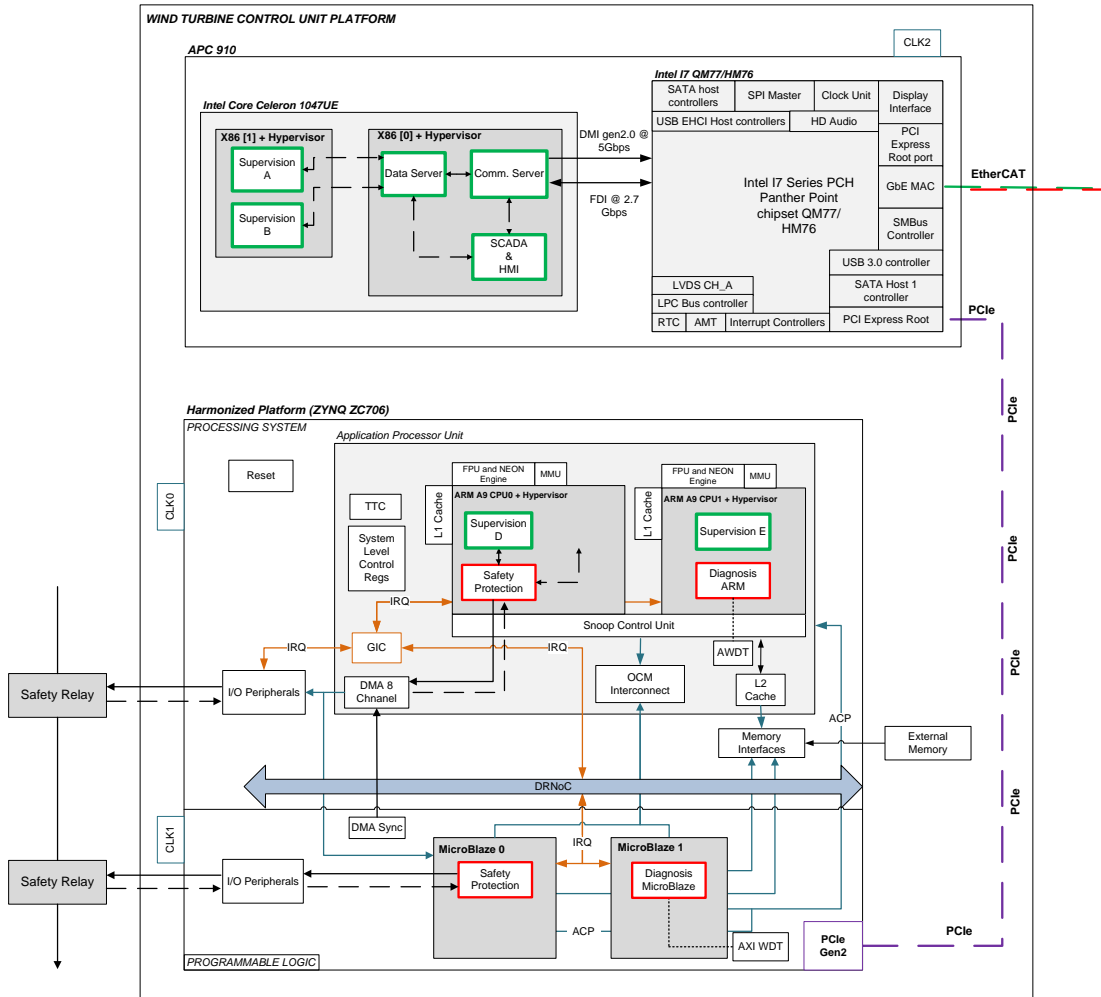


Figure 8.19: Possible deployments of the DREAMS Wind Turbine Control Unit (adapted from [DRE721])

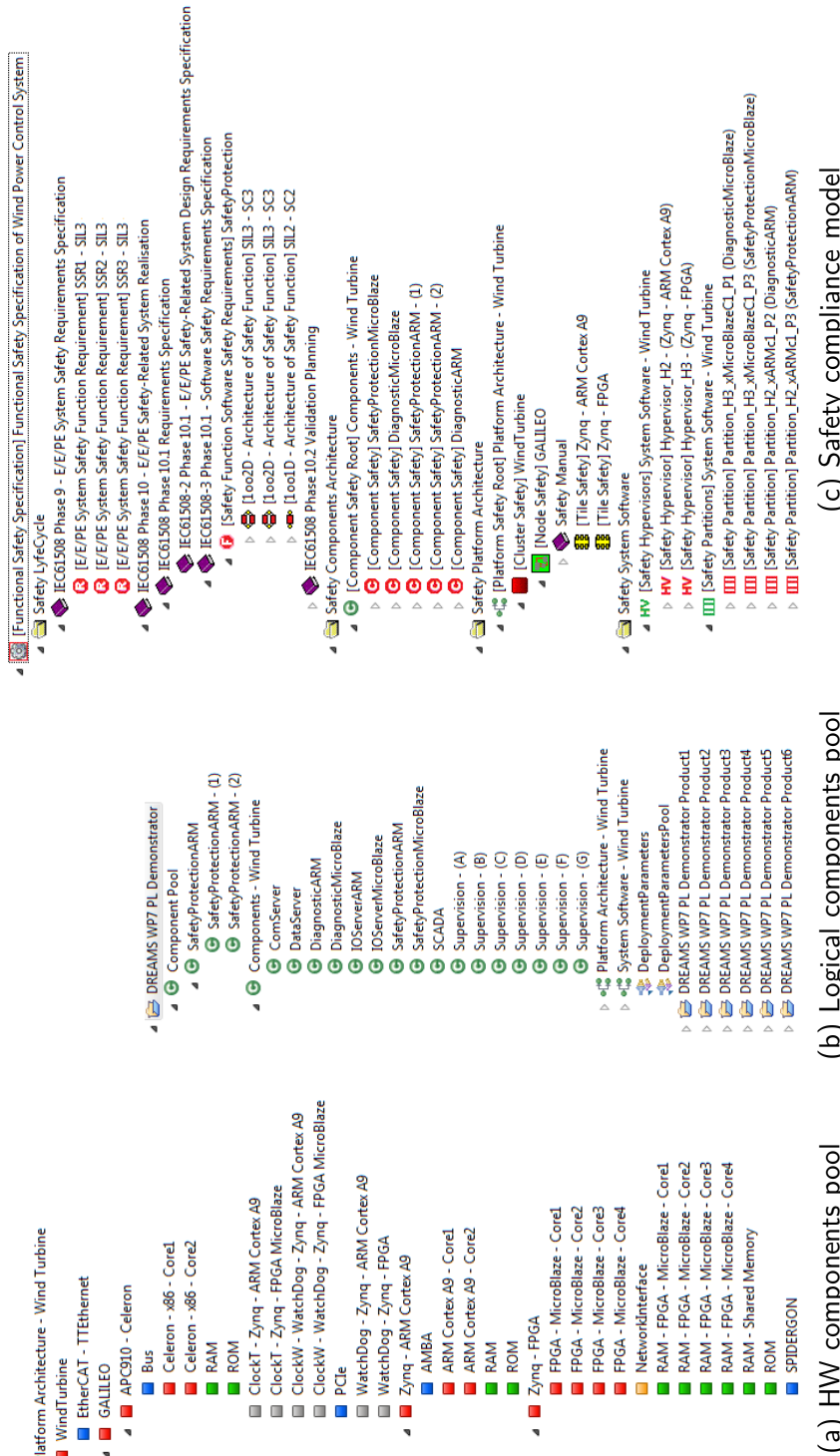


Figure 8.20: DREAMS libraries for HW / SW components and specification of safety compliance (source: [DRE721])

Preliminary Work

Preliminary work consists of compiling in advance the technical data and the certification argument fragments for the certifiable/pre-certified components to be considered later at the **DSE**. For the sake of accessibility, the certification arguments would be stored in a database, including the safety certificates (if available) and any safety-relevant information from the safety manuals, e.g., assumptions to prove valid for a safe integration of the component. The counterpart of the certification data is the component-properties database provided for designing the mixed-criticality product line in the **AF3** environment.

The Wind Power Demonstrator resembles an industrial-grade development of a mixed-criticality safety product line, combining a variety of functional requirements of both safety-critical (for possibly different integrity requirements) and non-safety-related kind. **DREAMS** deliverable D 7.2.1 [**DRE721**] provides a detailed overview of the application and its safety requirements. For certification purposes, a number of supportive evidences should be generated beforehand at this stage (e.g., risk analysis results, **FTA**, **FMEA**, **FMEDA**). This information constitutes the basis for eliciting the safety requirements for the system. Note that the safety requirements as scoped in D 7.2.1 apply to the whole system (i.e., including the wind turbine and its environment) and thus consider additional elements besides the programmable electronics.

The top safety claims would be represented as **GSN Goals** while evidences would be represented as **GSN Solutions** in the argumentation model. As safety analysis would normally require using complementary techniques and formalisms. Ideally the evidence nodes, i.e., the solutions, will include references to the document outputs generated in this analysis process.

Allocation of Safety Requirements to **DREAMS** platform

The next step consists of refining, decomposing and allocating the system safety requirements to the parts of the system. For demonstration purposes, we focus on deriving a set of safety requirements for the Programmable Electronics (**PE**) components of **WTCCS**, limiting the scope to both the **GALILEO** controller and the **DREAMS** harmonized platform. We will represent these safety requirements allocated to **PE** as top-level goals and context in the **GSN** argumentation graph. The same target safety requirements shall also be declared in the **AF3** environment, alongside the permissible variation points for the system composition.

Variability Resolution and Product Line Optimization

During the optimization process at **DSE**, the **DREAMS** toolset benchmarks a number of candidate product configurations against a set of properties that includes the maximum achievable safety scores. The evaluation operates on a set of automatically generated models of the system from which the **DSE** selects a Pareto-optimal product line configuration that,

once validated by the Safety Compliance Constraints & Rules Checker (SCCRC) tool, is presumed to satisfy the safety requirements. Variability resolution relies in three models:

- The '150 % model' sums up all the selectable features of the MCPL, defining a set of variability parameters for a "partial resolution" step, based on business decisions.
- The '125 % model' is refined from the '150 % model', by applying the business choices of product features. '125 % models' represent families of possible product configurations implementing a given set of properties, leaving technical variability features to be optimized by the DSE.
- A '100 % model' derives from the '125 % model' for a selection of technical features, defining a single product on which a preliminary safety assessment will be carried out.

Building the '150 % model'

Herein we show how to build a '150 % model' of the WTCU that includes a 'Component Pool' and bounds for the number of the replicas of one component (see §8.3.4). The '150 % model' is built starting from an initial model with no variability (i.e., a '100 % model'), which is iteratively extended until all possible variants are included in it.

The '150 % model' bundles all the available safety functions into a single logical architecture model. During the resolution of business variability, the DREAMS Base Variability Resolution (BVR) derives concrete architectures by eliminating the components of a subset of safety functions.

Figure 8.21 shows a '150 % model' that contains two channels of the wind turbine's safety function. Each channel in Figure 8.21 consists of a logic component that represents the actual safety function and a diagnostic unit that provides status information about the logic component, but does not have a direct influence on triggering the safety relays of the turbine. From this model the following architectures can be derived:

- A safety architecture with two independent channels (ARM and μ Blaze), each channel consisting of logic components deployed on separate hypervisor partitions, that in turn execute in different processors (ARM / μ Blaze). This configurations has $HFT=1$.
- A single-channel architecture (either ARM or μ Blaze), by removing the unneeded pair of logic and diagnostic components (e.g., 'SafetyProtectionARM' and 'DiagnosticARM'). This configurations has $HFT=0$.

The '150 % model' is modified to represent the technical variability, introducing abstract placeholders for which functionally equivalent variants provide the concrete implementations. In the example from Figure 8.21 'DiagnosticMicroblaze' and 'DiagnosticARM' are subsumed in the abstract 'Diagnostic' component, while the safety protection functions 'SafetyProtectionMicroblaze' and 'SafetyProtectionARM' are represented by

'SafetyProtection'. The resulting model supports the exploration of different safety architectures by the DSE. Figure 8.22 depicts the model derived from Figure 8.21.

In addition, a set of candidate safety architectures available for each particular safety function is provided separately. Hence, the safety-compliance model contains a list of candidate architectures that now reference the abstract components (see Figure 8.23).

Deriving the '125 % model'

The '150 % model' bundles all the possible features in the MCPL. The next step in the DREAMS workflow consists of specifying the desired product-line features using the AF3 environment. This triggers the BVR to resolve the business variability, generating the '125 % model' that only has technical variation points to be solved by the DSE when seeking an optimal product configuration. If some products generated by BVR may only be implemented with a subset of the safety architectures of a safety function, the disallowed choices are removed in the resolution step that generates the '125 %' safety-compliance model that is consumed by the DSE.

Generation of the '100 % models'

After selecting the business features, the DREAMS user has to set the optimization goal for the DSE, in order to resolve the technical variability. The DSE operates on the '125 %' safety-compliance model to build '100 % models', relying on an evolutionary search algorithm

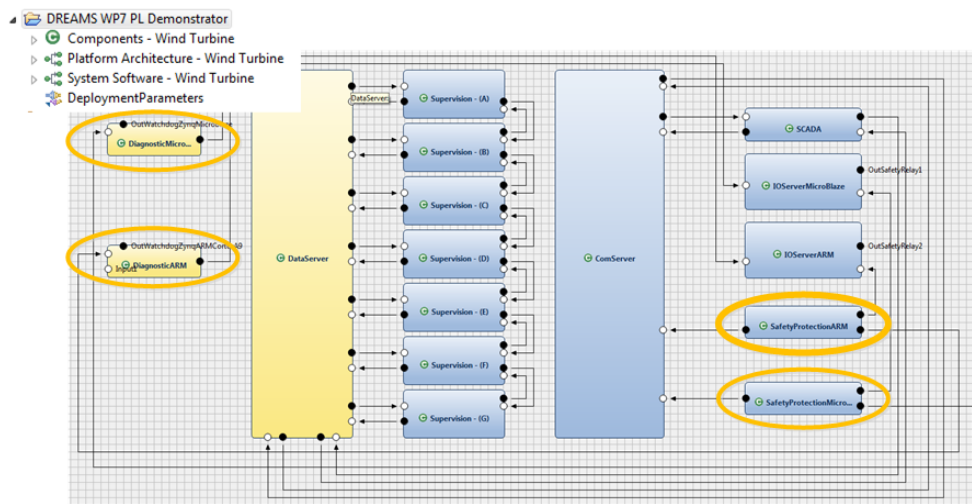


Figure 8.21: '150 % model' architecture of the Wind Turbine Control Unit in AF3 (source: [DRE433])

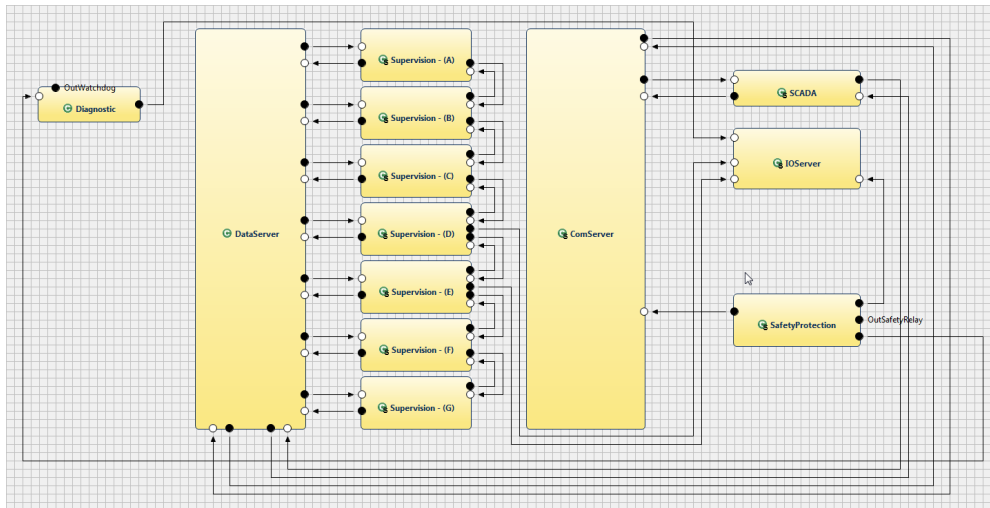


Figure 8.22: Updated '150 % model' with abstract safety components
(source: [DRE433])

to select candidate product configurations. The safety properties of each configuration picked by the **DSE** are evaluated by the **SCCRC**. Those component-to-execution unit mappings positively assessed by **SCCRC** are added to the set of presumably valid **MCPL** realisations. These valid '100 %' product models can be further processed by the **DREAMS** toolset, e.g., to generate a **DREAMS** deployment model or configuration artefacts.

Figure 8.24 shows the generated component architecture of a safety compliant deployment whose safety function has a '1oo2' safety architecture. It also consists of diagnostic units that are not connected to the relays that trigger the actual safety function of the wind turbine, but which can report malfunctions of the actual safety protection components. Here, each channel of the safety function has its own diagnostic unit. In this example each safety channel consists of a 'SafetyProtection - (1)' and a 'Diagnostic - (1)' component, which forward their calculation results to an 'IOServer' component per channel. The components to realise this architecture of the function instantiated from the 'Component Pool'. Hence, the channels of a safety function can also use software diversity to increase the safety metrics of the system.

8.3.5 Safety Case Argumentation

The **DREAMS SCCRC** tool helps at **DSE**, verifying that eligible architectures and deployments for safety functions do not contain errors that, from a safety perspective, would eventually prevent the safety certification of the products. A certification process according to a given standard requires a compliance argumentation that demonstrates

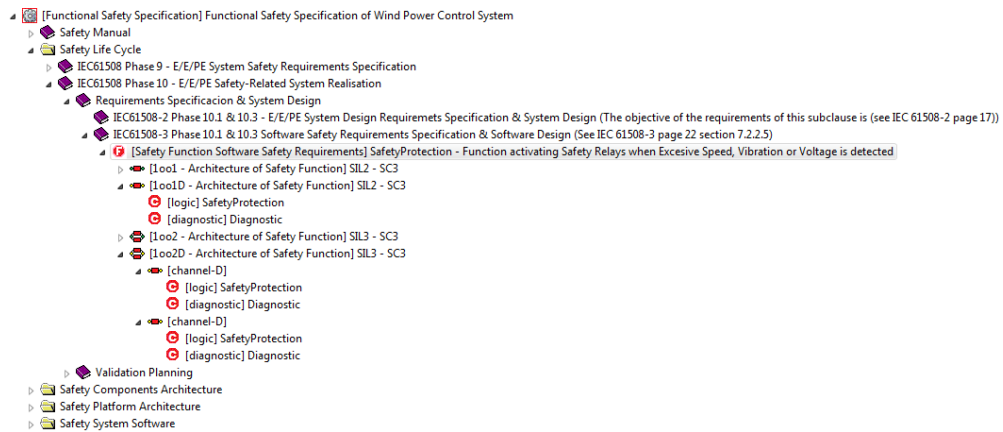


Figure 8.23: '150 %' safety compliance model of an abstract safety function (source: [DRE433])

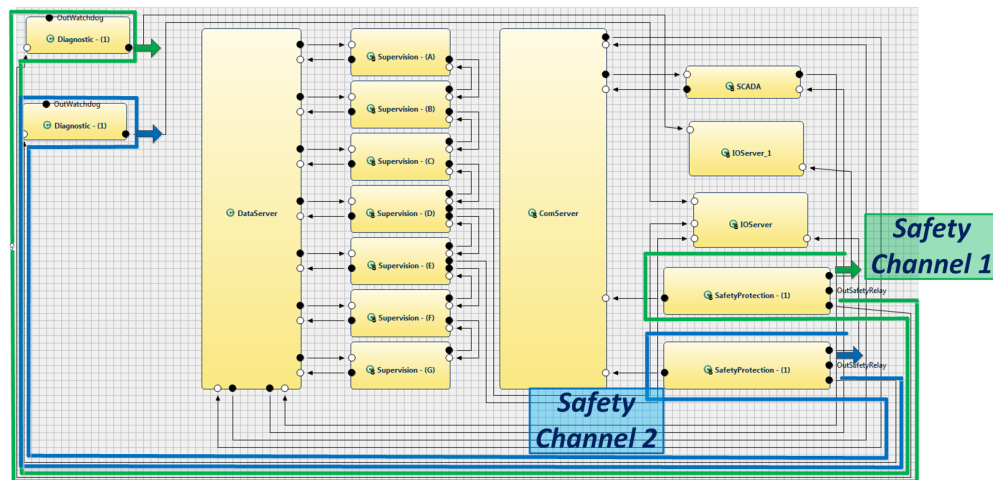


Figure 8.24: Generated '100 %' component model with a '1oo2' safety function architecture (source: [DRE433])

how standard recommendations have been followed. In this sense, a solid and coherent argumentation of safety claims and pieces of safety evidence must be submitted to certification authorities. The **SCCRC** argumentation module aims at partially automating the compilation of arguments and evidences, while checking their validity. However, direct use of the safety-check results as pieces of evidence in a certification process would require the qualification of the **SCCRC** tool itself, which is a very costly process. A workaround is to

demand **SCCRC** to produce a human readable document so that the certification authority can review and verify the checks done by the tool, in an argued way.

To support the automated compilation of safety arguments, we adopted modular **GSN** to describe the **DREAMS MSCs**. The pre-built arguments are stored in a Database Management System (**DBMS**) system, from which the **SCCRC** retrieves the argument models and then connects them with the safety-evaluation rationale elaborated at **DSE**, for each product configuration.

The product-line approach followed in **DREAMS** also takes advantage of a development process that relies on composition of modules (with their own proven safety argumentation) to design the products, so that the final products reuse safety-assurance artefacts either at subsystem or component levels.

Therefore, the **SCCRC** must also be able to automatically compose safety argumentations of subsystems for a given product. Further, it must be able to automatically produce at least part of the safety argumentation of any valid variant of the product line. This is a cost-effective certification process that saves certification costs because it both reuses already certified modules (e.g., a hypervisor or a **COTS** processors) and automatically constructs the basis of an argumentation of the composed product (no need to construct a complete argumentation from scratch for every variant). Moreover, a set of refined safety rules enhances the error-detection capability of the safety tool. Particularly important during the composition is the ability to check if usage constraints of reused sub-modules (expressed in Safety Manuals as Usage Constraints) are met by the composed modules using them. Argumentation must also document the fact that usage constraints are met. In summary, the features of the **SCCRC** tool are:

- The automatic translation of safety constraints and rule checks into a **GSN** based safety argumentation.
- The automatic composition of **GSN** based safety argumentations of sub-modules following the **GSN** certification methodology proposed in Chapter 7.
- Safety rules enhancements as, for example, checking of submodules usage constraints.

As a **DSE** post-processing activity, a partial **GSN** argument model for certification is automatically built for the resulting **MCPL** configuration. 'Partial' means that the argument has to be completed with evidences to be produced in a subsequent phase of the safety product realization (e.g., verification results from testing the actual **PE**). The certification argument consists of five packages, as shown in Figure 8.25, namely:

1. *Safety Argumentation for the Wind Turbine Demonstrator module*: This top-level package compiles the uppermost claims for the system, including the claim about the satisfaction of the allocated safety requirements and all the relevant assumptions for the system. This module is common to all the possible product configurations.

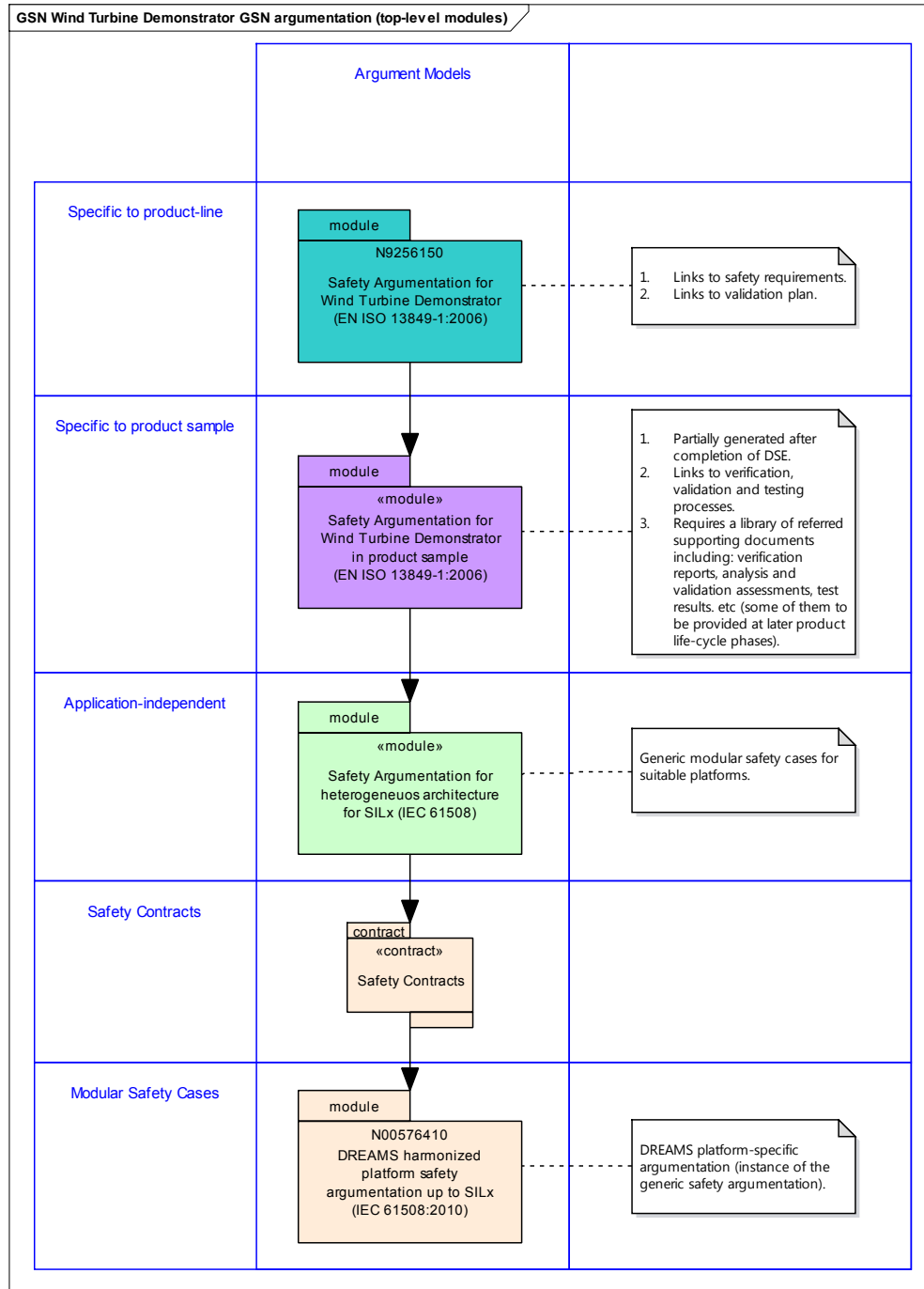


Figure 8.25: Layered GSN modules for compiling the WTCU certification arguments

2. *Safety Argumentation for Wind Turbine Demonstrator in product sample module:* This intermediate module instantiates an argument specific to a product configuration. It is initially composed at the **SCCRC** post-processing and is partially supported by the generic safety arguments of the DREAMS heterogeneous platform.
3. *Safety Argumentation for heterogeneous architecture:* This is a set of pre-built certification arguments for generic re-usable DREAMS platform components [[DRE511](#), [DRE512](#), [DRE513](#)], according to the requirements of the IEC 61508:2010 safety standard. It should eventually isolate the main product-line argument from a specific component selection, thus easing component substitution (e.g., use of an alternate safety hypervisor). Note that arguments of compliant development and integration of a component shall be provided for each candidate variant (e.g., a valid safety certificate for the component emitted by a certification body).
4. *Safety Contracts:* This is a set of pre-built certification arguments that details how a specific component choice satisfies the assumptions of the generic safety argumentation for re-usable DREAMS components.
5. *DREAMS harmonized platform safety argumentation up to SILx:* This is a set of pre-built the IEC 61508:2010 certification arguments for the specific re-usable safety components provided by the DREAMS harmonized platform (i.e., Zynq COTS processor, Xtratum hypervisor, STNoC and the TTEL). Some of these arguments are detailed in deliverables D 5.1.1 [[DRE511](#)], D 5.1.2 [[DRE512](#)] and D 5.1.3 [[DRE513](#)].

