Contents

Abstract iii

Acknowledgments ix

1 Introduction 1

1.1 Historical Perspective 2

1.1.1 IEC 61508 and Related Standards 3

1.2 Model-Driven Engineering 4

1.2.1 Model-Driven Development 4

1.2.2 Model-Driven Development using IEC 61499 5

1.2.3 Model-Based Safety Assessment 6

1.3 Contributions 7

1.4 Thesis Organisation 9

2 Background 11

2.1 Boiler Control System 11

2.2 Defects, Errors and Failures 13

2.3 System Reliability . . . 14
2.4 Reliability Modelling and Analysis . . . 15
2.4.1 Probability of Failure . . . 16
2.4.2 Failure Mode and Effect Analysis . . . 17
2.4.3 Reliability Block Diagram . . . 18
2.4.4 Fault Tree Analysis . . . 19
2.4.5 Markov Analysis . . . 21
2.5 Formal Verification . . . 22
2.5.1 Model Checking . . . 22
2.5.2 Temporal Logic . . . 23
2.5.3 Safety and Liveness Properties . . . 23
2.5.4 Probabilistic Verification . . . 24
2.6 Recap . . . 24
2.7 Related Work . . . 25
2.7.1 Classification . . . 25
2.7.2 AADL based approaches . . . 26
2.7.3 SAML . . . 27
2.7.4 Figaro . . . 27
2.7.5 Hip-HOPS . . . 27
2.7.6 Simulink based approaches . . . 28
2.7.7 SysML and UML-based approaches . . . 28

2.7.8 AltaRica 28

2.7.9 Stochastic Function Blocks 29

2.7.10 Discussion 30

3 IEC 61508 in a Nutshell: *An Introduction to Functional Safety* **31**

3.1 Basic Concepts 32

3.1.1 Hazard and Risk 32

3.1.2 As Low as Reasonably Possible (ALARP) 33

3.1.3 Safety Function 34

3.1.4 Safety Integrity Levels 34

3.2 Meeting Basic Requirements 34

3.2.1 Establishing SIL Targets 37

3.2.2 Function Safety Assessment 39

3.3 Meeting Hardware Requirements 41

3.3.1 E/E/PE System Design Requirements Specification 42

3.3.2 E/E/PE System Safety Validation Planning 42

3.3.3 E/E/PE System Design and Development 42

3.3.4 Reliability Modelling 43

3.3.5 Safe Failure Fraction 43

3.4 Meeting Software Requirements 44

3.4.1 Software Functional Safety Plan 45

3.4.2 V-Model for Software Development 45

3.4.3 Software Safety Specification and Validation 46

3.4.4 Software Design and Development 47

3.4.5 Integration and Testing 47

3.4.6 Operation and Modification 47

3.4.7 Software Verification 48

3.4.8 Software Functional Safety Assessment 48

3.4.9 IEC 61508-3 Annex. A 48

3.4.10 IEC 61508-3 Annex. B 48

3.5 Summary 49

3.6 Discussion 50

4 An Introduction to IEC 61499: *Model-Based Design using Function Blocks* **51**

4.1 Distribution station 51

4.2 Basic function block 53

4.2.1 A function block interface 53

4.2.2 Execution control chart (ECC) 54

4.2.3 Algorithms 55

4.3 Composite function blocks 56

4.3.1 Type specification 56

4.4 Service interface function blocks 58

4.5 System, Devices, and Resources 62

4.5.1 Device model 62

4.5.2 Resource model 62
4.5.3 System model 63
4.5.4 Implementation of the Distribution Station 64
4.6 Execution models for Function Blocks 66
4.6.1 FBRT 67
4.6.2 FORTE 67
4.6.3 FUBER 68
4.6.4 ISaGRAF 68
4.6.5 Synchronous Execution 68
4.7 Discussion 69

5 Converting Function Blocks to Prism 71

5.1 The Boiler Control System 72
5.2 Formalisation 73
5.2.1 Basic Function Blocks 73
5.2.2 Synchronous Execution of BFBs 76
5.2.3 Composite Function Blocks 77
5.2.4 Synchronous Execution of CFBs 80
5.3 The Prism Language 81
5.3.1 Markov Decision Processes 81
5.3.2 Prism Model and Modules 83
5.3.3 Execution of a Prism Module 86

5.4 Converting Function Blocks to Prism 87
5.4.1 Generating Variables 90
5.4.2 Generating Commands 94
5.4.3 Algorithm for Generating the Prism Model 102
5.5 Preserving the Execution Semantics 105
5.5.1 Execution States 105
5.5.2 Transitions 108
5.5.3 Bisimulation Equivalence 109
5.5.4 Model Verification 115
5.6 Discussion 115

6 Stochastic Function Blocks: *Model-Based Safety of IEC 61499 Systems* **117**

6.1 Overview 118
6.2 Stochastic Function Blocks (SFB) 120
6.2.1 Formalisation 121
6.2.2 Execution of SFBs 122
6.2.3 Composition of Stochastic Function Blocks 124
6.3 Transformation to PRISM 125
6.3.1 Illustration 125
6.3.2 Semantics Preserving Rule-Based Transformation 127
6.3.3 Modifying Inherited Rules for SFBs 127
6.3.4 Encoding Stochastic Transitions 129

6.3.5 Algorithm for Generating the Prism Model . . . 133
6.4 Preserving the Execution Semantics . . . 136
6.4.1 Transition . . . 137
6.4.2 Simulation Relation . . . 137
6.4.3 Preservation of LTL Properties . . . 141
6.5 Safety Analysis . . . 142
6.5.1 Probability of Failure . . . 142
6.5.2 Failure Mode and Effect Modelling . . . 143
6.5.3 Probabilistic Model Checking . . . 144
6.6 Results . . . 145
6.6.1 Scalability . . . 147
6.7 Discussion . . . 148

7 BlokIDE: *An IDE for Model-Based Design and Safety* **149**

7.1 Contributions . . . 150
7.2 BlokIDE – Design and Implementation . . . 151
7.2.1 IEC 61499 as a Domain Specific Language . . . 152
7.2.2 Compilation . . . 160
7.2.3 Simulation . . . 161
7.2.4 Interfacing with Static Analysers . . . 167
7.3 BlokIDE and Functional Safety . . . 169
7.3.1 Reliability Modelling and Fault Tolerance . . . 171

Bestpfe.com

7.3.2 Software Realisation . . . 172

7.3.3 Recommendations in IEC 61508-3 Annex. B . . . 177

7.4 Conclusion . . . 181

8 Conclusion 183

8.1 Model-Based Safety . . . 183

8.1.1 Summary and Contributions . . . 183

8.1.2 Future Work . . . 185

8.2 Function Blocks to Prism Conversion . . . 185

8.2.1 Summary and Contributions . . . 185

8.2.2 Limitations and Future Work . . . 187

8.3 Meeting IEC 61508-3 Requirements with BlokIDE . . . 188

8.3.1 Summary and Contributions . . . 188

8.3.2 Future Work . . . 189

Appendices 191

A 193

A.1 BlokIDE Screenshots . . . 193

A.2 Prism Screenshots . . . 203

List of Figures

1.1 IEC 61508 and related functional safety standards 4

1.2 Overview of the proposed approach 7

1.3 Thesis organisation 9

2.1 Piping and instrumentation diagram (P&ID) of a boiler 12

2.3 Reliability block diagram of boiler control system 18

2.4 Fault tree analysis of boiler system (pressure over-run scenario) 20

2.5 Markov chain of boiler system 21

2.6 State-space analysis of presented discrete Markov chain 21

2.7 A typical development cycle involving model checking 22

3.1 Tolerable risk and ALARP (source: [1]) 33

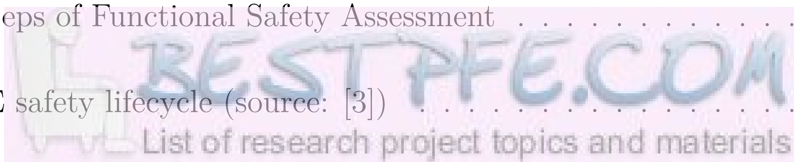
3.2 Primary cause of incidents by phase (source: [2]) 35

3.3 Overall safety lifecycle (source: [3]) 36

3.4 An example of a risk graph (source: [1]) 38

3.5 Seven steps of Functional Safety Assessment 40

3.6 E/E/PE safety lifecycle (source: [3]) 41



3.7 Software safety lifecycle (source: [3]) 45

3.8 V-Model for the design and development of E/E/PE Software (source: [4]) 46

4.1 System design of *Distribution Station* representing the various components 52

4.2 DistStnArm: an example of a basic function block 54

4.3 Execution control chart of the DistStnArm function block 55

4.4 DropItem algorithm from the *Arm* controller BFB 55

4.5 A composite function block with an encapsulated function block network . 57

4.6 Publish-subscribe communication function blocks. The publisher function block is configured to publish a single data element, SD_1. The subscriber function block is configured to subscribe a single data element, RD_1 . . . 58

4.7 Sequence diagram depicting the three scenarios for connection establishment between the publisher and the subscriber 60

4.8 Sequence diagram depicting normal unidirectional data transfer between the publisher and the subscriber 61

4.9 Sequence diagram depicting publisher-initiated and subscriber-initiated disconnections 61

4.10 IEC 61499 system containing two devices with nested resources 63

4.11 Application model of the *Distribution Station* system 63

4.12 A distributed IEC 61499 implementation of *Distribution Station* 65

5.1 IEC 61499 implementation of the boiler control system 72

5.2 Execution Control Chart of the Controller function block 76

5.3 A composite function block 79

5.4 An example of Markov decision processes [5] 82

5.5 The process of converting a FBN to a Prism model presented as a flow chart 88

5.6 Lookup source of wire-connections for Prism syntax generation 95

5.7 Illustration of Rule T5 for encoding deterministic transitions. The source state-transition structure is shown above and the generated Prism commands are shown below. 101

6.1 A hazardous gas detection and ventilation system 118

6.2 IEC 61499 implementation of the hazardous gas detection system using stochastic function blocks 119

6.3 Process diagram for the proposed approach 119

6.4 *GasSensor* stochastic function block 121

6.5 GasSensor SFB annotated with rules and lines numbers (see Figure 6.6) . . . 126

6.6 Prism modules generated for the GasSensor SFB (see Figure 6.5) 126

6.7 Illustration of Rule T7 showing probabilistic SFB transitions above and the equivalent Prism command below 130

6.8 Illustration of Rule T8 showing non-deterministic SFB transitions above and the equivalent Prism commands below 132

6.9 Modelling (a) *per-time* and (b) *on-demand* probabilistic failures 143

6.10 Modelling non-deterministic behaviours 144

7.1 UML component diagram of BlokIDE [6] 151

7.2 Meta-model of the DSL implementing IEC 61499 153

7.3 Metal-model for the BusinessModel class 154

7.4 Meta-model for FunctionBlockInterface and FunctionBlockReference 154

7.5 Various meta-model classes for Signals 155

7.6 Meta-model for ECCs of BFB and SFB 156

7.7 WireConnection and ProxyConnection relationships between FunctionBlockPorts 158

7.8 Meta-model of IEC 61499 implemented as a domain specific language . . . 159

7.9 A composite function block 160

7.10 Structure and outline of the generated code 161

7.11 Application model of the *Distribution Station* system 161

7.12 Sequence diagram for simulation 162

7.13 Toolbar commands to (a) Start, or (b) Continue, Stop and Restart simulation. Simulation can also be started from the Debug menu (c) 165

7.14 Simulation Data tool for taking user inputs 166

7.15 Tick Stack tool window shows tick history in reverse chronological order . . 166

7.16 Tick Snapshot tool window with hierarchical view of instances and signals . 167

7.17 Activity diagram for integration with static analysers 168

7.18 Model verification process showing *wait* and *success* dialogs 168

7.19 Timing analysis *success* dialog 169

7.20 Safety critical distribution station system 170

7.21 Validating SIL target using the proposed approach 171

7.22 V-Model for software realisation using BlokIDE and companion tools . . . 172

7.23 Illustration of simulation of a basic function block 174

7.24 Validating qaualitative safety property using the proposed approach 175

7.25 Onboard compilation and device deployment over SSH 176

A.1 Creating a new BlokIDE project 193

A.2 Creating a new Basic Function Block 194

A.3 Function Block Interface Editor 195

A.4 Creating a new Composite Function Block 195

A.5 Creating states and transitions 196

A.6 Edit transition priority 196

A.7 Adding state-entry actions to states 197

A.8 Function Block Network Editor 198

A.9 Refreshing the reference to load changes to referred function block interface 198

A.10 Model Explorer tool window showing a BFB (a) and CFB (b) 199

A.11 Simulation of an ECC of a BFB 200

A.12 Simulation of the encapsulated FBN of a given CFB 201

A.13 Project properties dialog for setting compiler path 202

A.14 IEC 61499 models on the right and the generated code on the left 202

A.15 Device Deployment over SSH 203

A.16 Prism model checker with a generated Prism model 204

A.17 Defining verification properties in Prism model checker 204

A.18 Start the verification process in Prism model checker 205

A.19 Result of probabilistic verification in Prism model checker 205

List of Tables

2.1 Classification of model-based safety assessment approaches 29

3.1 Likelihood of hazards 32

3.2 Severity of hazards 32

3.3 Risk Matrix 33

3.4 Safety Integrity Levels (SILs) (source: [3]) 35

3.5 Maximum achievable SIL for a given SFF and fault tolerance 44

3.6 Tables provided in Annex. A of IEC 61508-3 49

3.7 Tables provided in Annex. B of IEC 61508-3 49

5.1 BNF for BFB algorithms 75

5.2 An example of a Prism module 83

5.3 Illustration of composition of probabilistic commands of two Prism modules 85

5.4 Generated Prism module from the Controller function block shown in Figure 5.2. Note that "_controller" postfix is omitted from variables names for readability. 89

5.5 Lookup table for translating syntax from IEC 61499 to Prism 98

5.6 Name lookup table for Controller FB interface IO 103

5.7 Illustration of Algorithm 5.4 on the Prism module generated from the Controller BFB shown in Figure 5.2. Note that _controller postfix is omitted for readability. 104

6.1 Rule guide for mapping SFB structure to Prism using transformation rules 125

6.2 Summary of macros used in Rules T1–T6 127

6.3 Summary of the macros used in Algorithm 5.4 133

6.4 Illustration of Algorithm 6.2 on the Prism module generated from the sensor1 instance of GasSensor SFB shown in Figure 6.2 134

6.5 Probabilistic verification of example systems. 146

6.6 Analysis times for probabilistic verification 147

6.7 Analysis times for the scalability experiment. 147

7.1 Tick request Xml packet 163

7.2 Tick response Xml packet 164

7.3 Keyboard shortcuts for controlling the simulation 166

7.4 Recommendations on Design and Coding Standard 177

7.5 Recommendations for Modelling 179

7.6 Recommendations on Modular Approach 180

1

Introduction

Industrial automation and control systems are safety-critical due to hazardous plants such as oil and gas, chemical, material handling, smart manufacturing and nuclear power plants. Such systems perform control and automation tasks that may affect human lives in a safety assured manner. Failures in such systems can cause catastrophic events and may even result in loss of human lives. This seriousness of consequences mandates addressing safety in its own rights, rather than being a by-product of good practices. This dedicated mode of analysis, design, implementation, validation, deployment and maintenance of safety functions is referred to as *functional safety* [7]. Functional safety standards e.g., IEC 61508 [3] provide a range of requirements and recommendations for the various phases in the design and development of safety-critical systems. Conformance to such standards helps establishing the confidence in the system safety and demonstrates that the risk has been mitigated to an acceptable level. Achieving conformance with functional safety standards is not possible through ad-hoc practices. For example, IEC 61508 requires a lifecycle approach to be followed that governs all aspects of a safety-critical system at various phases in the planning, design, implementation, validation, deployment, maintenance, and decommissioning. In addition, it prescribes adoption of design methodologies that manage complexity of the hardware and the software in a scalable, verifiable, and unambiguous manner.

Bestofe.com

Industrial automation systems are complex real-time systems that require rigorous design and analysis techniques to manage their complexity and guarantee the safety assured operation. *Model-based design* and *synchronous languages* are design-methodologies that are proven in the industry to be capable of addressing these requirements [8, 9]. Amenability of these techniques towards static analysis techniques and *correct by construction* code generation makes them highly suitable for design and implementation of safety critical system software. In the recent years, several model-based approaches have been proposed [10] based on IEC 61499, which is a recent standard for design and development of industrial process control and measurement systems from the *International Electrotechnical Commission* [11]. However, due to semantic ambiguities in the execution guidelines of the standard [12], several interpretations of the execution model emerged [12–14]. Inspired by [8, 9], we have adopted the synchronous execution semantics of IEC 61499 function blocks [15] as the basis of this work.

In this thesis, we address functional safety of industrial automation systems using a technique called *model-based safety assessment* (MBSA). The proposed approach uses the system development models for the purpose of safety assessment and integrates seamlessly with an existing model-driven development approach based on IEC 61499. It addresses the three main aspects of model based safety: modelling safety-related aspects of the systems such as stochastic failures and their effects, unified analysis of system hardware (plant) and software (controller), and a model-driven development tool-chain that offers support for conformance to the IEC 61508 functional safety standard. In the following sections, we present a brief introduction to the various concepts that are required to understand the proposed approach that appears in this thesis.

1.1 Historical Perspective

Industrial control systems (ICS) have gone through decades of continued evolution. In the 60's ICS were implemented using ladders of electrical relays that were hard to configure and troubleshoot. After the initial proposal of *standard machine controller* presented by General Motors in 1968, early efforts of designing a *programmable controller* were soon kicked off, and by 1971 Allen-Bradley had created a device called *programmable logic controller* (PLC) [16]. This device allowed flexibility and was easy to configure, deploy and troubleshoot. Adoption of PLCs was initially difficult, however within just a decade of its invention PLCs started getting attention from the industry and eventually became the primary means for implementing ICS.

Before the 80's, system failures could usually be traced back to failures of physical components such as relays, sensor and actuators. However, increasing capabilities of hardware and introduction of software in control led to significant increase in complexity. Reasons for failures in such systems were often not apparent. This gave rise to a two-pronged approach i.e., to perform quantitative analysis to address random failures in physical components and to adopt qualitative techniques to avoid systematic errors in hardware and software design and implementation [17]. Subsequently, safety standards were established to provide guidelines for the validation of robustness and reliability of safety-critical systems, with an aim to validate that these systems mitigate the risk to human lives, as much as *reasonably* possible. Conforming to recommended safety standards is not only a legal requirement but is also a prudent financial decision as it can avoid expensive recalls and reduce development and maintenance costs through early detection of errors.

> *"The prevention of accidents must be seen not as a regulation prescribed by law but as a dictate of human obligation and sound economic sense"*
> — Werner von Siemens, 1880

1.1.1 IEC 61508 and Related Standards

Safety-critical systems exist in many fields such as automotive, avionics, nuclear, medical equipment, electric drives. Depending on the application, a safety-related system must comply to one or more safety standards like IEC 61513 (nuclear power plants) [18], IEC 62061 (machinery) [19], IEC 61511 (process industry) [20]. Although, the work in this dissertation is applicable to many functional safety standards, we use IEC 61508 as our principal guide, which is a functional safety standard for *generic electric, electronic, and programmable electronic* (E/E/PE) systems [3]. Several industry-specific functional safety standards refer to it as a basic guide as represented in Figure 1.1.

IEC 61508 defines four distinct safety integrity levels (SIL) of the safety-related system i.e., SIL1 to SIL4 with SIL4 being the most stringent. Each of the SILs recommends a set of qualitative analysis techniques for the detection and elimination of systematic errors such as software bugs and coding errors. On the other hand, physical components are prone to random errors that may be caused by manufacturing defects as well as environmental and operational strains. Eliminating such errors is not possible and therefore, the system must implement fault tolerance through redundancy. Validation against such failures entails estimation of system reliability through quantitative analysis such

BESTPFE.COM
List of research project topics and materials

as *Markov analysis* and ensuring that it satisfied an acceptable range of probability of failures. Readers are advised to refer to Chapter 3 for a brief introduction to IEC 61508 and its various hardware and software safety requirements.

1.2 Model-Driven Engineering

Model-driven engineering (MDE) is the *state-of-the-art* for the design, implementation and validation of control and automation systems. MDE is an engineering methodology where system models are developed and used for various engineering tasks such as model-driven architecture of the system, model-driven development (MDD) of the control software, model-based testing, and other related activities. MDE is well-suited and proven in the industry to address the complexity of safety-critical systems [21]. Tools like SCADE and Simulink have been successfully applied in design and development of safety-related software and provide tool-specific guidelines for achieving conformance to functional safety standards [22–24].