Argumentation Check

The **GSN** modular extensions support modelling the variability in an abstract argumentation graph [[Kel07](#)]. For the certification process, we have to instantiate this argumentation, resolving all the options for the variants. Once we get a particular product sample, the **DREAMS SCCRC** component performs sanity checks on the argumentation following these rules:

- Concrete argumentation shall not contain optional elements.
- All the goals shall be supported by strategies.
- All the strategies shall be supported by other goals or solutions.
- At the final development stage, all the related goals, strategies and solutions shall be developed and instantiated.

Generated GSN Argument

The composition of a Safety Case argumentation begins once the DSE has found valid deployments and valid safety architectures for the variants produced by the BVR. The process consists of the steps enumerated below:

1. A pre-condition to run the Safety Case Generator is to provide a library of safety arguments in a form so that they can be processed by a computer program. Therefore, we populate a GSN argumentation-model repository stored in a relational database by using the Enterprise Architect (EA) UML editor [EA]. This model library describes the safety rationale of the DREAMS Modular Safety Cases (MSCs) [Lar17] for the re-usable components in the DREAMS platform as a set of composable GSN argumentation patterns.
2. The first step to generate a product SC is to import the required SC GSN argumentation patterns from the DBMS to the AF3 project. In the example from Figure 8.26 SCCRC created: (1) an empty *Safety Argumentation Package* to contain the argumentation, (2) a 'GSN Root' diagram, (3) a 'Safety Function Architecture' diagram and (4) GSN diagrams for *1oo1*, *1oo1D*, *1oo2* and *1oo2D* architectures.
3. Then, the DREAMS user sets the source and destination folders for the GSN argumentation in the *Safety Compliance Model*, selects the root node in the *Safety Compliance Model* and starts the argumentation process.

Figure 8.27 shows three of those generated GSN model fragments:

- The top fragment is the GSN root diagram 'Safety Case Root', which is an instance of the source GSN Root pattern and does not contain uninstantiated nodes. The bottom left node in this fragment ('G0: Architecture ...') is an *AwayGoal* linking to the GSN fragment in the middle of the Figure.
- The middle GSN fragment is a development of the safety argument that demonstrates the capabilities of the DREAMS argumentation generator:
 - As the original argumentation pattern declared that there may be multiple Safety Functions, in this example the Safety Argument Generator splits the reasoning into two safety argumentation lines.
 - For each argumentation line, the generator continues by choosing the right architecture node in the safety-compliance model (note the '*1-out-of-n*' option node in the pattern). As this example has two Safety Functions, two lines are opened, one for the *SafetyProtectionOverSpeed* function and the other for the *SafetyProtectionOverVoltage* function.

specify their names, the **SIL**, the Systematic Capability and the **HFT** values.

As in the top fragment, bottom nodes in the middle **GSN** fragment are *AwayGoals* linking to other two **GSN** diagrams, one per candidate architecture.

- The bottom **GSN** fragment shows the developed argumentation for the *1oo2D*.

This process ends up linking the argument models to the re-usable **DREAMS** components (e.g., *Hypervisor*, *Safety Partitions*, *COTS Processors*, etc.) for which **DREAMS** provides libraries of **MCSs** as **GSN** models stored in the argumentation database (**DB**). This also applies to application-specific components (e.g., a piece of **SW**) that had been previously granted a safety qualification.

8.3.6 Preliminary Safety-Case Generation

SCCRC outputs a preliminary certification argument model after completion of **DSE**. Besides, **SCCRC** features a reporting tool to dump the rationale of the certification arguments into a suitable document template: the preliminary Safety-Case Report (**SCR**). This way, each safety evaluation carried out by the **SCCRC** component will be detailed in a format amenable to human review, which shall be a mandatory **VVT** activity. The automated generation of the preliminary safety case helps to keep the overall documentation synchronized and eases the completion of the argument with new safety-relevant information collected at later development stages.

This post-processing **SCCRC** feature generates a detailed description of the safety arguments in document form. To this end, **SCCRC** traverses the argumentation model for the feasible product variants and writes a \LaTeX transcript with a pre-defined safety-case template, adapted from [**SCDM06**] (the **SCR** structure is described in Appendix §D.1). Appendix E presents an example **SCR** generated by **DREAMS SCCRC** automatically from a safety argumentation model.

Pre-certification Document Libraries

Across the development process, the certification argument model provided by **SCCRC** will be piecewise completed with other independently developed arguments. Once all the required evidences become available, the enhanced argumentation model eases the compilation of pre-certification document libraries (including the available safety certificates for pre-certified safety components,) through document links annotated in the modular arguments.

Post-DSE Completion of Safety Cases

A mandatory requirement for developing an **MCS** is to implement redundancy in the development process itself, i.e., an independent **VVT** team shall plan, design and execute a number of checks, analysis and tests to demonstrate the correctness of both the safety

8.3. Safety Cases for Wind Turbine Controllers

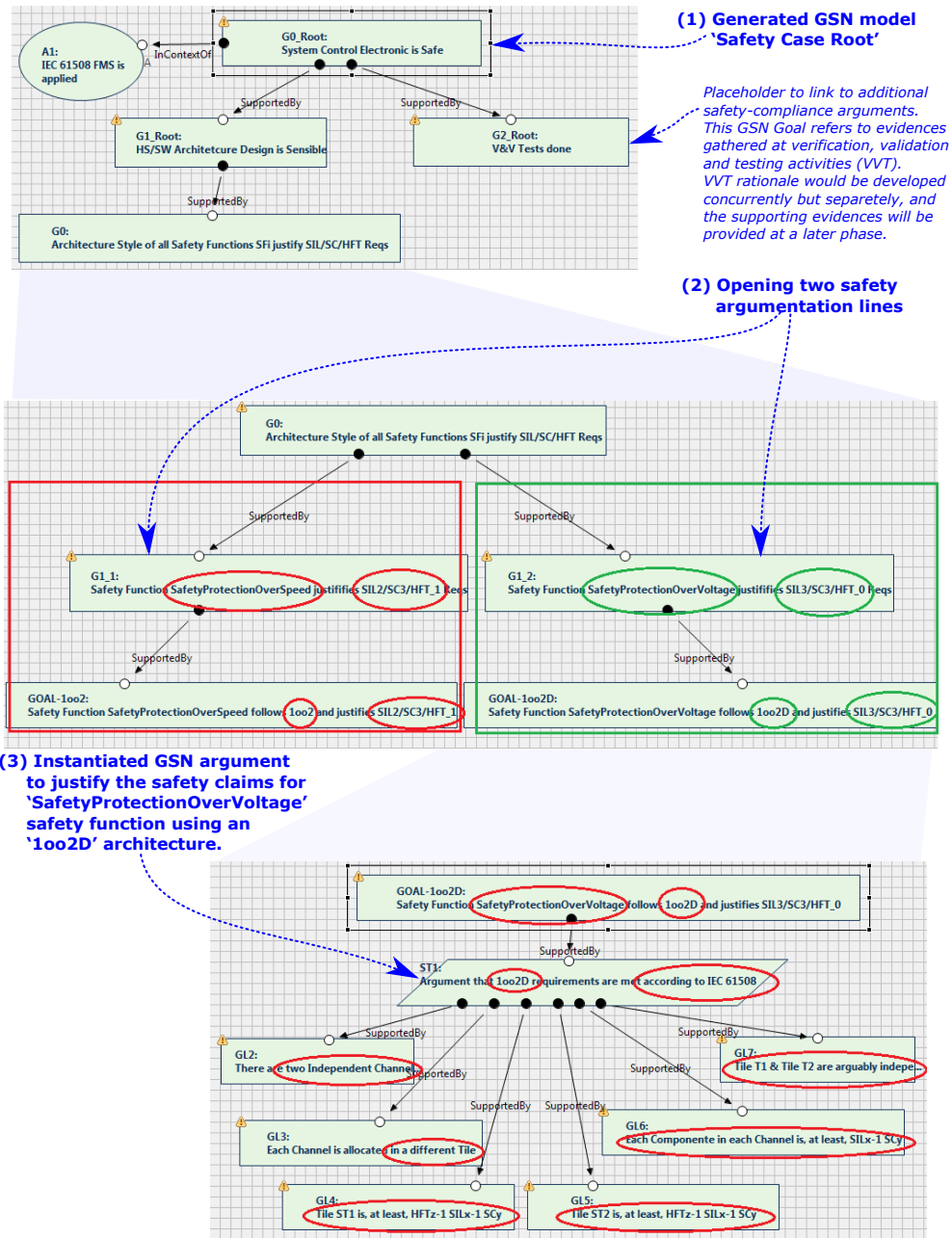


Figure 8.27: Refinement of a GSN safety argument for the DREAMS Controller (source: [DRE433])

product and the methodology. The outcome of these activities would be a set of evidences sufficient to support the safety claims regarding the product.

The rationale about the quality of the planned **VVT** process and the credibility of the resulting evidences (e.g., analysis of validity of test results) would be represented by an additional **GSN** argumentation model. The **VVT** will afterwards interleave their claims and evidences (or counter-evidences) with the **GSN** argumentation model resulting from the **DSE** process. A careful review of the evidence strength shall be carried out, as to effectively demonstrate the validity of the system with regard to the safety requirements.

By using the DREAMS tools and approach **VVT** activities shall provide evidences for, at least, the following arguments:

1. *Verification and Validation of Development Compliance*: An independent verification team shall develop an argumentation about the development-process compliance with the safety-standard requirements, including the aptitude of the staff, satisfaction of training requirements, tool qualification and usage, quality control processes, etc. This argument fragments can be linked to the global safety argument for referencing and reporting purposes.
2. *Verification and Validation on a model or simulator of the MCS*: Independently from the **DSE** optimization, an independent **VVT** team shall plan, design and execute a number of checks, analyses and tests targeting the system properties predicted by the available system models. Ideally, the **VVT** team shall select techniques that would allow re-checking the properties predicted by the models on the actual system, to prevent systematic faults arising from incorrect/over-simplistic system models or inexact model parameters.
3. *Verification and Validation of the MCS Programmable Electronics*: Similarly to the **VVT** activities targeting the models, the bulk of safety evidences will arise from the execution of verification activities on the actual system implementation. These again can be linked to the global safety argument for traceability and reporting purposes.

Some of the above-mentioned evidences will be available in post-**DSE** phases. As a consequence, the preliminary Safety-Case Report (**SCR**) would remain incomplete, or at least, unsupported, until the **VVT** processes yield the required evidence items. Thanks to the automated generation of **SCRs**, the certification documents can be rebuilt automatically once the evidences are referred to within the **GSN** model, then replacing uninstantiated **GSN** Solutions with the newly attached information.

8.3.7 Discussion

DREAMS certification aims at providing a coherent and global rationale about the safety properties of a mixed-criticality product line. To achieve this objective, DREAMS relies on

a global argumentation model that can be incrementally enhanced by different tools and teams, starting from pre-built arguments for re-usable safety components.

The automated argument re-construction and reporting alleviate the huge manual document rework that safety product lines would require otherwise. For instance, Figure 8.28 shows a product certification argument resulting from the composition of the modular argument fragments for the components. Turquoise packages belong to Safety Argumentation for Wind Turbine Demonstrator module, common to all the solution space, while the purple packages, corresponding to Safety Argumentation for Wind Turbine Demonstrator in product sample, are specific to a given product configuration. As the certification argument model may grow up to a very complex structure and thus becomes difficult to apprehend, a mapping shall be defined in order to divide the whole argument into a number of manageable fragments, preserving a similar level of detail.

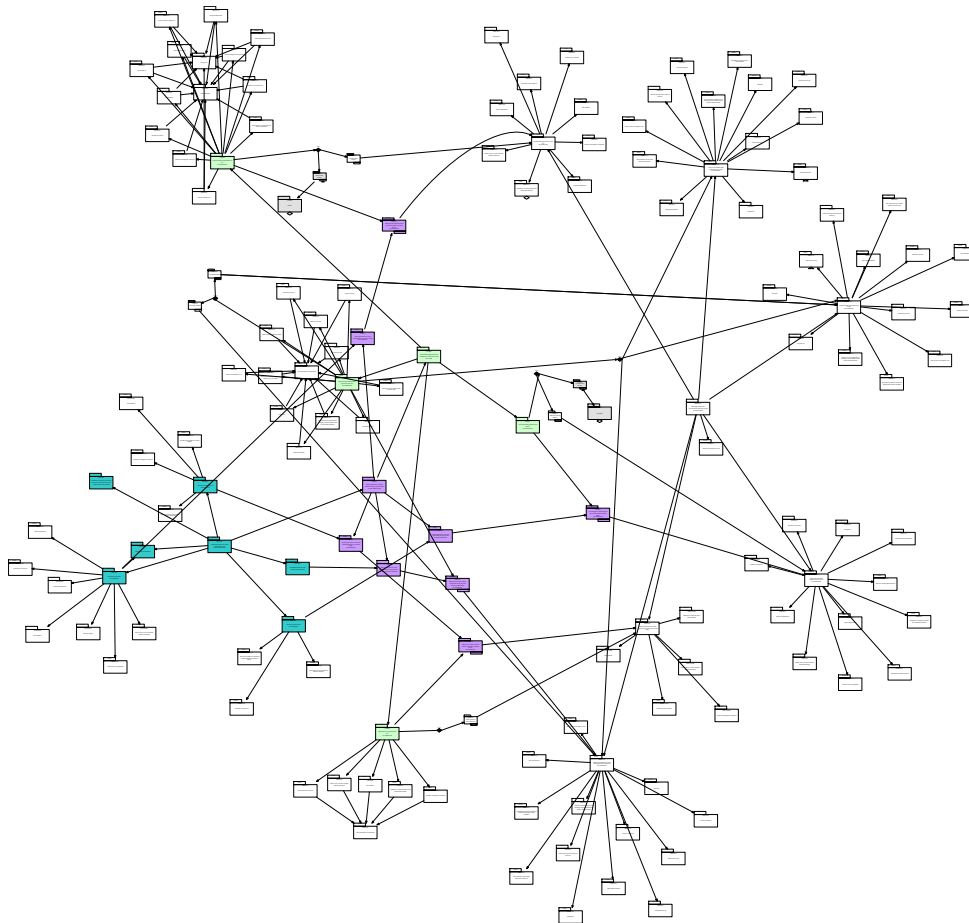


Figure 8.28: Hand-crafted wind turbine GSN certification argument (modules & contracts)

The main challenge was handling an incomplete argumentation structure while splitting and cross-referencing the information according to a sensible documentation structure that shall be defined in the project Functional Safety Management (FSM) procedure which was defined in [DRE561]. To document the safety cases, a mapping from the argument fragments to the FSM documents is required.

To tackle the incremental completion of safety-related information, we could rely on cross-references to existing documents, e.g., risks and faults analysis, or documents to be provided at a later development stage, e.g., compilation of test results and their analysis. To assemble the library of cross-referenced document artefacts we require a shared file system or a configuration-management server, where digital versions of the evidences would be stored as available. A relational DBMS system also provides a common storage point for other development tools. While this suffices for low-complexity products, for higher complex safety systems better scalability would be required. This could be attained by switching to application interfaces enabling a loosely coupled tool framework, e.g., OSLC.

8.3.8 Conclusion

The DREAMS platform-based design supports re-using pre-certified components to deploy mixed-criticality systems. These HW and SW elements enable a partially-automated design-space exploration while easing the generation of the design-rationale documentation as is required by the certification process. To this end, DREAMS provides a collection of safety arguments as a foundation to argue about the satisfaction of the overall safety requirements. The safety-case approach supports modularity for developing product lines where a per-product safety analysis and the justification of compliance are required to certification. Justification includes the linking analyses of the components, the freedom from interferences between the components and the prevention of systematic errors in the development process.

A database of modular certification arguments provides a convenient information arrangement to support the modular composition of safety arguments. Our work shows how this can be even partially automated using the GSN to model the re-usable safety arguments. As an example, we developed the safety arguments for a generic IEC 61508 compliant wind-turbine product line which consists of a DREAMS wind turbine product sample composed of a set of commercial components. Furthermore, we identified several variation points that may extend the modular argumentation database. Those variation points include the variability of safety-related standards (i.e., DO 178C, ISO 26262) and the integrity level of the components (i.e., SIL1 to 4 according to IEC 61508).

Future developments of the argumentation support would include additional attributes to represent the credibility of a given argument. Those attributes will enable capturing the subjective evaluation of the argumentation as done by a certification body. It is noteworthy that gathering this information is a challenging task. However, based on previous safety assessments and experiences with a certification body, a GSN model can represent a valuable asset to detect in advance the weakest link in the argumentation chain before actually facing

the certification process.

Related Publications

The results presented herein relate to the following publications: [[NEL+17](#), [LMN+17](#), [LPN+16](#), [LAN+15](#)].

Part IV

Conclusion

9

Conclusion

This chapter closes the dissertation with the conclusions drawn from the research work. We examine the validity of the initial hypotheses, discuss the limitations of the proposed approaches, and summarize the lessons learnt during the course of this thesis. Finally, we end up posing possible future research paths.

9.1 Review

The product development process for safety-critical applications requires a thorough verification of the system. Some authors identify the verification activities as the main cost factor in many safety-related developments. An early verification process would reduce the development cost by showing up ambiguities and errors in the specification before implementing the actual system, thus avoiding costly iterations during the development.

Model-Based Engineering (**MBE**) sets the verification forward by building models of the system for static and dynamic analyses. Analogously, Model-Based Testing (**MBT**) supports the early design of test cases by using models of the system under test. Moreover, the test suites represent a form of behavioural specification as they illustrate the expected behaviour of the system by test cases. This is an advantage with regard to the enhancement of comprehensibility, understandability and maintainability of the system during its life-cycle. These test cases enable the examination of the candidate system design or implementation by taking snapshots of its behaviour for a set of experiments.

Nowadays safety standards recommend **MBE** for the development of high-integrity systems. **MBE** makes the design amenable to humans by abstracting, partitioning

and layering the design. This constitutes a toolset to tackle with the complexity of currently available electronic platforms. Model-Based Development (MBD) enables the experimentation with a mathematical representation of the system. MBT enables the description of test cases, test components and test architectures in a similar way than MBD does for the system design. By sharing the same modelling language for MBD and MBT, we enable interleaving (e.g., incremental iterative refinement) and model interchange between design and testing. By developing the system and the testing architecture concurrently we can unveil defects earlier and easier at the cross-reviews of the models.

Despite criticism about the feasibility of testing by simulation for complex architectures due to its cost and the reliability of the models, we cannot avoid testing in practice. Alternate approaches promote relying on completely predictable computing infrastructures and programming constructs to achieve predictability, but currently these techniques are not mainstream. Our research proposal focuses on the cost-efficiency of MBT of Dependable Embedded Systems (DESs). We sought a structured verification process where models evolve as we incorporate further details into our examination.

This thesis presented several re-use strategies that aim at reducing the overall cost for verifying and certifying families of embedded dependable systems: (i) test specification re-use by integrating an Automatic Test Executor (ATE) that resembles actual ATEs used for real test-benches into modelling and simulation environments; (ii) model and code re-use to recover the validation costs by using proven-in-use components to build the test architectures; and, (iii) model-based framework to re-use certification arguments supporting the design of certifiable mixed-criticality product lines based on the DREAMS harmonized platforms. This section reviews the achievement of the Operative Goals, the validation of the Hypotheses and the current limitations of the approaches.

9.1.1 Hypotheses Validation

Three hypotheses were formulated for this dissertation (see §4.2). They have been validated with the design, implementation and validation of three contributed approaches: (i) an integrated ATE to automate the validation of fault tolerance on models of safety-critical embedded systems with a Time-Triggered (TT) architecture, (ii) a modelling framework and workflow that yields test components for hardware-in-the-loop (HiL) test-benches, and (iii) an argumentation framework supporting the re-use of Mixed-Criticality Systems (MCSs) and an assisted compilation of Safety Cases (SCs). Herein we examine the actual achievements.

Hypothesis A argues that *“An Automatic Test Executor (ATE) can be integrated in simulation frameworks so that it (i) enables the early analysis of fault-tolerance mechanisms in dependable systems with redundant structures at various levels of abstraction, (ii) automates the simulation of fault-injection experiments, and (iii) facilitates test re-use across system models at different levels of abstraction, and even up to a real test-bench”*.

This thesis motivated the **PS-TTM ATE**, a synchronous Python interpreter integrated in the Platform-Specific Time-Triggered Model (**PS-TTM**) modelling framework. **PS-TTM** is used to build low-complexity executable models of **TT** safety-critical systems at different abstraction levels, assuming that all the logical components stick to Logical Execution Time (**LET**) / Executable Time-Triggered Model (**E-TTM**) models of computation (**MoCs**). The **PS-TTM** simulation relies on the **E-TTM** implementation [PNO+10, Per11], running on top of the SystemC simulation library. The **PS-TTM** simulator also embeds the fault-injection library from [Aye15], which is wrapped in a Python Application Programming Interface (**API**) to the **ATE**. Additional Python **APIs** let the **ATE** control the allocation of test points, as well as the definition of timed sequences of test inputs, i.e., the *stimuli*.

When designing a dependable system with **PS-TTM** in a top-down approach, the first step consists of defining a pure functional model: the Platform-Independent Model (**PIM**). The **PIM** is an abstraction which aims at exercising the logical components against a set of fault scenarios. The purpose of the fault scenarios is to predict the behaviour of the system under the fault hypotheses elicited in the **FMEA** analysis. A test script drives the simulation of fault scenarios in the **ATE**, which guarantees repeatable test results, and also enables the chained execution of multiple simulations.

A second stage is the refinement of the dependable-system model as to resemble the redundant structures used to increase the reliability of the safety system. This phase comes up with a new model, the Platform-Specific Model (**PSM**), that incorporates additional details (e.g., diversified logical components, cross-diagnostics, etc.). A refined **FMEA** analysis yields an extended collection of possible faults. These are transcribed into **XML** fault configurations, that the **PS-TTM ATE** loads to dynamically insert signal saboteurs in the appropriate communication channels defined in the **PS-TTM** model.

The experimental evaluation on the Railway Controller Case Study (§8.1) shows how the simulation performance of the combined **ATE-PS-TTM** simulator achieves a time acceleration of approximately $200x$ for **PIMs**, and $35x$ for **PSMs**. The automation provided by the integrated Python **ATE** supported the simulation of fruitful simulated fault-injection campaigns, that actually highlighted some weaknesses in the *2003* odometry system concept. This demonstrates the achievement of Operative Goal A (§4.3.1), which validates Hypothesis A (i) and (ii). However, (iii) was not feasible, as the development of a realistic test-demonstrator and the cost of equipment required for it were far beyond the possibilities of this research.

Hypothesis B argues that “*Generic models of sensors and instruments can be designed and validated in a Commercial-Off-The-Shelf (COTS) modelling environment, such that these could be re-used to build real-time test components for a COTS heterogeneous HiL system to verify dependable systems, in which the obtained test components may*

execute in parallel, keeping a strict timing synchronisation”.

The Elevator Simulator Case Study shows how Model of Computation (MoC)-agnostic Simulink models can be designed, transformed, and synthesized on-demand to program the Field-Programmable Gate Array (FPGA) of an heterogeneous HiL NI Compact RIO (cRIO) platform. The parallel execution in the Programmable Logic (PL) eased the augmentation of a progressively complex HiL elevator simulator, while the proposed workflow supported a concurrent development by a team of people with different computing backgrounds. The model-based test components included position sensors, plants or special-purpose instruments that were first developed and validated in simulation, using model test-harnesses.