1.2.1 Model-Driven Development

MDD provides a basis for seamless integration of various model-based techniques. System models constructed for MDD activities, also known as system development models, can be used as input for the various model-based activities. A typical MDD workflow involves creating a high-level model of the system and performing subsequent iterative refinements in a top-down manner. At each stage of the design process, the system can be tested against various validation requirements, which enables early detection and prevention of flaws. Upon completion, the system model is used for *automatic code generation* that

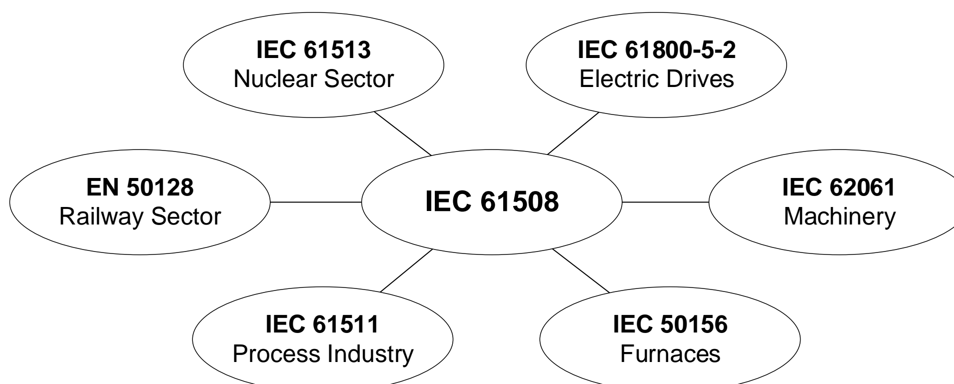


Figure 1.1: IEC 61508 and related functional safety standards

ensures a *correct-by-construction* implementation of the system software [25]. Due to these advantages, MDD has gained significant approval in academia and industry. One such MDD approach [15] already exists for industrial automation domain, which uses IEC 61499 function blocks for developing system models.

1.2.2 Model-Driven Development using IEC 61499

IEC 61499 is an open standard for design and implementation of *industrial process control and measurement systems* [11]. It provides concepts of *System*, *Application*, *Device* and *Resource* to model the system architecture. The implementation of control behaviours is performed using interconnected *networks* of *function blocks* that encapsulate Moore-like state machines called *execution control charts* (ECC). ECCs accept inputs from an encapsulating *function block interface* and are provisioned to execute state-entry actions such as the invocation of textual algorithms and emission of output events. Through this mechanism, function block networks read inputs from external environments (e.g., sensors, human-machine interface, timers) and produce desired behaviours through control outputs for actuation. Please refer to Chapter 4 for a summary of the IEC 61499 standard and related topics.

A popular design pattern for MDD using IEC 61499 named *model-view-controller* (MVC), involves creating two separate function block networks i.e., *plant-model* and *controller* [26, 27]. Plant-model mimics the expected behaviour of a plant as read from sensors and other inputs, and is connected in a closed-loop with the controller. The controller implements the automation and control logic using state machines and algorithms and controls the plant-model as if it was a physical plant. At a later stage, the controller is used to for automatic code generation and is deployed to the physical plant. Under this setting, a range of validation and verification (V&V) techniques can be used to confirm correctness and safety of the overall system such as closed-loop formal verification and other static analyses [28–31]. These analyses are useful for safety-related applications, but these approaches are qualitative in nature. Performing only qualitative analyses of safety-critical systems is dangerous since, such systems are also susceptible to random failures due to component failure rates [32]. Unfortunately, all existing IEC 61499 safety analysis techniques [28, 29, 33–42] are qualitative in nature.

In the absence of quantitative analysis approaches, safety assessment of such systems is performed manually by using reliability estimation techniques such as *reliability block diagrams* and *fault tree analysis* [43]. However, the manual application of these

approaches is largely dependent on practitioners' skills and experience [44]. Model-based safety analysis (MBSA) ameliorates this problem and improves the quality of safety analysis through the use of system models [45].

1.2.3 Model-Based Safety Assessment

Increasing capabilities of hardware resulted in a configuration-oriented market, where a product was no longer a monolithic object that could be just certified once. Also, the increasing role of software in the control systems meant that traditional method of safety analysis like *fault tree analysis* and *reliability block diagrams* were no longer feasible. Even with their current application, these traditional techniques neglect software altogether [45]. With the ever increasing complexities of these systems, application of these techniques become more and more dependent on the practitioners skills and experience. MBSA started becoming more popular due to these reasons and has gained significant interest in the safety community.

MBSA uses model for performing overall system safety assessment. These models may be constructed for the express purpose of MBSA, or automatically derived from extensions of the models used for system development [46]. The latter approach offers seamless integration of model-based development with the safety analysis practices, where the model of the nominal behaviour of the system is merged with the corresponding failure models and related quantitative values e.g., reliability and failure rates [44]. Thus, the resultant model contains the failure occurrence and propagation information that is, how the failure is generated and propagated through dependent components, as well as the failure effect information i.e., the failure-affected behaviour of the components [47].

A range of MBSA approaches has emerged in the recent years for domains other than industrial automation. However, the general idea is to use the system model to extract fault trees (e.g., [48]), critical event sequences (e.g., minimal cut-sets [49]), to perform quantitative analysis to compute probabilities of failures (e.g., [50]) and other similar safety-related analyses. A common pattern among several of these approaches is to perform an automatic transformation of system models into a secondary format for the purpose of quantitative analysis. Esteve et al. [51] presented a practical application of MBSA that converts AADL [52] models into Markov chains in order to perform dependability analysis of *failure detection, isolation and recovery system* of a satellite. A similar application is presented in [53] where AltaRica modelling language [54] is used to describe system models, which is later converted to Markov chains for quantitative

reliability assessment. The approach presented by Güudemann et al. [55] also bears this similarity where system models are constructed using a tool independent specification language named SAML , which can later be translated into Markov chains for quantitative analysis by various probabilistic verifiers. Even though the above-mentioned approaches are grounded in practical applications and are supported by compatible tool-chains, however, these system models are dedicated for safety analysis and do not integrate well with MDD. On the other hand, approaches like [56, 57] utilise system development models i.e., Simulink/ Stateflow models for the purpose of safety analysis. The benefit of such approaches is two-fold. Firstly, such an approach would seamlessly integrate with existing model-based development practices, and secondly the model analysed will be same as the one used for eventual deployment. However, none of these approaches can be used for IEC 61499 function blocks.

1.3 Contributions

In this thesis, we propose an MBSA approach for industrial automation systems based on IEC 61499, where an extended plant-model is designed using IEC 61499 such that. This extended plant-model is constructed using a novel structure based named *Stochastic Function Block* (SFB). SFBs are based on the semantics of *Markov Decision Processes* (MDP) [58] and can be used to model stochastic behaviours of the external plant e.g., probabilistic failures and their respective effects on the nominal plant behaviour. On the other hand, the controller is implemented using IEC 61499 standard compliant function blocks, which are later used for automatic code generation and deployment. This way, the overall system remains IEC 61499 standard compliant for the original purpose of model-driven development, as shown in Fig. 1.2. The added expressiveness of the plant-model is

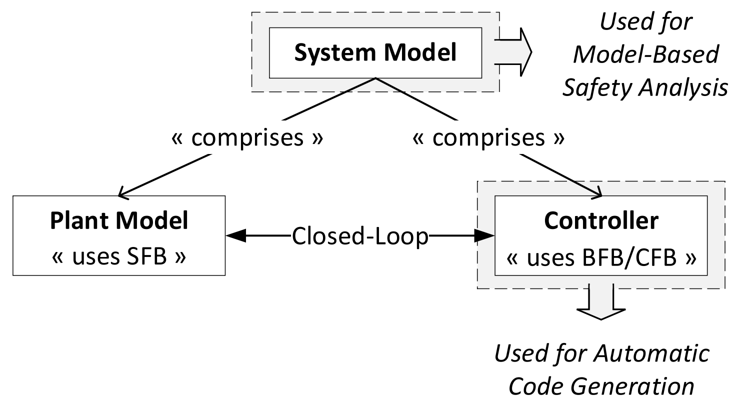


Figure 1.2: Overview of the proposed approach

leveraged to automatically derive Markov models using a rule-based transformation. This transformation preserves the synchronous execution semantics of the controller as well as the MDP semantics of the plant-model. The generated Markov models can be analysed by employing a probabilistic model checker e.g., the Prism model checker [59]. This not only helps practitioners with early detection of systematic software errors but can also give quantitative estimates of the overall system failure. The key contributions of this thesis are as following.

1) A novel structure based on IEC 61499 basic function blocks named *stochastic function block*, which can be used for modelling probabilistic and non-deterministic behaviours. This structure is used for representing the random errors in the plant-model under the MVC design pattern. The resultant model is a closed-loop system that contains approximated behaviours of the plant-model along with the identified random errors, and the controller (software) with its systematic errors (if any).

2) A rule-based transformation for semantic preserving translation of IEC 61499 function blocks to Prism language, which is a well-known language for representing stochastic models. This automatic translation helps developers construct Markov models without any error-prone manual effort. The generated Prism model is a Markov decision process that represents the probabilistic and non-deterministic aspects of the system due to its random errors.

3) A technique for a unified qualitative and quantitative analysis using *probabilistic verification*. The result of this analysis is particularly useful in the early design stages for validation as well as to manage modifications in the physical and logical design.

4) An MDE tool-chain named BlokIDE, for the purpose of modelling, implementation, and validation of IEC 61499 systems. BlokIDE provides tooling support for the proposed structures, which enables stochastic error modelling. The algorithm for automatic translation to Prism language is implemented as *export* functionality, which enables integration with the Prism model checker for the purpose of qualitative and quantitative analyses.

5) A proposed guide for conforming to IEC 61508 requirements using IEC 61499 model-based approach. This proposal is inspired by the IEC 61508 guide provided by SCADE [24]. It shows how various IEC 61499 artefacts can be used for the purpose of specification and design on various stages of a proposed V-Model. Furthermore, it discusses the various qualitative requirements of IEC 61508 software safety lifecycle and the compatibility of the companion tool-chain.

1.4 Thesis Organisation

The remaining of this thesis comprises six chapters, which are organised as following. We begin with presenting some necessary background concepts about safety and reliability modelling in Chapter 2.

These concepts are immediately useful to the reader to understand the related existing approaches for MBSA. Chapter 3 continues on the theme of safety and presented a functional safety standard namely IEC 61508. In this chapter, we described the various functional safety concepts and safety requirements on hardware and software of E/E/PE systems.

Chapter 4 presents a basic introduction of IEC 61499, which entails the discussion about its structure and semantics. We presented various types of function blocks and various execution semantics used to interpret their behaviours. This introduction is necessary for understanding the proposed model transformation-based approach, which is presented in Chapter 5. Here, we present a set of semantics-preserving rules to convert IEC 61499 functions blocks to Prism language. Stochastic function blocks are described in Chapter 6, which further extended the transformation rule to generate stochastic Prism models as Markov decision processes.

The concepts of IEC 61499 and IEC 61508 are combined together in an effort to implement the proposed MBSA approach in a tool-chain named BlokIDE as presented in Chapter 7. In this chapter, we also present a proposal for conformance to the requirements imposed by IEC 61508, using the system design artefacts of IEC 61499. This proposal comprises a customised V-Model and its compatibility with the safety recommendations of IEC 61508.

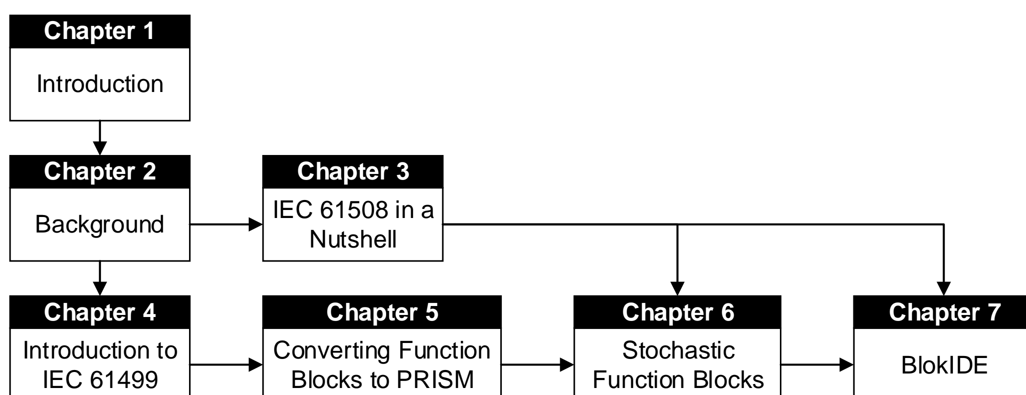


Figure 1.3: Thesis organisation

2

Background

This chapter provides some basic concepts related to system reliability, reliability modelling and safety analysis. We use a motivating example of a safety-related boiler control system to illustrate these concepts. We further describe a few existing approaches for safety analysis of system models. This is followed by a critical comparison of these approaches with the model-based safety assessment approach proposed in Chapter 6.

2.1 Boiler Control System

We present a boiler controller implemented as an embedded control system with hardware components and software control. This system can be understood from its *piping and instrumentation diagram* (P&ID) presented in Figure 2.1. It consists of a cylinder mounted with a pressure sensor. The contents of the cylinder are heated using a heat exchanger (e.g., a gas burner). The pneumatic control of the heat-exchanger allows altering the amount of heat being transferred to the cylinder. A pressure relief control valve allows reducing the pressure of the boiler in case the pressure rises above the desired value. Flow indicators (FI) are mounted in 1-out-of-2 (1*oo*2) redundant fashion to confirm the pressure

Bestpfe.com

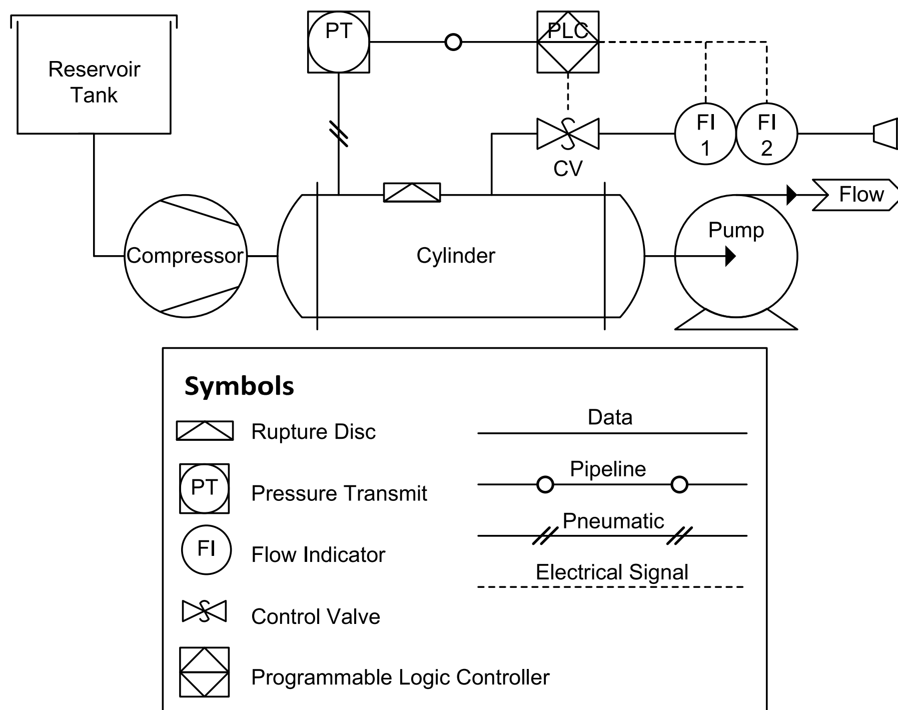


Figure 2.1: Piping and instrumentation diagram (P&ID) of a boiler

relief. The desired behaviour for this system is to boil the contents of the cylinder.

Risk analysis of the presented example identifies several possible hazards, for example if the pressure increases beyond a threshold, the contents of the cylinder may be rendered useless or may even result in an explosion. The sensor installed on the boiler provides the pressure information, which is used by the controller to avoid the said *pressure overrun hazard*. Using sophisticated verification and validation techniques, it is possible to create a safety-critical control software that assures safety in such scenarios, that is the system contains no *systematic errors*. However, the behaviour of the software control depends on the correct function of hardware components, which are prone to *random errors*. In the case of a malfunction of sensors, the controller alone will not be able to prevent accidents.

For the purpose of simplicity, we restrict our further discussion around the specific scenario where pressure increases beyond the desired value and the pressure relief valve is to be opened by the controller. This scenario imposes a safety requirement and hence, a corresponding safety function is to be implemented. From the perspective of functional safety, two main aspects have to be considered in this specific scenario namely, the reliability of physical components (pressure transmitter (PT), programmable logic controller (PLC), flow-indicators (FI)) and correctness of the safety function implemented as a software component. Here, achieving safety involves addressing the reliability of the physical

components of the system and correctness of the corresponding software. In the following sections, we use this system as an illustrative example and present some basic required concepts.

2.2 Defects, Errors and Failures

A *defect* is defined as a deviation from a given specification that may be caused due to incorrect requirement analysis or flaws in construction. These flaws can be conceptual as well as material, for example, insufficient details, incorrect understanding of the specification, quality of raw material used, or lack of precision of control in manufacturing processes. But not all defects are immediately encountered. Some defects may go unnoticed for long periods of time. When a defect is encountered, the system may behave incorrectly, which is referred to as an *error*. In an error state, the system behaviour may deviate from expectations. Some errors are detectable and can be corrected through error detection, prevention and correction measures. Such errors do not become obvious to the external environment, for example, as effects on equipment under control (EUC), not accepting commands from human operators, incoherency in the controlled process or incorrect measurements. The errors that are not detected and corrected induce a system *failure*, which is observable and needs to be addressed using failure recovery mechanisms. Failures can lead the system into undesired states and may even cause unpredictable behaviours. Failures in safety-critical systems can be dangerous and are, therefore, analysed for their robustness through avoidance, detection, and tolerance e.g., through validation, testing, and redundancy. Due to the close relationship between errors and failure, sometimes these terms are used interchangeably with the understanding that a failure is caused due to an error.

Depending on the type of defects, some errors (and corresponding failures) may occur in a repeatable fashion under a particular set of conditions. Such errors are referred to *systematic errors* and can be traced back to the source defect. Examples of such errors are incorrect specifications, lack of clear understanding, human errors in design, implementation and deployment etc. These errors are usually dealt with avoidance measures such as validation and quality assurance techniques. Through rigorous processes and validation, it possible to significantly reduce the number of systematic errors. However, some failures occur randomly without a predictable set of steps to reproduce the failure. Such failures are referred to *random failures*. These failures are still predictable through statistical methods, for example, by operating a large population of components the rate of failure can be predicted if the individuals in the population are true representatives of

BESTPFE.COM List of research project topics and materials

their class. This is also referred to as *failure rates*.

2.3 System Reliability

Reliability of a system may be understood as a time-based perspective of quality i.e., the ability of a system or product to perform its mandated function over a duration of time. This time period can either be set by the manufacturer (i.e., a warranty period) or specified in the system requirements (i.e., service period). Inside this mandated period, the system may fail to perform the expected function with some uncertainty. However, this uncertainty needs to be quantified as a probability of correct or incorrect behaviour. This preamble gives way to the formally accepted definition of reliability, presented as follows.

Definition 2.3.1 (Reliability)**.** "The probability that an item will perform a required function without failure under stated conditions for a stated period of time." [60]

Reliability can be quantified using some statistical attributes measured over a population of items (e.g., sensors). Some of the commonly used measures are:

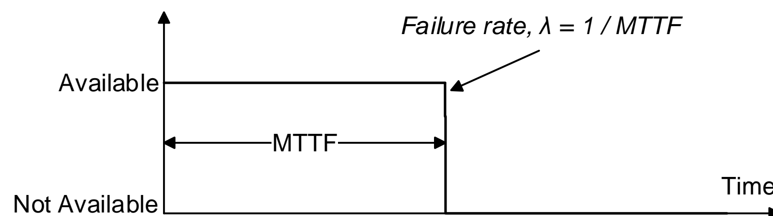
- *Failure Rate* (λ) — mean number of failures in a given time.
- $MTBF$ — mean time between failures (applicable to repairable items).
- $MTTR$ — mean time to repair (applicable to repairable items).
- $MTTF$ — mean time to failure (applicable to non-repairable items).

There is a fundamental difference on how repairable and non-repairable items are treated mathematically for reliability. The classic definition of reliability is applicable to non-repairable items where reliability is a measure of the estimation of time to first failure. Examples of such systems include electronic components, for example, miniature-capacitors, resistors, microprocessors. Statistical measures like MTTF can be used as a characteristic of reliability for such items or systems. Similarly, for repairable systems measures like MTBF can be used as a characteristic of reliability. However, this is valid only under constant failure rates, where $\lambda = 1/MTBF$. Depending on several factors, a controlled manufacturing process may offer a constant failure rate for specific items, which renders MTBF as a useful measure of reliability [60]. For a constant mean repair time, we can derive another statistical attribute referred to as *repair rate*(μ) such that,

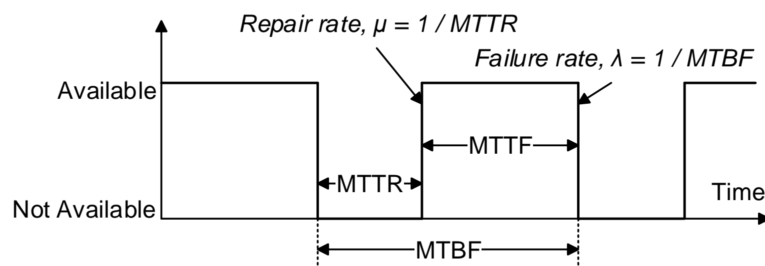
$\mu = 1/MTTR$ [61]. Figure 2.2 visually illustrates the concept of MTBF, MTTR, MTTF, failure rate, and repair rate.

It is desirable to estimate the *availability* of a repairable system, which depends on the rate of failure and time taken by the repair process. The statistical measures of MTBF and MTTR can be used derive the availability of repairable systems as presented in Equation 2.3.1.

$$availability = \frac{MTBF}{MTBF + MTTR} = \frac{\mu}{\lambda + \mu} \tag{2.3.1}$$



(a) Non-repairable system



(b) Repairable system

Figure 2.2: Availability of non-repairable and repairable systems (adapted from [61])

2.4 Reliability Modelling and Analysis

Reliability of a system depends directly on the reliability of individual components and their configuration. Redundancy is a common way to increase the reliability of a system where a component with low reliability is installed in parallel with a redundant component.

It becomes gradually difficult to estimate the reliability of systems where a large number of components and redundancies are employed. This issue is further compounded by the fact that there are several types of redundancies:

1. *Active Redundancy* is the simplest form of redundancy where two identical components are active at the same time. Successful operation is guaranteed if *either or both* components work correctly.
2. **m**-*out-of*-**n** is a form of active redundancy with a total n number of identical components. Successful operation is guaranteed if m number of components work correctly, where $m < n$.
3. *Standby Redundancy* is when a redundant component takes over the responsibility when the *primary* component fails. Successful operation is guaranteed if both components do not fail one after the other.

Redundancy, in general, aims to increase the reliability of the overall system. However, it is necessary to evaluate redundant components for any *common cause failures* (CCF), which can cause all redundant components to fail at the same time. A system may also be composed of various levels of redundancies, where an entire set of components is installed in parallel to a redundant set. These complexities raise the need for using a formal technique for calculating the estimates on system reliability. We shall discuss formal techniques for reliability modelling of systems that utilise the known probabilities of failure of components for calculating the overall system reliability.

2.4.1 Probability of Failure

For mission critical systems, the probability of failure during operation is estimated and validated against given requirements. For known values of failure rates, the probability of failure can be calculated from the *Poisson distribution* [43]. Poisson distribution is a *discrete probability distribution* that is used to calculate the probability of a random event occurring during a specified amount of time, where the average rate of occurrence of the event is known. Relating this concept to the *failure rates* (λ) of components, we can calculate the probability of failure using the Poisson distribution with mean(μ) and variance(σ^2) set to the failure rate ($\mu = \lambda$ and $\sigma^2 = \lambda$).

The probability of failure depends on a few factors and is calculated differently for repairable and non-repairable systems. In repairable systems, repair is often performed by

replacing a failed component with a spare without any delay. The reliability of the overall system, in this case, depends on the number of available spares and the respective failure rate. With a given N_{sp} number of spares, the reliability of such systems is the measure of the probability of $N_{sp}+1$ failures within a given time interval $(0,T)$. Reliability can, therefore, be calculated as the sum of probabilities all mutually exclusive events i.e., no failures in $(0,T)$, one failure in $(0,T)$, ... , N_{sp} failures in $(0,T)$. A higher value of this measure thus signifies a lower reliability of the system.

$$R(T) = \sum_{k=0}^{N_{sp}} \frac{(\lambda T)^k e^{-\lambda T}}{k!} \tag{2.4.1}$$

For non-repairable systems with no replacements, the reliability is simply the probability of the first failure occurring within mandated time. The reliability of such systems can be derived from the above equation using $N_{sp}=0$, which is essentially the probability of operation without failure within mandated time interval $(0,T)$. In this setting the value of the reliability measure falls in the range $[0,1]$ i.e., $0 \leq R(T) \leq 1$, and higher value signifies lower reliability.

$$R(T) = e^{-\lambda T} \tag{2.4.2}$$

Discussion on the probability of failure of generic repairable equipment, such as a mechanical lever or rewindable electric coil that requires manual labour is beyond the scope of this document.

2.4.2 Failure Mode and Effect Analysis

Failure mode and effect analysis (FMEA) [62] is a systematic methodology for reliability study of all components of a system. In this study various operation modes of the system and individual components are analysed for the possibility of failures measured both in qualitative and quantitative values. The immediate effects of these failures and their long run consequences are also analysed. Furthermore, it is also studied if a failure in one component can affect other components or the overall system behaviour. This data collected in the studies is populated in a worksheet, which allows easy look up for analysis

safety. For example, these worksheets are used to find relationships between component failures and their reliability data for deriving reliability block diagrams and fault trees.

2.4.3 Reliability Block Diagram

Reliability block diagram (RBD) is a well-known method for modelling system reliability. An RBD presents a network of blocks in series or parallel configuration. A block in an RBD may represent a simple component (e.g., resistor) or a complex component that comprises of multiple sub-components (e.g., relay). The connections between blocks represent the relationship of individual components' reliabilities and may not necessarily represent the physical or functional layout of the system. These relationships can be derived from systematic reliability study of a system (e.g., by using FMEA). Various types of redundancies in the system can be represented in an RBD to calculate the overall system reliability. A component (or an assembly of more than one components) is presented in parallel with its redundant counterparts. A combining operator is used to merge the connections and represent the redundancy type with respective symbols: '+' for active redundancy, 'm/n' for **m**-*oo*-**n** redundancy and a switch symbol for standby redundancy. All singular components are presented in series fashion. In this manner, various types and levels of redundancies can be expressed. RBD of the example scenario from the boiler system is presented in Figure 2.3, where flow indicators are installed in active redundancy and all other components are singular.

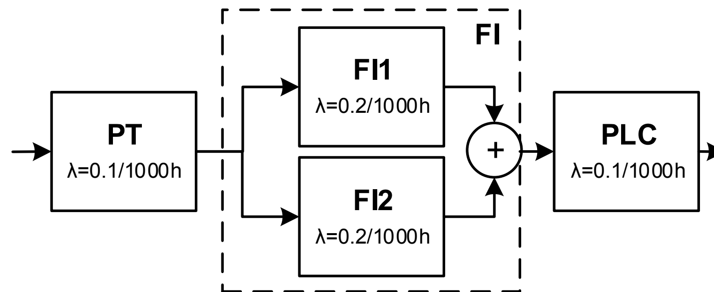


Figure 2.3: Reliability block diagram of boiler control system

$$R_S = R_{PT} \times R_{FI} \times R_{PLC} \tag{2.4.3}$$

$$\textit{Where, } R_{FI} = 1 - [(1 - R_{FI1}) \times (1 - R_{FI2})] \tag{2.4.4}$$

Reliability for operation of $T = 1000h$ can be calculated as following,

$$R_{PT} = e^{-\lambda_{PT}T} = 0.9048$$

$$R_{FI} = 1 - [(1 - e^{-\lambda_{FI1}T}) \times (1 - e^{-\lambda_{FI2}T})] = 0.9671 \quad \textit{(from eq. 2.4.4)}$$

$$R_{PLC} = e^{-\lambda_{PLC}T} = 0.9048$$

$$R_S = 0.7917 \quad \textit{(from eq. 2.4.3)}$$

2.4.4 Fault Tree Analysis

Fault tree analysis (FTA) is a reliability modelling technique that focuses on effects of system failure, referred to as *Top-Level* events. Unlike RBD, FTA is event-centric and has to be performed on a per-event basis. The analysis in FTA is started from the top-level event and logically broken down into possible reasons of the failure. A standardised visual notation is used to express various relationships among events in the system, with each event assigned a unique identifier. Boolean logical operators are used to express the connections of hazards on individual components with each other. A logical `AND` gate suggests that all hazard must occur to cause consequences to the next level of the hierarchy. Similarly, a logical `OR` gate suggests that even a single hazard on the lower level will affect the next level of the consequential hierarchy. This cause and effect information can be derived from systematic reliability studies of the system (e.g., using FMEA).

Even though FTA is most commonly used as a qualitative technique, quantitative analysis of the system is also possible i.e., by using probabilities of system failures. An example from the boiler system is presented in Figure 2.4 such that, the probability of pressure failure (E1) in the boiler system is derived from the FTA as following.

$$\mathtt{P(E1)} = \mathtt{P(F1 \cup F2 \cup E2)}$$

Using the addition rule for the union of two sets:

$$\begin{aligned}\mathtt{P(E1)} &= \mathtt{P(F1)} + \mathtt{P(F2)} + \mathtt{P(E2)}\\ &\quad - (\mathtt{P(F1)} \cap \mathtt{P(F2)}) - (\mathtt{P(F1)} \cap \mathtt{P(E2)})\\ &\quad - (\mathtt{P(F2)} \cap \mathtt{P(E2)}) - (\mathtt{P(F1)} \cap \mathtt{P(F2)} \cap \mathtt{P(E2)})\end{aligned}$$

Using the multiplcation rule for the intersection of two sets:

$$\begin{aligned}\mathtt{P(E1)} &= \mathtt{P(F1)} + \mathtt{P(F2)} + \mathtt{P(E2)}\\ &\quad - (\mathtt{P(F1)} \times \mathtt{P(F2)}) - (\mathtt{P(F1)} \times \mathtt{P(E2)})\\ &\quad - (\mathtt{P(F2)} \times \mathtt{P(E2)}) - (\mathtt{P(F1)} \times \mathtt{P(F2)} \times \mathtt{P(E2)}) \qquad (2.4.5)\end{aligned}$$

Similarly,

$$\begin{aligned}\mathtt{P(E2)} &= \mathtt{P(F3 \cap F4)}\\ \mathtt{P(E2)} &= \mathtt{P(F3)} \times \mathtt{P(F4)} \qquad (2.4.6)\end{aligned}$$

Probability of failure for operation of $T = 1000h$ is calculated as following,

$$\begin{aligned}\mathtt{P(F1)} &= 1 - e^{-\lambda_{PT}T} = 0.0952\\ \mathtt{P(F2)} &= 1 - e^{-\lambda_{PLC}T} = 0.0952\\ \mathtt{P(F3)} &= 1 - e^{-\lambda_{FI1}T} = 0.1812\\ \mathtt{P(F4)} &= 1 - e^{-\lambda_{FI2}T} = 0.1812\\ \mathtt{P(E1)} &= 0.1124 \quad \textit{(from eq. 2.4.5 and 2.4.6)}\end{aligned}$$

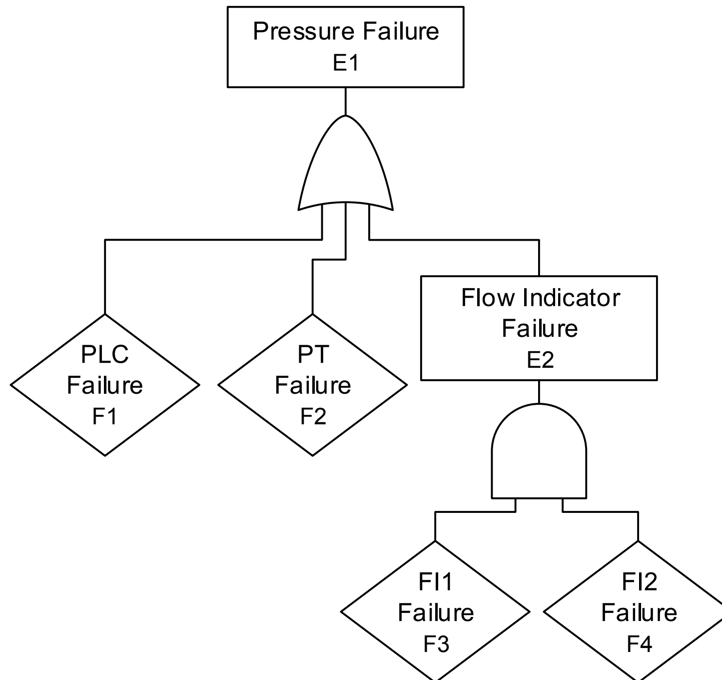


Figure 2.4: Fault tree analysis of boiler system (pressure over-run scenario)

2.4.5 Markov Analysis

Markov analysis is suitable for systems with components that can be in discrete states such as *failed* and *non-failed*. The probability of being in one state or the other is associated with the respective state and probabilistic transitions in a corresponding *Markov chain* [63]. State-space analysis can thus be used to find the probability of being in a particular state after a time interval [61]. The 1*oo*2 redundancy of flow indicators in the boiler system can be modelled as a Markov chain as shown in Figure 2.5. State-space

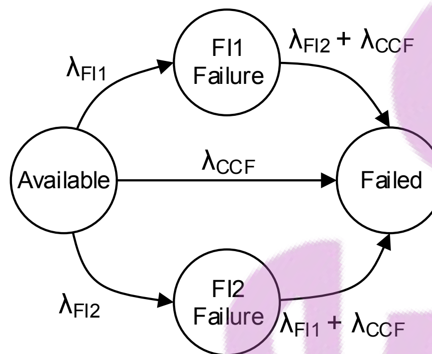


Figure 2.5: Markov chain of boiler system

analysis of the Markov chain for two intervals of $1000h$ can be seen in Figure 2.6, where each horizontal line represents an advancement of time unit and possible transitions with respective probabilities. Thus, the probability of being in a failed state after two intervals with *common-cause-failure* can be calculated using this state-space analysis, presented as follows.

$$\mathtt{P}_2(\mathtt{F}) = \lambda_{CCF} + \lambda_{FI1}[\lambda_{FI2} + \lambda_{CCF}] + \lambda_{FI2}[\lambda_{FI1} + \lambda_{CCF}] \quad (2.4.7)$$
$$\mathtt{P}_2(\mathtt{F}) = 0.0814, \text{ for } \lambda_{CCF} = 0.001$$

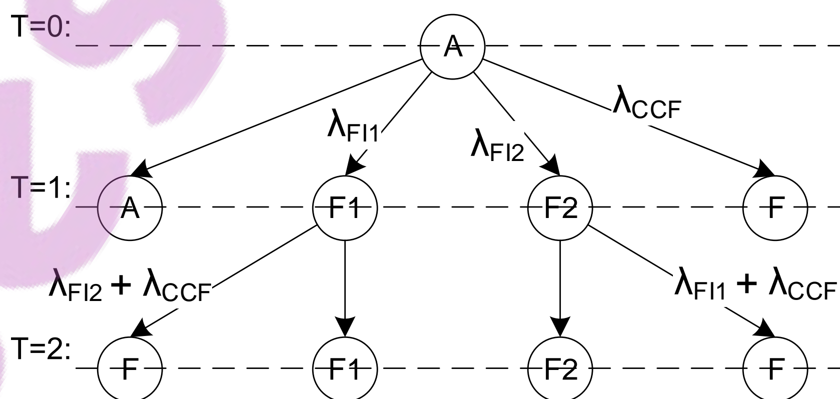


Figure 2.6: State-space analysis of presented discrete Markov chain

2.5 Formal Verification

Formal verification is a mathematical technique to prove or disprove a given specification property against a model of the system. Being exhaustive in nature, formal verification locates errors that may have been undiscovered by validation techniques (e.g., testing or simulation). While a range of techniques exists for formal verification of software, we shall discuss model checking [64], which allows automatic verification of system models against temporal specifications.

2.5.1 Model Checking

Model checking is an automated technique for verifying system models. The verification process explores all possible execution paths to discover execution traces that do not satisfy the given specification. Such an execution trace is referred to as *counter-example* and can be extracted to significantly reduce the debugging time. Due to its ease of application, model checking is often suggested to become part of the development lifecycle, especially for the development of mission critical systems. A typical system development cycle involving model checking is presented in Figure 2.7.

Model checking relies on an automatic derivation of system models from development artefacts. This suits well to model-based design paradigm where system development

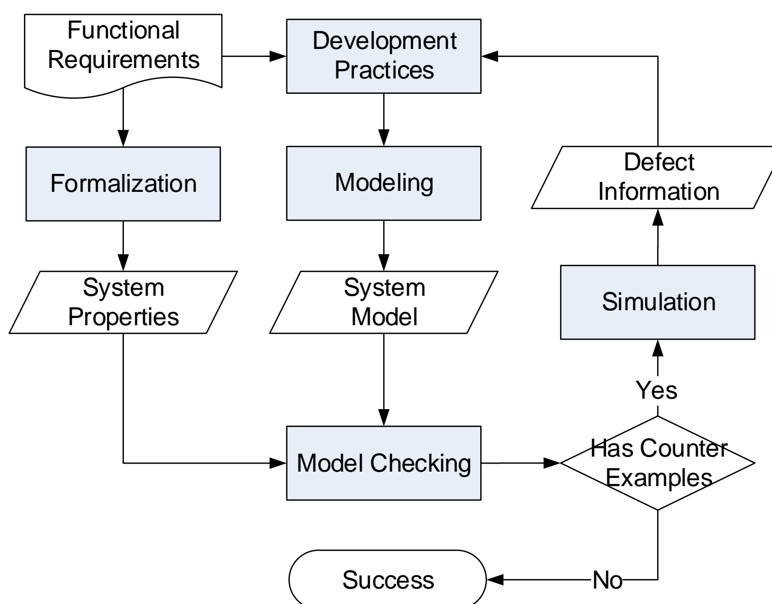


Figure 2.7: A typical development cycle involving model checking

model itself may be used for the verification process, thus providing seamless integration. Since model checking is considered only as good as the model itself, this also enhances the confidence in verification results against the actual deployed system. Another crucial aspect to model checking is the specification being verified, which is most commonly provided using temporal logic. For a given model $\mathcal{M}$ and a system property φ, the notation $\mathcal{M} \vDash \varphi$ is used to denote that the model satisfies the property. Conversely, we use the notation $\mathcal{M} \nvDash \varphi$ to express that the model does not hold the given property.

2.5.2 Temporal Logic

Starting from the initial state as the root node a system under execution may take several possible execution paths, which forms an execution tree. Temporal logic is a genre of notations that specify branching and linear behaviours of such system execution trees i.e., we can specify desirable and undesirable paths. Linear-time temporal logic (LTL) and computation tree logic (CTL) [65] are the well-known temporal logic notations for linear and branching time reasoning respectively. The expressiveness of the notations does overlap but does not completely coincide. For example in LTL it is not always possible to express requirement of the existence of a specific path that satisfies a specific sub-property. Similarly in CTL it is not always possible to express universal behaviours over a range of paths. This problem is resolved by combining the expressiveness of the two notations into a superset notation called CTL*. Using this notation, both linear and branching behaviours can be specified. An example CTL* property from the boiler control system is presented in Equation 2.5.1 namely, *"On all paths, pressure relief will not be performed until the pressure exceeds the threshold value, and a path will exist where pressure relief can be enabled"*.

$$A \; [\; (\; \overline{\texttt{relief}} \;\; U \; (\texttt{pressure} > \texttt{threshold})) \;\; \& \;\; E \; (\; \texttt{relief} \;) \;] \tag{2.5.1}$$

2.5.3 Safety and Liveness Properties

System properties can be divided into two general categories i.e., *safety* and *liveness* [66]. Safety properties model the global absence of undesirable occurrence, for example, *"something bad"* must never occur in a system during the course of its execution. Whereas the liveness properties model the global presence of desirable occurrences, for example,

List of research project topics and materials

"something good" will happen eventually. A safety property from the boiler control system is, *"when the pressure exceeds a threshold, the pressure-relief control valve will always be opened"*. Similarly liveness property from the boiler system is, *"pressure relief valve will be closed most of the time i.e., to perform useful boiler operation"*.