For our test application we tried out a single-model approach instead of a dual-model development (i.e., a platform-independent specification model and a platform-specific implementation model). In both developments we achieved the required real-time simulator performance. The FPGA provided the execution parallelism to add new functions with virtually no interference with the operation of the rest of the HiL simulator. The main advantages of preserving the function of the HiL test system as a set of models were an improved communication across the project team, easier maintenance of the test components, and better portability throughout heterogeneous computing platforms.

Currently we have 2 HiL simulator variants derived from the same Simulink model: cRIO-9082 (FPGA Xilinx Spartan-6 LX150), and cRIO-9039 (FPGA Xilinx Kintex-7 325T). Although the FPGA variants impose different timing constraints, by profiling the transformation settings we got both implementations showing the same real-time behaviour. Nowadays we can modify or enhance the features of the test system and initially validate them by relying on the models of computation supported by the modelling environment. Formerly, the maintenance and validation of former HiL elevator simulators was cumbersome, time-consuming and error-prone: the real-time behaviour of the simulator was confronted with the actual system-under-test that was also under development (and therefore unstable) and, even worse, the constraints of a given HiL computing platform limited the evolution of the test system, requiring at times a major overhauling.

The experimental evaluation on the Elevator Simulator Case Study (§8.2) shows the achievement of Operative Goal C (§4.3.1), which validates Hypothesis B.

Hypothesis C: argues that *“The capabilities of current COTS HiL platforms support the deployment of virtual replicas of nodes in a distributed dependable system, yielding a versatile and cost-effective test architecture through re-using code from actual devices”.*

This research provided a light-weight platform para-virtualisation approach that enables the code re-use of existing embedded application code to build a scalable

Restbus simulator that could be integrated in a **COTS** Heterogeneous Computing Platform (**HCP**) **HiL**. The approach was evaluated on a case-study for improving the testability of an elevator distributed control system, where a Controller Area Network (**CAN**) Restbus simulator replaced multiple remote **I/O** devices with virtual replicas.

We integrated our Restbus simulator into an overall plant **HiL** simulator, where the virtual instances of the simulated nodes were stimulated by **CAN** messages, timing events, signals computed by the plant model and commands from a test execution environment. The real-time Restbus **HiL** simulator enabled a more thorough validation of real-time embedded controllers, as it is software-reconfigurable, reduces the overall cost of the test-bench by replacing external hardware, eases the insertion of node-mutants, allows integration into a **HiL** test infrastructure and improves the test automation.

However, the proposed virtualisation approach has several shortcomings due to the pre-requisites and assumptions regarding the coding and execution flow of the embedded applications. Besides, the technique to share a single communication port between several virtual devices cannot be generalized to every type of network and thus, won't be useful for other kinds of distributed dependable systems, especially in case of using **HW**-generated timestamps to guard the nodes against wrong message sequences.

The experimental evaluation on the Elevator Simulator Case Study (§8.2) shows the achievement of Operative Goal D (§4.3.1). This only constitutes a partial validation of Hypothesis C, due to the limited applicability of the technique.

Hypothesis D: argues that “A model-based argumentation framework enables the compilation of interrelated sets of information to assist safety engineers in developing safety cases for Mixed-Criticality Product Lines (**MCPLs**), following a platform-based design (**PBD**) supported by a set of pre-built Modular Safety Cases (**MSCs**), and where results from analysis, verification and testing activities can be incorporated to the set of supporting evidences required to argue about the safety of a system”.

The experimental evaluation on the Wind-Turbine product line case study (§8.3) shows that the the model-based **DREAMS** argumentation database supports the staged completion of Safety Cases according to the safety requirements for each product variant. The argument models enable the representation of the **MCSs** that conform to the **DREAMS** platform support. Although **DREAMS MSCs** were designed to meet the requirements of the IEC 61508 safety standard, alternative argumentation patterns could be incrementally incorporated to the database, as to fulfil possible different styles required by other safety application domains. The argumentation database provides a fine-grained documentation of the safety properties of the used components.

The experimental evaluation on the Wind Turbine Case Study (§8.3) shows the achievement of Operative Goal E (§4.3.1), which validates Hypothesis D.

9.1.2 Limitations

The first re-use strategy is supported by the developed **ATE** adaptation to the modelling and simulation environments.

1. *Model Re-use Framework for **HiL** test-benches:* Although **FPGAs** are gaining acceptance for computing-intensive applications, the implementation of floating-point arithmetic IPs in an **FPGA** is still very area- and resource-consuming, and also increases the critical timing paths. As a consequence, in many cases it is needed to rely on fixed-point arithmetic replacements which operand bit-lengths and computation time varying with the value ranges for the inputs and outputs. This has the drawback that the fixed-point optimizations eventually depend on the whole computation chain, i.e., each time a signal / parameter-value range potentially affecting the computation chain is modified the fixed-point conversion should be re-worked. Current automated model-to-code transformation technology for **FPGA** targets still do require an assisted adaptation of models including arithmetic operations, which actually yields a second variant of fixed-point models, which would complicate the evolution and maintenance of the model libraries. A possible workaround for Simulink model libraries is to have a **MoC**-agnostic model library (as in the style used in Ptolemy II), i.e., a library of models where the signal data-types and value ranges are inherited from a container model. Container models for desktop simulations would rely on floating-point arithmetic, and would minimize toolbox-dependencies. Container models for **HiL/FPGA** implementations would incorporate fixed-point data-types that the hardware description language (**HDL**) Coder will propagate throughout the **MoC**-agnostic reference mode. Optimizations specific to the **FPGA**, like pipe-lining or the selection of the register bit-lengths for computation Intellectual Property (IPs) will then be applied on the **FPGA**-specific model. This model should be touched-up on range modifications or when deploying to a different **FPGA**.
2. *Issues of **HiL** device virtualisation for other networking technologies:* In the elevator system case study, the para-virtualisation with code re-use enabled the replacement of multiple slave I/O nodes with a **CAN** Restbus simulator using a single standard **CAN** port. In our approach, the illusion of multiple devices communicating through a **CAN** bus was achieved with an algorithm that re-ordered the queue of outgoing messages according to their priority. However, this is not generalizable to every possible networking technology. In particular, some **DES** integrate safety communications that incorporate **TT**-networks, redundant channels, or buses with **HW**-timestamps, and these cannot be replicated this way.
3. *Lack of semantic interpretation of safety-certification arguments:* The current version of the **DREAMS** AutoFOCUS 3 (**AF3**) toolset has no semantic checking of the re-usable argument models stored in the database (**DB**). Except for some completeness

checks, the quality of the Goal Structuring Notation (GSN) models eventually depends on the information put in the database by component designers and safety engineers.

9.2 Lessons Learned

The experimentation with the proposed re-use approaches herein yielded these lessons:

Re-usable test models ease cross-domain collaboration: For the elevator HiL test application we tried out a single-model approach instead of a dual-model development (i.e., a platform-independent specification model and a platform-specific implementation model). The project team consisted of 3 people: a seasoned Simulink user developed the bulk of the Simulink model, a developer with a mid-level background in both Simulink and LabVIEW programming integrated the single-encoder cRIO-9082 variant, and an FPGA expert with programming knowledge about cRIO controllers and LabVIEW synthesized the dual-encoder cRIO-9039 variant. In both developments we achieved the required real-time simulator performance. The FPGA provided the execution parallelism to add new functions with virtually no interference in the operation of the rest of the HiL simulator. The main advantages of preserving the function of the HiL test system as a set of models were: an improved communication across the project team, easier maintenance of the test components, and better portability throughout heterogeneous computing platforms.

Portable para-virtualisation economizes development effort: We integrated our Restbus simulator into an overall plant HiL simulator, where the virtual instances of the simulated nodes were stimulated by CAN messages, timing events, signals computed by the plant model and commands from a test execution environment. The real-time Restbus HiL simulator enabled a more thorough validation of real-time embedded controllers, as it is software-reconfigurable, may reduce the overall cost of the test-bench by replacing external hardware, eases the insertion of node-mutants, allows integration into a HiL test infrastructure and enhances test automation. Moreover, after completing the initial CAN Restbus simulator, the software (SW) was branched to support the development of a replacement for a legacy communication library that had to be integrated in a different device. The SW modifications were extensively tested using the simulator, this time using the elevator controller as a part of the test environment. In this phase no In-Circuit Emulator (ICE) or special instrumentation HW was initially required: the desktop simulator was the prototyping platform and the original test setup consisted of actual elevator components interoperable with the system under development. Once validated, the new library release was compiled for the target Micro-controller Unit (MCU) and underwent integration testing: at this stage virtually no problems were detected, actually reducing the integration effort to one fifth of the initially foreseen workload.

Enforcing argumentation models improves knowledge-sharing across a team:

Building Mixed-Criticality Systems on top of modular platforms like the **DREAMS** heterogeneous **HW / SW** is hindered by the complexity of the possible interactions, the number of developers involved, the required compliance with a predefined Functional Safety Management (**FSM**) protocol, and the exhaustive documentation required by the certification processes. Modular Safety Cases (**MSCs**) provide re-usable arguments to support the safety claims in a range of diverse applications. To that purpose, the **MSC** has to be tailored, i.e., instantiated, to suit the safety requirements for a product configuration. Then, the particularized **MSC** is merged into a connected chain of reasoning to argue about the validity of the safety claims. Argument models provide an infrastructure to compose **MSCs** with complementary information.

The **GSN** modelling language is currently accepted by several certification bodies as a technique to structure the argumentation. The safety argumentation integrates all the information about the product itself, but also about how it was developed, how it was validated and verified, etc. Thus a product safety case derives from a connected information network provided by possibly independent sources: the project management, the application experts, the developers and the verification and validation staff. In this context, the enforcement of argumentation patterns to support a partial automation of the Design Space Exploration (**DSE**) also benefits the development: defined boundaries between argument modules, abstractions (e.g., *to be instantiated, undeveloped*), and a clear structure to support or challenge a claim iteratively enable the completion of the argument as information becomes available.

In the **DREAMS** project we opted for a relational database server to store these re-usable assets. Without loss of generality, we completed an Unified Modelling Language (**UML**) meta-model extension to design modular **GSN** models using a standardized **HMI** front-end: Enterprise Architect from Sparx Systems. A distinctive feature of EA is that it supports Database Management System (**DBMS**) storage of models. This also improves usability and enables a concurrent completion of the **GSN** models.

A fully automated model-based safety documentation is challenging: Although

some researchers propose using **GSN** models for the formal analysis of arguments, in the safety domain there are somehow subjective criteria, e.g., “expert judgement”, that can be hardly expressed mathematically. In some application domains position papers and explanatory guidelines aim at addressing ambiguities, e.g., [**CAP760**, **CAST**, **TR50506-2**, **IEC62741**]. However, an automated safety reporting process does not guarantee the readability of the generated documents: this relates to an ‘adequate’ level of detail, as well as a ‘reasonable’ partitioning of the information that ultimately depend on the subjective criteria of the documentation reviewers. Due to this, the **DREAMS** argumentation framework sought a limited automation at generating product safety-cases. Instead, it provides basic sanity

checks that support the extraction of snapshots to show the actual progress and the pending evidences and a programmable safety-case documentation tool that could be tailored to suit the safety-documentation structure defined in the **FSM** procedures.

9.3 Future Work

The contribution from this dissertation could be further extended in future research directions, such as:

- *Integration of Virtual Testing and Virtual Platforms on PS-TTM:* Virtual platform simulators can reproduce the functional and timing behaviour of **HW** / **SW** systems by exercising the intended **SW** against models of the **HW** components. Platform simulators like **OVP** [**OVP**] or **QEMU** [**QEMU**] provide libraries of models for different processors and peripherals, which developers can exploit to analyse the system feasibility at different levels of abstraction and simulation accuracy. Virtualisation of a complex system including **HCPs** may include cross-platform Instruction-Set Simulators (**ISSs**) and custom models of peripherals and interconnects (e.g., SystemC models). On one hand, developers can exercise an unmodified **SW**-stack of a target system, even for mixed-processor architectures, using the **ISSs** simulator as a debugging platform to accelerate the development. On the other hand, the test engineers can customize **HW**-models to perform fault injection at low abstraction levels. Some safety-**MCUs** integrate fault-injection to assess the fault tolerance of the dependable system. This increases the potential to re-use test artefacts, as the same fault-injection tests could be likely exercised on either the platform simulator or on the actual **HW**.
- *Integration of cross-platform Instruction-Set Simulators (ISSs) in real-time (RT) HiL environments:* A common feature of Virtual Platform Simulators is the virtualisation of the memory. Memory virtualisation provides separate execution contexts for each instance of the virtual processor. The execution of a **SW** stack by an **ISS** becomes immune to the memory usage in the hosted application, overcoming the limitations of the lightweight para-virtualisation approach. The integration of **ISSs** within the **HiL** platform would ease the re-use of validated **SW** components to build real-time test systems, while maintaining the versatility of a **SW**-defined Restbus simulator.
- *Parallel PS-TTM simulator:* The Railway Controller case-study served as a benchmark for the simulation times for Platform Independent Time-Triggered Model (**PI-TTM**) and **PSM** models. The **PS-TTM** model mimics the redundant structure of the system, increasing both, the computation burden and the number of possible fault scenarios to examine. This reduces the time-acceleration factor, lengthening the time to complete a fault-simulation campaign. Simulation time gets even longer in case of including components modelled at an **RTL** level of detail, for which a single-threaded SystemC

simulation would become impractical. An improved **PS-TTM** would rely on a parallel-SystemC engine [ECC+09,SLP+10,DCH+11], exploiting the multi-core architectures to increase simulation performance and reduce the validation costs.

- *Parallel Python testing:* Functional testing of dependable platforms usually requires the synchronous observation and correlation of multiple inputs in the test architecture to evaluate the actual performance of a given implementation under test. Conventional testing relies on sequential programming to define the test procedures, but complex test architectures would require the coordinated operation of multiple processors, which also have to be programmed. From a maintenance perspective it is desirable to keep a homogeneous test programming environment. The Python language has been adopted for Big Data analysis, yielding parallel Python implementations that also could be adapted to control functional test-benches [Hin07, PMD16].
- *Stateless Simulink models to customize parallelism in Programmable Logic:* Heterogeneous Computing Platforms for hardware-in-the-loop (**HiL**) test systems ease the execution of test components at different sampling rates, typically deploying low-frequency logic on computing cores while the high-frequency elements get synthesized in Programmable Logic (**PL**). Sometimes the **HiL** plant model requires multiple instances of a subsystem model in the **PL**. Instantiation is straightforward when the subsystem model is transformed to sequential code and deployed in a conventional processor or core, as by using dynamic memory allocation the number of component instances is defined at run-time (provided that computing power and memory resources suffice). However, when the component model targets the **PL** there are a number of challenges:
 - Each instance of a subsystem model translates to **HDL** code that will end-up owning computational resources. Thus, the higher the number of instances, the more **PL** area will be required by the plant model. While this is optimal for timing performance, it could be disadvantageous for arithmetic-intensive subsystems, as the number of available **FPGA** resources is constrained (e.g., **DSPs**), eventually limiting the maximum number of feasible subsystem instances.
 - Simulink (**SL**) library models favour data encapsulation to preserve the model abstraction and enhance the model comprehensibility. But data encapsulation implicitly adds internal memory registers to store the state of each block / subsystem instance (e.g., as in a unit-delay block). The hidden memory registers in **SL** blocks make the customization of parallelism in **PL** implementations difficult, as there is no direct control on the memory resources owned by each subsystem instance. Therefore conventional Simulink blocks may hinder the parallelization of the subsystem computations using alternative scheduling algorithms (e.g., with some C/C++ to **HDL** translators we can customize the parallelism of loops).

In some test scenarios the effective sampling rate of multiple-instance subsystems could be relaxed (i.e., when the required sampling time is several times the propagation time of the critical path in the **FPGA**). This was the case in the Hardware-in-the-Loop Elevator Simulator (**HiLES**) simulator, where the number of instances that could be deployed in a single **FPGA** was constrained by the allocation of Digital Signal Processor (**DSP**) blocks to the computation **IP**, while there was a surplus of unused memory blocks. *Stateless **SL** models* would enable an optimized trade-off between the usage of **DSP** and memory resources in the **FPGA**, by extracting all state memory to a **R/W** external register. Instantiation would be realized by switching the external memory at each update cycle and sampling and holding the **I/O** registers to preserve the time semantics. With this structure, a reduced number of arithmetic **IPs** would be required, freeing **DSP** resources that could be reserved for a future scaling-up of the plant model. Stateless models would preserve artefact re-usability from model-in-the-loop (**MiL**) to **HiL** test configurations, and offer additional customization to optimize resource usage in the **HiL** platform

- *Extended argumentation framework for other safety standards:* This dissertation contributed an argumentation framework to derive product safety-cases from a library of Modular Safety Cases (**MSCs**) compliant to the IEC 61508 safety standard. The adaptation and re-use of the **MSCs** library is straightforward for domain-specific safety standards rooted in IEC 61508, like ISO 26262 (automotive), EN 5012X (railway) or ISO 13849 (machinery). However, safety standards from other application domains (e.g., DO-178C for airborne systems) would require different argumentation models and a specific documentation structure. Future work may extend the argument database with additional argument fragments suited for those safety-standards.
- *Extended argumentation framework to other safety-related aspects:* While safety aims at protecting a system against unintended hazards, security aims at protecting a system against intended threats. Nowadays security is a requirement for safety-systems, specially for Mixed-Criticality Systems (**MCSs**) where non-safety components can connect to uncertain environments. In the security domain we find a modular approach similar to the safety case: the *security case*. Currently the re-usable argumentation framework supports the compilation of partial safety arguments to elaborate safety cases. A future extension to the argumentation framework can also incorporate security cases as **GSN** argumentation models [WLG07, AHK11], where the shared database approach enables linking safety arguments to security arguments. This eases the analysis and documentation of a complete chain of safety and security arguments resulting from the composition of a complex **MCS**.

Part V

Bibliography and Appendix

Bibliography

- [Accellera] "Accelera Systems Initiative," <http://www.accelera.org/home/>, accessed: 2014/10/07. 24
- [ACDM16] "Airport collaborative decision making (ACDM) safety case guidance material," <https://www.EUROCONTROL.int/sites/default/files/publication/files/20160204-apt-a-cdm-safety-case-guidance-material-v2.0.pdf>, 2016, EUROCONTROL. 272
- [CAE] Adelard, "Claims, Arguments and Evidence (CAE)." [Online]. Available: <http://www.adelard.com/asce/choosing-asce/cae.html> 29
- [AF3] AutoFOCUS 3. <http://af3.fortiss.org/>. Fortiss GmbH. 120
- [AHK11] R. Alexander, R. Hawkins, and T. Kelly, "Security Assurance Cases: Motivation and the State of the Art," High Integrity Systems Engineering Group, Department of Computer Science, University of York, UK, Tech. Rep., 2011. 227
- [ALR+04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, 2004, DOI: 10.1109/TDSC.2004.2. 46
- [ANP+14a] I. Ayestaran, C.-F. Nicolas, J. Perez, A. Larrucea, and P. Puschner, "A novel modeling framework for time-triggered safety-critical embedded systems," in *Proceedings of the 2014 Forum on Specification and Design Languages (FDL)*, vol. 978-2-9530504-9-3. IEEE, Oct 2014, Conference Paper, pp. 1–8, DOI: 10.1109/FDL.2014.7119343. 74, 169
- [ANP+14d] —, *A Simulated Fault Injection Framework for Time-Triggered Safety-Critical Embedded Systems*, ser. Lecture Notes in Computer Science. Springer, May 2014, vol. 8666, pp. 1–16, DOI: 10.1007/978-3-319-10506-2_1. 152, 153, 157, 160, 162, 166, 167, 169, 248, 250
- [ANP+14b] —, "Modeling and Simulated Fault Injection for Time-Triggered Safety-Critical Embedded Systems," in *2014 IEEE 17th International Symposium on Real-Time Distributed Computing (ISORC)*, vol. 978-1-4799-4430-9. IEEE, May 2014, Conference Paper, pp. 180–187, DOI: 10.1109/ISORC.2014.9. 165, 169
- [ANP+14c] I. Ayestaran, C.-F. Nicolas, J. Perez, and P. Puschner, "Modeling Logical Execution Time based Safety-critical Embedded Systems in SystemC," in *2014 3rd Mediterranean Conference on Embedded Computing (MECO)*. IEEE, June 2014, pp. 77–80, DOI: 10.1109/MECO.2014.6862662. 74, 169
- [AS5506A] "SAE Architecture Analysis and Design Language (AADL)," 2009, SAE Aerospace. 6
- [ASAMHiL] "ASAM AE HiL Programmer's Guide," p. 162, 2009, ASAM HiL Workgroup. 46, 73
- [ASAMXiL] (2013) ASAM XiL 2.0. <https://wiki.asam.net/display/STANDARDS/ASAM+XiL/>. 55, 73
- [ASCE] "ASCE," <http://www.adelard.com/asce/user-group/4-Jun-2015>, Adelard. 30, 126
- [Aye15] I. Ayestaran, "Simulated Fault Injection for Time-Triggered Safety-Critical Embedded Systems," PhD thesis, Faculty of Informatics, Vienna University of Technology, Austria, 2015. 8, 10, 71, 159, 168, 169, 219, 247, 251, 252, 255

- [Bau11] C. Baumann, T. Bormer, H. Blasum, and S. Tverdyshev, "Proving Memory Separation in a Microkernel by Code Level Verification," in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. IEEE, March 2011, pp. 25–32, DOI: [10.1109/ISORCW.2011.14](https://doi.org/10.1109/ISORCW.2011.14). 2
- [BBF+08] G. Beltrame, C. Bolchini, L. Fossati, A. Miele, and D. Sciuto, "ReSP: A non-intrusive Transaction-Level Reflective MPSoC Simulation Platform for design space exploration," in *2008 Asia and South Pacific Design Automation Conference*, March 2008, pp. 673–678, DOI: [10.1109/ASPDAC.2008.4484036](https://doi.org/10.1109/ASPDAC.2008.4484036). 46
- [BBL+05] P. Baker, P. Bristow, C. Jervis, D. King, R. Thomson, B. Mitchell, and S. Burton, "Detecting and resolving semantic pathologies in UML sequence diagrams," in *10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*. Lisbon, Portugal: ACM, 2005, pp. 50–59, DOI: [10.1145/1081706.1081716](https://doi.org/10.1145/1081706.1081716). 20
- [BCG+97] F. Balarin, M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki, and B. Tabbara, *Hardware-software co-design of embedded systems: the POLIS approach*. Kluwer Academic Publishers, 1997. 2, 74
- [BEG+15] O. Bringmann, W. Ecker, A. Gerstlauer, A. Goyal, D. Mueller-Gritschneider, P. Sasidharan, and S. Singh, "The Next Generation of Virtual Prototyping: Ultra-fast Yet Accurate Simulation of HW/SW Systems," in *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, ser. DATE '15. IEEE, 2015, pp. 1698–1707, DOI: [10.7873/DATE.2015.1105](https://doi.org/10.7873/DATE.2015.1105). 57
- [Ber07] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," in *Future of Software Engineering 2007 (FOSE'07)*. IEEE, 2007, pp. 85–103, DOI: [10.1109/FOSE.2007.25](https://doi.org/10.1109/FOSE.2007.25). 48
- [BG09] A. Bandyopadhyay and S. Ghosh, "Test Input Generation Using UML Sequence and State Machines Models," in *Software Testing Verification and Validation, 2009. ICST '09. International Conference on*. IEEE, 2009, pp. 121–130, DOI: [10.1109/ICST.2009.23](https://doi.org/10.1109/ICST.2009.23). 20
- [BGB+08] J. C. Baraza, J. Gracia, S. Blanc, D. Gil, and P. J. Gil, "Enhancement of Fault Injection Techniques Based on the Modification of VHDL Code," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, no. 6, pp. 693–706, 2008, DOI: [10.1109/TVLSI.2008.2000254](https://doi.org/10.1109/TVLSI.2008.2000254). 46
- [BGL+07] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "A subset of precise UML for model-based testing," 2007, DOI: [10.1145/1291535.1291545](https://doi.org/10.1145/1291535.1291545). 20
- [BKZ+11] M. Broy, K. Helmut, J. Zimmermann, and S. Kirstan, "Model-based software development - its real benefit," 2011. [Online]. Available: <http://www.automotivedesign-europe.com/en/model-based-software-development-its-real-benefit.html> 5
- [BKS+10a] C. Buckl, A. Knoll, I. Schieferdecker, and J. Zander-Nowicka, "Model-Based Analysis and Development of Dependable Systems," in *Model-Based Engineering of Embedded Real-Time Systems*, ser. Lecture Notes in Computer Science (LNCS), H. Giese, G. Karsai, B. Rumpe, and B. Schätz, Eds. Berlin Heidelberg New York: Springer, 2010, pp. 271–293, DOI: [10.1007/978-3-642-16276-3](https://doi.org/10.1007/978-3-642-16276-3). 6
- [BMS08] C. Bolchini, A. Miele, and D. Sciuto, "Fault Models and Injection Strategies in SystemC Specifications," in *Digital System Design (DSD) 2008, 11th EUROMICRO Conference on*. IEEE, 2008, pp. 88–95, DOI: [10.1109/DSD.2008.35](https://doi.org/10.1109/DSD.2008.35). 46
- [Boe11] Boehm, Barry, *Some Future Software Engineering Opportunities and Challenges*, 1st ed. Springer, 2011, ch. 1, pp. 1–32, DOI: [10.1007/978-3-642-15187-3_1](https://doi.org/10.1007/978-3-642-15187-3_1). 44

- [BP03] A. Benso and P. Prinetto, *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation*. Kluwer Academic Publishers, 2003. 46
- [Bro11] M. Broy, “Seamless Method- and Model-based Software and Systems Engineering,” in *The Future of Software Engineering*, 1st ed., S. Nanz, Ed. Springer, 2011, pp. 33–47. 6, 54
- [BTZ05] J.-F. Boland, C. Thibeault, and Z. Zilic, “Using MATLAB and Simulink in a SystemC Verification Environment,” in *Design and Verification Conference (DVCon’05)*, 2005. [Online]. Available: <http://www.iml.ece.mcgill.ca/people/professors/zilic/DVCon05.pdf> 20
- [Bul16] Bull, Stephen, “Improving European Aviation Safety Approvals,” 2016. [Online]. Available: https://WWW.ascos-project.eu/downloads/ascos_paper_bull.pdf 60
- [BUSM] Busmaster. <http://rbeietas.github.io/busmaster/>. 56
- [CEN11] CENELEC, “EN 50128: Railway Applications — Software for Railway Control and Protection Systems,” 2011. 8
- [CAST] Certification Authorities Software Team (CAST). CAST Position Papers. [Online]. Available: http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/ 224
- [CF09] M. Conrad and I. Fey, “Testing Automotive Control Software,” in *Automotive Embedded Systems Handbook*, N. Navet and F. Simonot-Lion, Eds. CRC Press, 2009. 41
- [CFS05] M. Conrad, I. Fey, and S. Sadeghipour, “Systematic Model-Based Testing of Embedded Automotive Software,” *Electronic Notes in Theoretical Computer Science*, vol. 111, p. 13–26, 2005, DOI: 10.1016/j.entcs.2004.12.005. 39
- [CiA406] “CiA 406 version 3.2.0: Device profile for encoders,” <https://www.can-cia.org/can-knowledge/canopen/cia406/>, 2006, CAN in Automation (CiA). 101, 102, 175
- [CiA301] “CiA 301 version 4.2.0: Application layer and communication profile,” 2011, CAN in Automation (CiA). 101, 102, 175
- [CiA417] “CiA 417 version 2.0.0: Application profile for lift control systems,” 2011, CAN in Automation (CiA). 101, 175
- [CAP760] Civil Aviation Authority (CAA), “Guidance on the Conduct of Hazard Identification, Risk Assessment and the Production of Safety Cases,” 2010. [Online]. Available: <http://publicapps.caa.co.uk/docs/33/CAP760.pdf> 30, 126, 224
- [CMK08] M. Chen, P. Mishra, and D. Kalita, “Coverage-driven automatic test generation for UML activity diagrams,” in *Great Lakes symposium on VLSI (GLSVLSI’08), Proceedings of the 18th ACM*. ACM, 2008, DOI: 10.1145/1366110.1366145. 20
- [Con07] M. Conrad, “Using Simulink and Real-Time Workshop Embedded Coder for IEC 61508 Applications. White Paper ,” Aug 2007 2007. [Online]. Available: <http://www.safetyusersgroup.com/documents/AR070002/EN/AR070002.pdf> 20
- [Con08] —, “Model-Based Testing of Embedded In-Vehicle Software: An Overview,” *AST Magazine*, vol. 2, no. 1, pp. 20–35, 2008. [Online]. Available: <http://www.associationforsoftwaretesting.org/drupal/June.2008.pdf> 48, 49
- [DAN+13] D. Dasari, B. Akesson, V. Nelis, M. A. Awan, and S. M. Petters, “Identifying the sources of unpredictability in COTS based multi-core systems,” in *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE, 2013, Conference Proceedings, pp. 39–48, DOI: 10.1109/SIES.2013.6601469. 2
- [DGJ+12] Dardar, R. and Gallina, B. and Johnsen, A. and Lundqvist, K. and Nyberg, M., “Industrial Experiences of Building a Safety Case in Compliance with ISO 26262,” in *Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on*. IEEE, 2012, pp. 349–354, DOI: 10.1109/ISSREW.2012.86. 124

- [DCH+11] R. Dömer, W. Chen, X. Han, and A. Gerstlauer, "Multi-core parallel simulation of System-level Description Languages," in *16th Asia and South Pacific Design Automation Conference (ASP-DAC 2011)*. IEEE, 2011, pp. 311–316, DOI: [10.1109/ASPDAC.2011.5722205](https://doi.org/10.1109/ASPDAC.2011.5722205). 226
- [DPH11] Denney, Ewen and Pai, Ganesh and Habli, Ibrahim, "Towards Measurement of Confidence in Safety Cases," in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. IEEE, 2011, pp. 380–383, DOI: [10.1109/ESEM.2011.53](https://doi.org/10.1109/ESEM.2011.53). 30, 126
- [DPP12] Denney, Ewen and Pai, Ganesh and Pohl, Josef, "AdvoCATE: An Assurance Case Automation Toolset," in *Computer Safety, Reliability, and Security. SAFECOMP 2012*, ser. LNCS, vol. 7613. Springer, 2012, DOI: [10.1007/978-3-642-33675-1_2](https://doi.org/10.1007/978-3-642-33675-1_2). 30, 126
- [Dij68] E. W. Dijkstra, "A constructive approach to the problem of program correctness," *BIT Numerical Mathematics*, vol. 8, pp. 174–186, 1968, DOI: [10.1007/BF01933419](https://doi.org/10.1007/BF01933419). 41
- [NS10] M. Di Natale and A. Sangiovanni-Vincentelli, "Moving From Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools," *Proceedings of the IEEE*, vol. 98, no. 4, pp. 603–620, 2010, DOI: [10.1109/JPROC.2009.2039550](https://doi.org/10.1109/JPROC.2009.2039550). 1, 2
- [DOL] Distributed Operation Layer (DOL). <http://www.tik.ee.ethz.ch/~shapes/dol.html>. 6
- [DREAMS] DREAMS, "DREAMS - Distributed real-time architecture for mixed-criticality systems," 2013. [Online]. Available: <http://www.uni-siegen.de/dreams/home/> 120
- [DRE551] DREAMS Consortium, "DREAMS Deliverable D5.5.1: State-of-the art of piecewise certification of mixed-criticality systems," EU, Tech. Rep., 2014. 127
- [DRE412] —, "DREAMS Deliverable D4.1.2 Definition of offline adaptation strategies for mixed criticality and initial implementation," EU, Tech. Rep., 2015. 130, 131
- [DRE431] —, "DREAMS Deliverable D4.3.2 Variability Analysis and Testing Techniques for Mixed-Criticality Systems," EU, Tech. Rep., 2015. 141
- [DRE511] —, "DREAMS Deliverable D5.1.1: Modular safety-case for hypervisor," EU, Tech. Rep., 2015. 29, 137, 205
- [DRE512] —, "DREAMS Deliverable D5.1.2 Modular safety-case for selected COTS multicore processors," EU, Tech. Rep., 2015. 29, 137, 205
- [DRE513] —, "DREAMS Deliverable D5.1.3 Modular safety-case for selected mixed-criticality networks," EU, Tech. Rep., 2015. 29, 137, 205
- [DRE721] —, "DREAMS Deliverable D7.2.1: Wind power demonstrator," EU, Tech. Rep., 2015. 193, 196, 197, 198
- [DRE433] —, "DREAMS Deliverable D4.3.3 Final implementation and improvement of variability analysis and testing techniques for mixed critical systems," EU, Tech. Rep., 2016. 126, 200, 201, 202, 207, 209
- [DRE541] —, "DREAMS Deliverable D5.4.1: Guidelines for process and tool integration," EU, Tech. Rep., 2016. 131
- [DRE553] —, "DREAMS Deliverable D5.5.3: Method for certifying mixed-criticality product lines," EU, Tech. Rep., 2016. 133
- [DRE561] DREAMS Consortium, "DREAMS Deliverable D5.6.1: Functional safety management for DREAMS architecture," EU, Tech. Rep., 2017. 212
- [EA] Enterprise Architect UML modeling tool. Sparx Systems. [Online]. Available: <http://www.sparxsystems.com/products/index.html#corp> 132, 206

- [ECC+09] P. Ezudheen, P. Chandran, J. Chandra, B. P. Simon, and D. Ravi, "Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines," in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS '09. IEEE Computer Society, 2009, pp. 80–87, DOI: [10.1109/PADS.2009.25](https://doi.org/10.1109/PADS.2009.25). 226
- [EJ09] C. Ebert and C. Jones, "Embedded Software: Facts, Figures, and Future," *Computer*, vol. 42, no. 4, pp. 42–52, 2009, DOI: [10.1109/MC.2009.118](https://doi.org/10.1109/MC.2009.118). 2
- [LIMAX] ELGO, "LIMAX02 Shaft Information System," 2013. [Online]. Available: <http://www.elgo.de/en/products/sensors/limax02.html> 101, 175
- [EN50129] EN, "Railway applications. Communication, signalling and processing systems. Safety related electronic systems for signalling." 2005. 149, 270
- [OPE23] Espinoza, Huáscar and Ruiz, Alejandra and Larrucea, Xabier and Campos, Sergio and Borgers, Erik and van der Zee, Sven and Rommes, Eelco, "OPENCROSS Platform Architecture," OPENCROSS Consortium, Tech. Rep., 2012. [Online]. Available: http://www.opencross-project.eu/sites/default/files/D2_3_OPENCROSS_Platform_Architecture_final.pdf 59
- [EZW] EZwave. http://www.mentor.com/products/ic_nanometer_design/analog-mixed-signal-verification/ezwave/. 79
- [FCM+12] Ferrari, Alberto and Carloni, Marco and Mignogna, Alessandro and Menichelli, Francesco and Ginsberg, David and Scholte, Eelco and Nguyen, Dang, "Scalable virtual prototyping of distributed embedded control in a modern elevator system," in *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*. IEEE, 2012, pp. 267–270, DOI: [10.1109/SIES.2012.6356593](https://doi.org/10.1109/SIES.2012.6356593). 56
- [FKM11] Y.-Y. Fan Jiang, J. Kuo, and S.-P. Ma, "An Embedded Software Modeling and Process by Using Aspect-Oriented Approach," *Soft. Eng. and Applications, J. of*, vol. 4, no. 2, p. 16, Apr. 2011, DOI: [10.4236/jsea.2011.42012](https://doi.org/10.4236/jsea.2011.42012). 121
- [FMI] Functional Mock-up Interface (FMI). <http://fmi-standard.org/>. Accessed: 2017/05/20. 262
- [FWI+14] V. Fernández, E. Wilpert, H. Isidoro, C. Ben Aoun, and F. Pêcheux, "SystemC-MDVP Modelling of Pressure Driven Microfluidic Systems," in *3rd Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 2014, DOI: [10.1109/MECO.2014.6862665](https://doi.org/10.1109/MECO.2014.6862665). 27
- [GBG+01] J. Gracia, J. C. Baraza, D. Gil, and P. J. Gil, "Comparison and Application of different VHDL-Based Fault Injection Techniques," in *Defect and Fault Tolerance in VLSI Systems, 2001. Proceedings. 2001 IEEE International Symposium on*. IEEE, 2001, pp. 233–241, DOI: [10.1109/DFTVS.2001.966775](https://doi.org/10.1109/DFTVS.2001.966775). 46
- [GD06] A. L. Guennec and B. Dion, "Bridging UML and Safety-Critical Software Development Environments," in *ERTS2006. 3rd Embedded Real Time Software Conference, 2006*, Toulouse, 2006. 20
- [GGR+14] M. Glaß, S. Graf, F. Reimann, and J. Teich, *Design and Evaluation of Future Ethernet AVB-Based ECU Networks*. New York, NY: Springer, 2014, pp. 205–220, DOI: [10.1007/978-1-4614-3879-3_12](https://doi.org/10.1007/978-1-4614-3879-3_12). 57
- [Ghe06] F. Ghenassia, *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, 2006. 26
- [GMS10] B. Güldali, M. Mlynarski, and Y. Sancar, "Effort Comparison for Model-Based Testing Scenarios," in *Software Testing, Verification, and Validation Workshops, 3rd Int. Conf. on (ICSTW 2010)*. Paris, France: IEEE, 2010, pp. 28–36, DOI: [10.1109/ICSTW.2010.15](https://doi.org/10.1109/ICSTW.2010.15). 51, 52

- [Gro05] H.-G. Gross, *Component-Based Software Testing with UML*. Berlin Heidelberg: Springer, 2005. 20
- [GSN] "Goal Structuring Notation (GSN)," http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf, 2011, GSN Working Group. 29, 265
- [GTKW] "GTKWave," <http://gtkwave.sourceforge.net/>, Accessed: 2014/10/23. 79
- [HK10] Habli, Ibrahim and Kelly, Tim, "A Safety Case Approach to Assuring Configurable Architectures of Safety-Critical Product Lines," 2010, DOI: 10.1007/978-3-642-13556-9_9. 30, 126, 265, 268
- [Ham03] R. Hammett, "Flight-critical distributed systems: design considerations [avionics]," *IEEE Aerospace and Electronic Systems Magazine*, vol. 18, no. 6, pp. 30–36, June 2003, DOI: 10.1109/MAES.2003.1209588. 1
- [Har08] M. J. Harrold, "ICST 2008 Keynote: Testing of Evolving Software: Achievements, Challenges, and Promises," in *ICST 2008. 1st International Conference on Software Testing Verification and Validation, 2008*, Lillehammer, Norway, 2008. [Online]. Available: <http://www.cs.colostate.edu/icst2008/MaryJeanKeynote.pdf> 53
- [HDL] "HDL Coder," MathWorks. [Online]. Available: <http://www.mathworks.com/products/hdl-coder/> 96, 108
- [HetSC] "HetSC," <http://www.teisa.unican.es/HetSC/>, accessed: 2014/07/14. 27
- [HHK03] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: a time-triggered language for embedded programming," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 84–99, 2003, DOI: 10.1109/JPROC.2002.805825. 6
- [HINCEP] "Catrene (CA701) H-Inception," <https://www-soc.lip6.fr/trac/hinception>, accessed: 2014-07-24. 27
- [Hin07] K. Hinsin, "Parallel Scripting with Python," *Computing in Science Engineering*, vol. 9, no. 6, pp. 82–89, Nov 2007, DOI: 10.1109/MCSE.2007.117. 226
- [HM13] S. Hutchesson and J. McDermid, "Trusted Product Lines," *Inf. Softw. Technol.*, vol. 55, no. 3, pp. 525–540, 2013. 30, 126
- [HON08] K. Hylla, J.-H. Oetjens, and W. Nebel, "Using SystemC for an extended MATLAB/Simulink verification flow," in *Forum on Specification and Design Languages, FDL'08*. Stuttgart, Germany: IEEE, 2008, pp. 221–226, DOI: 10.1109/FDL.2008.4641449. 20
- [HSV05] F. Herrera, P. Sánchez, and E. Villar, "Heterogeneous System-Level Specification in SystemC," in *Advances in Design and Specification Languages for SoCs*, P. Boulet, Ed. Springer US, 2005, pp. 199–216. 27
- [HV06] F. Herrera and E. Villar, "A Framework for Embedded System Specification under Different Models of Computation in SystemC," in *Design Automation Conference, 2006 43rd ACM/IEEE*. ACM, 2006, pp. 911–914, DOI: 10.1145/1146909.1147140. 27
- [HV08] —, "A Framework for Heterogeneous Specification and Design of Electronic Embedded Systems in SystemC," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, pp. 1–31, 2008, DOI: 10.1145/1255456.1255459. 27
- [HVF+05] J. Hartmann, M. Vieira, H. Foster, and A. Ruder, "A UML-based approach to system testing," *Innovations in Systems and Software Engineering*, vol. 1, no. 1, pp. 12–24, 2005, DOI: 10.1007/s11334-005-0006-0. 20
- [IEC61508] IEC, "Functional safety of electrical/electronic/programmable electronic safety-related systems (Second edition)," April 2010. 30, 31, 32, 123

- [IEC61508-2] IEC61508-2, "Functional safety of electrical/electronic/programmable electronic safety-related systems – Part 2: Requirements for electrical/electronic/programmable electronic safety-related systems," April 2010. 194
- [IEC62741] IEC, "Demonstration of dependability requirements – The dependability case," 2015. 224, 269
- [IEEE1076] IEEE, "IEEE Standard VHDL Language Reference Manual," pp. c1–626, Jan 2009, DOI: 10.1109/IEEESTD.2009.4772740. 96
- [IEEE1666-2011] "IEEE Standard for Standard SystemC Language Reference Manual," pp. 1–638, Jan 2012, DOI: 10.1109/IEEESTD.2012.6134619. 24
- [IEEE1666.1-2016] "IEEE Standard for Standard SystemC(R) Analog/Mixed-Signal Extensions Language Reference Manual," pp. 1–236, April 2016, DOI: 10.1109/IEEESTD.2016.7448795. 27
- [ISO04] "ISO/IEC 17000 Conformity assessment – Vocabulary and general principles," 2004. 2
- [ISO13849] "Safety of machinery — Safety-related parts of control systems (Second edition)," November 1 2006. 192, 195
- [JAR+94] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson, "Fault Injection into VHDL Models: The MEFISTO Tool," in *Fault-Tolerant Computing, 1994. FTCS-24. Digest of Papers., Twenty-Fourth International Symposium on*, 1994, pp. 66–75, DOI: 10.1109/FTCS.1994.315656. 46
- [Joh13] Johansen, Martin F., "Testing Product Lines of Industrial Size: Advancements in Combinatorial Interaction Testing," Ph.D. dissertation, Faculty of Mathematics and Natural Sciences, 2013. [Online]. Available: <https://WWW.duo.uio.no/bitstream/handle/10852/38097/dravhandling-johansen.pdf> 58
- [KB03] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," in *Proceedings of the IEEE*, vol. 91. IEEE, 2003, p. 15, DOI: 10.1109/JPROC.2002.805821. 17, 18, 19
- [KDV+97] B. Kienhuis, E. Deprettere, K. Vissers, and P. van der Wolf, "An approach for quantitative analysis of application-specific dataflow architectures," in *Application-Specific Systems, Architectures and Processors, 1997. Proceedings., IEEE International Conference on*. IEEE, 1997, pp. 338–349, DOI: 10.1109/ASAP.1997.606839. 2, 74
- [Kel98] Kelly, Tim, "Arguing safety – a systematic approach to managing safety cases," PhD thesis, University of York, 1998. [Online]. Available: <https://www-users.cs.york.ac.uk/tpk/tpkthesis.pdf> 124
- [Kel07] T. Kelly, "Modular Certification: Acknowledgements to the Industrial Avionic Working Group (IAWG)," in -, 2007, Conference Paper. 205
- [KOS+07] H. Kopetz, R. Obermaisser, C. El Salloum, and B. Huber, "Automotive Software Development for a Multi-Core System-on-a-Chip," in *Software Engineering for Automotive Systems, 2007. ICSE Workshops SEAS '07. Fourth International Workshop on*, 2007, pp. 2–2. 18
- [Kop98] H. Kopetz, "The Time-Triggered Architecture," in *Object-Oriented Real-time Distributed Computing, 1998. (ISORC 98) Proceedings. 1998 First International Symposium on*. IEEE, 1998, pp. 22–29, DOI: 10.1109/ISORC.1998.666765. 17
- [Kop98b] —, "The Time-Triggered Model of Computation," *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*, p. 16, 1998, DOI: 10.1109/REAL.1998.739743. 17
- [Kop11] —, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2011, DOI: 10.1007/978-1-4419-8237-7. 17, 18, 19, 251
- [EAGSN] Kouno, Takeshi. (2015) MDG Technology for GSN (Goal Structure Notation). [Online]. Available: <http://community.sparxsystems.com/community-resources/> 132, 135

- [KS03] H. Kopetz and N. Suri, "Compositional Design of RT Systems: A Conceptual Basis for Specification of Linking Interfaces," in *Object-Oriented Real-Time Distributed Computing, 2003. Sixth IEEE International Symposium on*. IEEE, 2003, pp. 51–60, DOI: [10.1109/ISORC.2003.1199236](https://doi.org/10.1109/ISORC.2003.1199236). 18
- [KS12] C. Kirsch and A. Sokolova, *The Logical Execution Time Paradigm*. Springer, 2012, ch. 5, pp. 103–120, DOI: [10.1007/978-3-642-24349-3_5](https://doi.org/10.1007/978-3-642-24349-3_5). 74
- [KZ10] S. Kirstan and J. Zimmermann, "Evaluating costs and benefits of model-based development of embedded software systems in the car industry – Results of a qualitative Case Study," 2010. [Online]. Available: http://www.esi.es/modelplex/c2m/docum/Paper_ECMFA_Altran.pdf 5
- [LAN+15] A. Larrucea, I. Agirre, C.-F. Nicolas, J. Perez, M. Azkarate-Askasua, and T. Trapman, "Temporal independence validation of an IEC-61508 compliant mixed-criticality system based on multicore partitioning," in *2015 Forum on Specification and Design Languages (FDL)*. IEEE, Sept 2015, Conference Paper, pp. 1–8, DOI: [10.1109/FDL.2015.7306359](https://doi.org/10.1109/FDL.2015.7306359). 132, 213
- [Lar17] A. Larrucea, "Development and Certification of Dependable Mixed-Criticality Embedded Systems based on DREAMS," PhD thesis, Naturwissenschaftlich-Technischen Fakultät. University of Siegen, Germany., 2017. 2, 10, 206
- [LMN+17] A. Larrucea, I. Martinez, C.-F. Nicolas, J. Perez, and R. Obermaisser, "Modular Development and Certification of Dependable Mixed-Criticality Systems," in *Digital System Design (DSD) 2017, 20th Euromicro Conference on*, aug 2017, Conference Paper. 213
- [LPA+15] A. Larrucea, J. Perez, I. Agirre, V. Brocal, and R. Obermaisser, "A Modular Safety Case for an IEC 61508 compliant Generic Hypervisor," in *Digital System Design (DSD), 2015 18th Euromicro Conference on*. IEEE, aug 2015, Conference Paper, DOI: [10.1109/DSD.2015.27](https://doi.org/10.1109/DSD.2015.27). 120, 132
- [LPN+16] A. Larrucea, J. Perez, C.-F. Nicolas, H. Ahmadian, and R. Obermaisser, "A Realistic Approach to a Network-on-Chip Cross-domain Pattern," in *Digital System Design (DSD), 2016 19th Euromicro Conference on*. IEEE, Oct 2016, Conference Paper, DOI: [10.1109/DSD.2016.66](https://doi.org/10.1109/DSD.2016.66). 120, 132, 213
- [LPO15] A. Larrucea, J. Perez, and R. Obermaisser, "A Modular Safety Case for an IEC 61508-compliant COTS multi-core device," in *DASC 2015 Conf. Proc.* IEEE, Oct. 2015, Conference Paper, DOI: [10.1109/CIT/IUCC/DASC/PICOM.2015.269](https://doi.org/10.1109/CIT/IUCC/DASC/PICOM.2015.269). 120, 132
- [LR11] W. Lu and M. Radetzki, "Efficient Fault Simulation of SystemC Designs," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*. IEEE, 2011, pp. 487–494, DOI: [10.1109/DSD.2011.68](https://doi.org/10.1109/DSD.2011.68). 46
- [LSP82] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Trans. Program. Lang. Syst.*, vol. 4, no. 3, pp. 382–401, Jul. 1982, DOI: [10.1145/357172.357176](https://doi.org/10.1145/357172.357176). 251
- [MAM+06] P. V. R. Murthy, P. C. Anitha, M. Mahesh, and R. Subramanyan, "Test ready UML statechart models," in *Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*. Shanghai, China: ACM, 2006, pp. 75–82, DOI: [10.1145/1138953.1138968](https://doi.org/10.1145/1138953.1138968). 20
- [MAR10] M. Malvezzi, B. Allotta, and M. Rinaldi, "Odometric estimation for automatic train protection and control systems," *Vehicle System Dynamics*, vol. 49, no. 5, pp. 723–739, 2010, DOI: [10.1080/00423111003721291](https://doi.org/10.1080/00423111003721291). 153
- [MARTE] "UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems," June 2nd, 2011 2011, Object Management Group (OMG). [Online]. Available: <http://www.omg.org/spec/MARTE/> 6

- [ML] MathWorks, "MATLAB." [Online]. Available: <http://www.mathworks.com/products/matlab/> 171
- [MAAB] MathWorks Automotive Advisory Board,, "Control Algorithm Modeling Guidelines Using MATLAB, Simulink, and Stateflow - Version 3.0," 2012. [Online]. Available: http://www.mathworks.com/solutions/automotive/standards/docs/MAAB_Style_Guideline_Version3p00_pdf.zip 94
- [Mat13] K. Matzker, "Magnetic encoders conquer safety-relevant applications," *CAN Newsletter special CANopen Lift*, 2013. [Online]. Available: http://www.can-newsletter.org/engineering/engineering-Miscellaneous/140811_can-newsletter-print_canopen-lift_2013 101, 175
- [MTest] M.E.S.GmbH, "MTest Classic - Model Test Manager for Simulink and TargetLink Models - Homepage," 2009. [Online]. Available: <http://www.mtest-classic.com/> 20
- [uPY] "MicroPython - Python for microcontrollers," <https://micropython.org/>. 72
- [MOG31] MOGENTES, "Fault Models," MOGENTES, Tech. Rep., 2009/12/29 2009. 46, 247
- [MPE09] S. Moazzeni, S. Poormozaffari, and A. Emami, "An Optimized Simulation-Based Fault Injection and Test Vector Generation Using VHDL to Calculate Fault Coverage," in *Microprocessor Test and Verification (MTV), 2009 10th International Workshop on*. IEEE, 2009, pp. 55–60, DOI: 10.1109/MTV.2009.22. 46
- [MK10b] A. Marrero Pérez and S. Kaiser, "Multi-Level Test Models for Embedded Systems," in *Software Engineering 2010*, ser. Lecture Notes in Informatics (LNI) - Proceedings, G. Engels, M. Luckey, and W. Schäfer, Eds. Paderborn, Germany: Köllen Druck+Verlag GmbH, Bonn, 2010, pp. 213–224. [Online]. Available: <http://subs.emis.de/LNI/Proceedings/Proceedings159/P-159.pdf> 6, 53
- [MK10c] —, "Top-Down Reuse for Multi-level Testing," in *Engineering of Computer Based Systems (ECBS), 2010 17th IEEE International Conference and Workshops on*. IEEE, 2010, pp. 150–159, DOI: 10.1109/ECBS.2010.23. 6, 53, 54
- [MK10a] A. Marrero Pérez and S. Kaiser, "Bottom-up reuse for multi-level testing," *Journal of Systems and Software*, vol. 12, no. 83, pp. 2392–2415, 2010, DOI: 10.1016/j.jss.2010.07.028. 6, 53
- [MSR+10] A. Michailidis, U. Spieth, T. Ringler, B. Hedenetz, and S. Kowalewski, "Test Front Loading in Early Stages of Automotive Software Development Based on AUTOSAR," in *Design, Automation and Test in Europe, 2010 (DATE10)*. ICC, Dresden, Germany: IEEE, 2010, DOI: 10.1109/DATE.2010.5457166. 42
- [MSU+09] J. Mendizabal-Samper, A. Salterain, S. Urcelayeta, J. M. Blanco, and A. Galarza, "UML-based Fault Injection for Software Model Testing (FISMT)," in *European Safety and Reliability Conference, ESREL 2009*, R. Briš, C. Guedes Soares, and S. Martorell, Eds. Prague, Czech Republic: CRC Press, 2009, pp. 2021–2026, DOI: 10.1201/9780203859759.ch278. 47
- [MVS07] S. Misera, H. T. Vierhaus, and A. Sieber, "Fault Injection Techniques and their Accelerated Simulation in SystemC," in *Digital System Design (DSD), 2007 10th Euromicro Conference on*. IEEE, 2007, pp. 587–595, DOI: 10.1109/DSD.2007.4341528. 46
- [MWSF17] "Developing S-Functions (R2017a)," http://www.mathworks.es/help/pdf_doc/simulink/sfunctions.pdf, 2017. 84
- [NAM+16] C.-F. Nicolas, I. Avestaran, I. Martinez, and P. Franco, "Model-Based Development of an FPGA Encoder Simulator for Real-Time Testing of Elevator Controllers," in *2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, May 2016, Conference Paper, pp. 53–60, DOI: 10.1109/ISORC.2016.17. 117, 176, 180, 181, 191