2.5.4 Probabilistic Verification

Formal verification through model checking verifies system for systematic errors that can always be avoided through design improvements and error correction. However, random failures often cannot be completely eliminated, for example, due to manufacturing defects in hardware components. Therefore applying qualitative model checking will always yield a negative result indicating the reachability of undesired execution paths. Probabilistic verification analyses *less-than-perfect* models, where system properties are quantitative in nature. For example in the boiler system, model checking can be used to verify the correct operation of the controller software, but the overall boiler system model also contains random failures due to hardware component failure rates. Therefore, such systems must be verified using probabilistic model checking that is, to verify that the system will work correctly with a high reliability. As presented earlier, a failure rate can be converted to an equivalent probability of failure by using a probability distribution, such as Poisson distribution (see Section 2.4.1). These probabilities are then used as annotations for the various behaviours on the system model. A probabilistic specification is then used to calculate the probability of the overall system model satisfying desirable outcomes. A probabilistic specification from the boiler system is presented as following i.e., *"what is the probability of failing to open the relief valve when needed"*.

$$P =? \ [\, (\texttt{pressure} > \texttt{threshold}) \ \ \& \ \ (\overline{\texttt{relief}}) \,] \tag{2.5.2}$$

2.6 Recap

So far in this chapter, we presented some basic concepts about failures, safety, system reliability and verification. These concepts are important for understanding the proposed model based safety assessment (MBSA) approach presented in Chapter 6, as well as the existing approaches. In the subsequent sections, we discuss some of the existing approaches for MBSA and provide a critical comparison with the proposed approach. We further provided a classification approach and describe some of the similarities and differences between them.

2.7 Related Work

In this section, we present a range of existing approaches for model-based safety assessment (MBSA) and classify them by their defining characteristics. Among these techniques, we narrow the focus of our survey to exclude those approaches that are not applicable to functional safety standards (e.g., IEC 61508). For example, we exclude software reliability modelling [67] techniques since IEC 61508 consider reliability models only for random errors in the hardware. Software errors are considered systematic in nature are dealt with avoidance measures such as using validation techniques on software specification, design and source code. Furthermore, we also excluded techniques that only perform qualitative analyses (e.g., [68]), as they do not address random errors in E/E/PE systems induced from hardware component failures.

2.7.1 Classification

Lisagor et al. [46] present a basis for classification of the existing MBSA approaches. The most important aspect of this classification is the engineering semantics for the modelling failures and nominal behaviours. This leads to the following categories.

- ***Failure effect modelling*** (FEM) that is, modelling the effects of failure on the nominal behaviour of the system model, for example, the *stuck at* failure pattern can be modelled by disabling the potential to change in the corresponding model component.
- ***Failure logic modelling*** (FLM) that is, modelling the conditions that induce the actual failure e.g., current overflow, pressure overload, or temperature overrun scenarios. These scenarios are considered beyond the nominal behaviours of the system and therefore, are often difficult to model. Furthermore, in a component-based design paradigm, the reuseability of the component should also be applied with additional care since, two similar components may fail under different circumstances.
- ***Hybrid failure modelling*** is a combination of both of the above approaches.

Another criteria for categorization is the approach used to construct the system safety models. Among these, we find that most existing techniques fall in one of the following categories.

- Using ***dedicated models*** built for the purpose of safety assessment. Such models are free from complexities, for example, engineering semantics that may be unnecessary for the purpose of safety analysis.
- Using system ***development models*** after manually or automatically extending it with safety related information (e.g., as annotations). The main benefit of this approach is the partial utilisation of the system model, which ensure the consistency between safety assessment results and the deliverables of the development processes.

We further inspect the syntax and structure used for representing the system model. Here, syntax and structure refer to the modelling language used for expressing the system behaviour and its constraints. Three categories can be derived from the choice of languages of the existing approaches, namely the following.

- Using an ***open standard*** for the purpose of model-based design such as AADL [69] and IEC 61499 [11]
- Using a ***proprietary standard*** of model-based development such as SCADE [70] and Simulink [71]
- Using a ***domain specific language*** that is developed for the explicit purpose of safety analysis such as AltaRica [72] and SAML [55]

Later in this section, we present a few approaches that are closely related to the proposed approach. These approaches use high-level modelling formalism for modelling system's nominal and failure behaviour and enable quantitative risk assessment and safety assessment.

2.7.2 AADL based approaches

Architecture analysis and design language (AADL) is an open standard by the *international society of automotive engineers* (SAE) [73]. AADL presents a textual and graphical language (similar to UML) for model-based design of large performance critical systems such as quality of service for video-on-demand. The error modelling annex [74] provides language features for modelling failures in embedded systems. These errors can be stochastic in nature and describe faults in system behaviours. A number of AADL based approaches have since emerged to that perform safety assessment or offer

reliability measure. Joshi et al. [75] present an algorithm for automatic extractions of fault tree from a given AADL model for the purpose of safety analysis. Rugina et al. [76] have presented a tool named ADAPT to generate generalised stochastic Petri-Nets [77] by performing a rule-based transformation of the given AADL model. The generated model can then be analysed by a range of existing tools such as GreatSPN [1].

2.7.3 SAML

SAML is a tool-independent framework for modelling qualitative and quantitative aspects of safety-related systems [78]. It adopts a formal approach for modelling stochastic and non-deterministic behaviours as Markov decision processes. The model can then be automatically converted into a range of tool-specific formats accepted by various analysis tools e.g., NuSMV [2] and Prism model checker [3] can be used for qualitative and quantitative analyses [55] respectively. Similar to ADAPT tool for AADL , SAML is also available as an Eclipse plugin [79] for enhanced developer experience.

2.7.4 Figaro

Similar to SAML , Figaro is yet another tool-independent framework for modelling failure related information [80]. Models are represented in Figaro language, which a text-based language and resembles object-oriented syntax. In addition, a pre-described knowledge base is also supported that acts like a library of rules thereby reducing the modelling time. The language supports both continuous dynamics and discrete time model. Upon completion, the represented model can be transformed into various tool-specific formats for qualitative and quantitative analyses such as Markov model generation and analysis, and Monte Carlo simulation.

2.7.5 Hip-HOPS

Hierarchically Performed Hazard Origin and Propagation Studies (Hip-HOPS) [81] is an integrated technique for safety analysis that simplifies *functional failure analysis* (FFA), *failure mode and effect analysis* (FMEA), and *fault tree analysis* (FTA). The system design

[1] `http://www.di.unito.it/~greatspn`
[2] `http://nusmv.fbk.eu`
[3] `http://www.prismmodelchecker.org`

is represented as a hierarchy of connected components. Failure behaviours can be specified as a failure components that connect with other components using inputs and outputs (IO) on their respective interfaces. Failure effects can be propagated between components using these IO. A detailed approach of performing temporal FTA using Hip-HOPS is presented in [82].

2.7.6 Simulink based approaches

Gomes et al. [50] present a Simulink based approach for safety assessment. The approach uses the component failure information gathered using FMEA to model failures. A rule-based translation of Simulink designs uses this information to generate continuous time Markov chains in Prism language [58]. The generated model is then subjected to probabilistic verification to perform quantitative risk assessment using Prism model checker. A similar approach is also presented by Beer et al. [57] where Simulink / State-Flow designs are translated to Prism language for the purpose of quantitative analysis.

2.7.7 SysML and UML-based approaches

Various approaches exist that utilise SysML and UML for the purpose of modelling safety-related systems and their safety aspects [48, 83–86]. These approaches are usually based on annotations that are later used to derive safety models automatically (e.g., fault trees). SysML based techniques that also perform quantitative analysis also exist, which are primarily based on probabilistic verification [87, 88].

2.7.8 AltaRica

AltaRica language [72] is designed for describing behaviours on system failures. The language uses the formalism of states and transitions to model nominal and failures as temporal sequences. Failure logic is modelled as undesired sequences of transitions that lead to bad states [89]. Several approaches based on AltaRica exist that extend the language syntax and semantics to solve different problems, for example, in addition to discrete-time models, hybrid and timed extensions also exist [90] that aim to verify the continuous dynamic and perform timing verification of real-time systems. Being a high-level language, AltaRica models can represent various safety-related aspects of the system. This information can be automatically extracted to perform various classical analyses. These include

automatic fault tree generation [91], Markov chain generation, stochastic simulation, as well as traditional qualitative formal verification [92].

2.7.9 Stochastic Function Blocks

In Chapter 6, we present a model-based safety approach that utilises a novel structure named stochastic function blocks or the purpose of modelling probabilistic and non-deterministic aspects e.g., for failure logic and effect modelling. These structure seamlessly integrate with the development models and allow unified safety analysis for both qualitative and quantitative verification of the overall system. We have used the classification criteria presented in [46] for comparing the proposed approach with the approached presented above presented the result in Table 2.1.

Table 2.1: Classification of model-based safety assessment approaches

	Model Origin	**Modelling Language**	**Failure Modelling**
SAML	Dedicated	Domain Specific	FEM
Figaro	Dedicated	Domain Specific	FEM
Hip-HOPS	Development	Domain Specific	FEM
Simulink based approach	Development	Proprietary	Hybrid
AADL based approaches	Dedicated	Open	Varies
AltaRica	Dedicated	Domain Specific	Hybrid
Proposed Approach	Development	Open Standard	Hybrid

In this table, the first criteria for the classification is the origin of the model i.e., whether the model was derived from system development model or constructed for the express purpose of safety analysis. The benefit of the earlier of two is the ability to seamlessly integrated into the model-driven practices e.g., model-based development and model-based testing. Furthermore, the model used for safety analysis is the as the one used for development, which avoids inconsistent results. From the said table we see that only a few approaches[50, 57, 81, 82, 90–92] utilise the development models including the approach proposed in this thesis.

The second criteria use for the classification, is the type of the language used for modelling safety related aspects e.g., failure effect and propagation, failure logic modelling. Based om this criteria we see that AADL based approaches are the only other approaches that uses an open standard [73, 74] for modelling failures and related aspects. Similar to the AADL based approaches, the proposed approach also uses an open standard namely IEC 61499, for the modelling purpose purpose.

Lastly, we examine the presented approaches for their capabilities of modelling failure information. We see that many approaches are only restricted to *failure effect modelling* (FEM). This type of failure information is easy to model and analyse. *Failure logic modelling* (FLM) on the other hand, is difficult to model but provides deeper understanding of the model i.e., through modelling *"how"* the failure occurs instead of just modelling the effects of failures. A hybrid approach allows both FEM and FLM and results in a unified model and is, therefore, superior to just performing FEM. Similar to the proposed approach, both Simulink and AltaRica based approaches offer hybrid mode of modelling failure logic as well as failure effects.

Based on this criteria, the proposed approach is the only approach that uses system development models constructed using an open standard (IEC 61499), which offers both failure logic and failure effect modelling. Furthermore, in the industrial automation domain, the proposed approach is the only model-based safety approach that offers a unified qualitative and quantitative analysis.

2.7.10 Discussion

In Chapter 6 we present a model-based safety assessment technique that is closely related with the approach presented above. Similar to AltaRica [72], Figaro [80], SAML [55], AADL [74] and Simulink [93] based approaches, the proposed approach also provides dedicated syntax and semantics for modelling stochastic behaviours and provides a pathway to perform qualitative and quantitative verification. However, unlike most of these approaches the proposed approach is based on an open standard IEC 61499 similar to AADL , which is also an open standard [73].

Among the presented approaches, SAML is the most closely related approach in the sense that it too adopts a formal approach towards modelling failures, performs qualitative and quantitative analysis in a unified manner, and performs rule-based transformation of the model to generate Prism models. However, unlike the proposed approach, it does not support the hierarchy of components and their dependencies through explicit inputs and outputs on their interfaces. Furthermore, most of these approaches do not utilise or only partially utilise (e.g., [81, 93]) system development models. This is unlike the proposed approach where system development models are incorporated in the safety analysis. The benefit of this approach is that it reduces the gap between the result of safety analysis and the actual deployed system. Also, it enables seamless integration of safety analysis processes into the model-driven development activities.

3

IEC 61508 in a Nutshell

An Introduction to Functional Safety

Safety is often defined as,*"freedom from unacceptable risk"* [7]. This definition demands criteria on what is deemed *acceptable*, and how to achieve *freedom* from it. Both of these questions can be addressed using the notion of *Functional safety*. Functional safety is the name of addressing safety in its own rights by establishing safety targets through analyses, and allocating safety functions to avoid the identified unacceptable scenarios. IEC 61508 is a standard for functional safety that applies to safety-related systems that incorporate electric/electronic/programmable electronic (E/E/PE) components and devices (latest edition published in 2010) [3]. IEC 61508 is generically applicable to all E/E/PE systems and not specific to any application or domain. It adopts a lifecycle approach where risk reduction techniques are applied throughout its stages. This safety lifecycle applies to all electric, electronic and programmable aspects of embedded systems. Furthermore, if a safety-related system consists of even a single E/E/PE component, all other components (mechanical, chemical, pneumatic) of the system must also be considered for IEC 61508 practices. In fact, all components of the system that contribute to risk and offer risk reduction practices must be considered.

Bestpfe.com

3.1 Basic Concepts

The follow sections present some basic concepts extracted from IEC 61508 that are helpful in understanding rest of this chapter.

3.1.1 Hazard and Risk

Hazard is defined as, *"an occurrence with undesirable consequences"*. The two attributes that arise from this definition are the likelihood of such an occurrence and the severity of consequences. According to IEC 61508, the likelihood of a hazard is quantifiable as presented in Table 3.1. Similarly, the severity of consequences is also a measurable attribute, as presented in Table 3.2. Together, these two attributes help us define *risk* i.e., *"risk the likelihood of hazard actually occurring"*.

Table 3.1: Likelihood of hazards

Category	***Definition***	***Range*** *(incidents/year)*
Frequent	Can occur many times in system lifetime	$> 10^{-3}$
Probable	Can occur several times in system lifetime	10^{-3} to 10^{-4}
Occasional	Can occur at least once in system lifetime	10^{-4} to 10^{-5}
Remote	Unlikely to occur in system lifetime	10^{-5} to 10^{-6}
Improbable	Very unlikely to occur in system lifetime	10^{-6} to 10^{-7}
Incredible	Do not believe that it would occur in system lifetime	$< 10^{-3}$

The said two attributes also helps us establish *risk matrix*, which is the basis of *risk mitigation* practices, presented in Table 3.3. Risk matrix is a tool to classify risk according to its severity, which helps to make various types of decisions regarding risk mitigation.

Table 3.2: Severity of hazards

Category	***Definition***
Catastrophic	Multiple losses of lives
Critical	Loss of a single life
Marginal	Major injuries to one or more persons
Negligible	Minor injuries at worst

Table 3.3: Risk Matrix

Likelihood/Severity	*Negligible*	*Marginal*	*Critical*	*Catastrophic*
Incredible	Class IV	Class IV	Class IV	Class IV
Improbable	Class IV	Class IV	Class III	Class III
Remote	Class IV	Class III	Class III	Class II
Occasional	Class III	Class III	Class II	Class I
Probable	Class III	Class II	Class I	Class I
Frequent	Class II	Class I	Class I	Class I

3.1.2 As Low as Reasonably Possible (ALARP)

Risks are broadly categorised in three categories as shown in Figure 3.1, where risk is either (1) so large that it is refused, or (2) is so negligible that it is ignored, or (3) neither too large to be addressable nor too small to be neglected. In the 3rd case, the risk hazard is reduced to what is practically reasonable. The principle of *As low as reasonably possible* (ALARP) is therefore used to determine the tolerable risk. For example, using this principle in conjunction with the risk matrix we can establish that all *Class I* and *Class II* are considered unacceptable. Whereas, risk belonging to a *Class III* are mitigated only if the required effort is justifiable. An example of acceptable classification based on ALARP is presented as follows.

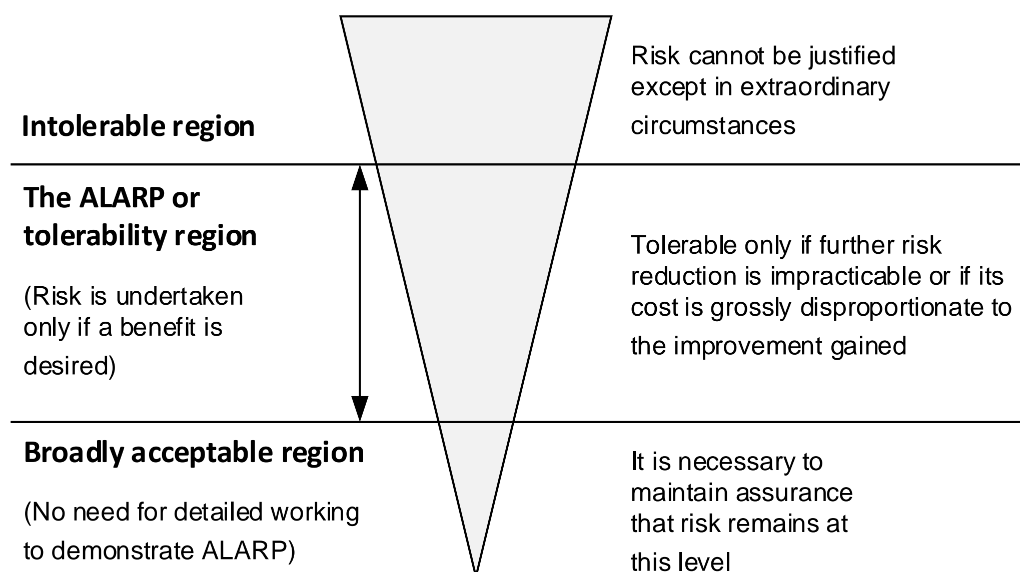


Figure 3.1: Tolerable risk and ALARP (source: [1])

BESTPFE.COM
List of research project topics and materials

- **Class I** - Unacceptable at any cost. Risks in this class must be eliminated i.e., through redundancy or other countermeasures.
- **Class II** - Highly undesirable. Risks in this class are only tolerated if the effort or cost to eliminate these are grossly disproportionate to the benefits achieved.
- **Class III** - Tolerable only the cost to eliminate outweighs the advantages.
- **Class IV** - Acceptable, but may require monitoring to avoid catastrophic occurrences.

3.1.3 Safety Function

Safety functions are implemented by safety-related systems to protect against a specific hazard and maintain the safe-state of the a system. Safety functions mitigate the risk of hazardous events through active detection and prevention in contrast with taking passive safety measures. Fault tolerance is often built into such systems to reduce the chances of unsafe failures. For such systems, an acceptable level of rigour in the design and implementation of such systems is often needed to be demonstrated, for example, for policy conformance, customer confidence, or as a legal requirement.

3.1.4 Safety Integrity Levels

Based on the application and its interaction with the environment, some systems may have to exhibit a higher level confidence than others, which mandates establishing an ordinal scale of confidence levels. *Safety integrity levels* (SIL) is a unified measure of confidence in the safety practices adopted for a system. IEC 61508 defines four distinct SILs: SIL1 to SIL4, with SIL4 being the highest. Table 3.4 shows the recommended values of acceptable failure rates of safety-related E/E/PE systems for different SILs, where a *high-demand rate* signifies high usage of the system during its operation. An example of *low-demand rate* is an elevator that may be used for less than an hour every day.

3.2 Meeting Basic Requirements

IEC 61508 adopts a lifecycle approach for performing system design, development, deployment and the eventual decommissioning. This allows early identification and addressing

Table 3.4: Safety Integrity Levels (SILs) (source: [3])

Safety-Integrity Level	***High demand rate***	***Low demand rate***
	Dangerous failures/hour	*Probability of failure on demand*
SIL4	$\geq 10^{-9}$ to $< 10^{-8}$	$\geq 10^{-5}$ to $< 10^{-4}$
SIL3	$\geq 10^{-8}$ to $< 10^{-7}$	$\geq 10^{-4}$ to $< 10^{-3}$
SIL2	$\geq 10^{-7}$ to $< 10^{-6}$	$\geq 10^{-3}$ to $< 10^{-2}$
SIL1	$\geq 10^{-6}$ to $< 10^{-5}$	$\geq 10^{-2}$ to $< 10^{-1}$

of safety issues that may be caused at various stages of the system development. An analysis of various safety related incidents in control systems [2] reveals that such incidents are caused in various phases of system development as shown in Figure 3.2. Therefore, adopting a structured lifecycle-based approach to the overall system development makes reasonable sense.