- [NAP+17] C.-F. Nicolas, I. Ayestaran, T. Poggi, G. G. Sagardui, and J. M. Martin, "A CAN Restbus HiL elevator simulator based on code reuse and device para-virtualization," in *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*. IEEE, May 2017, Conference Paper, DOI: [10.1109/ISORC.2017.02](https://doi.org/10.1109/ISORC.2017.02). 173, 187, 191
- [CERTWARE] NASA. Certware - Eclipse-based, open source tools for safety, assurance, or dependability cases. [Online]. Available: <http://nasa.github.io/CertWare/> 30, 126
- [cRIO9039a] National Instruments, "cRIO-9039 CompactRIO Controller." [Online]. Available: <http://www.ni.com/en-us/support/model.crio-9039.html> 93, 178
- [LV] National Instruments, "LabVIEW System Design Software." [Online]. Available: <http://www.ni.com/labview/> 92
- [NI9853] —, "NI 9853 2-Port, High-Speed CAN Module for NI CompactRIO." [Online]. Available: <http://sine.ni.com/nips/cds/view/p/lang/en/nid/201972> 179
- [cRIO] National Instruments, "NI CompactRIO." [Online]. Available: <http://www.ni.com/compactrio/> 93
- [cRIO9082] —, "cRIO-9081/9082 Operating Instructions and Specifications," July 2011 2011. [Online]. Available: <http://www.ni.com/pdf/manuals/375714a.pdf> 93, 178
- [cRIO9039b] —, "cRIO-9039 Specifications," 2015. [Online]. Available: http://www.ni.com/pdf/manuals/375697a_02.pdf 178
- [NEL+17] C.-F. Nicolas, F. Eizaguirre, A. Larrucea, S. Barner, F. Chauvel, G. Sagardui, and J. Perez, "GSN Support of Mixed-Criticality Systems Certification," in *"Dependable Smart Embedded Cyber-physical Systems and Systems-of-Systems" at SAFECOMP 2017 (DECSoS '17), 17th ERCIM/EWICS/ARTEMIS Workshop on.,* May 2017, Conference Paper, (accepted paper). 213
- [Nik11] R. S. Nikhil, "Abstraction in hardware system design," *Commun. ACM*, vol. 54, pp. 36–44, October 2011, DOI: [10.1145/2001269.2001284](https://doi.org/10.1145/2001269.2001284). 6
- [NIPXIHIL] Hardware-in-the-Loop. <http://www.ni.com/es-es/innovations/automotive/hardware-in-the-loop.html>. National Instruments. 93
- [Obe04] R. Obermaisser, "An Integrated Architecture for Event-Triggered and Time-Triggered Control Paradigms," Ph.D. dissertation, Institut für Technische Informatik, 2004. 1
- [ARM] Argumentation Metamodel (ARM). <http://www.omg.org/spec/ARM/>. 29
- [UML] Object Management Group, "Unified Modeling Language (UML)." [Online]. Available: <http://www.omg.org/spec/UML/> 6
- [OPALRT] Hypersim. <http://www.opal-rt.com/systems-hypersim/>. OPAL-RT. 93
- [OPE47] OPENCROSS Consortium, "Methodological Guide Common Certification Language," OPENCROSS Consortium, Tech. Rep., December 2015. [Online]. Available: http://www.opencross-project.eu/sites/default/files/D4.7_Methodological_Guide_Common_Certification_Language_V1.0.pdf 59
- [OVP] Open Virtual Platforms. [Online]. Available: <http://www.ovpworld.org> 57, 225
- [Pel96] J. Peleska, "Formal Methods and the Development of Dependable Systems," Habilitation Thesis, Christian Albrechts Universität zu Kiel, Kiel, Germany, 1996. 43
- [Per11] J. Perez, "Executable Time-Triggered Model (E-TTM) for the Development of Safety-Critical Embedded Systems," Ph.D. dissertation, Technischen Universität Wien, 2011. 71, 79, 219

- [PGJ+13] Perez, Jon and Gonzalez, David and Trujillo, Salvador and Trapman, Ton and Garate, Jose Miguel, "A Safety Concept for a Wind-Power Mixed-Criticality Embedded System based on Multicore Partitioning," in *Mixed-Criticality Systems WMC, RTSS, 1st Workshop on*, 2013, pp. 25–30. [61](#)
- [PMD16] K. Paul, S. Mickelson, and J. M. Dennis, "A new parallel python tool for the standardization of earth system model data," in *2016 IEEE International Conference on Big Data (Big Data)*, Dec 2016, pp. 2953–2959, DOI: [10.1109/BigData.2016.7840946](#). [226](#)
- [PNO+10] J. Perez, C.-F. Nicolas, R. Obermaisser, and C. El Salloum, "Modeling Time-Triggered Architecture Based Real-Time Systems Using SystemC," in *Forum on specification and Design Languages (FDL) 2010*, T. J. Kaźmierski and A. Morawiec, Eds. Springer, Sept 2010, DOI: [10.1007/978-1-4614-1427-8_8](#). [74](#), [219](#)
- [PP05] A. Pretschner and J. Philipps, "Methodological Issues in Model-Based Testing," in *Model-Based Testing of Reactive Systems*, M. Broy, Ed. Springer, 2005, vol. LNCS 3472, pp. 281–291, DOI: [10.1007/11498490_13](#). [49](#), [54](#)
- [AMASS] "AMASS (Architecture-driven, Multi-concern and Seamless Assurance and Certification of Cyber-Physical Systems)." [Online]. Available: <http://www.amass-ecsel.eu/> [60](#), [145](#)
- [ASCOS] "ASCOS - Aviation Safety and Certification of new Operations and Systems," <http://www.ascos-project.eu/>. [60](#)
- [CESAR] "Cost-efficient methods and processes for safety relevant embedded systems." [Online]. Available: <http://www.cesarproject.eu> [57](#)
- [OPENCROSS] "OPENCROSS - Open Platform for Evolutionary Certification of Safety-critical Systems." [Online]. Available: <http://www.opencross-project.eu/> [59](#), [145](#)
- [PROXIMA] "Probabilistic real-time control of mixed-criticality multicore and manycore systems (PROXIMA)." [Online]. Available: <http://www.proxima-project.eu/> [60](#)
- [NEFAB] "NEFAB Project: Safety Case Report, Version 3.01," 2011. [Online]. Available: http://ec.europa.eu/transport/modes/air/single_european_sky/doc/2011_08_26_nefab_anx10.pdf [124](#), [272](#)
- [VERDE] "VERDE: VERification-oriented and component-based model Driven Engineering for real-time embedded systems," 2012. [Online]. Available: <https://itea3.org/project/verde.html> [58](#)
- [PS04] H. D. Patel and S. K. Shukla, "Towards A Heterogeneous Simulation Kernel for System Level Models: A SystemC Kernel for Synchronous Data Flow Models," in *Proceedings of the 14th ACM Great Lakes symposium on VLSI*. ACM, 2004, pp. 248–253, DOI: [10.1109/FDL.2008.4641452](#). [27](#)
- [PS06] —, *SystemC Kernel Extensions for Heterogeneous System Modeling: A Framework for Multi-MoC Modeling & Simulation*. Kluwer Academic Publishers, 2006. [27](#)
- [PSU+08] J. Pérez, J. F. Sevillano, S. Urcelayeta, and I. Vélez, "System behaviour capture: from UML to SystemC," in *Forum on Specification and Design Languages, FDL'08*. IEEE, 2008, pp. 235–236, DOI: [10.1109/FDL.2008.4641452](#). [28](#)
- [PtolemyII] Ptolemy II. <http://ptolemy.eecs.berkeley.edu/ptolemyII/>. Accessed: 2017/05/20. [262](#)
- [Python] Python. <https://www.python.org/>. [72](#)
- [PyFMI] PyFMI: A package for working with dynamic models compliant with the Functional Mock-Up Interface standard. <https://pypi.python.org/pypi/PyFMI>. Accessed: 2017/05/20. [262](#)
- [PYW] "Pythonwin documentation," 2012. [Online]. Available: <http://sourceforge.net/projects/pywin32> [79](#)

- [QEMU] Quick EMUlator (QEMU). [Online]. Available: <http://www.qemu-project.org/> 57, 225
- [RGG+12] P. Radojkovic, S. Girbal, A. Grasset, E. Quiñones, S. Yehia, and F. J. Cazorla, "On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–25, 2012, DOI: 10.1145/2086696.2086713. 2
- [RL11] K. Rustan and M. Leino, "Tools and Behavioral Abstraction: A Direction for Software Engineering," in *The Future of Software Engineering*, 1st ed., S. Nanz, Ed. Springer, 2011, pp. 1–32. 55
- [RPV+13] S. Reiter, M. Pressler, A. Viehl, O. Bringmann, and W. Rosenstiel, "Reliability assessment of safety-relevant automotive systems in a model-based design flow," in *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*. IEEE, 2013, pp. 417–422, DOI: hrefhttp://dx.doi.org/10.1109/ASPDAC.2013.650963210.1007/s11334-009-0105-4. 46
- [RTTM] Real-Time Target Machines. <https://www.speedgoat.com/products-services/real-time-target-machines>. Speedgoat. 93
- [OPE53] Ruiz, Alejandra and Espinoza, Huáscar and Nair, Sunil and Attwood, Katrina and Conmy, Philippa, "Compositional Certification Conceptual Framework," OPENCOSS Consortium, Tech. Rep., December 2013. [Online]. Available: http://www.opencoss-project.eu/sites/default/files/D5.3_Compositional_Certification_Conceptual_Framework_final.pdf 59, 129, 143
- [SACM] "Structured Assurance Case Metamodel (SACM) Version 1.1," <http://www.omg.org/spec/sacm>, July 2015, the formal version of SACM is a combination of Argumentation Metamodel (ARM) and Software Assurance Evidence Metamodel (SAEM) documents. 29, 59
- [SCALEXIO] SCALEXIO. https://www.dspace.com/en/ltd/home/products/hw/simulator_hardware/scalexio.cfm/. dSPACE. 93
- [SCDM06] "Safety Case Development Manual version 2.2," Nov 06 2006, EUROCONTROL. [Online]. Available: <http://www.EUROCONTROL.int/sites/default/files/Article/content/documents/nm/link2000/safety-case-development-manual-v2.2-ri-13nov06.pdf> 30, 124, 126, 208, 265, 269, 272, 274, 275
- [SGH+10] M. Streubühr, J. Gladigau, C. Haubelt, and J. Teich, *Efficient Approximately-Timed Performance Modeling for Architectural Exploration of MPSoCs*. Dordrecht: Springer, 2010, pp. 59–72, DOI: 10.1007/978-90-481-9304-2_4. 57
- [SIMICS] Wind River Simics. [Online]. Available: <https://www.windriver.com/products/simics//> 57
- [SimVision] SimVision. <http://www.cadence.com/products/fv/simvision>. Accessed: 201/10/23. 79
- [SL] "Simulink," MathWorks. [Online]. Available: <http://www.mathworks.com/products/simulink/> 92, 171
- [SLDV] "Simulink Design Verifier," MathWorks. [Online]. Available: <http://www.mathworks.com/products/sldesignverifier> 20
- [SLP+10] C. Schumacher, R. Leupers, D. Petras, and A. Hoffmann, "parSC: Synchronous parallel SystemC simulation on multi-core host architectures," in *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. IEEE, 2010, pp. 241–246, DOI: 10.1145/1878961.1879005. 226
- [SLUG] "Simulink User Guide," MathWorks. [Online]. Available: <https://www.mathworks.com/help/simulink/ug/> 20
- [SLVV] "Simulink Verification and Validation, commercial model-based verification and validation tool." [Online]. Available: <http://www.mathworks.com/products/simverification/> 20

- [SRH08] R. A. Shafik, P. Rosinger, and B. Al-Hashimi, "SystemC-based Minimum Intrusive Fault Injection Technique with Improved Fault Representation," in *2008 14th IEEE International On-Line Testing Symposium*. IEEE, July 2008, pp. 99–104, DOI: [10.1109/IOLTS.2008.25.46](https://doi.org/10.1109/IOLTS.2008.25.46)
- [SCB+04] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi, "Benefits and challenges for platform-based design," in *Proceedings of the 41st annual conference on Design automation - DAC'04*. ACM, 2004, p. 5, DOI: [10.1145/996566.996684](https://doi.org/10.1145/996566.996684). 121
- [SM01] A. Sangiovanni-Vincentelli and G. Martin, "Platform-based design and software design methodology for embedded systems," *IEEE Design and Test of Computers*, vol. 18, no. 6, p. 10, 2001, DOI: [10.1109/54.970421](https://doi.org/10.1109/54.970421). 56, 121
- [SWS06] A. Sinha, C. E. Williams, and P. Santhanam, "A measurement framework for evaluating model-based test generation tools," *IBM Systems Journal*, vol. 45, no. 3, pp. 501–514, 2006, DOI: [10.1147/sj.453.0501](https://doi.org/10.1147/sj.453.0501). 51
- [SysML] "Systems Modeling Language," Object Management Group (OMG). [Online]. Available: <http://www.omg.sysml.org/> 6, 21
- [TCA+14] S. Trujillo, A. Crespo, A. Alonso, and J. Pérez, "MultiPARTES: Multi-core partitioning and virtualization for easing the certification of mixed-criticality systems," *Microprocessor and Microsystems*, vol. 38, no. 8, pp. 921–932, November 2014, DOI: [10.1016/j.micpro.2014.09.004](https://doi.org/10.1016/j.micpro.2014.09.004). 61
- [The03] T. Thelin, "Automated Statistical Testing Suite for Software Validation," in *11th European International Conference on Software Testing, Analysis & Review (EuroSTAR'03)*, 2003. 20
- [TLMP93] S.-Y. Tzou, J.-J. Lim, J. Menon, and D. Palmer, "A Distributed Development Environment for Embedded Software," *Software—Practice & Experience*, vol. 23, 1993. 1
- [Tou58] S. E. Toulmin, *The Use of Argument*. Cambridge University Press, 1958, no. 241. 29
- [TR50506-2] "PD CLC/TR 50506-2:2009 Railway applications. Communication, signalling and processing systems. Application guide for EN 50129. Part 2: Safety assurance," 2009, CENELEC. 124, 224, 269, 270, 271
- [MULTIPARTES] Trujillo, S. and Crespo, A. and Alonso, A., "MultiPARTES: Multicore Virtualization for Mixed-Criticality Systems," in *Digital System Design (DSD), 2013 16th Euromicro Conference on*, 2013, pp. 260–265. 61
- [TSA+09a] K. Tomasena, J. F. Sevillano, N. Arrue, A. Cortés, and I. Vélez, "Embedding Matlab in SystemC Transaction Level Modeling for Verification," in *XXIV Conference on Design of Circuits and Integrated Systems (DCIS'09)*, Zaragoza, Spain, 2009. [Online]. Available: <http://dcis2009.unizar.es/FILES/CR2/p22.pdf> 28
- [TSP+09] K. Tomasena, J. F. Sevillano, J. Pérez, A. Cortés, and I. Vélez, "A Transaction Level Assertion Verification Framework in SystemC: An Application Study," in *Advances in Circuits, Electronics and Micro-electronics, 2009. CENICS '09. Second International Conference on*. IEEE, 2009, pp. 75–80, DOI: [10.1109/CENICS.2009.24](https://doi.org/10.1109/CENICS.2009.24). 27
- [V-ECUc] Ulrich Lauff, "Virtual ECUs. ETAS ISOLAR-EVE in application." [Online]. Available: <http://www.etas.com> 56
- [UTP05] UTP, "UML Testing Profile. Version 1.0. OMG document: formal/05-07-07," 2005 2005, <http://utp.omg.org/>. 20
- [VARIES] VARIES Consortium,, "VARIability In safety-critical Embedded Systems (VARIES) Project ARTEMIS-2011-1 295397." [Online]. Available: <http://www.varies.eu/> 58
- [VAR48] —, "D4.8 Challenges and Potentials of Certifying Product Lines," VARIES Consortium,, Tech. Rep., 2014. 123

- [VAR47] —, “D4.7 Variability Analysis Solutions,” VARIES Consortium,, Tech. Rep., 2015. [Online]. Available: http://www.varies.eu/wp-content/uploads/sites/8/2013/05/VARIES_D4.7_v01_PU_FINAL.pdf 58
- [VHC+15] Vasilevskiy, Anatoly and Haugen, Øystein and Chauvel, Franck and Johansen, Martin Fagereng and Shimbara, Daisuke, “The BVR tool bundle to support product line engineering,” pp. 380–384, 2015, DOI: 10.1145/2791060.2791094. 58
- [CANoe] Vector, “ECU Development & Test with CANoe.” [Online]. Available: https://vector.com/vi_canoe_en.html 56
- [V-ECUa] Virtual Validation. [Online]. Available: https://www.dspace.com/en/pub/home/products/systems/virtual_validation.cfm 33, 56
- [BIP] Verimag, “Rigorous Design of Component-Based Systems - The BIP Component Framework.” [Online]. Available: <http://www-verimag.imag.fr/Rigorous-Design-of-Component-Based.html> 6
- [VMM+09] A. Villaro, J. Mendizabal, J. R. Martín, I. Adin, and A. Galarza, “Fault Injection techniques for the safety and availability characterization of an N out of M System,” in *XXIV Conference on Design of Circuits and Integrated Systems (DCIS'09)*, Zaragoza, Spain, 2009. [Online]. Available: <http://dcis2009.unizar.es/FILES/CR2/p40.pdf> 47
- [WG09] P. Winter, B. Guiot, and I. U. o. Railways, *Compendium on ERTMS: European Rail Traffic Management System*. Eurail Press, 2009. 150, 151
- [Win09] A. Windisch, “Search-based testing of complex Simulink models containing Stateflow diagrams,” in *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*. IEEE, 2009, pp. 395–398, DOI: 10.1109/ICSE-COMPANION.2009.5071030. 20
- [WLG07] C. Weinstock, H.-F. Lipson, and J. Goodenough, “Arguing Security – Creating Security Assurance Cases,” Carnegie Mellon University, Tech. Rep., 2007. 227
- [WSB01] J. Wegener, H. Sthamer, and A. Baresel, “Application Fields for Evolutionary Testing,” in *9th European International Conference on Software Testing Analysis & Review (Eurostar 2001)*, Stockholm, Sweden, 2001. [Online]. Available: <http://www.systematic-testing.com/documents/eurostar2001.pdf> 40
- [Xil15] Xilinx, “7 Series FPGAs Overview,” 2015. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf 179
- [XTR] XtratuM Hypervisor. [Online]. Available: <http://www.fentiss.com/en/products/xtratum.html> 120, 195
- [Zan08] J. Zander, “Model-based testing of real-time embedded systems in the automotive domain,” PhD Thesis, Fakultät IV – Elektrotechnik und Informatik, Technische Universität Berlin, Germany, 2008. 1, 20
- [ZAV04] H. Ziade, R. Ayoubi, and R. Velazco, “A Survey on Fault Injection Techniques,” *The International Arab Journal of Information Technology*, vol. 1, p. 16, 2004. 46
- [ZC05] Y. Zhan and J. A. Clark, “Search-based mutation testing for Simulink models,” in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '06. ACM, 2005, pp. 1061–1068, DOI: 10.1145/1068009.1068188. 20
- [ZC06] —, “The state problem for test generation in Simulink,” in *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '06. ACM, 2006, DOI: 10.1145/1143997.1144319. 20
- [ZC08] —, “A search-based framework for automatic testing of MATLAB/Simulink models,” *J. Syst. Softw.*, vol. 81, no. 2, pp. 262–285, 2008, DOI: 10.1016/j.jss.2007.05.039. 20

- [ZDS+05] J. Zander-Nowicka, Z. R. Dai, I. Schieferdecker, and G. Din, "From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing," in *IFIP 17th International Conference on Testing Communicating Systems (TestComm 2005)*. Montral, Canada: Springer, 2005. [Online]. Available: http://www.fokus.fraunhofer.de/de/motion/ueber_motion/unser_team/zander_justyna/TestCom2005_MDATestingv_final1.pdf 20
- [ZMS07b] J. Zander-Nowicka, A. Marrero Pérez, I. Schieferdecker, and Z. R. Dai, "Test Design Patterns for Embedded Systems," in *CONQUEST 2007, 10th International Conference on Quality Engineering in Software Technology*, Potsdam, Germany, 2007. [Online]. Available: http://www.fokus.fraunhofer.de/de/motion/ueber_motion/unser_team/zander_justyna/Conquest_07_vfinal_DV2.pdf 20
- [ZSM06] J. Zander-Nowicka, I. Schieferdecker, and A. Marrero Pérez, "Automotive Validation Functions for On-line Test Evaluation of Hybrid Real-time Systems," in *Autotestcon, 2006 IEEE*, ser. 2006 IEEE AUTOTESTCON - IEEE Systems Readiness Technology Conference. Anaheim, CA: IEEE, 2006, pp. 799–805, DOI: 10.1109/AUTEST.2006.283767. 20
- [ZXS08] J. Zander-Nowicka, X. Xiong, and I. Schieferdecker, "Systematic test data generation for embedded software," in *Proceedings of the 2008 International Conference on Software Engineering Research and Practice, SERP 2008*, ser. 2008 International Conference on Software Engineering Research and Practice, SERP 2008, Las Vegas, NV, 2008, pp. 164–170. [Online]. Available: <http://www.scopus.com/inward/record.url?eid=2-s2.0-62749156482&partnerID=40> 20
- [ZSJ10] J. Zhu, I. Sander, and A. Jantsch, "HetMoC: Heterogeneous Modelling in SystemC," in *Specification & Design Languages (FDL 2010), 2010 Forum on*. IET, 2010, pp. 1–6, DOI: 10.1049/ic.2010.0139. 27



The PS-TTM Modelling Framework

This appendix recalls the **PS-TTM** modelling framework contributed by I. Ayestaran [Aye15].

A.1 Fault Injection Libraries

Test engineers may inject faults by selecting the desired fault effects in the fault-configuration **XML** file. The **ATE** pre-processes the **PS-TTM** model of the system, then automatically extends the original model to include the pre-built fault injectors as specified in the fault configuration. The Fault-Injection Unit (**FIU**) component of the **ATE** includes the libraries of executable fault models for both platform-independent and platform-specific models.

A.1.1 Fault Library for Platform-Independent Models

Platform-independent models abstract from aspects related to the target platform in **PS-TTM**, thus the library of faults focuses on faults at signal level. This fault library draws on the failure mode functions (FMFs) defined in the MOGENTES project [MOG31]. These failure modes represent the “*effects of faults / errors that would lead to a failure in a system*”.

Table A.1 summarizes the fault models for platform independent models currently provided by the library. The fault library for **PIMs** natively provided by the **PS-TTM ATE** for Platform-Independent Model (**PIM**) models covers a wide range of failures that are likely to occur in systems and provides a straightforward way to simulate them at a platform-independent level. The **PIM** fault library defines faults for signals of boolean, integer and floating-point datatypes, which are intended to simulate the scenarios listed below:

- **Amplification:** The Amplification fault effect can simulate a number of faults, e.g., erroneous sensor positioning and / or orientation, wrong parametrization of components, bugs due to misunderstanding of parameter units, etc.
- **Amplification Range:** The purpose of the Amplification Range fault effect is to introduce noise in integer- or continuous-range signals.
- **Constant:** This fault effect can be used to simulate several different situations, such as a broken encoder that keeps providing a constant value to the system, a faulty

Table A.1: Fault library for PI-TTM models

(source: [ANP+14d]).

	Fault effect	Configuration parameters	Description
Boolean	<i>Invert</i>	-	Boolean value is inverted.
	<i>Stuck At</i>	stuck_value	Signal gets stuck at a given value.
	<i>Stuck</i>	-	Signal gets stuck at the current value.
	<i>Stuck If</i>	stuck_value, condition	Signal gets stuck at {stuck_value} if a given {condition} holds.
	<i>Open Circuit</i>	-	Wire is disconnected, signal takes an arbitrary value (noise)
	<i>Delay</i>	delay	Signal is delayed by an amount of time.
Integer/Float	<i>Constant</i>	constant_value	Signal gets stuck at a given constant value.
	<i>Amplification</i>	ampl_value	Signal is amplified by fixed value.
	<i>Amplification Range</i>	min_amp_value, max_ampl_value	Signal is amplified by a randomly selected value (between given max. and min. values)
	<i>Drift</i>	drift_value	At each time stepc, the signal drifts away from its nominal value by a given value
	<i>Offset</i>	offset_value	A given fixed offset is added to the signal.
	<i>Offset Range</i>	min_offset_value, max_offset_value	A randomly selected offset value is added to the signal (between given max. and min. values).
	<i>Stuck</i>	-	Signal gets stuck at the actual value.
	<i>Random</i>	min_value, max_value	Signal takes an arbitrary value (between given max. and min. values).
	<i>Delay</i>	delay	Signal is delayed by an amount of time.

sensor measuring a constant value, or a rotational component that is blocked by an external object.