A typical product safety lifecycle under IEC 61508 is depicted in Figure 3.3. The first few phases of this lifecycle address the concept and scope of the system. The hazard and risk analysis phase involves identifying hazards using techniques like *Preliminary Hazards and Operability study* (HAZOP) and *Layers of Protection Analysis* (LOPA) etc. and establishing safety requirements to prevent the said hazards. Overall safety requirements are allocated to the corresponding subsystems and safety functions are identified. This is followed by the realisation phase, where hardware E/E/PE system and software are designed and implemented. This phase is of the most relevant to our work and, therefore, demands the most attention in our context. Other activities including installation,

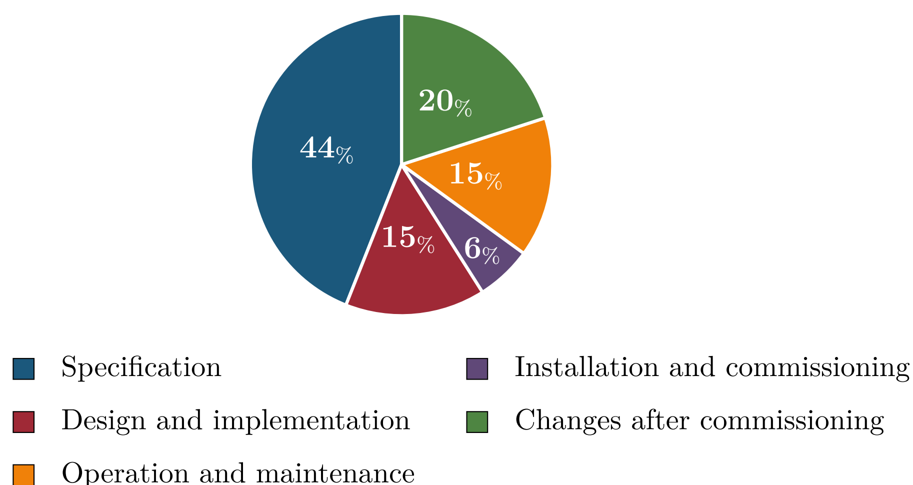


Figure 3.2: Primary cause of incidents by phase (source: [2])

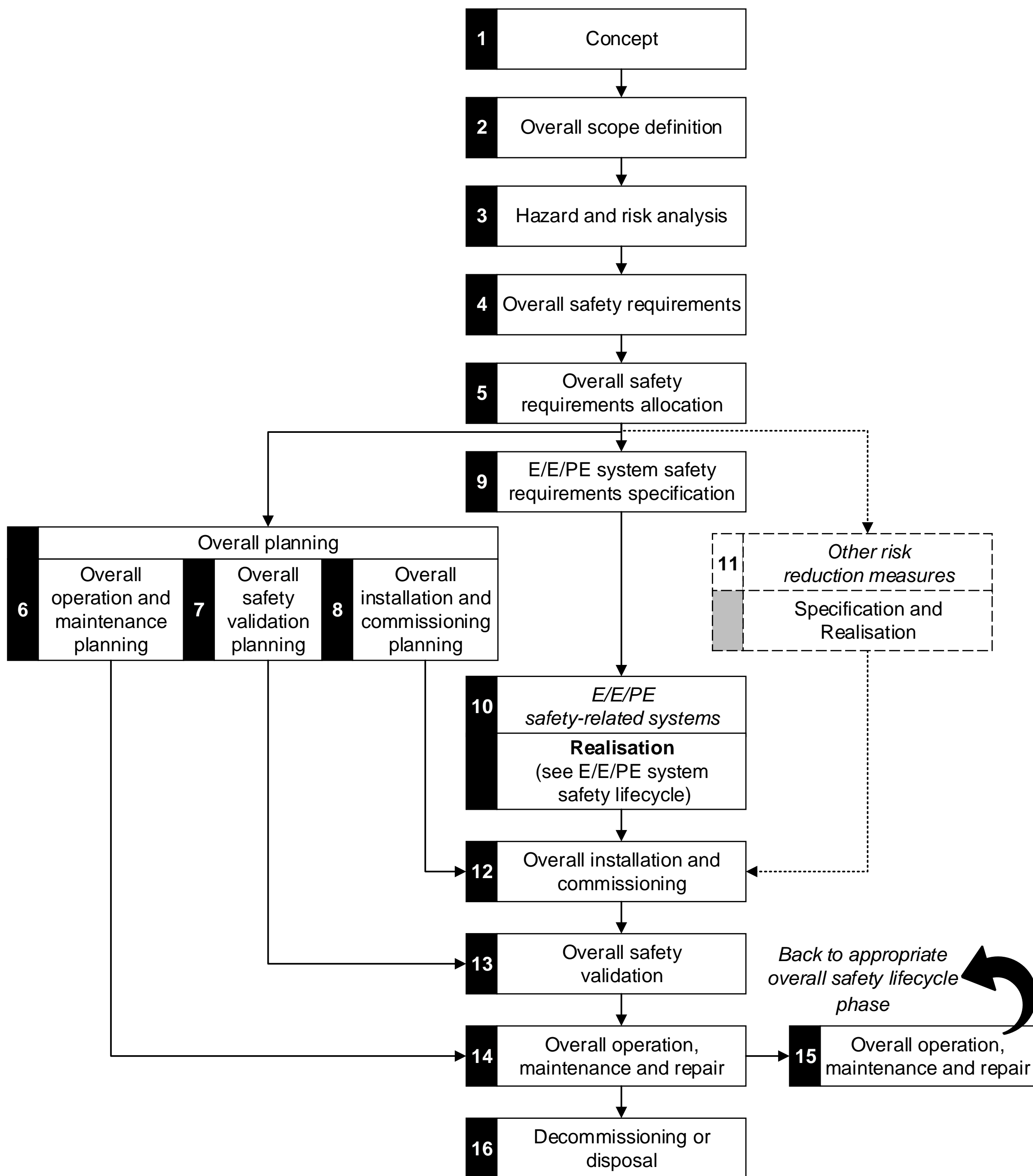


Figure 3.3: Overall safety lifecycle (source: [3])

maintenance, modification, and decommissioning are not discussed in detail due to the scope of discussion. Readers are advised to consult the standard for more details.

3.2.1 Establishing SIL Targets

The target SIL can be established using any of the three prescribed ways namely, using the quantitative approach, layers of protection analysis (LOPA) or the risk graph approach.

3.2.1.1 The Quantitative Approach

A quantitative approach based on maximum tolerable risk is prescribed in the standard. This approach can be applied both to the low-demand and high-demand safety functions. In low-demand safety functions, the maximum tolerable risk is estimated using the rate of fatalities per incident. For example, assume that an incident has a maximum probability of occurrence $A = 10^{-6}$ per annum, and $B = 10^{-3}$ of these incidents lead to fatality. This makes the maximum tolerable risk $C = A/B = 10^{-3}$. Now if the reliability model of the system predicts a failure once per seven years ($D = 1.43 \times 10^{-1}$), the probability of failure on demand needs to be $E = C/D = 7 \times 10^{-3}$. Using low-demand column of Table 3.4, the system needs to target SIL2. Similarly, take an example of a high-demand safety function where a maximum 10^{-5} fatalities are tolerable per-annum. If it is known (e.g., through a statistical study) that 1 out of 50 incidents lead to a fatality, we get the maximum tolerable failure rate of $10^{-5} \times 50 = 5 \times 10^{-4}$ per annum, which is same as 5.708×10^{-8} per hour. Using the high-demand column of Table 3.4, the system should target SIL3.

3.2.1.2 Layers of Protection Analysis (LOPA)

LOPA is a semi-quantitative approach is only applicable to low-demand systems and safety functions. The approach assumes multiple layers of protections against various hazards and the probability of failure is estimated by multiplying the source of hazardous events by the probability failure of a particular layer. Next, these probabilities are assigned severity and likelihood using a reference table lookup based approach, which is referred to as *mitigated consequence frequency*. These values are then compared against the policies and historical data to as criteria to establish where additional risk reduction is required. This data is then used to prepare LOPA worksheets, which gives the maximum tolerable risk. This technique is often disused in favour of the quantitative approach presented above. More details about this technique are given Annex. F of IEC 61508-5.

3.2.1.3 Risk Graph Approach

This is a qualitative approach for establishing a target SIL that is suitable for low-demand safety functions. Even though the standard allows this approach, it is often disused in the favour of the approaches presented above. Risk graph is a structure that is built using the qualitative parameters for risk mitigation i.e., the following.

- C = Consequence risk parameter
- F = Frequency and exposure time risk parameter
- P = Possibility of failing to avoid risk parameter
- W = Probability of unwanted occurrence

These parameters are arranged in increasing order in a ladder-like structure such that, $C_A < C_B < C_C < C_D$, $F_A < F_B$, $P_A < P_B$, and $W_1 < W_2 < W_3$. This way, a risk with higher consequence, frequency, possibility of failure and occurrence imposes more rigorous level risk mitigation target. The overall SIL target is estimated by evaluating the graph for all identified risks.

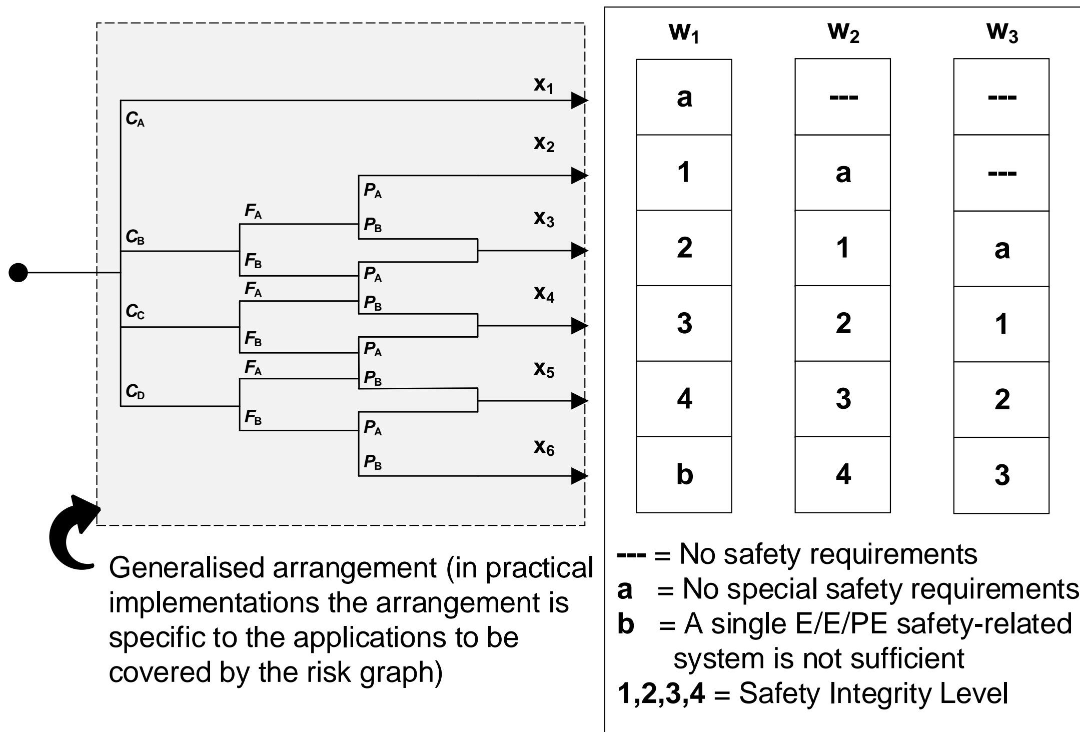


Figure 3.4: An example of a risk graph (source: [1])

Using the pressure-overrun hazard from the boiler system example (see Chapter 2), we can assign qualitative values to the various factors as follows. The consequence of this hazard is not a lot since we have a rupture disk and evacuation alarm installed, so assign $C = C_B$. Next, the frequency of this hazard is high due to high pressure involved in operations, so we assign $F = F_B$. Next, we assign a high value to the possibility of failing to avoid risk $P = P_B$. Lastly, we assign a probability of unwanted occurrence as low, since we have multiple layers of protection i.e., $W = W_1$. Starting from the left side, we reach SIL 2, which means that the safety function implemented for this hazard must adhere to recommendations of SIL 2. More details on this technique are available in IEC 61508-5 Annex. E.

3.2.2 Function Safety Assessment

Functional safety assessments (FSA) are supervisory processes and are carried out to demonstrate the compliance with an established SIL target. These assessments are used as a tool during the lifecycle to evaluate and improve the system design and development process. FSA is divided into seven steps [17] as shown in Figure 3.5. These steps are labelled with the relevant part of the IEC 61508 standard, which is split in a total of seven parts. First three parts provide recommendations on achieving functional safety. Whereas, last four parts provide supplementary materials such as worked out examples, definitions, tables, application guidelines etc. Figure 3.5 labels each of the step in FSA with the relevant part. We briefly describe these steps as following.

- **Step 1** — Establish functional safety capability and competence of the assessor and/or design organisation through management processes. This involves development, procurement and recruitment of the necessary resources.
- **Step 2** — Establish a risk target. This involves hazard identification and setting the maximum tolerable failure rates.
- **Step 3** — Identify safety-related functions through a formal approach such as fault tree analysis, which is useful for identifying the consequences of hazardous events. This results in the identification of protection system(s) for which the SIL is required.
- **Step 4** — Establish SIL for the safety-related elements using the recommended qualitative and/or quantitative techniques as described in IEC 61508-1 [3] such as the numerical assessment and risk-graph approach.

- **Step 5** — Quantitative assessment of the safety-related system is performed in accordance with the recommendations provided in IEC 61508-2 [94]. Reliability modelling techniques such as, reliability block diagrams, fault tree analysis and Markov analysis are used to determine system reliability. These methods use failure rates of safety-related elements to calculate the overall failure rates and reliability. For a basic introduction to these concepts please refer to Chapter 2.

- **Step 6** — Qualitative assessment against the target SIL is performed in accordance with the recommendations provided in IEC 61508-3 [4]. Several lifecycle activities are performed to avoid the systematic errors. Several recommendations for software development practices are provided namely, the *V-Model* of development and software specific recommendations for development, validation and verification (IEC 61508-3 annex A, B, C [4]).

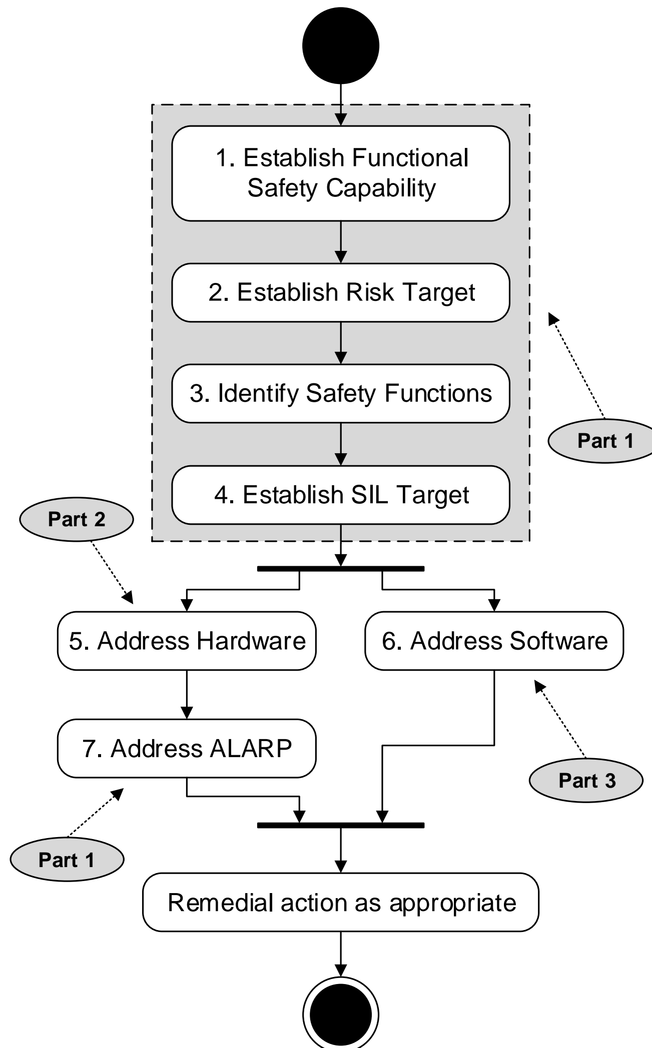


Figure 3.5: Seven steps of Functional Safety Assessment

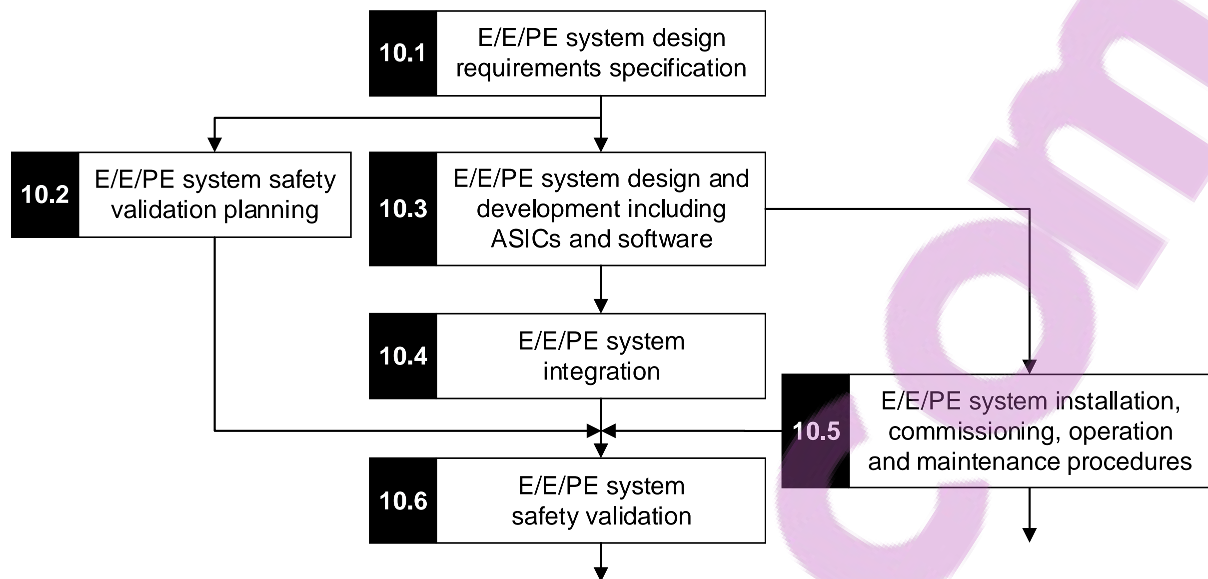


Figure 3.6: E/E/PE safety lifecycle (source: [3])

- **Step 7** — Achieve as low as reasonably possible (ALARP) risk. Meeting the targets established in step 4 is not enough. All practices and improvements that can reduce the failure rate must be adopted until a broadly acceptable failure rate is met.

3.3 Meeting Hardware Requirements

The E/E/PE safety lifecycle addresses the various phases of the hardware development and is shown in Figure 3.6. Every E/E/PE component of in the system follows this process, which becomes the basis for its certification. Safety requirements allocated to a particular component are specified and the hardware component is designed/procured that satisfy the target SIL requirements. These requirements are evaluated using the guidelines provided in Part 2 of the standard. These guidelines include clear instruction on safety specification, addressing common cause errors through separation of functions, evaluating random errors though reliability modelling, and various configurations of redundancies including on-chip redundancy. Furthermore, this part of the standard also proposes a V-Model for designing *Application Specific Integrated Circuits* (ASIC) that may help avoidance of design flaws through validation.

3.3.1 E/E/PE System Design Requirements Specification

The objective of IEC 61508-2 Clause 7.2 is to specify the design requirements for E/E/PE equipment and components, which are derived from E/E/PE system safety requirements. These requirements need to be clear, unambiguous and easy to understand and must contain all information required for the E/E/PE design and development activities such as all necessary hardware and software, information on individual components and their respective interfaces. Furthermore, all diagnostics and nominal operation modes their corresponding behaviours are listed. The requirements on the capability of these components to work under extreme conditions such as, weather, temperature, magnetic interference is also specified. Self-diagnostics and actions to be taken upon detection of dangerous errors is also included.

3.3.2 E/E/PE System Safety Validation Planning

The Clause 7.3 of IEC 61508-2 calls for planning the validation of E/E/PE safety-related system, which information about the procedures as well as the techniques to employ. This encompasses the steps involved in validation, the conditions and environments to use for the validation, acceptance criteria for the validations, and how to use the failure outcome of the validation to initiate a change request. The safety parameters and operation modes described in the design requirements are used to perform the actual validation, for example, validating immunity against electromagnetic interference.

3.3.3 E/E/PE System Design and Development

The Clause 7.4 of IEC 61508-2 and its various sub-clauses present requirements on the design and development of E/E/PE safety-related system. This includes requirements on the various components such as actuators, sensors, microprocessors, ASIC, software and its integration with the hardware. The requirements address the systematics capability of the E/E/PE system and its independence from common-cause errors, as well as random errors and fault redolence through reliability modelling and estimating safe failure fraction. In order to establish the systematic capability, one of the following routes can be taken.

- $\mathbf{1_S}$: Demonstrate compliance to requirements on avoidance of systematic faults i.e., by following recommendations from Clauses 7.4.6 and 7.4.7, as well as with IEC

61508-3 as appropriate

- **2_S** : Demonstrate compliance by providing evidence of proven use in the industry as required in IEC 61508-2 Clause 7.4.10
- **3_S** : Pre-existing software components can demonstrate their compliance with requirements of IEC 61508-3

Similarly, demonstrating conformance of hardware with the requirements has two routes.

- **1_H** : Demonstrate compliance based on concepts of hardware fault tolerance and safe failure fraction as appropriate for the target SIL
- **2_H** : Demonstrate compliance using user feedback reliability data, hardware fault tolerance and increased confidence levels from proven use as appropriate for the target SIL

3.3.4 Reliability Modelling

Reliability data from end user feedback can be used estimating E/E/PE system reliability using reliability modelling techniques. These techniques include *reliability block diagrams*, *quantitative fault trees*, and *Markov analysis*. With these techniques, practitioners can utilise reliability data of the individual components and their (possibly redundant) configuration to estimate the overall system reliability. Detailed information about these techniques and their example applications are presented in IEC 61508-6. Please refer to Chapter 2 for a basic introduction of these techniques.

3.3.5 Safe Failure Fraction

An acceptable level of estimated reliability may be achieved through implementing redundancy. However, a minimum level of redundancy must be implemented in the system to achieve a particular level of fault tolerance as described by *safe failure fraction* (SFF). SFF is a measure of what fraction of failures lead the system to a safe state and is calculated by using the formula given in Equation 3.3.1. This value is to be used to determine how much fault tolerance is required in a given E/E/PE equipment. The recommendations for complex components with potentially unknown failures like *integrated circuits*

BEST PFE.COM
List of research project topics and materials

and *microprocessors* is presented in Table 3.5. For example, it recommends that a maximum of SIL2 may be achieved with an SFF in [60%, 90%) if the E/E/PE system can tolerate one failure e.g, through a 1*oo*2 redundancy. Similarly, for an SFF ($< 60\%$), a minimum of two failures must be tolerable for achieving SIL2.

$$SFF = \frac{\lambda_{SD} + \lambda_{SU} + \lambda_{DD}}{\lambda_{SD} + \lambda_{SU} + \lambda_{DD} + \lambda_{DU}} \tag{3.3.1}$$

Where, λ_{SD} = Safe detected failures
λ_{SU} = Safe undetected failures
λ_{DD} = Dangerous detected failures
λ_{DU} = Dangerous undetected failures

Table 3.5: Maximum achievable SIL for a given SFF and fault tolerance

Safe Failure Fraction	**Hardware Fault Tolerance**		
	0	**1**	**2**
$< 60\%$	Not Allowed	SIL1	SIL2
$60\% - < 90\%$	SIL1	SIL2	SIL3
$90\% - < 99\%$	SIL2	SIL3	SIL4
$\geq 99\%$	SIL3	SIL4	SIL4

3.4 Meeting Software Requirements

The requirements on safety-related software are met by following the software safety lifecycle and following the SIL specific recommendations for each phase as shown in Figure 3.7. These recommendations are qualitative in nature and address the various aspects of specification, design, implementation, installation and maintenance. The realisation phase of software is executed in tandem with the corresponding E/E/PE equipment, which influences specification and various design decisions. The standard does not impose any particular on the choice of language or tools for the design and development of the software, however, a check-list of requirements must be satisfied. V-Model is proposed for the actual design and development activities such that each design stage in the process has a corresponding validation stage, which helps early detection and addressing of the identified issues. We further describe these requirements and suggestions in Section 3.4.

3.4.1 Software Functional Safety Plan

IEC 61508-3 Clause 6 outlines the requirement for creating a software functional safety plan encompassing the administration of relevant activities such as software design, development, procurement and modification. The objective is to ensure that the software safety is analysed and maintained as appropriate for the target SIL. This is achieved through ensuring that all software artefacts including source code, third-party components, tools for development and test etc., are appropriately stored and documented. Any modification must be recorded as a change request along with appropriate details including the reason for the change, criteria to analyse its impact on safety, and to carry our appropriate tests to ensure the software safety. Each of these activities is further qualified by the various sub-clauses of IEC 61508 Clause 7.

3.4.2 V-Model for Software Development

IEC 61508-3 Clause 7 requires following the V-Model for software realisation. V-Model is a process model in which every development stage has corresponding validation stage. Unlike the well-known waterfall model, practitioners of V-Model can go back to a previous stage either as a result of failing validation criteria or finding a flaw for which a previous stage is to be blamed. This allows early detection of errors and design defects and is therefore highly desirable for developing safety-related systems.

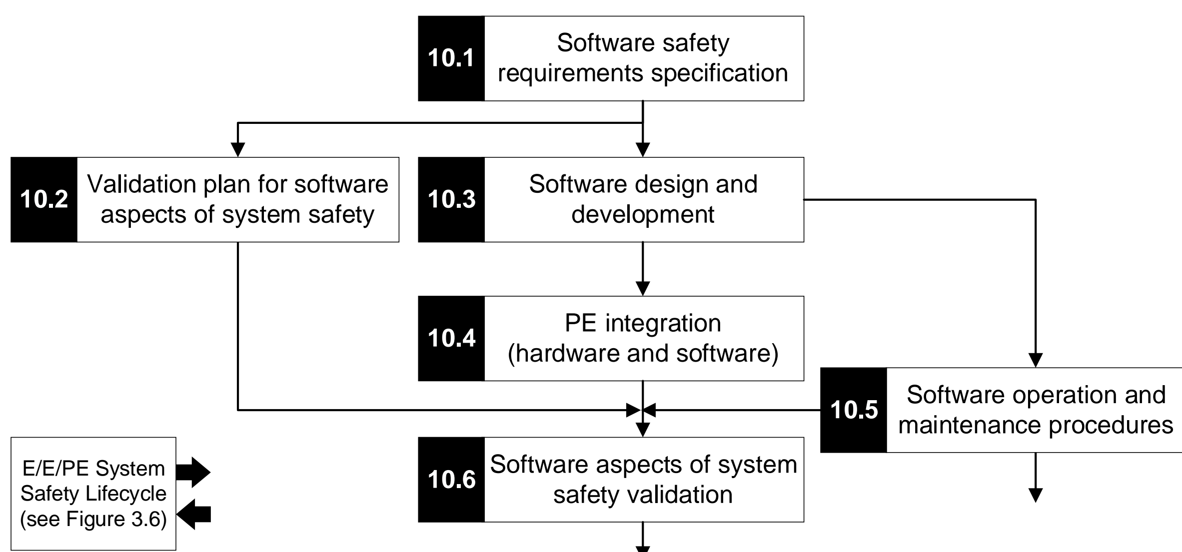


Figure 3.7: Software safety lifecycle (source: [3])

3.4.3 Software Safety Specification and Validation

IEC 61508-3 Clause 7.2 necessitates the development of software safety specification that is unambiguous, verifiable, feasible and is appropriate for the target SIL. This specification is often created as a document that is derived from the functional safety requirements and must contain enough information to perform the functional safety assessment of the software. Clauses 7.3 and 7.7 of IEC 61508-3 are related and require validation planning and its execution respectively. The planning document must provide a complete detail of activities to be performed on the specified equipment including who will be performing it, when and under which conditions. It also outlines all the various modes of operations that must be validated on the safety related software, both under normal and abnormal conditions. Furthermore, concise acceptable criteria must also be present in the specification, which determines whether a test was successful or failed. The execution of this validation is often performed on the software separately as well in an integrated fashion with the actual E/E/PE equipment. The list of activities and their outcomes are recorded along with the conditions under which the validation was performed. Any failure or discrepancy in the validation is used to issue an action item, which results in detection of the cause and is traced as a change request.

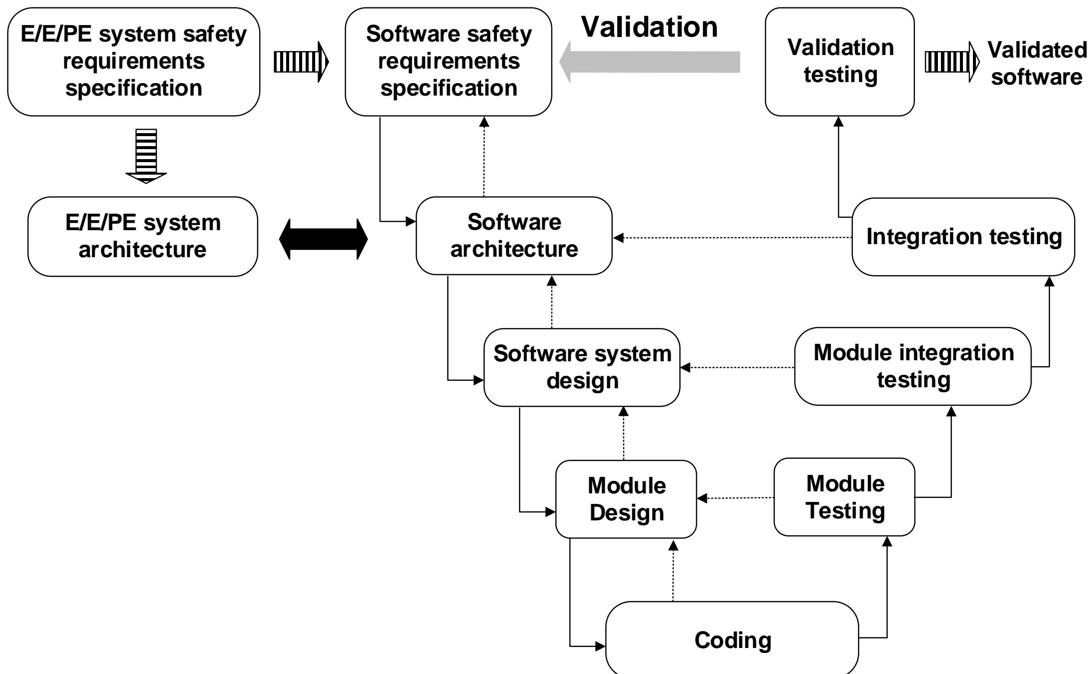


Figure 3.8: V-Model for the design and development of E/E/PE Software (source: [4])

3.4.4 Software Design and Development

IEC 61508-3 Clause 7.4 presents requirements on software design and development. Design techniques that enable good software engineering principals such as, abstraction and modularity are recommended. These design techniques need to be well understood and free from ambiguities. The flow of control and data should be clear and free from side effects on data integrity and scope. The overall complexity of the design, development and maintainability practices should be manageable. The entire software design should be considered safety related unless demonstrated otherwise. This includes implementation of diagnostic features that should not interfere with the nominal behaviour of the system. The overall architecture of the software should be clearly specified including interactions of the modules or components, their interdependencies and data flow, as well as the criteria for its integrated validation with the E/E/PE equipment. The development of the software is to be performed using quality assurance measures such as coding conventions, design patterns and documentation. Language features that may lead to undesirable states or unclear context are considered unsafe and are prohibited. The resulting software implementation should be understandable, testable and maintainable. Furthermore, code traceability is required for each module all they way up to specification stage. This ensures that the software is free from ad-hoc alterations and is built to its specifications.

3.4.5 Integration and Testing

Clause 7.5 of IEC 61508-3 presents requirements on the integration of the software and the hardware on the target E/E/PE equipment and performing testing. The requirements for this testing are created in the design and development phase that provide test cases and test suites to execute, information on tools, techniques and configurations to use for testing. Furthermore, it includes testing criteria and procedure to records the outcome of this testing along with instruction on how to initiate change request or a corrective action.

3.4.6 Operation and Modification

Software modification for enhancements, correction and adaptations must be done through an authorised process (IEC 61508-3 Clauses 7.6 and 7.8). The authorization information includes the reason for modification, hazards that are affected by this change and actual

modification details i.e., what modifications are performed. Criteria are established to validate that correctness and completeness of the modification against its requirements. Furthermore, this modification must be free from side effects or and unwanted behaviours, and should be testable and verifiable as appropriate for to the target SIL.

3.4.7 Software Verification

Software Verification (IEC 61508-3 Clause 7.9) aims to verify the consistency of outputs of each phase in the software safety lifecycle with the inputs it receives. This verification is performed in parallel with the design and development activities. A validation plan is created to verify each of the stage in the V-Model including software safety requirements specification, software architecture design, software system design, module design, source code, data, timing performance, software module testing, software integration testing, E/E/PE integration testing, and validation of software safety aspects.

3.4.8 Software Functional Safety Assessment

Clause 8 of IEC 61508 requires performing the functional safety assessment of software which is similar to already described in Section 3.2.2. Further guidelines on FSA are provided in IEC 61508-1 and IEC 61508-3 Annex. A.

3.4.9 IEC 61508-3 Annex. A

Annex. A of IEC 61508 comprises ten detailed tables that provide guidelines for selection of techniques for the various tasks procedures related to software safety lifecycle. Each technique or measure offers SIL specific recommendation as `R` (recommended), `NR` (not recommended), `HR` (highly recommended) or blank with no recommendation. These tables provide recommendations for the following phases.

3.4.10 IEC 61508-3 Annex. B

Annex. B of IEC 61508 provides nine detailed tables on techniques that are referred in Annex. A. These tables provide the following details.

Table 3.6: Tables provided in Annex. A of IEC 61508-3

Table	Clause	Purpose
A.1	7.2	Software safety requirements specification
A.2	7.4.3	Software architecture design
A.3	7.4.4	Support tools and programming language
A.4	7.4.5 and 7.4.6	Detailed design
A.5	7.4.7 and 7.4.8	Software module testing and integration
A.6	7.5	Programmable electronics integration (hardware & software)
A.7	7.7	Software aspects of system safety validation
A.8	7.8	Modification
A.9	7.9	Software verification
A.10	8	Functional safety assessment

Table 3.7: Tables provided in Annex. B of IEC 61508-3

Table	Referenced By	Purpose
B.1	A.4	Design and coding standards
B.2	A.5 and A.9	Dynamic analysis and testing
B.3	A.5, A.6 and A.7	Functional and black-box testing
B.4	A.10	Failure analysis
B.5	A.7	Modelling
B.6	A.5 and A.6	Performance testing
B.7	A.1, A.2 and A.4	Semi-formal methods
B.8	A.9	Static analysis
B.9	A.4	Modular approach

3.5 Summary

IEC 61508 provides a generic yet comprehensive guide for procedure and techniques to adopt for the specification, design, development and modification of E/E/PE equipment. IEC 61508-2 specifically addresses the requirements on the E/E/PE hardware and its integration with the software, whereas IEC 61508-3 provides specific requirements on the software. A large amount of effort goes into specification and planning phases, which is consistent with the study that finds the majority of causes of the incidents arising from these phases. Rigorous design and development techniques are recommended to avoid systematic errors i.e., through following SIL specific recommendations provided in Tables A.1-A.10 and B.1-B.9. Validation planning and its execution ensure that design and implementation are traceable to their specification and conform to the acceptance criteria. Any modification of the system must be authorised to avoid introduction of inadvertent errors or undesired behaviours.

3.6 Discussion

IEC 61508 imposes several requirements on the hardware and software including the validation of different phases involved in their design and development. Our current work addresses two specific requirements namely validation of hardware reliability against random errors and software against systematic errors. These two activities are largely performed in isolation with each other. The function correctness of the overall system is only established through integration tests, which is a non-exhaustive technique and also, it is performed too late in the development. In this thesis, we propose a model-based safety framework (see Chapter 6) for industrial automation systems where hardware reliability and software correctness is verified in a unified manner. The proposed approach is exhaustive in nature and offers an early estimation of system reliability and detection of systematic errors, which benefits the management of the overall system safety. Furthermore, we present a tool-chain (see Chapter 7) that implements the proposed approach along with an extended V-Model for design and development of safety-related industrial automation system. This provides developers with the necessary tools to design safety-related systems and provides a pathway to demonstrate conformance with IEC 61508 and related safety standards.

4

An Introduction to IEC 61499

Model-Based Design using Function Blocks

This chapter gives a brief introduction of IEC 61499 [11] that is tailored to fit the scope of this thesis and should be considered a summary of the basic concepts. In the first few sections, we present the concepts regarding *structure*, such as the different design elements of IEC 61499 and how they fit with each other to build complex control systems. This involves a discussion on various types of function blocks and the hierarchical system model. In the later sections, we discuss the *semantics*, i.e., the manner in which inputs are processed to implement the behaviour of a control system. In order to illustrate these concepts, this chapter uses a *Distribution Station* [95] example, with a control system implemented using IEC 61499.

4.1 Distribution station

The *Distribution Station* is a mechanical assembly that picks and places workpieces on a network of conveyor belts. Figure 4.1 shows the labelled diagram of a typical *Distribution*

Bestpfe.com

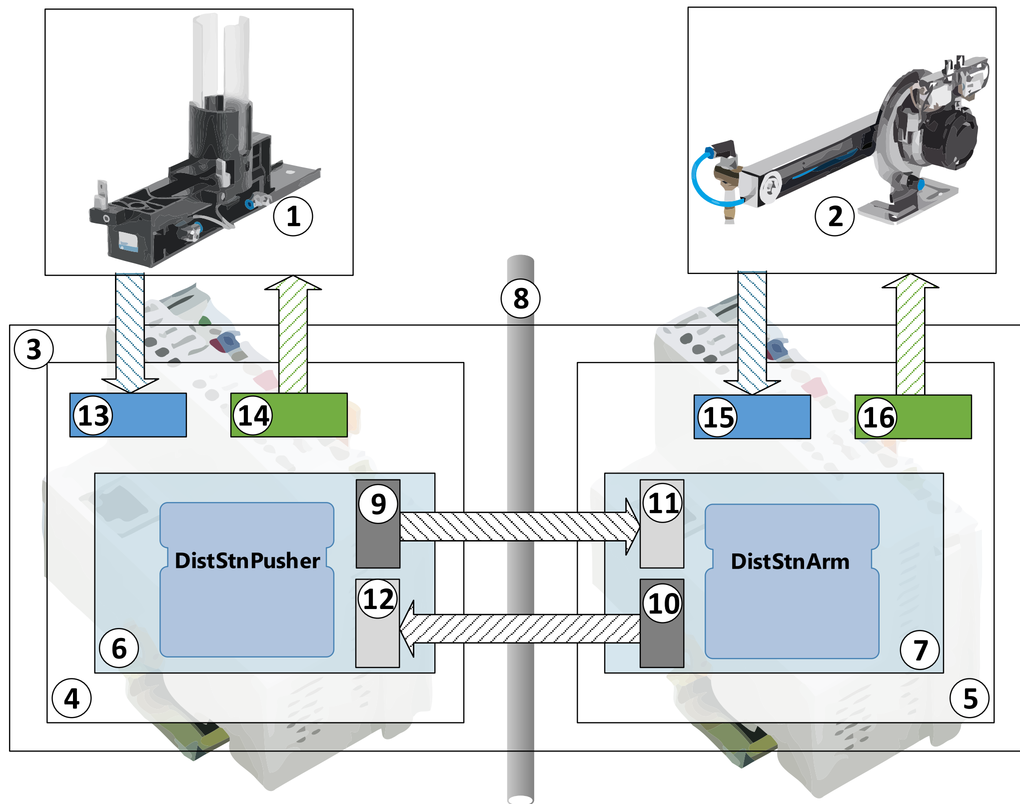


Figure 4.1: System design of *Distribution Station* representing the various components

Station, which consists of a *Pusher* ① that places workpieces on a pick-up location, and an *Arm* ② that picks up the placed items and puts them on a conveyor belt. This process, therefore, requires coordination between the independently operating mechanical apparatus that are controlled by individual *programmable logic controllers* (PLC). IEC 61499 allows programming such systems in an *object-oriented* manner, whereby all physical and logical components of the system are modelled using IEC 61499 design elements.