- **Delay:** The `DeLay` fault effect can be used to simulate lags introduced by a data-traffic overhead in a communication system or by the excessive length of wires in the propagation of a signal.
- **Drift:** This effect is used to simulate incremental faults in components, such as a failure in a counter that causes it to count more events than it should at each iteration.
- **Inversion:** This fault effect can be used to simulate bit flips in memory cells, such as Single Event Upsets (**SEUs**) induced by energetic particles.
- **Offset:** This fault effect can be used to reproduce situations in which a sensor has not been correctly calibrated.
- **Offset Range:** This fault effect is mainly used to introduce noise in signals.
- **Open Circuit:** This fault effect is used to simulate different situations in which a signal can get a random value, such as a broken wire, a defective connection or a communication through a very noisy environment.
- **Random:** This fault effect is used to reproduce situations in which a signal can get a random value, such as a broken wire or a defective electrical connection.
- **Stuck:** This fault effect can be used to simulate several different situations, such as a broken encoder that keeps providing a constant value to the system, a faulty sensor measuring a constant value, or a rotational component that is blocked by an external object.
- **Stuck At:** The `Stuck At` fault effect is used to reproduce a failure in a memory cell or connector that provokes a given data to stuck at a constant value.
- **Stuck:** The `Stuck` fault effect can reproduce a communication loss between two components, what would result in keeping the value of the variable unaltered.
- **Stuck If:** The `Stuck If` can be used to simulate different phenomena also reproduced by the `Stuck` or `Stuck At` effects, but adding a pre-condition to fire the fault effect.

A.1.2 Fault Library for Platform Specific Models

In our modeling approach the platform specific models of the **PS-TTM** represent the **HW** components at a high abstraction level. Hence, the fault library for **PSMs** is composed by the observable high-level effects of HW-related faults.

The **ATE** emulates faulty **HW** components by sabotaging all their outgoing signals according to the fault effect specified in the fault configuration, i.e., from the perspective of the **PS-TTM** ATE, a faulty **HW** component is a black box whose push activities are all sabotaged, thus giving the illusion of being a faulty component. This approach provides an equivalent effect to modifying the **HW** component models to simulate an erroneous behaviour, but having the advantage of its non-intrusiveness, as it does not require to apply modifications to the model of the System Under Test (**SUT**), such as replacing a correct component by an erroneous one (mutant based fault injection).

The **PSM** fault library adds the four fault effects listed in Table A.2, where **HW**-related fault effects are reproduced by fault injection during the communication phases of the simulation. The intended usages of the **HW**-related fault effects are:

- **Corruption:** This fault effect enables to reproduce a situation in which the functionality of the hardware component performs incorrectly. Thus, it can be used to simulate different situations that can provoke a faulty behavior in the value domain, such as noisy environments or defective electrical connections.
- **No execution:** This fault effect can be used to simulate errors caused by faults in power supplies, cuts in wires, or misbehaviors of **HW** components due to corrupted or incorrect data.
- **Out of time:** The Out of Time fault effect can be used to introduce a delay in the response of a **HW** component, which reproduces the effect of a number of anomalies in the system, such as an overload of the **CPU**, an excess of traffic in the communication system, or any other timing-related error caused by a misbehavior of a component.
- **Babbling** This fault effect can be used to simulate environments with very unfavorable

Table A.2: Fault library extensions for **PS-TTM** models

(source: [ANP+14d]).

Fault effect	Configuration parameters	Description
<i>Corruption</i>	-	The functionality is performed incorrectly. The data provided by the output interface is corrupted.
<i>No execution</i>	-	The functionality is not executed. No information is provided as a result.
<i>Out of time</i>	delay_value	Time bounds of the functionality are not respected. Data is provided later than expected.
<i>Babbling</i>	delay_value	Information at the interface is erroneous both in content and time.

conditions, such as those including high levels of noise (which would lead to corruption of data) and overhead of data-traffic in communications (which would cause the system to miss required deadlines).

A.2 Symmetric and Asymmetric Fault Injection

In a **PS-TTM** model where a signal provided by a job is forwarded to more than one component, the signal has to be replicated in different channels. In case of a fault in such a signal, the consistency between the values read by the receiving components might be compromised. When the incorrect service is equally perceived by all the consumers, the failure is considered consistent; on the contrary, if some of the receivers perceive differences in the incorrect service, the failure is called an inconsistent failure or 'byzantine failure' [LSP82]. Byzantine failures are difficult to handle, since they have the potential to confuse the correct components. In extreme cases a receiver might classify the failing component as erroneous whereas another receiver might identify it as correct, thus leading to an inconsistent view of the failed component among the correct components [Kop11].

Safety systems typically rely on redundant structures where functionally equivalent components may interchange information for cross-checking the behaviour of each other. This cross-monitoring may be compromised by a byzantine fault, and it should be assessed that a candidate design of a **DES** detects and reacts properly in such fault context. As the **FIU** of the **PS-TTM ATE** can inject faults in any signal of the system under test, the test engineers can straightforwardly perform both symmetric and asymmetric fault injection on the **PIM / PSM** models, depending on if the fault is injected at the instant in which a signal is being sent or when it is being received, as sketched in Figure A.1.

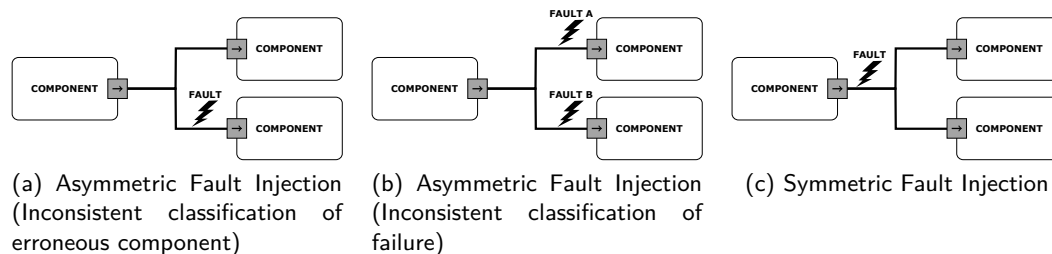


Figure A.1: Symmetric and Asymmetric Fault Injection

(source: [Aye15])

Figure A.2 shows the generic UML meta-model of the Fault Injection Configuration files.

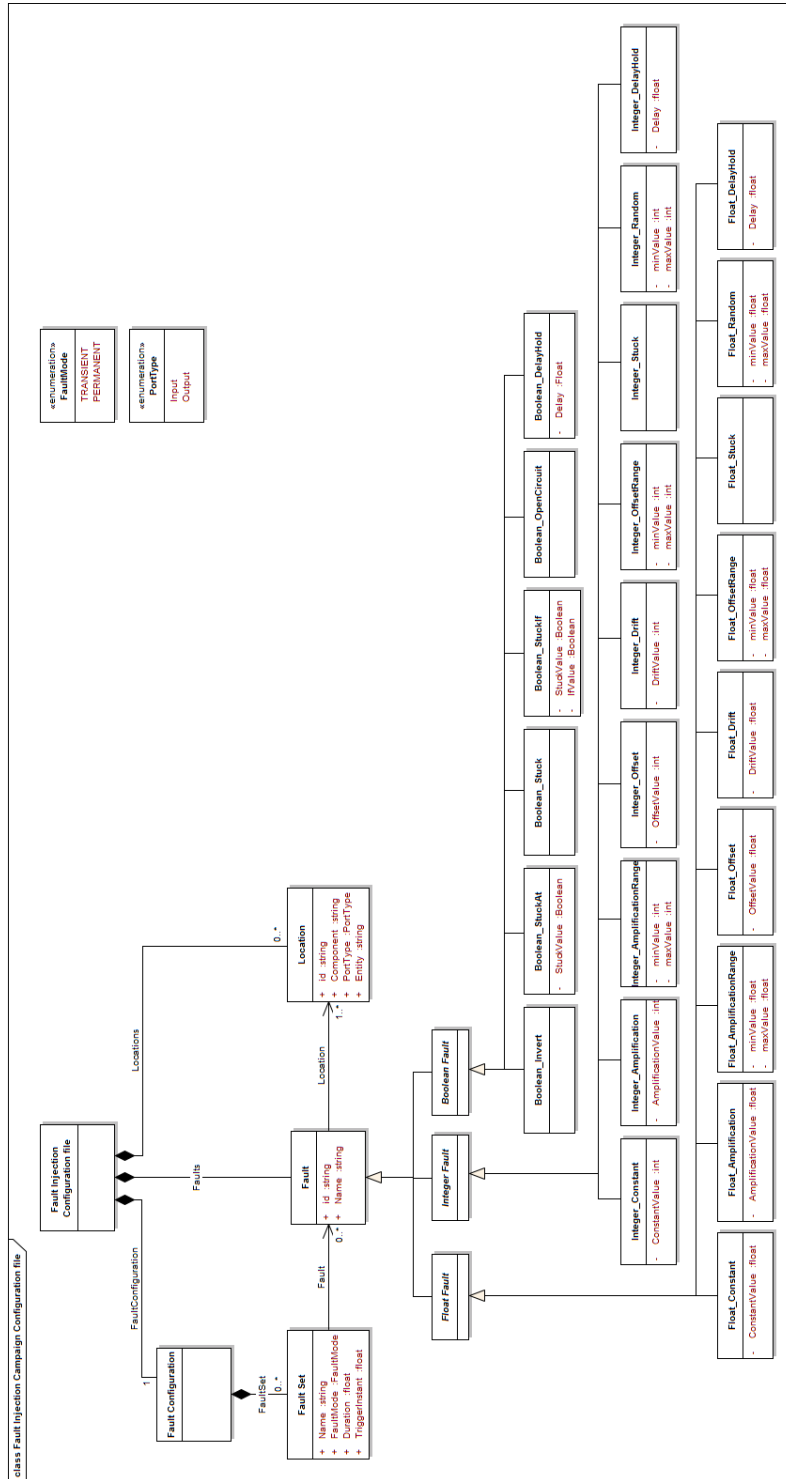


Figure A.2: UML Meta-model of Fault Injection Configuration XML files (source: [Aye15])

A.3 The Simulated Fault Injection Unit (FIU)

The Fault-Injection Unit (FIU) is the PS-TTM subsystem that provides simulated fault injection capabilities to the ATE. FIU supports two different fault modes: transient and permanent. Permanent faults will remain active from their activation specified by the trigger time until the end of the simulation. However, *transient faults* are temporary misbehaviours, so their configuration requires to specify a duration in addition the triggering instant. Once the injection of a *transient fault* is finished, the signal affected by the fault returns back to a non-faulty state.

Fault campaigns are optional: if omitted, the ATE will run a fault-free simulation. The fault-injection campaigns are structured in three blocks:

1. *Fault configuration with fault sets*: The fault configuration may include one or more fault sets. Each fault set, defined by the *FaultSet* tag, specifies a *name*, a *fault mode*, a *triggering instant*, and references to one or more faults. The fault mode must be specified either as 'TRANSIENT' or 'PERMANENT'. Therefore, in case the *fault mode* is set to 'TRANSIENT' the fault set must also specify a *duration* for the fault in addition to the rest of the configuration parameters. A fault set might be defined in the *Faults* block.
2. *Faults*: Each fault must specify a unique *id*, a *name*, a *fault effect* from the fault libraries (see section A.1) and a set of attributes that depend on the specific effect of the fault. When the fault configuration is edited and saved with the fault editor, the XML generator automatically sets the unique *id* of each fault to prevent duplicates.
3. *Locations*: The Locations block specifies a set of locations where the faults might be injected. Each location shall be defined by a unique *id* and the hierarchical *location* of the component that must simulate a faulty behaviour. In the case of faults from the PIM library which are related to signals, the name of the *entity* to be sabotaged and its *port type* (input or output) must also be specified.

The XML schema for the definition of fault-injection campaigns was derived from the ASAM AE HIL 1.0.0 standard for HiL testing. Although the aim of this work is to perform fault-injection experiments in model / software-in-the-loop configurations, close adherence to the standard eases the forward reuse of the fault-injection campaigns in the final prototyping phase, provided that fault injecting equipment is available. Listing A.1 shows an example of a fault configuration file for a PIMs model, whereas Listing A.2 shows an example XML specification of a permanent timeout HW-fault. As opposite to the fault specifications for PIMs, this fault specification ommits the *PortType* and an *Entity* attributes, that become meaningless for the considered HW abstraction.

Listing A.1: Example of an XML fault configuration file for the FIU module

```

1 <FaultConfiguration>
2   <FaultSet>
3     <Name>FC4</Name>
4     <FaultMode>TRANSIENT</FaultMode>
5     <Duration>4.0</Duration>
6     <TriggerInstant>86.0</TriggerInstant>
7     <Fault ref="_p5LWtRbKEe0w28kCcXd1kQ" />
8   </FaultSet>
9 </FaultConfiguration>
10 <Faults>
11   <Fault id="_p5LWtRbKEe0w28kCcXd1kQ">
12     <Name>i4</Name>
13     <Location ref="_LeWBYBbJEe0EgpzBm6gXmA" />
14     <FaultEffect>Integer_Constant</FaultEffect>
15     <ConstantValue>600</ConstantValue>
16   </Fault>
17 </Faults>
18 <Locations>
19   <Location id="_LeWBYBbJEe0EgpzBm6gXmA">
20     <Component>system_railwayss.das_superv.das_odo.job_odo</Component>
21     <PortType>Input</PortType>
22     <Entity>DAS_ODO_ENCODER1</Entity>
23   </Location>
24 </Locations>

```

Listing A.2: Example of a FIU configuration file for an 'Out of time' HW-fault

```

1 <FaultConfiguration>
2   <FaultSet>
3     <Name>FC37</Name>
4     <FaultMode>PERMANENT</FaultMode>
5     <Duration></Duration>
6     <TriggerInstant>120.0</TriggerInstant>
7     <Fault ref="hV4tRVQKifNGh5L1vztK"/>
8   </FaultSet>
9 </FaultConfiguration>
10 <Faults>
11   <Fault id="hV4tRVQKifNGh5L1vztK">
12     <Name>outoftime6</Name>
13     <Location ref="ZjYD4BVP03DD4Tbw0Uon"/>
14     <FaultEffect>HW_OutOfTime</FaultEffect>
15     <Delay>0.50</Delay>
16   </Fault>
17 </Faults>
18 <Locations>
19   <Location id="ZjYD4BVP03DD4Tbw0Uon">
20     <Component>system_railwayss.clrail.nodeevcC.procevc.coreA</Component>
21     <PortType></PortType>
22     <Entity></Entity>
23   </Location>
24 </Locations>

```

A.3.1 The PS-TTM toolset and workflow

The **PS-TTM** simulation engine derives from the standard SystemC library, extended with a set of macros and mechanisms to help the designers in the definition of the platform-independent and platform-specific models. In addition, a set of auxiliary **SW** applications assist the designers during the development and testing processes of the systems [Aye15]:

SFI Configuration Tool: This is an editor to select and configure fault effects for the **FIU** subsystem from PI / **PS-TTM** simulator, according to the fault meta-model described in the Appendix §A.1. This application supports the definition of comprehensive Simulated fault-injection campaigns, generating **FIU** configuration files compliant with pre-defined **FIU XML** schemas.

Test Case Generator: The Test-Case Generator retrieves output data obtained from a MATLAB / Simulink simulation, and converts them to timed sequences of input signals in a **PS-TTM**-compliant **XML** file. This application converts signal traces recorded in other simulation environments into test vectors optimized for the Test-Case Interpreter (**TCI**) component of the **PS-TTM** simulator.

Test Result Interpreter: In order to assess the behaviour of the model under test, the Test Point Manager of the **PS-TTM ATE** outputs the results of the simulations as a value-change-dump file (*.vcd). The VCD format is intended for storing waveforms captured by instruments in a standard format, and the test user can use specific viewer applications to retrieve and examine the information contained in VCD files. However, if the assessment of the system requires sophisticated data analysis, then it is convenient to export data to other **COTS** environments. Comma-Separated-Values files (*.csv) are commonly supported for data transfer between several applications. The Test Result Interpreter provides a vcd-to-csv file translator, easing the analysis the simulation results provided by the **PS-TTM ATE**.

Graphical Modeling Tool (alpha version): Manually typing a time-triggered system model in plain text is error-prone and might become a tedious work, particularly for complex systems with many interconnections. A graphical design front-end enhances the comprehension of the design for the users, presenting the model structure as a set of diagrams. A preliminary Graphical Modeling Tool was contributed by [Aye15], including graphical modeling capabilities for both **PIM** and **PSM** models. **PIM** models are built hierarchically, by composing **Systems**, **DASes** and **jobs**, which are represented as blocks.

To build a **PSM** model, the tool guides the user through these steps:

1. First the user must define the target platform.

2. Afterwards the tool recalculates the temporal properties of jobs, based on the specifications contained in the **PIM**. These temporal properties of jobs might be modified by users if necessary.
3. The user specifies the deployment model, allocating the functional assemblies from the **PIM** to the platform components in the **PSM**.
4. Finally, an integrated code generator generates the source files for both **PIM** and **PSM** models. Once compiled and linked with the **E-TTM** SystemC engine, the **PS-TTM** simulator is ready to simulate the fault-injection experiments for the verification of fault-tolerance properties on the system model, at **PIM** and **PSM** abstraction levels.

Figure A.3 shows how the previously described tools are integrated into the workflow specified by the **PS-TTM** approach.

As the figure shows, the design of the system should start with the specification of the functional and non-functional requirements with a dedicated requirement management tool. Once the requirements have been identified, the testing teams can use the graphical SFI campaign designer tool to define their SFI campaigns, and generate the necessary environmental models with Simulink. When these tasks are finished, the SFI **XML** code generator and test-case generator script will automatically generate the corresponding **XML** files for the evaluation of the system.

Simultaneously, the system designers can start the design of the **PIM** with the graphical modeling tool, and generate the textual **PIM** with the automatic code generator. The **PS-TTM ATE** will take the SFI campaign, test cases and test-point configuration file (developed manually), and perform the specified simulations by means of the **PI-TTM** execution engine.

The results of the simulation can then be immediately translated into CSV files by means of the vcd-to-csv translator. The test designers might then use any csv-compliant software to analyse and validate the results, and suggest any modification to the system in case the system does not fulfil any of the requirements.

If the **PIM** model is considered correct, it is deployed into the platform model by means of the graphical tool, and automatically generate the textual version of the **PSM**. The **PS-TTM ATE** can then be used to execute the model against the test campaigns defined from the functional and non-functional requirements, and the test developers can again use the vcd-to-csv translator in order to validate the results with their favourite csv-compliant software.

A.3. The Simulated Fault Injection Unit (FIU)

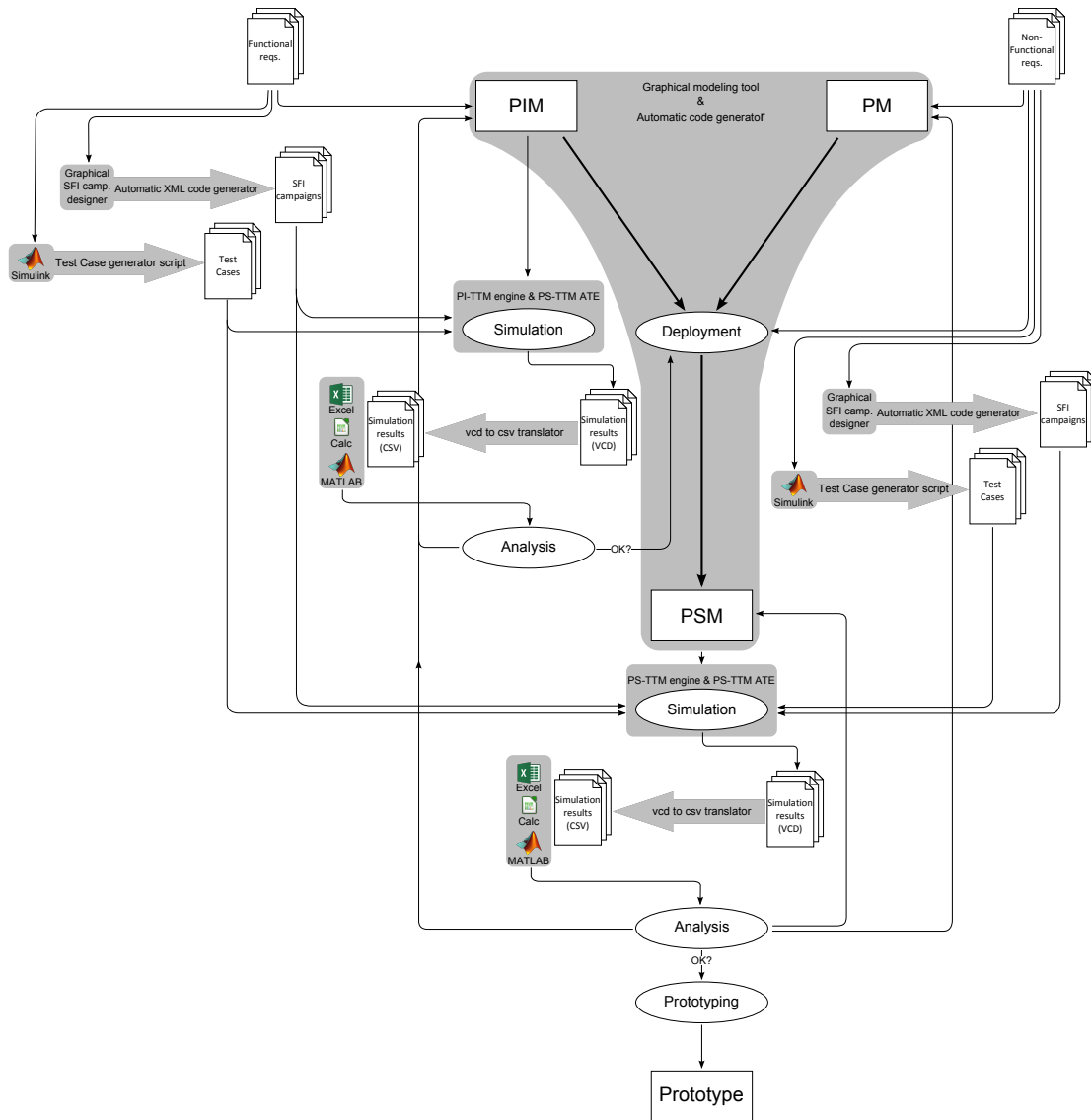


Figure A.3: Integration of tools in the PS-TTM workflow

B

Re-using Virtual Devices for Virtual Testing

This dissertation contributed a light-weight para-virtualization approach to provide more versatile test architectures for verifying distributed Dependable Embedded System (DES) integrating remote Input/Output (I/O) nodes. When the remote I/O nodes are not considered as parts of the System Under Test (SUT), then these devices are included amongst the components of the test-bench. When the test requirements for the SUT impose test cases with different layouts of the distributed DES (e.g., variable number of I/O nodes, alternative network topology, etc.), then the overall cost of testing with actual devices may increase, due to the effort spent in re-configuring the test setup.

Figure B.1 shows an example of a simple test architecture of an Automatic Test Executor (ATE) that operates an SUT consisting of a single device: the Device Under Test (DUT). The ATE integrates a Python interpreter to run the test scripts that interact with the DUT through Controller Area Network (CAN)-networked I/O devices. The slave I/O devices have I/Os paired with those of the DUT. The ATE, the SUT and the I/O devices communicate using a CAN bus. The SUT and the embedded I/O devices rely on their integrated CAN ports, while the ATE computer accesses a communication interface via PCI or USB. In the configuration depicted in Fig. B.1 the ATE uses an IXXAT CAN interface, where the VCI component represents the vendor drivers specific to the CAN adapter and the test computer configuration. To abstract the test specification (e.g., a test sequence programmed in Python) from dependencies on the test platform, a layered structure of software (SW) components encapsulates the platform-dependent features:

- *PyDevProxy* is a *Python* class that exposes the remote I/O devices as proxies to the test scripts, hiding the native implementation of the communication library.

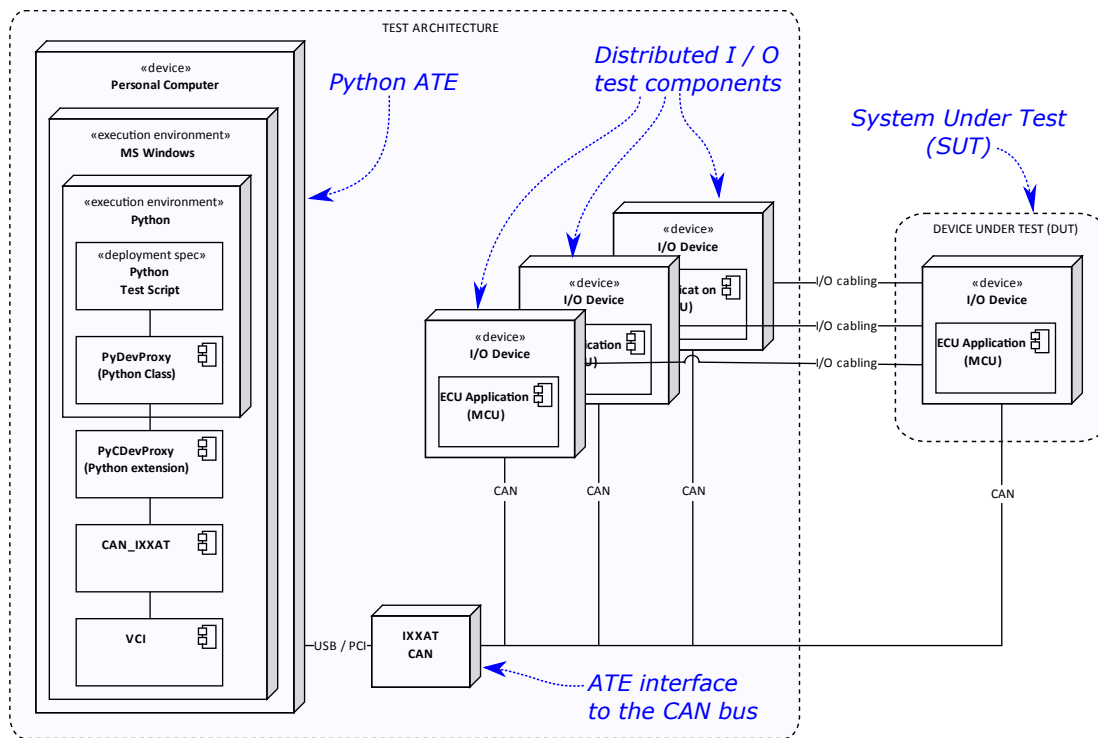


Figure B.1: A distributed test architecture for automated functional verification

In this example the **DUT** is also an **I/O device**, where a new application developed for the **DUT** target undergoes testing. In this scenario the remote **I/O nodes** and the **DUT** may exhibit different behaviours.