A top-down approach of implementation begins by creating a *system* ③ that contains two devices (labelled as ④ and ⑤), where each device represents a *programmable device* (e.g., PLC, PAC, microcontroller). A network segment ⑧ connects the two devices, thus enabling the communication and coordination between the tasks assigned to each device. Device models in IEC 61499 host device-specific behaviours, such as device drivers to control the external I/O (inputs ⑬,⑮ and outputs ⑭,⑯), as well as to provide the automation logic as *resources* ⑥, ⑦. This control logic is implemented as a network of interconnected function blocks that model the *execution behaviour* of the desired task, as well as any related dependencies, such as timers and communication infrastructure (e.g., ⑨-⑫). This systematic approach of modelling renders a resource as an independent operational unit of IEC 61499. While, in general, a device may contain more than one resource, in the *Distribution Station* example, each device hosts only a single resource,

i.e., ⑥ and ⑦ respectively.

In the following sections, we begin by describing the various types of function blocks and their respective attributes. Subsequently, we revisit the concepts of *system*, *devices*, and *resources*, and discuss the IEC 61499 implementation of the *Distribution Station* example.

4.2 Basic function block

Basic function blocks (BFB) are the atomic units of execution in IEC 61499. A BFB consists of two separations, i.e., a function block interface and an execution control chart (ECC) that operates over a set of events and variables. The execution of a BFB entails accepting inputs from its interface, processing the inputs using the ECC, and emitting outputs. We elaborate on these in the following.

4.2.1 A function block interface

A BFB is encapsulated by a function block interface, which exposes the respective inputs and outputs using ports. These input or output ports may be classified as either event or data ports. Figure 4.2 shows the interface of the function block that implements the *Arm* control logic. This interface exposes input events (PosChange, ItemStatus), output events (ArmCtrl, ArmStatus), as well as input variables (PosReadyToPickup, PosReadyToDropoff, ItemNeedsPickup), and output variables (ArmToPickup, ArmToDropoff, ArmSuck, ArmRelease, ArmClear).

Event ports are specialised to accept or emit events, which are pure signals that represent status only, i.e., they are either *absent* or *present.* On the other hand, data ports can accept or emit valued signals that consist of a typed-value, such as *Integer*, *String*, or *Boolean.* Variable ports of a special type *"Any"* can accept data from a range of typed-values. In addition, a concept of multiplicity is also applicable to data ports, which allows accepting or emitting arrays of values.

A data port can be associated with one or more event ports, as shown in Figure 4.2. For example, ItemNeedsPickup is associated with ItemStatus. However, this association can only be defined for ports of the matching flow direction, that is, input data ports can only be associated with input event ports. This event-data association regulates the data flow

BESTPFE.COM
List of research project topics and materials

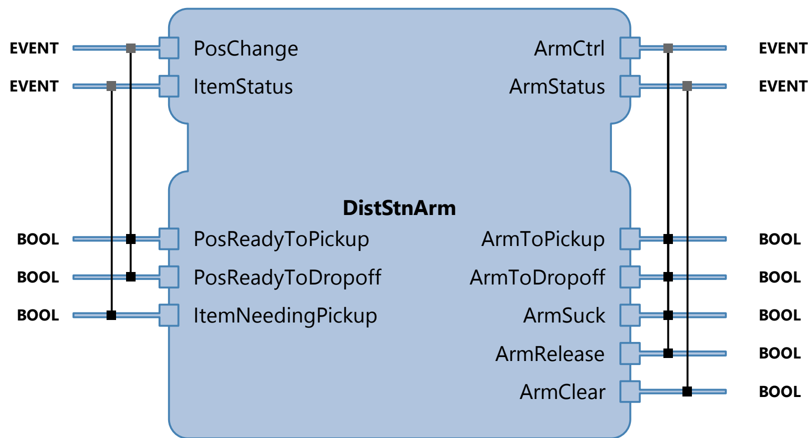


Figure 4.2: DistStnArm: an example of a basic function block

in and out of a BFB, i.e., new values are loaded or emitted from the data ports on the interface when an associated event is present. The lifetime of an event and mechanism to load or emit new data values is a source of ambiguity and has lead to a number of interpretations of the standard [12, 96]. We adopt the synchronous execution semantics (see Section 4.6.5) to eliminate these ambiguities.

4.2.2 Execution control chart (ECC)

The behaviour of a BFB is expressed as a Moore-type state machine, known as an execution control chart (ECC). An ECC reacts to input events and performs actions to generate the appropriate outputs. Figure 4.3 shows the ECC of the *Arm* controller BFB, which consists of six states, i.e., PickingUp, Waiting, Drop etc. States in ECCs have provision to execute algorithms and emit output events upon ingress, which are represented as ordered elements in their respective action-sets. As an example, the algorithm DropItem is executed, and the ArmCtrl and ArmStatus events are emitted upon entering the Drop state.

The execution of an ECC starts from its initial state (Waiting in Figure 4.3) and progresses by taking transitions, which are guarded by an input event and an optional Boolean expression over input and/or internal variables. Upon evaluation, a transition is considered to be enabled if the respective guard condition evaluates to `true`. The ECC will then transition to the next state by taking the enabled egress transition from the source state to the corresponding target state.

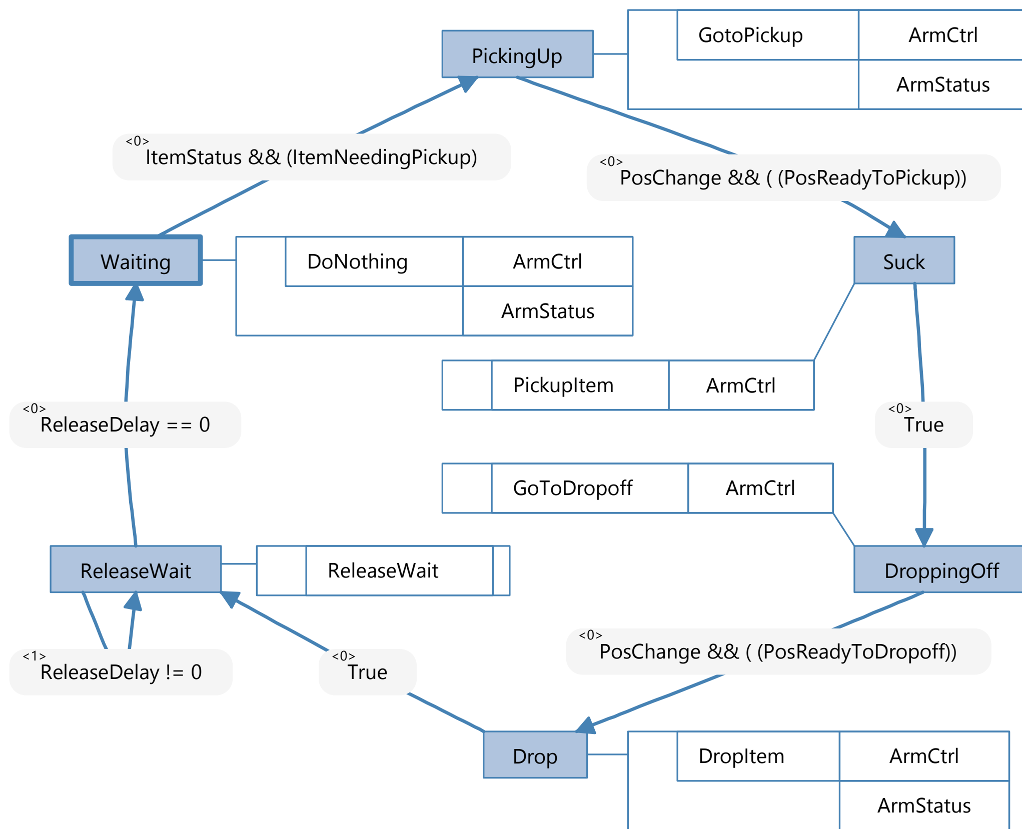


Figure 4.3: Execution control chart of the DistStnArm function block

```
1 me->ArmSuck = false;
2 me->ArmRelease = true;
3 me->ReleaseDelay = 5;
```

Figure 4.4: DropItem algorithm from the *Arm* controller BFB

4.2.3 Algorithms

An algorithm is a finite set of ordered statements that operate over the ECC variables. Typically, an algorithm consists of loops, branching, and update statements, which are used to consume inputs and generate outputs. The IEC 61499 standard allows algorithms to be specified in a variety of implementation-dependent languages. Common languages allowed by various implementations include Structured Text (ST) (e.g., [97]), Java (e.g., [98]), and C (e.g., [99, 100]). The DropItem algorithm from the *Arm* controller BFB is presented as follows using the C language. Here, the "`me`" is a *pointer* reference to the current instance of the function block, which is synonymous to the "`this`" keyword used in many object-oriented programming languages.

4.3 Composite function blocks

Composite function blocks (CFB) facilitate the representation of the structural hierarchy. CFBs are similar to BFBs in the sense that they too are encapsulated by function block interfaces. However, unlike a BFB, the behaviour of a CFB is implemented by a network of function blocks.

4.3.1 Type specification

Basic and composite function blocks may have different *type specifications*, which are referred to as function block types (FBTypes). A function block network (FBN) may consist of instances of various FBTypes, where any given FBType may be instantiated multiple times. This concept is very similar to the object-oriented programming paradigm, which contains *classes* (analogous to FBTypes) and their instances, namely *objects* (analogous to FB instances). These FB instances connect and communicate with each other using *wire connections*, and with external signals via the encapsulating function block interface of a CFB. This facilitates the structural hierarchy, i.e., a given FBN may contain instances of other CFBs that encapsulate sub-FBNs.

Figure 4.5 shows a function block network with two function block instances that communicate with each other using wire connections, for example, a Boolean output value ItemPresent of the *Pusher* instance can be read as ItemNeedingPickup by the *Arm* instance. Furthermore, some signals directly flow from the interface of the top-level CFB into the encapsulated FBN, for example, the event InputsChange is read from an external source and made available to the PosChange input event of both the *Pusher* and *Arm* instances. However, only compatible signals flow in this manner, meaning that an input event on a CFB interface can only flow into an input event of nested FB interfaces. Similarly, data flow in this manner must also conform to datatype compatibility, that is, a Boolean input on the CFB interface cannot flow into a String type input of the nested FB interface. One exception to this rule is the *"Any"* type, which as the name suggests, can accept any data type.

This mode of signal flow is thus directly responsible for effecting the interface definition of a CFB, i.e., if a nested FB needs an input from an external source, there must be an input defined on the CFB interface which flows into the said nested FB. This encapsulation of nested FBs from external sources simplifies the reuse of FBTypes.

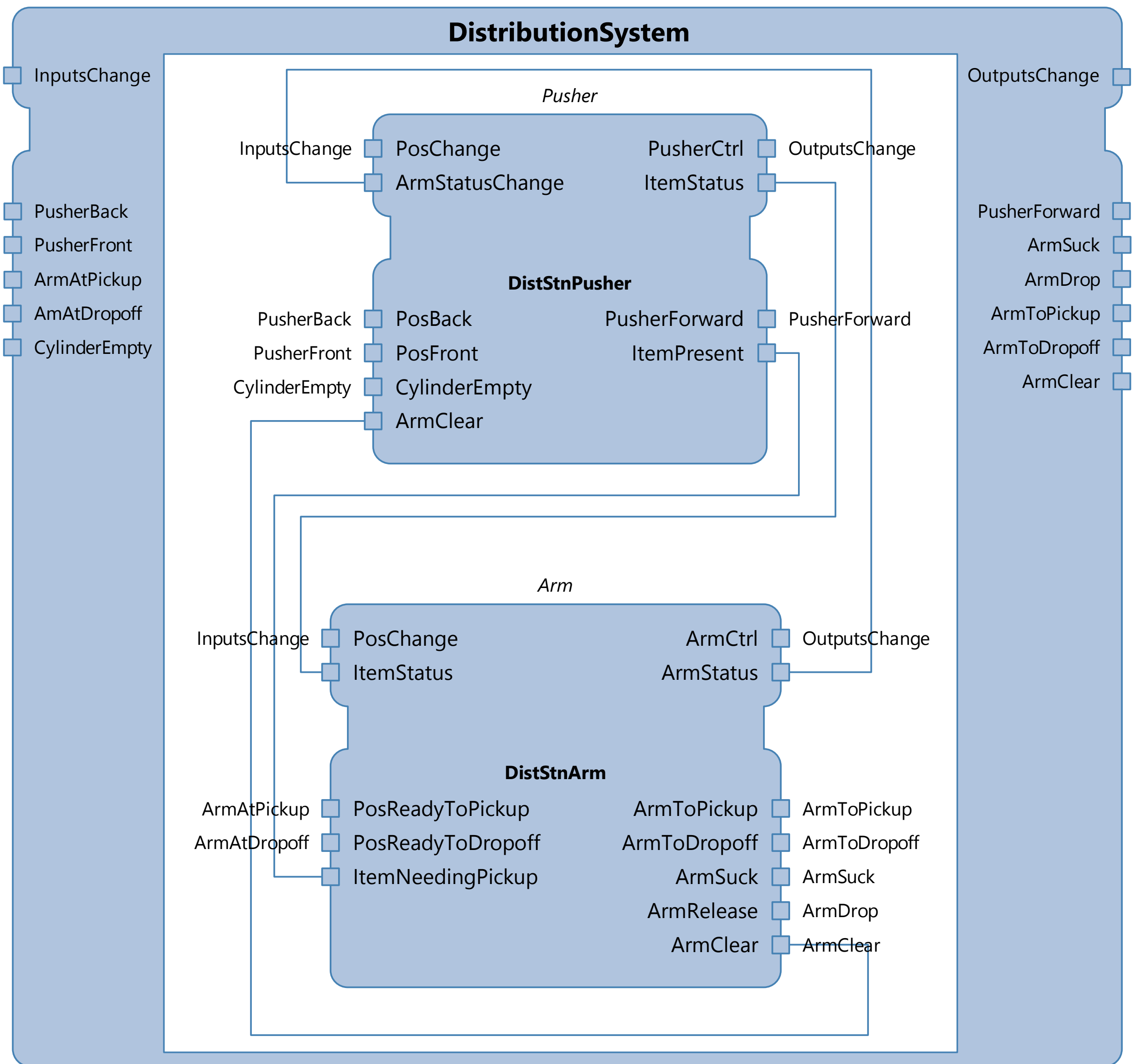


Figure 4.5: A composite function block with an encapsulated function block network

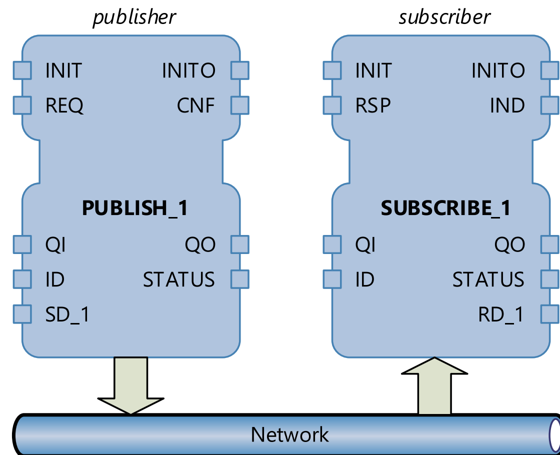


Figure 4.6: Publish-subscribe communication function blocks. The publisher function block is configured to publish a single data element, SD_1. The subscriber function block is configured to subscribe a single data element, RD_1

4.4 Service interface function blocks

Service interface function blocks (SIFB) can be considered as device drivers that connect the external environment with function block applications. These blocks are used to provide services to a function block application, such as the mapping of I/O pin interactions to event and data ports, and the sending of data over a network. Figure 4.1 shows an example of device drivers (see ⑬-⑯ in Figure 4.1) that are used to control the *programmable device's* I/O for actuation and sensing of the physical environment. There are two categories of SIFBs described in the standard, namely, *communication function blocks* and *management function blocks*.

While CFBs capture centralised entities, resources are reminiscent of tasks and devices represent PLCs. Hence, both resources and devices need specific entities that facilitate either task-level (inter-resource) or distributed (inter-device) communication. Communication function blocks are SIFBs that provide interfaces that enable communication between IEC 61499 resources. Different types of communication function blocks may be used to describe a variety of communication channels and protocols. Figure 4.1 shows an example of a pair of communication blocks that are used to achieve coordination between the *Arm* and the *Pusher* (see ⑨-⑫ in Figure 4.1). On the other hand, management function blocks are SIFBs which are used to coordinate/manage application-level functionality by providing services, such as starting, stopping, creating, and deleting function block instances or declarations. They are somewhat analogous to a task manager in

a traditional operating system.

Unlike BFBs, where the behaviour is specified using an ECC, SIFBs are specified using *time-sequence diagrams* from ISO/IEC 10731 [101]. Here, we present an example of such diagrams depicting the communication between publish-subscribe communication function blocks. The publish-subscribe pair is intended for unidirectional one-to-one or one-to-many communication. Figure 4.6 presents a pair of publish-subscribe communication function blocks, which sends a single data element from the publisher block to the subscriber block. This communication mechanism is used in the *Distribution Station* where the two devices coordinate with each other, as shown with labels ⑨ and ⑪, which represent a *publisher* and a *subscriber* receptively. The meaning of each input and output port on the SIFB interfaces is presented below:

- INIT – Event to initialise the SIFB.
- INITO – Event to indicate that the SIFB initialisation has been completed, which may or may not have been successful.
- REQ – Event to request the publisher block to transfer a data element over the network.
- CNF – Event to confirm a successful data transfer has been completed by the publisher.
- RSP – Event to indicate to the subscriber block that the application has processed the received data element.
- IND – Event to indicate that data arrived successfully in the subscriber.
- QI – A Boolean to indicate that the SIFB should be initialised when *true*, or to otherwise terminate the SIFB service when *false*.
- QO – A Boolean to indicate successful initialisation when *true*, or initialisation failure when *false*.
- ID – A communication identification string, such as the IP address and the port number.
- SD_1 – The data to be sent.
- RD_1 – The received data.

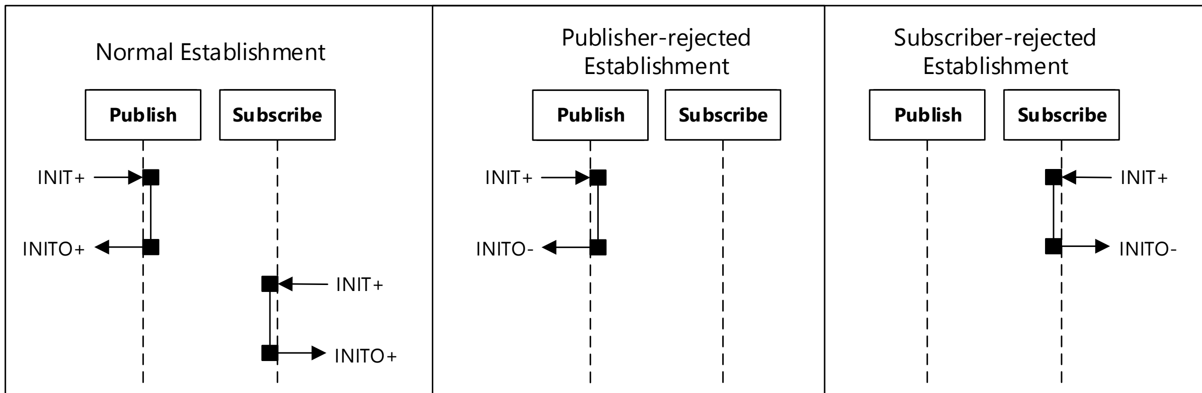


Figure 4.7: Sequence diagram depicting the three scenarios for connection establishment between the publisher and the subscriber

The publish-subscribe block has three separate phases of execution: *connection establishment*, *data transfer*, and *disconnection*. Firstly, Figure 4.7 shows the sequence diagram depicting the three scenarios for *connection establishment*, namely: *normal establishment*, *publisher-rejected establishment*, and *subscriber-rejected establishment*. For both the publisher and subscriber blocks, the INIT input event is used to establish or disconnect a communication depending on the Boolean value of QI. As normative in the IEC 61499 standard, the suffix "+" is used in conjunction with an input/output event name to indicate that the value of the QI/QO input/output is true at the occurrence of the associated event, while the suffix "–" is used to indicate otherwise. The lines connecting events indicate a cause and effect, where the event on top is emitted prior (cause) to events at the bottom (effect).

Normal establishment occurs when the publisher and subscriber function blocks are successfully initialised. At this point, a connection is established. In the normal establishment scenario, both publisher and subscriber set QI and QO to 'true', respectively, to indicate successful initialisation. Publisher-rejected establishment occurs when the publisher function block tries to initialise, but a connection to the corresponding subscriber block was not established. Subscriber-rejected establishment occurs when the subscriber function block tries to initialise, but a connection to the corresponding publisher block was not established. In either rejection scenario, the QO value will be set to 'false' to indicate the failure to initialise (these scenarios are depicted in Figure 4.7).