- *PyCDevProxy* is a *Python extension* programmed in C/C++, implementing the communication protocols to connect the **ATE** to either the **SUT** or the remote **I/O nodes**. It is intended to re-use available application C code to realize the communication with the remote **I/O devices**. It accesses the **CAN** port through a standardized Application Programming Interface (**API**) interface.
- *CAN_IXXAT* is an optional abstraction layer to provide a generic **API** for the **CAN** interface, to prevent the upper **ATE SW** layers from the vendor-specific dependencies. It is an interface adapter to the selected **CAN** programming support (in the figure we show the IXXAT VCI library).

The para-virtualization enables the partial re-use of the actual source from the **I/O device** application, preserving the platform-independent functionality but replacing the low-level interfaces (e.g. inputs, outputs, communication ports, ...) with replicas suited for the interfaces required in a X-in-the-loop simulator: memory **I/O** interfaces replace the physical **I/Os**, while message queues replace the communication ports integrated the real device.

For some real-time networks (e.g. CAN bus), multiple virtual devices can share a single communication port linking to the SUT, which ultimately could replace most of the slave devices by a special test component: a Restbus simulator. In a real-time hardware-in-the-loop (HiL) test setup like the Hardware-in-the-Loop Elevator Simulator (HiLES) the HiL computing platform provides enough computing performance to emulate hundreds of slave I/O nodes in soft real-time.

Recalling the test scenario depicted in Figure B.1, the Restbus simulator enables the replacement of physical I/O nodes, where the test architecture could be simplified as shown in Figure B.2. In this configuration we re-use the original ATE system unmodified, therefore

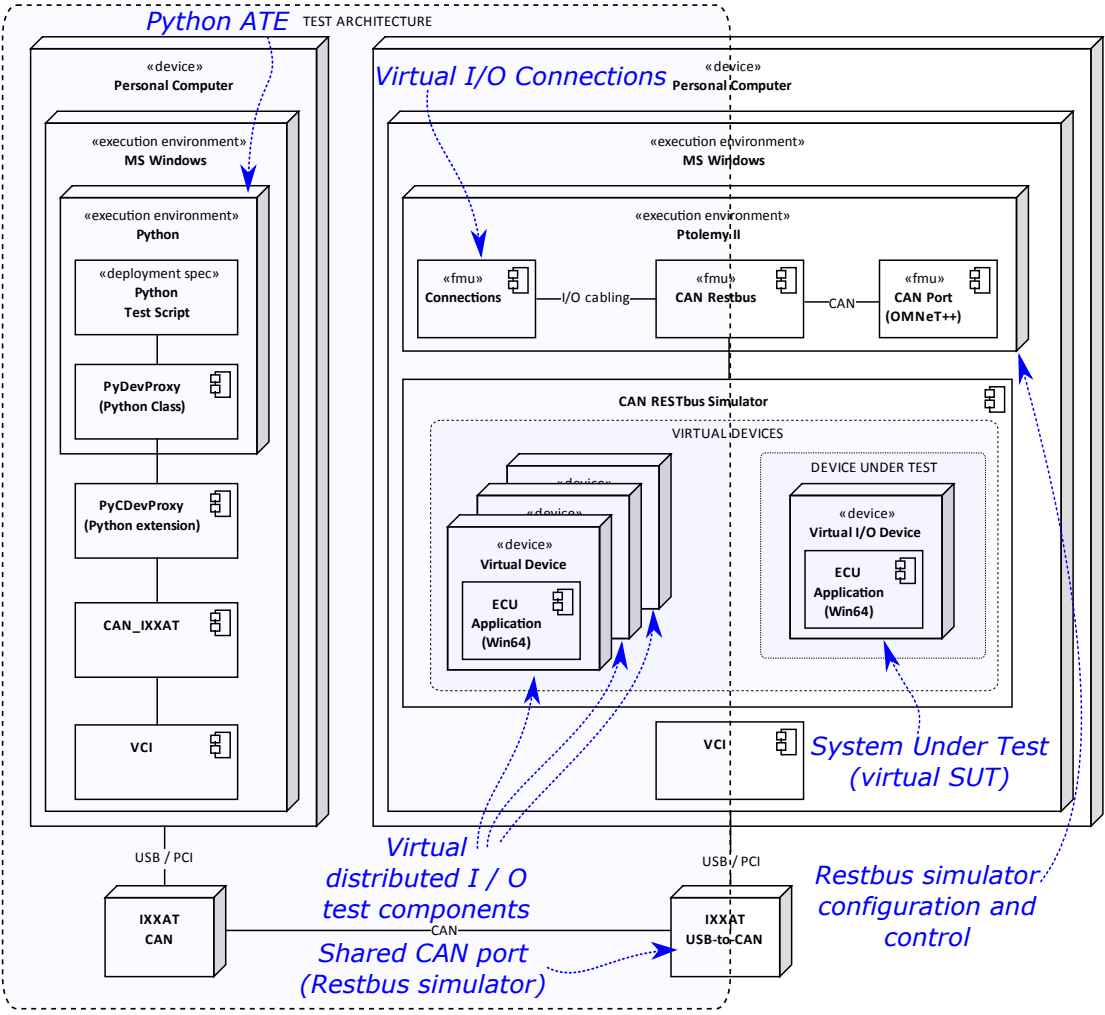


Figure B.2: Partial virtualization of a test architecture using a RESTBUS simulator

it requires a real CAN network to interact with the virtualized nodes. The virtual components are deployed in a separate execution context hosted in an additional computer with a shared CAN interface. To improve composability, the Restbus simulator is encapsulated in a Functional Mock-up Unit (FMU) [FMI], such that it can be instantiated from a simulation environment (e.g. Ptolemy II [PtolemyII]), or even another Python interpreter using the PyFMI adapter [PyFMI].

The virtualization can be further extended to the ATE itself as shown in Figure B.3. Here a bus simulator (e.g. an OMNeT++ model) replaces the real CAN network. As a

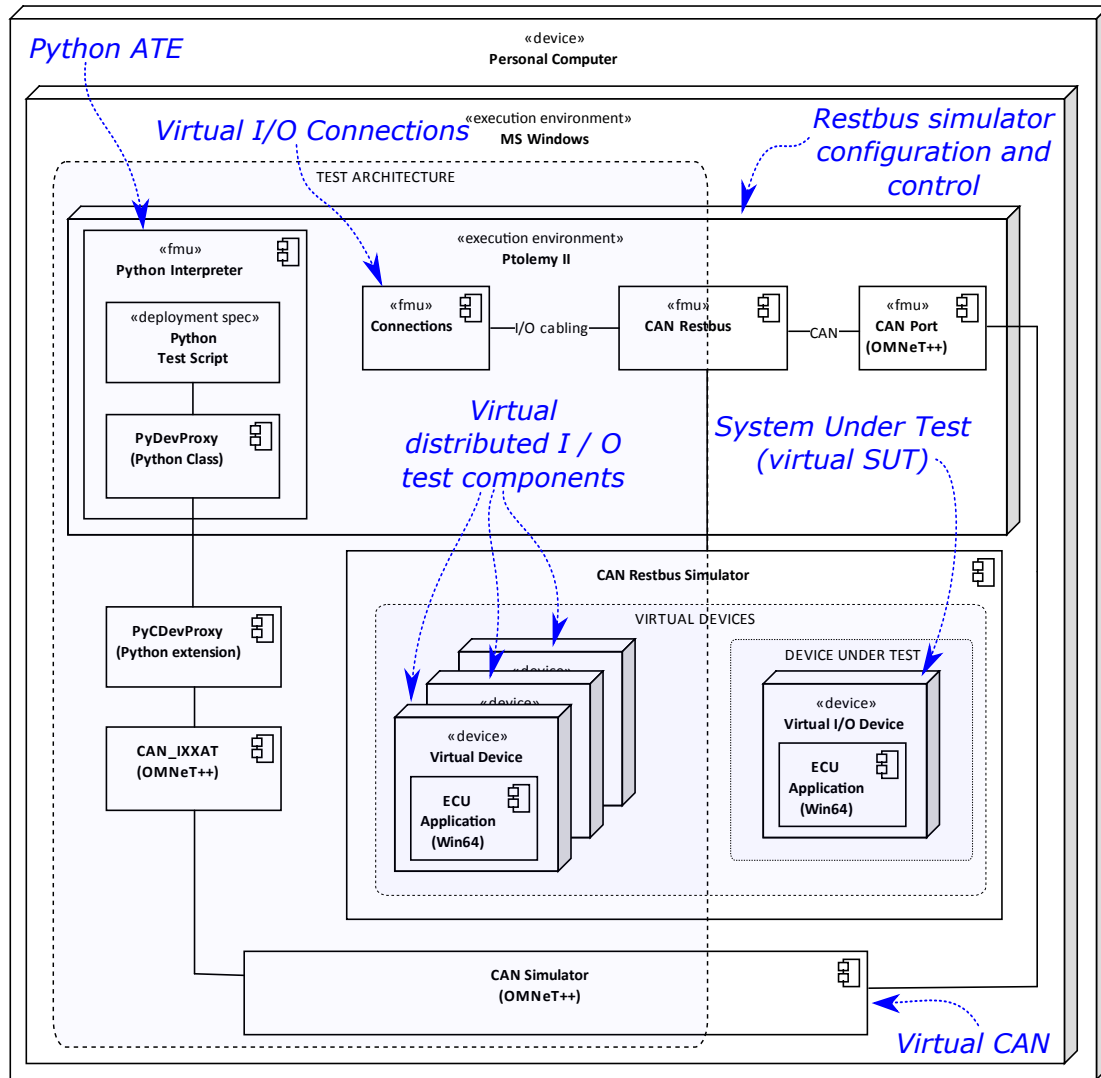


Figure B.3: Virtual Testing: Full virtualization of a test architecture

consequence, the **CAN** abstraction layer in the Python **ATE** is replaced by an OMNeT++ adapter, but the test scripts remain unmodified. Moreover, the Python interpreter for the **ATE** is adapted to be synchronized with the virtual **I/O** devices and the virtual **DUT**. An **FMU** encapsulation of the **ATE** eases the composition of a pure virtual test architecture from the selected simulation environment. In this configuration all the virtualized components can run in virtual time, which enables a synchronization that could better reproduce the temporal behaviour of the original test architecture depicted in **B.1**.

The resulting test infrastructure is well suited for a pure *Virtual Testing* approach, where the test specifications can be exercised against virtual equivalents of hardware (**HW**) components, yet enables the functional verification of the **SW** components, and makes test re-use and test validation more cost-effective.



A Primer on GSN Notation

This appendix introduces the Goal Structuring Notation (**GSN**) notation used in the **GSN** diagrams included in this document, compiling information from Eurocontrol's Safety Case Development Manual [[SCDM06](#)], the GSN standard [[GSN](#)] and the publication from Habli and Kelly [[HK10](#)].

C.1 GSN Basic Notation

Table [C.1](#) lists the basic **GSN** stereotypes.

C.2 GSN Modular Extensions

GSN modular extensions support the development of modular and compositional safety cases [[HK10](#)], reducing the effort of the reassessment of a safety case after system modifications. Using the modular extensions listed in Table [C.2](#) we can partition the argumentation into a set of scoped argument modules whose composition defines the system safety case. Argument packages include Argument Modules and Contract Modules. Each argument module is specified by an interface comprising the goals, context and evidence contained in the module and the references to elements from other modules supporting the module arguments. Inter-module dependencies state the reliance on assumptions, context, goals and evidences, i.e., solutions, found in other modules.

Table C.1: GSN Entities and Relationships


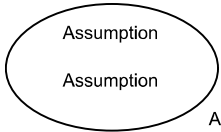
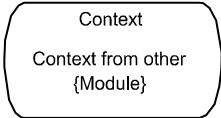



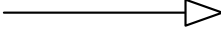
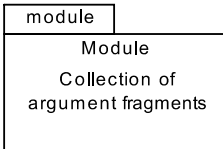
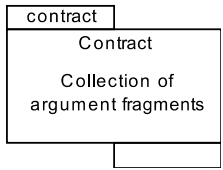
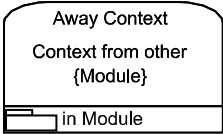
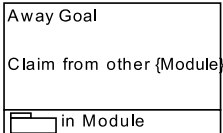
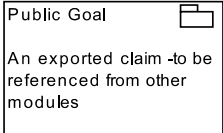
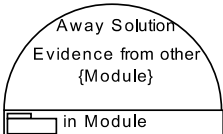
Graphical depiction	Description
	<p>A <i>Goal</i> presents an argument or claim that should take the form of a simple predicate, i.e., a statement that can be shown to be only true or false. Goals can be decomposed into lower-level arguments. For a logical argumentation structure to be sufficient, it is mandatory that at each level of decomposition: 1) The set of arguments covers everything needed to show that the parent claim is true, and 2) there is no valid counter-argument that would undermine the parent claim.</p>
	<p>An <i>Assumption</i> is a statement whose validity has to be relied upon in order to make an Argument. Assumptions may be attached to other GSN elements, including Strategies and Solutions (evidences).</p>
	<p><i>Context</i> provides information needed to understand or specify an Argument or other GSN element. Context may include a statement which limits the scope of an argument in some way.</p>
	<p>A <i>Justification</i> is used to give a rationale for the use or satisfaction of a particular argument or Strategy. More generally it can be used to justify the change that is the subject of a safety case.</p>
	<p>A <i>Solution</i> represents the evidence to support the claims. To consider that an argument structure is complete, every branch must terminate in a reference to the item of evidence that supports the argument to which it is attached. Evidence must be: 1) appropriate to and necessary to support the related Argument. Spurious evidences should be avoided in the logical structure for the sake of comprehension and 2) sufficient to support the related Argument (inadequate evidence undermines the related Argument), as well as all the higher levels of the structure relying on the latter.</p>
	<p><i>SupportedBy</i>, rendered as a line with a solid arrowhead, states a relationship of inferential type (i.e., declares an inference between goals in the argument) or evidence type (i.e., links a goal to the evidence used to substantiate it).</p>
	<p><i>InContextOf</i>, rendered as a line with a hollow arrowhead declares a contextual relationship.</p>

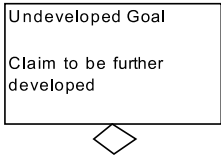
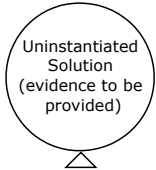
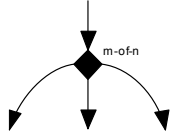
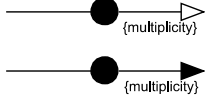
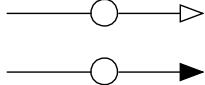
Table C.2: GSN modular extensions to support the development of safety cases.

Graphical depiction	Description
	<p><i>Argument Modules</i> contain the safety arguments, built upon Assumptions, Contexts, Goals, Solutions, and their connections, expressed as InContextOf and SupportedBy relationships.</p>
	<p><i>Contract Modules</i> specify interrelationships between the public elements of the argument modules. Argument contracts aim at preserving the overall integrity of the modular safety-case, minimizing the impact of change between interrelated safety-case modules. The contract links the goals to be supported (in the generic argumentation) with the supporting Goals (in the specific argumentation). Essentially, a <i>safety-case</i> contract captures a 'rely-guarantee' relationship between two argument modules.</p>
	<p>An <i>Away Context</i> is a reference to contextual information contained in another Argument Module.</p>
	<p>An <i>Away Goal</i> is a Goal reference that is used to support or provide contextual backing for an argument presented in one module. However, the argument supporting that goal is presented in another module (hence creating interdependencies between the safety-case modules).</p>
	<p>A <i>Public Goal</i> is a Goal exported for reference from arguments in other modules or contracts.</p>
	<p>An <i>Away Solution</i> is a reference to a Solution presented in another Module that is used to support or provide contextual backing for an argument presented in the module containing the Away Solution reference.</p>

C.3 GSN Pattern Extensions

GSN Pattern Extensions were introduced to model reusable patterns for safety-cases [HK10]. To this end, GSN was extended with structural abstractions for linking the modules and entity abstractions, supporting in turn the generalisation/specialisation of GSN elements (see Table C.3).

Table C.3: GSN pattern extensions to support the abstraction of arguments.

Graphical depiction	Description
	<p>A hollow diamond attached to an entity (e.g., Goal) denotes an <i>Undeveloped Entity</i> that is a placeholder for an element requiring further development. At some later stages, the entity shall be decomposed and supported by sub-entities.</p>
	<p>A hollow triangle attached to an entity (e.g., Goal, Solution) denotes an <i>Uninstantiated Entity</i> that is an 'abstract' element that shall be replaced (i.e., instantiated) with a concrete instance.</p>
	<p>A solid diamond indicates a <i>Choice</i> amongst several possible alternatives to satisfy the relationship (InContextOf or SupportedBy). It can represent a <i>1-of-n</i> or <i>m-of-n</i> selection. Choices are used while developing an argument to show a decision point. However, they must be resolved and removed before completing a safety argument.</p>
	<p>A solid ball indicates that there may be <i>many</i> (meaning zero or more) instances of the relationship (i.e., InContextOf or SupportedBy). The multiplicity label indicates the cardinality of the relationship.</p>
	<p>A hollow ball indicates that the relationship (i.e., InContextOf, SupportedBy) is <i>optional</i> (meaning zero or one).</p>

D

Safety Cases

This annex presents some safety-case documentation structures currently in use for railway signalling and European Air Traffic Management (**EATM**) applications. For the latter, we also review the issues found to automatically generate '*quality*' safety cases, according to the guidelines defined by Eurocontrol.

D.1 Example Safety-Case Structures

This section recalls three safety-case documentation structures: (a) the EN50129 safety-case guideline for railway signalling applications CLC/TR 50506-2 [[TR50506-2](#)], (b) the Safety-Case Development Manual (**SCDM**) for European Air Traffic Management (**EATM**) from Eurocontrol [[SCDM06](#)], and (c) the IEC 62741:2015 standard for Dependability Cases

D.1.1 IEC 62741:2015 Dependability Cases

Closely related to Safety Cases, in 2015 the International Electrotechnical Commission (**IEC**) published the IEC 62741:2015 [[IEC62741](#)], a standard guideline to prepare dependability cases. This standard gives guidance on the content and application of a dependability case and establishes general principles for the preparation of a dependability case.

In order to achieve dependability of a system, dependability requirements should be established, the risks of not meeting them identified and a suitable set of activities developed to meet and demonstrate the requirements and manage the risks. A dependability case provides a convenient and convincing means of recording the output of these activities in

a single location and presenting an argument, supported by evidence, that risks have been treated and that the necessary dependability has been or will be achieved and will continue to be achieved over time.

Dependability cases serve as the main means of communication on dependability among customers, suppliers and other stakeholders and promotes cooperation among them. This is essential for dependability achievement and providing assurance. Certification bodies and regulators may examine a submitted dependability case to support their decisions and users of the system may update/expand the case, particularly where they use the system for a different purpose.

Preparing a dependability case can also improve dependability through the actions taken to prepare and develop the argument within the dependability case. It can improve the cost effectiveness of a dependability programme because if an activity does not provide evidence to support the case, this may indicate that the activity is not necessary.

Therefore, the scope and aim from the certification approach presented in Chapter 7 are aligned with the general principles found in standard IEC 62741:2015. This standard could provide the foundation for an alternative argumentation structure to organize and collate the different safety evidences generated in the development of Mixed-Criticality Systems (MCSs) by application of the contributed framework.

D.1.2 Safety Case Structure for EN 50129

The IEC 61508:2010 safety standard does not provide specific guidelines to develop safety cases. However, the related EN 50129 [EN50129] safety standard for railway applications defines a standardized safety-case template, published as technical report CLC/TR 50506-2. The safety-case information is structured as a set of inter-related documents, summarized in Table D.1 (see [TR50506-2] for a detailed overview). This report also addresses the safety cases modularity and cross-project re-use of certified components, considering the following three layers of safety cases:

1. Generic Product Safety Case (GPSC): Safety case for a component used in generic applications, possibly containing more supportive hardware (HW) and software (SW) Commercial-Off-The-Shelf (COTS) products.
2. Generic Application Safety Case (GASC): Safety case for standard developments shared across multiple projects. These safety cases may rely on GPSCs.
3. Specific Application Safety Case (SASC): Safety case specific to a project, addressing particular data configuration and installation. Complex safety cases may rely on other Safety Cases.

Table D.1: EN 50129 Safety Case documentation structure for railway applications
(adapted from [TR50506-2])

Section	Section Contents
System Definition	This section shall include the system description and general issues.
Quality Management Report	This section shall compile the evidences of Quality Management throughout the entire life cycle of the system under consideration.
Safety Management Report	The Safety Management Report shall demonstrate that the project has been defined, developed and produced in accordance with a safety management process compliant with EN 50126-1 standard, and that the organisational structure satisfies the requirements of EN 50129:2003.
Technical Safety Report (TSR)	<p>Summary of the safety principles, the extent of safety measures and the list of reference standards relevant to the considered system, subsystem, equipment. The TSR shall include the following sections:</p> <p><i>Introduction</i></p> <p><i>Assurance of Correct Operation</i> Shall demonstrate that the correct operation under fault-free normal conditions and to describe how safety requirements are fulfilled.</p> <p><i>Effects of Faults</i> Shall demonstrate that safety requirements continue to be met in the event of random hardware faults.</p> <p><i>Operation with External Influences</i> Shall provide evidences of operation according to the requirements of the safety standard.</p> <p><i>Safety-related application conditions (SRAC)</i> Compilation of exported assumptions and contextual information, generated by safety analysis at each level of the project. For Generic Products or a Generic Applications these constraints shall be explicitly listed in the corresponding Safety Case, so that they can be exported and taken into account by the specific application using the generic ones.</p> <p><i>Safety Qualification Tests</i> Shall demonstrate that the specified operational requirements can be fulfilled by the system / subsystem / equipment under real operating conditions.</p>
Related Safety Cases	When the current Safety Case relies on other Safety Cases these should be referred in this section –e.g., GPSCs.
Conclusion	<p>Sums up the evaluation on the safety element stating that:</p> <ul style="list-style-type: none"> the system is suitable for its intended use, the Quality / Management Process is controlled and no quality issues are open, the Safety Management Process is controlled and all safety issues are closed or forwarded to the system's SRACs, the requested safety target has been reached, the system's SRACs are defined, the SRACs of the related Safety Cases are closed or forwarded to the system's SRACs.

D.1.3 Eurocontrol Safety Case Structure for EATM

Safety Cases (SCs) represented as Goal Structuring Notation (GSN) models have been used for a while in Air Traffic Management. This is covered by the SCDM guideline, which we adopted to build the prototype DREAMS SCCRC Safety-Case Report (SCR) generator. Table D.2 reproduces the format and structure to document a safety case as described in the Safety-Case Development Manual (SCDM) [SCDM06] from EUROCONTROL, intended for air-traffic operations (see examples in [NEFAB, ACDM16]).

Table D.2: Eurocontrol defined Safety Case template for ATM applications
(adapted from [SCDM06])

Section	Section Contents
Executive Summary	The Executive Summary presents guidance on the Safety Case layout.
Introduction	<p>The Introduction should provide the reader with an overview of what the Safety Case is about, what it is trying to show and for whom, a summary of the conclusions and caveats (see below) and recommendations (if any). This section should include:</p> <p><i>Background</i> An outline of, for example, the circumstances which led to the need for, and development of, the Safety Case;</p> <p><i>Aim</i> A simple statement of the aim –i.e., what the Safety Case seeks to demonstrate. It should be related directly to the top-level Claim;</p> <p><i>Purpose</i> The purpose of the Safety Case –i.e., why, and for whom, it has been produced;</p> <p><i>Scope</i> The scope and boundary of the Safety Case. It is important to explain what is included and what is not included;</p> <p><i>Layout</i> The purpose of each of the sections of the document. In general, the main part of the document should be structured along the lines of the Safety Argument.</p>
Service / System Description	Provide a description of the system to which the Safety Case applies, including its operational environment, interfaces and boundaries of responsibility.
Overall Safety Argument	<p>This section should describe and explain the highest levels of the Safety Argument structure, including:</p> <p><i>Claim</i> The Claim –i.e., the top-level statement which asserts that the service / system (etc) is <i>safe</i>;</p> <p><i>Criteria</i> The Safety Criteria which define what is meant by safe in the context of the Claim;</p> <p><i>Context</i> A description of the operational context to which the Safety Case applies;</p> <p><i>Justification</i> When the Safety Case applies to a system modification, this section shall provide the justification for the change, where the Safety Case addresses a change to a service and/or system that is not being made mainly for reasons of improving safety, and therefore potentially for incurring some risk;</p>

Table D.2: Eurocontrol defined Safety Case template for ATM applications
(continued)

Section	Section Contents
	<p><i>Principal Safety Arguments</i> The principal Safety Arguments –i.e., the first level of decomposition of the top-level Claim – these should be reasoned and well structured, showing how the Safety Criteria are satisfied and the rationale for the approach taken in the decomposition;</p> <p><i>High-level Assumptions</i> The key Assumptions on which the highest levels of the Safety Argument critically depend –e.g., the level of risk prior to the introduction of a change is acceptable. Other Assumptions, applicable to the lower levels of the Safety Argument structure should be included in the Assumptions section.</p>
Safety Argument and Evidence sections	<p>These sections should present each of the principal Safety Arguments in turn, together with the supporting Evidence which shows that each of the Arguments is valid.</p> <p>It is recommend that, where applicable, each section be structured as follows:</p> <ul style="list-style-type: none"> Objective (of the section) –related directly to the principal Safety Argument; Strategy (breakdown of the principal Safety Argument into lower-level arguments); Rationale (for the Strategy); Lower-level Arguments / Evidence; Conclusions (of section).
Assumptions	<p>Present directly, and/or by reference, all the Assumptions on which the Safety Case depends, including the high-level Assumptions mentioned above. Assumptions usually relate to matters outside of the direct control of the organisation responsible for the Safety Case but which are essential to the completeness and/or correctness of the Safety Case. Each Assumption must be shown to be valid or at least reasonable according to the circumstances.</p>
Issues	<p>List any outstanding safety issues that must be resolved before the Claim can be considered to be valid, together with the responsibilities and timescales for clearing them.</p>
Limitations	<p>State and explain any limitations or restrictions that need to be placed on the deployment and / or operation of the system.</p>
Conclusions	<p>The main conclusion should refer to the original Claim and, if applicable, reassert its validity, subject to the following caveats:</p> <ul style="list-style-type: none"> the Scope –especially what the Safety Case does not cover; the operational Context to which the Safety Case applies; the Assumptions that have had to be made; the outstanding Issues; any limitation placed on the deployment and/or operation of the service / system.

Table D.2: Eurocontrol defined Safety Case template for ATM applications
(continued)

Section	Section Contents
Recommendations	<p>Recommendations are not mandatory and any that are made should not be temporary in nature. For example, it might be appropriate to make recommendations on the use of the Safety Case by its recipients, but not concerning its approval.</p> <p>Recommendations must not contain any statements that would undermine, or add further caveats to, the Conclusions.</p>

D.2 Issues in Automated Verification of Safety Cases

This section examines some of the difficulties arising when attempting to automatically generate quality Safety-Case Reports (SCRs) to describe the safety rationale justifying the safety claims.

Before attempting the certification of a safety system by using safety cases, it is advisable to check several aspects of the documentation to be presented to a certification body, ranging from the format and the logical structure to the semantics of the contents. Safety cases have been in use for several years in the Air Navigation domain. As a result, the European Organization for the Safety of Air Navigation published a safety-case development manual [SCDM06]. Although this guideline targets systems and process for Air Traffic Management, most of the recommendations would also be applicable to the development of other types of certifiable safety systems, in particular the subject of DREAMS project: the production of safety cases for Mixed-Criticality Product Line (MCPL).

Table D.3 recalls the check rules enumerated in [SCDM06], where subjective terms are written in italics on a distinctive background colour, and also identifies which rules are susceptible of automated checking. It should be recalled that the purpose of a safety case is to present the information in a way amenable to examination by humans, and noticeably most of the rules contain subjective qualifiers (e.g., *‘actually’*, *‘appropriate’*, *‘clear’*, *‘correctly’*) and therefore are not suited to automated processing by a tool as DREAMS Safety Compliance Constraints & Rules Checker (SCCRC).

Table D.3: Safety Case check rules and verification roles
(adapted from [SCDM06])

Rule No.	Property to check	Verifiable by ¹²	
		SCCRC	Expert Reviewer
Safety Case Presentation: General			
1	Is the aim of the Safety Case explained and <i>clear</i> ?		●
2	Is the purpose of the Safety Case <i>explained</i> and <i>clear</i> ?		●
3	Is the scope of the Safety Case <i>explained</i> and <i>clear</i> ?		●
4	Is a justification given as to why the subject of the Safety Case is <i>necessary</i> ?		●
5	Are the 'system' and its environment <i>completely</i> and <i>correctly</i> described and bounded?		●
6	Is the operational concept <i>described</i> ?		●
7	Is the regulatory context <i>described</i> ?		●
8	Is the Safety Case <i>structured along the lines</i> of the Argument?		●
9	Is the Argument structure <i>apparent</i> in the layout of each of the core sections?		●
Argument Structure			
10	Is the overall Claim a single, <i>clear</i> and <i>unambiguous</i> statement of what the Safety Case is trying to demonstrate?		●
11	Is the Claim expressed <i>in a positive way</i> – ie does it accept the "burden of proof"?		●
12	Is the context <i>clear</i> ?		●
13	Are the criteria for being 'acceptably safe' <i>appropriate</i> and <i>adequately</i> specified?		●
14	Are the initial assumptions <i>explicitly</i> stated?	▸	●
15	Is the decomposition of the Argument structure <i>adequately</i> explained by "Strategies"?	▸	●
16	Is the level of decomposition <i>appropriate</i> to the complexity of the Safety Case and/or Evidence?		●
17	Is each level of decomposition <i>necessary</i> and <i>sufficient</i> to show that the parent Argument is true?		●
18	Is each Argument set out as a <i>simple</i> predicate?		●
19	Is the Argument structure free of negative and <i>inconclusive</i> Arguments? [Lack of evidence of risk ≠ Evidence of lack of risk]		●
20	Does the Argument structure appear to be <i>immune to possible counter Arguments</i> which could undermine the top-level Claim?		●
21	Is the distinction between product- and process-based Arguments <i>clear</i> ?		●
22	Are Arguments supposedly related to the observable properties of the related product (i.e. Direct Arguments) <i>actually addressing</i> the outputs of a process?		●
23	Are Arguments supposedly related to the observable properties of the related processes which generated that product (i.e. Backing Arguments) <i>actually addressing</i> the process?		●

¹² Legend: ▸ = Partially achievable; ● = Feasible

Table D.3: Safety Case check rules and verification roles
(continued)

Rule No.	Property to check	Verifiable by ¹²	
		SCCRC	Expert Reviewer
24	Are Direct Arguments and Evidence supported by enough Backing Arguments and Evidence to give <i>sufficient confidence</i> in the former?		●
25	Where process-based Arguments are used as Direct Arguments, is this <i>appropriate</i> ?		●
26	Is each branch of the Safety Argument structure terminated in Evidence?	●	●
Evidence			
27	Is all the presented Evidence <i>necessary</i> to support the Argument to which it relates?		●
28	Are all the presented Evidences <i>clear</i> , objective, <i>relevant</i> and <i>conclusive</i> in showing the related Argument to be true?		●
29	Is the rigour of the Evidence <i>appropriate</i> to the associated risk –i.e. is it to the required level of assurance?		●
30	Has the Evidence been produced from following an <i>accepted</i> and <i>recognised</i> methodology?		●
31	Is the underlying safety analysis <i>sound</i> ?		●
32	Does the safety analysis address both the <i>desired</i> and <i>undesired</i> behaviour of the ‘system’?		●
33	Are the various possible types of Evidence –design, test, previous usage, etc.– used <i>appropriately</i> ?		●
34	Where Evidence is contained in appendices or external documents, is an <i>adequate</i> summary presented in the body of the Safety Case alongside the related Argument?		●
35	Where Evidence is based on compliance with standards, is its usage <i>appropriate</i> and <i>justified</i> ?		●
36	Does the Evidence <i>actually</i> relate to the system / configuration under consideration?		●
Caveats			
37	Have all the Assumptions been <i>clearly</i> stated and validated, or responsibilities for validation been stated?		●
38	Have all the <i>outstanding</i> Issues been cleared, or responsibilities for clearing them been stated?		●
39	Have Limitations on the scope of the analysis been <i>clearly</i> stated?		●
40	Have Limitations on the deployment / operation of the ‘system’ been <i>clearly</i> stated?		●
Conclusions			
41	Is there a <i>clear</i> statement of what the Safety Case concludes, which relates to the initial, overall Claim?		●
42	Is it made <i>clear</i> that the conclusions are subject to the stated Caveats (see above)?		●

¹² Legend: ◐ = Partially achievable; ● = Feasible



DREAMS Preliminary Safety Case Report

This annex presents an example of a preliminary Safety-Case Report (SCR) generated using the DREAMS toolset described in Chapter 7. The SCR is the transcription of the rationale implicit to the automated safety validation carried out by the Safety Compliance Constraints & Rules Checker (SCCRC) component. The SCR documents the safety argumentation for a single product sample in the Mixed-Criticality Product Line (MCPL), making it amenable to human review by safety certification bodies. The overall safety argumentation for a whole MCPL would consist of a collection of product-specific SCRs. When using the DREAMS workflow to develop an MCPL, the product line can be refined using the Design Space Exploration (DSE) tool. The DSE calls the SCCRC twice:

1. The first pass corresponds to the DSE optimization phase, when the SCCRC examines the satisfiability of the safety requirements by each design under examination, according to the IEC 61508 safety standard recommendations. This involves several rounds of safety evaluations for candidate product samples, for which the SCCRC feeds back a PASS / FAIL verdict.
2. The second pass corresponds to the SCR documentation phase, when SCCRC re-visits the safety assessment for a previously validated product configuration, this time in verbose mode, generating a \LaTeX transcript justifying the design validity.

The SCR included herein was generated for the *Wind Turbine Controller Case Study* (see §8.3). 'TODO' markers are reminders of unavailable information in the GSN argumentation model when generating the report, to be completed later. SCCRC instantiates a product-specific GSN model each time eventually linking to the DREAMS Modular Safety Cases

(MSCs), and could refer to other available evidence sources, e.g., analysis and testing results.

WP7 Wind Power Test Case Safety Case Report

DREAMS Consortium
Wednesday 31st August, 2016

1

Contents

1 Executive Summary	4
2 Introduction	5
2.1 Background	5
2.2 Aim	5
2.3 Purpose	5
2.4 Scope	5
2.5 Layout	5
3 System Description	6
3.1 Main System Description	6
3.1.1 (SSR) System Safety Requirements Specification	6
3.1.2 (SSFR) System Safety Functions Requirements Specification	7
3.1.3 (SSFSWR) System Safety Functions Software Safety Requirements	7
3.1.4 System Safety Usage Constraints	8
3.2 System/Subsystems Description	8
4 Overall Safety Argumentation	9
4.1 Claim	9
4.2 Safety Criteria	10
4.3 Context	10
4.4 Justification	10
4.5 Main Safety Argument	10
4.6 Key Assumptions	10
5 Safety Argument and Evidence Sections	11
5.1 Safety Case: Safety Case	11
5.1.1 Argument Module: GSN Root	11
5.2 Safety Case: Safety Case	13
5.2.1 Argument Module: SRRule-IECG1508-2-Check-JustifiableSILfrom-SafetyFunctionArchs	13
5.3 Safety Case: Safety Case	14
5.3.1 Argument Module: SRRule-IECG1508-2-Check-JustifiableSILfrom-SafetyFunctionArchs[tooID]	14

2

5.4 Safety Case: Safety Case	15
5.4.1 Argument Module: SRRule-Check-UsageConstraints	15
6 Assumptions	18
7 Issues	19
8 Limitations	20
9 Conclusions	21
10 Recommendations	22
11 Annex A - Template usage guide.	23

3

Chapter 1

Executive Summary

This should provide the reader with an overview of what the Safety Case is about, what it is trying to show and for whom, a summary of the conclusions and caveats (see below) and recommendations (if any).

This Safety Case layout follows the proposal described in the *Safety Case Development Manual* (Edition : 2.2, Edition Date : 13 Nov 2006, Document Identifier DAP/SSH/091) developed by EATM (European Air Traffic Management). Examples, relating to EATM can be found in the EUROCONTROL Pre- and Post-Implementation Safety Cases for RVSM:

1 The EUR RVSM Pre-Implementation Safety Case, Edition 2.0, 14 August 2001

2 The EUR RVSM Post-Implementation Safety Case, Edition 2.0, 28 July 2004

TODO: Add content

4

Chapter 2

Introduction

2.1 Background

TODO: Background should include an outline of, for example, the circumstances which led to the need for, and development of, the Safety Case.

2.2 Aim

TODO: Aim should include a simple statement of the aim - **what** the Safety Case seeks to demonstrate. It should be related directly to the top-level Claim (see below).

2.3 Purpose

TODO: Purpose should include the purpose of the Safety Case - **why**, and **for whom**, it has been produced.

2.4 Scope

TODO: Scope should include the scope and boundary of the Safety Case. It is important to explain what is included **and** what is not included.

2.5 Layout

TODO: Layout should include the purpose of each of the sections of the document. In general, the main part of the document should be structured along the lines of the Safety Argument.

5

Chapter 3

System Description

Provide a description of the system to which the Safety Case applies, including its operational environment, interfaces and boundaries of responsibility.

TODO: Add content

3.1 Main System Description

This section provides a description of the system from the safety point of view, including:

- System Safety Requirements Specification
- System Safety Functions Requirements Specification
- System Safety Functions Software Safety Requirements
- System Safety Usage Constraints

3.1.1 (SSR) System Safety Requirements Specification

This section list the IEC-61508 Phase 9 - E/E/PE **System Safety Requirements** Specifications of the system:

- **SSR:** SSR1 - When Speed Sensor Value $\geq 100\%$ => activate SafetyRelays in less than 50 ms
 - SIL3
 - LowDemandMode
 - Response Time 50ms

6

- Traceability - This requirement is provided by system Safety Function Software Safety Requirement **SSFSWR:** SafetyProtection

- **SSR:** SSR2 - When Vibration Sensor Value $\geq 100\%$ => activate SafetyRelays in less than 50 ms

- SIL3
- LowDemandMode
- Response Time 50ms

- Traceability - This requirement is provided by system Safety Function Software Safety Requirement **SSFSWR:** SafetyProtection

- **SSR:** SSR3 - When Voltage PT Sensor Value $\geq 100\%$ => activate SafetyRelays in less than 50 ms

- SIL3
- LowDemandMode
- Response Time 50ms

- Traceability - This requirement is provided by system Safety Function Software Safety Requirement **SSFSWR:** SafetyProtection

3.1.2 (SSFR) System Safety Functions Requirements Specification

This section provides the list of functional specification of the **System Safety Functions Requirements** capable of being performed by the system. These functions are the following:

- **SSFR:** SafetyProtection - Function activating Safety Relays when Excessive Speed, Vibration or Voltage is detected

- Traceability - This function is provided by system Safety Function Software Safety Requirement **SSFSWR:** SafetyProtection

3.1.3 (SSFSWR) System Safety Functions Software Safety Requirements

This section provides the list of the **System Safety Functions Software Safety Requirements** that provide the **System Safety Functions Requirements** of the system. These software function requirements are the following:

- **SSFSWR:** SafetyProtection - Function activating Safety Relays when Excessive Speed, Vibration or Voltage is detected

7

3.1.4 System Safety Usage Constraints

3.2 System/Subsystems Description

This section provides a description of the system/subsystems elements that compose the system from the safety point of view, including:

- System Safety Compliant SW/HW Items
 - Safety SW Components
 - Safety HW Platform Architecture
 - Safety SW Hypervisors/Partitions

For each system/subsystem the following information is detailed:

- System Safety Requirements
 - System Safety Requirements Specification
 - System Safety Functions Requirements Specification
 - System Safety Functions Software Safety Requirements
- System Safety Usage Constraints

8

Chapter 4

Overall Safety Argumentation

This section describes and explains the highest levels of the Safety Argument structure, including:

- the Claim - ie the top-level statement which asserts that the service / system (etc) is safe;
- the Safety Criteria which define what is meant by safe in the context of the Claim; a description of the operational context to which the Safety Case applies;
- the Context - ie a description of the operational context to which the Safety Case applies;
- the justification for the change, where the Safety Case addresses a change to a service and/or system that is not being made mainly for reasons of improving safety, and therefore potentially for incurring some risk;
- the principal Safety Arguments - ie the first level of decomposition of the top-level Claim - these should be reasoned and well structured, showing how the Safety Criteria are satisfied and the rationale for the approach taken in the decomposition;
- the key Assumptions on which the highest levels of the Safety Argument critically depend - for example, the level of risk prior to the introduction of a change is acceptable. Other Assumptions, applicable to the lower levels of the Safety Argument structure should be included in the Assumptions section.

4.1 Claim

TODO:

9

4.2 Safety Criteria

TODO:

4.3 Context

TODO:

4.4 Justification

TODO:

4.5 Main Safety Argument

TODO:

4.6 Key Assumptions

TODO:

10

Chapter 5

Safety Argument and Evidence Sections

These sections presents each of the principal Safety Argument Modules in turn, together with the supporting Evidence which shows that each of the Arguments is valid. Each argument module is structured as follows:

- Root Claim related directly to the principal Safety Argument;
- Strategy (breakdown of the principal Safety Argument into lower-level arguments);
- Rationale (for the Strategy);
- Lower-level Arguments / Evidence;

5.1 Safety Case: Safety Case

5.1.1 Argument Module: GSN Root

Goal: G0-Root

Claim:

- System Control Electronic is Safe

Supported By:

- Goal: G1-Root - HS/SW Architecture Design is Sensible
- Goal: G2-Root - V&V Tests done

In Context Of:

- Assumption: A1 - IEC 61508 FMS is applied

11

Goal: G1-Root

Claim:

- HS/SW Architecture Design is Sensible

Supported By:

- Away Goal: G0-UsageConstraints - System's Usage Constraints covers all Usage Constraints of Safety SubSystems. System's Usage Constraints are:

[

SystemListUsageConstraints]

- Away Goal: G0 - Architecture Style of all Safety Functions SFi justify SIL/SC/HFT Reqs

In Context Of:

- No InContextOf nodes defined

Goal: G2-Root

Claim:

- V&V Tests done

Supported By:

- **TODO: Unsupported claim** - No solution has been defined to support the claim

In Context Of:

- No InContextOf nodes defined

Assumption: A1

Claim:

- IEC 61508 FMS is applied

Away Goal: G0-UsageConstraints

Claim:

- System's Usage Constraints covers all Usage Constraints of Safety SubSystems. System's Usage Constraints are:

[

SystemListUsageConstraints]

12

Away Goal: G0
Claim:

- Architecture Style of all Safety Functions SFi justify SIL/SC/HFT Reqs

5.2 Safety Case: Safety Case
5.2.1 Argument Module: SFRule-IEC61508-2-Check-JustifiableSILfro
Goal: G0
Claim:

- Architecture Style of all Safety Functions SFi justify SIL/SC/HFT Reqs

Supported By:

- Goal: G1 - Safety Function SafetyProtection justifies SIL2/SC3/HFT-0 Reqs

In Context Of:

- Context: C000 - IEC 61508 architectures 1oo1, 1oo1D, 1oo2, 1oo2D are described in IEC61508 Part 6
- Justification: J000 - If each Safety Function complies with ...

Goal: G1
Claim:

- Safety Function SafetyProtection justifies SIL2/SC3/HFT-0 Reqs

Supported By:

- Away Goal: GOAL-1oo1D - Safety Function SafetyProtection follows 1oo1D and justifies SIL2/SC3/HFT-0

In Context Of:

- Assumption: A000 - 1ooX[D] architecture is followed by Safety Functions

Context: C000
Claim:

- IEC 61508 architectures 1oo1, 1oo1D, 1oo2, 1oo2D are described in IEC61508 Part 6

Justification: J000
Claim:

- If each Safety Function complies with ...

Away Goal: GOAL-1oo1D
Claim:

- Safety Function SafetyProtection follows 1oo1D and justifies SIL2/SC3/HFT-0

Assumption: A000
Claim:

- 1ooX[D] architecture is followed by Safety Functions

5.3 Safety Case: Safety Case
5.3.1 Argument Module: SFRule-IEC61508-2-Check-JustifiableSILfro
Goal: GOAL-1oo1D
Claim:

- Safety Function SafetyProtection follows 1oo1D and justifies SIL2/SC3/HFT-0

Supported By:

- Strategy: ST1 - Argument that 1oo1D requirements are met according to IEC 61508

In Context Of:

- No InContextOf nodes defined

Strategy: ST1
Claim:

- Argument that 1oo1D requirements are met according to IEC 61508

Supported By:

- TODO: Unsupported claim** - No solution has been defined to support the claim

In Context Of:

- No InContextOf nodes defined

5.4 Safety Case: Safety Case
5.4.1 Argument Module: SFRule-Check-UsageConstraints
Goal: G0-UsageConstraints
Claim:

- System's Usage Constraints covers all Usage Constraints of Safety Sub-Systems. System's Usage Constraints are: LISTA

Supported By:

- Goal: G1-UsageConstraints-Components - System's Usage Constraints covers all Usage Constraints of Components

In Context Of:

- No InContextOf nodes defined

Goal: G1-UsageConstraints-Components
Claim:

- System's Usage Constraints covers all Usage Constraints of Components

Supported By:

- Goal: G1.1-UsageConstraints-Components - Component SafetyProtectionMicroBlaze Usage Constraints -RANGE in [-25.0 .. 60.0]-VERSION >= 3.0are covered
- Goal: G1.1-UsageConstraints-Components - Component DiagnosticMicroBlaze Usage Constraints Usage Constraints defined for this component are covered

In Context Of:

- No InContextOf nodes defined

Goal: G1.1-UsageConstraints-Components
Claim:

- Component SafetyProtectionMicroBlaze Usage Constraints -RANGE in [-25.0 .. 60.0]-VERSION >= 3.0are covered

Supported By:

- Solution: S1.1.1-UsageConstraints-Components - Usage Constraints are covered are Covered by Parent's Usage Constraints

In Context Of:

- No InContextOf nodes defined

Goal: G1.1-UsageConstraints-Components
Claim:

- Component DiagnosticMicroBlaze Usage Constraints Usage Constraints defined for this component are covered

Supported By:

- Solution: S1.1.1-UsageConstraints-Components - Usage Constraints are covered are Covered by Parent's Usage Constraints

In Context Of:

- No InContextOf nodes defined

Solution: S1.1.1-UsageConstraints-Components
Claim:

- Usage Constraints are covered are Covered by Parent's Usage Constraints

Supported By:

- TODO: Unsupported claim** - No solution has been defined to support the claim

In Context Of:

- Justification: J1.1.1-UsageConstraints-Components - Usage Constraints are covered as follows: -RANGE in [-25.0 .. 60.0] is covered by TEMP-RANGE in [-25.0 .. 60.0]-VERSION >= 3.0 is covered by OS-VERSION >= 3.0
- Justification: J1.1.1-UsageConstraints-Components - Usage Constraints are covered as follows: Usage Constraints defined for this component.

Solution: S1.1.1-UsageConstraints-Components
Claim:

- Usage Constraints are covered are Covered by Parent's Usage Constraints

Supported By:

- TODO: Unsupported claim** - No solution has been defined to support the claim

In Context Of:

- Justification: J1.1.1-UsageConstraints-Components - Usage Constraints are covered as follows: -RANGE in [-25.0 .. 60.0] is covered by TEMP-RANGE in [-25.0 .. 60.0]-VERSION >= 3.0 is covered by OS-VERSION >= 3.0
- Justification: J1.1.1-UsageConstraints-Components - Usage Constraints are covered as follows: Usage Constraints defined for this component.

Justification: J1.1.1–UsageConstraints–Components

Claim:

- Usage Constraints are covered as follows: –RANGE in [-25.0 .. 60.0] is covered by TEMP–RANGE in [-25.0 .. 60.0]–VERSION >= 3.0 is covered by OS–VERSION >= 3.0

Justification: J1.1.1–UsageConstraints–Components

Claim:

- Usage Constraints are covered as follows: Usage Constraints defined for this component.

Justification: J1.1.1–UsageConstraints–Components

Claim:

- Usage Constraints are covered as follows: –RANGE in [-25.0 .. 60.0] is covered by TEMP–RANGE in [-25.0 .. 60.0]–VERSION >= 3.0 is covered by OS–VERSION >= 3.0

Justification: J1.1.1–UsageConstraints–Components

Claim:

- Usage Constraints are covered as follows: Usage Constraints defined for this component.

17

Chapter 6

Assumptions

This section presents directly, and/or by reference, all the Assumptions on which the Safety Case depends, including the high-level Assumptions mentioned above. Assumptions usually relate to matters outside of the direct control of the organisation responsible for the Safety Case but which are essential to the completeness and/or correctness of the Safety Case. Each Assumption must be shown to be valid or at least reasonable according to the circumstances.

TODO:

18

Chapter 7

Issues

This section lists any outstanding safety issues that must be resolved before the Claim can be considered to be valid, together with the responsibilities and timescales for clearing them.

TODO:

19

Chapter 8

Limitations

This section states and explains any Limitations or restrictions that need to be placed on the deployment and/or operation of the system.

TODO:

20

Chapter 9

Conclusions

TODO:

This section should not merely repeat the conclusions from each section here. The main conclusion should refer to the original Claim and, if applicable, re-assert its validity, subject to the following caveats:

- the Scope - especially what the Safety Case does not cover;
- the operational Context to which the Safety Case applies;
- the Assumptions that have had to be made;
- the outstanding Issues;
- any Limitations placed on the deployment and/or operation of the service / system.

21

Chapter 10

Recommendations

Recommendations are not mandatory and any that are made should not be temporary in nature. For example, it might be appropriate to make recommendations on the use of the Safety Case by its recipients, but not concerning its approval.

Recommendations must not contain any statements that would undermine, or add further caveats to, the Conclusions.

TODO:

22

Chapter 11

Annex A - Template usage guide.

DREAMS WP4 tools generate this basic skeleton of Safety Case Report collecting information contained in the models of the product, particularly from the Safety model and the GSN argumentation generated from the model.

The present document represents the generated basic skeleton of Safety Case Report for a given product. This document must be completed manually according to the following rules:

- Texts in Black must be kept as-is, therefore cannot be modified.
- Texts in Blue represent suggestions/recommendations that explain the nature and purpose of the chapter/section in which they appear. These texts in blue must be eliminated from the final Safety Case Report.
- Texts in Red represent a TODO section. This section must be written manually by the authors of the Safety Case Report.

IMPORTANT: This Annex itself must be eliminated from the final Safety Case Report.

23