Secondly, Figure 4.8 shows the sequence diagram depicting *normal data transfer*. During normal data transfer, the publisher block receives the REQ event and sends the data at the SD_1 port to the subscriber block. Once the subscriber block receives the data, it emits the IND event to indicate that data has been received and sends that data to other function blocks in the application though the RD_1 port. The publisher block

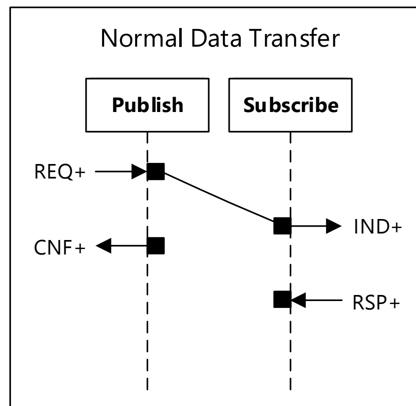


Figure 4.8: Sequence diagram depicting normal unidirectional data transfer between the publisher and the subscriber

also emits the CNF event to indicate a successful transfer. When the data is processed by the function block application, the function block application sends the RSP event to the subscriber block.

Lastly, Figure 4.9 shows the sequence diagram depicting the *disconnection phase*. The disconnection can be initiated by either the publisher or the subscriber. The disconnection is initiated when either the publisher or subscriber block receives an INIT event and a false value on the QI port. If the disconnection is initiated by the publisher block, a signal is sent to the subscriber block to disconnect the connection. Once termination is successful, the publisher and subscriber blocks emit their respective INITO events and set the QO ports to false. If the disconnection is initiated by the subscriber block, the connection is disconnected without notifying the publisher block because of the unidirectional nature of the publisher-subscriber pair.

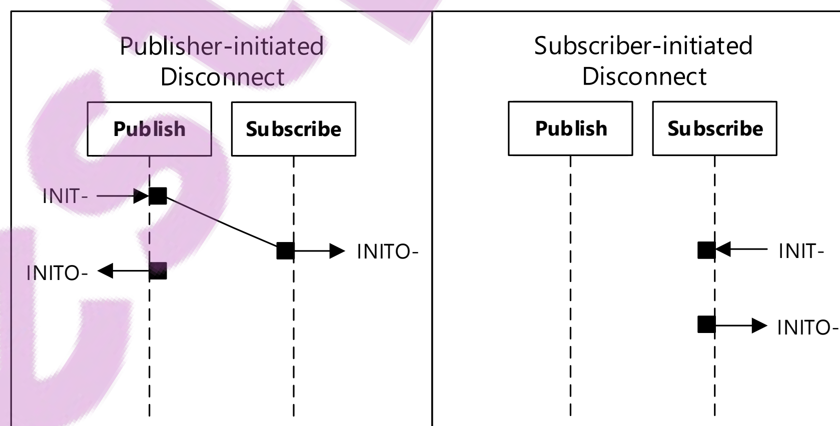


Figure 4.9: Sequence diagram depicting publisher-initiated and subscriber-initiated disconnections

Bestpfe.com

4.5 System, Devices, and Resources

Device and *Resource* models are defined in IEC 61499 to reduce the gap between the physical components of a system such as microcontrollers, PLCs, sensors, and actuators, and the logical components of the automation logic, i.e., the various function blocks types. This method of modelling automation systems bears resemblance to the object-oriented paradigm, where the *System* model sits at the highest level of the object definition.

4.5.1 Device model

IEC 61499 defines a device as, "*an independent physical entity capable of performing one or more specified functions in a particular context and delimited by its interfaces.*" A device model, therefore, is the functional definition of a physical component in a larger distributed system. Each device may contain some inherent behaviour owing to its physical sub-components, such as timers and reset interrupts, as well as some mapped behaviours such as an automation task modelled using a function block network. In order to manage the complexity of devices, the concept of resource models is used. A device may contain zero or more resources encapsulating independent function or tasks. Figure 4.1 shows two devices (④, ⑤), with one resource each (⑥, ⑦ respectively).

4.5.2 Resource model

IEC 61499 defines a resource as, "*a functional unit having independent control of its operation, and which provides various services to applications including scheduling and execution of algorithms.*" A resource model, therefore, is the functional definition of an independent task executing on a device. Such tasks are segregated from each other in such a way that a particular system resource (e.g., a sensor or an actuator) may only be accessed and operated upon by a single resource. In the absence of shared variables, resources and devices communicate using communication function blocks in order to perform the coordination between tasks.

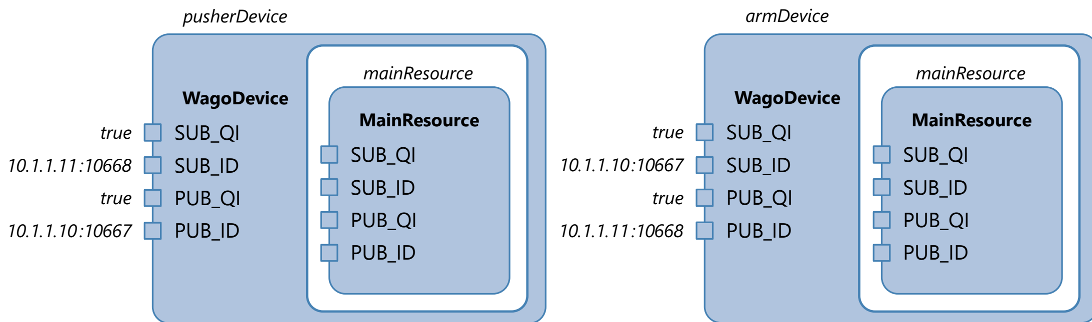


Figure 4.10: IEC 61499 system containing two devices with nested resources

4.5.3 System model

The system model is used to represent an overall automation system and is defined as, "*a collection of devices interconnected and communicating with each other by means of a communication network consisting of segments and links*." Each device is capable of performing a set of independent tasks that coordinate by means of a communication network and, thus, constitute a distributed system. Figure 4.10 shows the system model for the *Distributed System* configured with two devices containing their respective resources.

A system consists of two separations, namely an *application* model, and a *device and resource configuration*. The former describes the actual automation logic, whereas the latter implements its execution. The application model is primarily an FBN that consists of instances of various types of function blocks, as shown in Figure 4.11. It is the top-most level of the hierarchy of FBNs and implements the automation logic of the overall system.

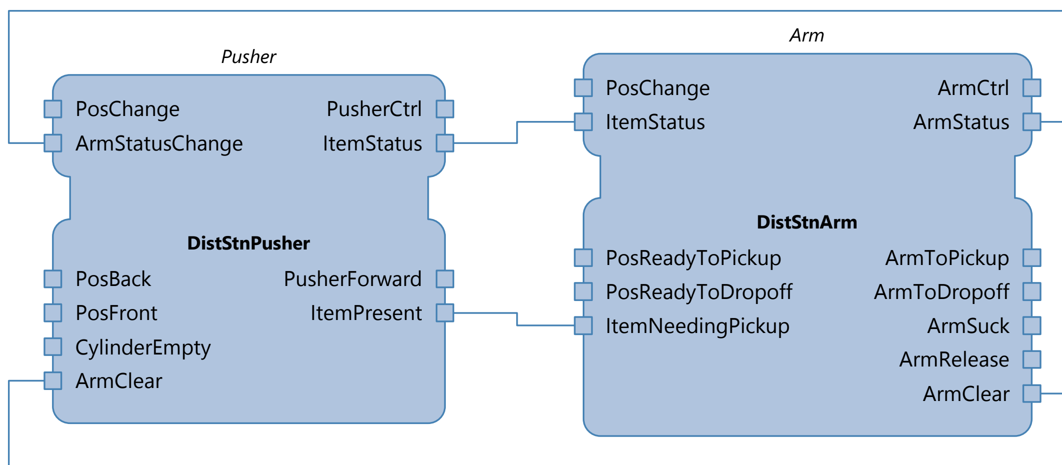


Figure 4.11: Application model of the *Distribution Station* system

BEST PFE.COM
List of research project topics and materials

This holistic view of an application model provides an unobstructed view of the overall system's behaviour, but must be partitioned in order to be implemented in a distributed fashion. For the said purpose, a subset of the application can be mapped onto a device containing zero or more resources in the configuration to implement localised sub-system/task, for example, the function block DistStnArm is mapped to the armDevice in the system implementation. Similarly, the function block DistStnPusher is mapped to the pusherDevice for the purpose of implementing the *Distribution Station* as a distributed system. This partitioning of an application can raise communication dependencies, for example, cross-device or cross-resource wire connections must be routed through a communication network. However, such dependencies can be detected automatically and resolved in a supervised manner.

4.5.4 Implementation of the Distribution Station

The *Distribution Station* is implemented using the standard IEC 61499 constructs and is shown in Figure 4.12. All function blocks in this implementation can be traced back to the high-level diagram shown in Figure 4.1. This implementation is performed by instantiating a device model twice in a system model. Each device, in turn, contains an instance of a resource model, thereby creating a total of two nested instances of the said resource model. On the other hand, a device-independent implementation of the automation logic is created as two BFBs, namely DisStnPusher and DistStnArm, which control the *Arm* and *Pusher*, respectively. The automation task is then assigned to the resource instances by means of a *mapping* process.

The mapping process enables the device specific behaviours to be decoupled from the functional behaviours, i.e., automation tasks are not made part of the resource definition. This approach isolates the automation logic from the physical model and allows reuse of device and resource models in a system, as well as easy reconfigurability. Figure 4.12 highlights the use of this concept, where:

- numbered labels match corresponding blocks in Figure 4.1;
- function blocks with a thick solid border depict the automation logic mapped to a resource;
- function blocks with a thick dashed border depict the *device drivers* that are defined in a device FBType; and
- the remaining function blocks are part of the resource FBType.

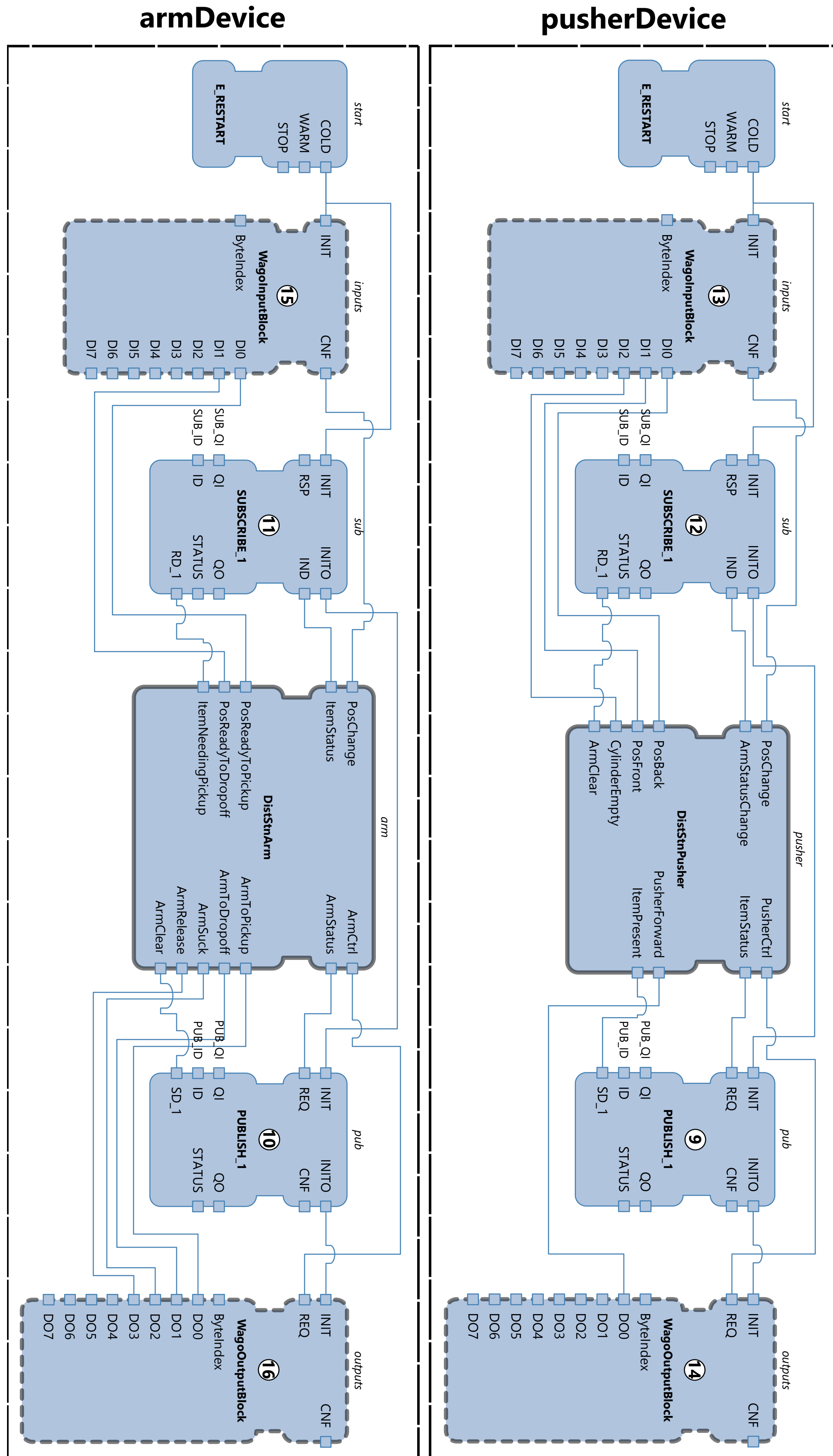


Figure 4.12: A distributed IEC 61499 implementation of *Distribution Station*

An aggregated FBN is constructed to allow these function blocks to interact seamlessly and to execute in the scope of a device.

Compilation of this implementation creates two binary executables, one for each device. The device specific IO blocks (⑬-⑯ in Figure 4.12) allow interaction with external devices, such as, *Arm* and *Pusher* for the purpose of actuation and sensing. The communication blocks (⑨-⑫ in Figure 4.12) allow the device to coordinate using the Ethernet network, thus, making the system ready to perform the desired distributed automation task.

4.6 Execution models for Function Blocks

We have so far covered the syntactic aspects of the standard. The semantic aspects deal with the mechanisms by which a given FB based design (such as the one shown in Figure 4.12) can be expected to provide the desired outcome. This section provides an overview of the semantic concepts, which dictate the execution behaviour of function blocks.

In order to interpret IEC 61499 models as behaviours, the structural definitions must be paired with semantics, i.e., rules defining how to execute ECCs and their interconnections. The execution semantics can be realised via a run-time environment (which is analogous to a scheduler in an OS kernel) that takes on the responsibility of scheduling events, function blocks, and the data transfer between them. An alternate approach embeds these semantics within the generated code, thus, making the execution independent of any run-time environment. Benefit of the latter approach are the higher performance due to lower demand for computation power, and the smaller memory footprint.

Scheduling of function blocks, i.e., when to execute a function block, can be performed in two different ways. The *event-triggered* scheduling executes a function block when a corresponding input event occurs. The subsequent execution may generate other events, which may, in turn, trigger the execution of other function blocks. In the presence of multiple events, usually a queuing mechanism is used to service events one at a time. Thus, the behaviour of the overall system depends on the event-queue and its management.

IEC 61499 run-time environments that adopt this execution approach are FBRT [98], FORTE [99], and FUBER [102]. The alternate approach for scheduling function blocks

is the cyclic execution model, which resembles the PLC scan cycle. In this approach, each function block in the given network executes once per cycle, sampling its inputs and producing outputs. Any event generated can be processed in the same cycle by other function blocks that are further down in the per-cycle order of execution. To achieve a robust execution order, a topological sort can be performed to schedule an event producer before an event consumer. Consequently, any cycle that is discovered needs to be resolved by using unit-length buffers to delay their processing by one cycle. *ISaGRAF* [97] and synchronous execution semantics [15] use the cyclic scan approach. However, *ISaGRAF* relies on a run-time environment, whereas the synchronous execution semantics relies on static scheduling. In the following subsections, a brief discussion about existing execution approaches is presented.

4.6.1 FBRT

The *Function Block Run-Time* (FBRT) [98] is a Java-based run-time environment. The *Function Block Development Kit* (FBDK) generates Java code for FBRT. The chosen execution scheme is the event-driven model, where the occurrence of an event in the system is mapped to a direct function call on the function block instance. This results in a *depth-first* model for event propagation. The advantage of this execution model is the simplicity of the generated code. However, it has several disadvantages. The generated code requires a Java virtual machine to run on the target device, which may not be suitable for resource-constrained implementations. Secondly, the depth-first event propagation may potentially require very deep memory stacks on the target device if an invocation results in a long chain of cascading events [103]. This is especially so in cases where event loop-backs are present in the function block network.

4.6.2 FORTE

FORTE is the companion run-time environment for the 4DIAC-IDE [99] function block editor and code generator. Unlike the approach used in FBRT [98], FORTE adopts a *breadth-first* event propagation scheme. All external and generated events are queued in a FIFO event buffer and are consumed by the respective function blocks in a sequential manner. This significantly reduces the depth of call stack for long event chains. A reported disadvantage [104] of this technique is the slow and bulky generated code due to multi-threading, which may not suitable for resource-constrained embedded systems.

4.6.3 FUBER

FUBER [102] is an interpreter for IEC 61499 designs that adopts a breadth-first event propagation approach, similar to FORTE [99]. However, unlike the global event buffer of FORTE, FUBER has chosen to create a local FIFO event-buffer for each function block instance. When an event has to be notified to a function block instance, a new event is queued in its event-buffer, and the function block instance is queued in a scheduler queue. The scheduler then executes the queued instances in a FIFO manner [104] to consume the events. This approach has similar advantages to the approach of FORTE.

4.6.4 ISaGRAF

ISaGRAF [97] adopts the PLC scan cycle execution model [96], where each function block is mapped to a separate PLC program. During a scan cycle, a function block is executed if at least one associated event is present. In this manner, function blocks are executed in a round-robin fashion with a specific order. The events produced during execution are immediately available to consumer function blocks. If the consumer block is scheduled after the producer block, the event can be consumed in the same scan cycle; otherwise, the event will be consumed in the next scan cycle. In this model of execution, the behaviour of the system is dependent on the order in which the function blocks have been scheduled.

4.6.5 Synchronous Execution

This approach does not require a run-time environment on the target device and has a higher performance than other function block execution models [100]. Under these semantics, function blocks execute in a cyclic manner in logic time units called *ticks*. In each tick, every function blocks executes one cycle of execution by sampling inputs, taking a transition, and performing state-entry actions. By the virtue of having a fixed order in transitions selection and invocation of algorithms, this execution is deterministic. Furthermore, all updates to events and variable values are delayed by one tick, thus, the order of execution of individual function blocks in a given tick does not matter. This renders this execution model as deterministic and dead-lock free [15]. Therefore, it not only suits various application domains but also supports a wide range of devices with varying computation power and memory capacity. Due to its suitability for safety critical applications, this execution model is adopted for the proposed model-based safety

assessment approach presented in this thesis in Chapters 5, 6.

4.7 Discussion

This chapter presented the basic concepts about structure and semantics of IEC 61499. We started with an overview of how distributed systems are designed, and how different types of function blocks fit in this design. The concept of system, devices, and resources facilitate an object-oriented approach for designing the overall system. Basic and composite function blocks are primarily used to model the behaviour of the system, whereas service interface function blocks are used to implement low-level functions, such as device drivers or communication interfaces. We further discussed how theses models are interpreted and executed using run-time environments and their respective mechanisms.

5

Converting Function Blocks to Prism

Previous chapters covered the background required for this thesis. Specifically, in Chapter 2 we presented model checking and temporal logic. In Chapter 4, we presented a brief summary of the IEC 61499 standard [11], which described the various types and structure of function blocks including the *basic* and *composite* function blocks. Later in that chapter, we presented various existing semantics of their execution including the *synchronous execution semantics* [105], which performs a cyclic execution of a given function block network in logical time units called *ticks*. In this chapter, we present a formal structure for IEC 61499 adopted from [106]. This formalism is used for devising a set of sound transformation rules that convert function blocks into Markov decision processes [58] in a semantics-preserving manner. We chose the Prism language [58] for representing the generated Markov decision processes such that a given function block network (FBN) is automatically converted into an equivalent Prism model.

Prism [59] is a probabilistic model checker for quantitative analysis of stochastic systems. It supports the Prism language [58] that is used for system and model specification in the form of variables and probabilistic commands. These models are then subjected to automated probabilistic verification of linear temporal logic (LTL) properties [107]. The correctness of this transformation allows verification of the generated Prism model such

Bestpfe.com

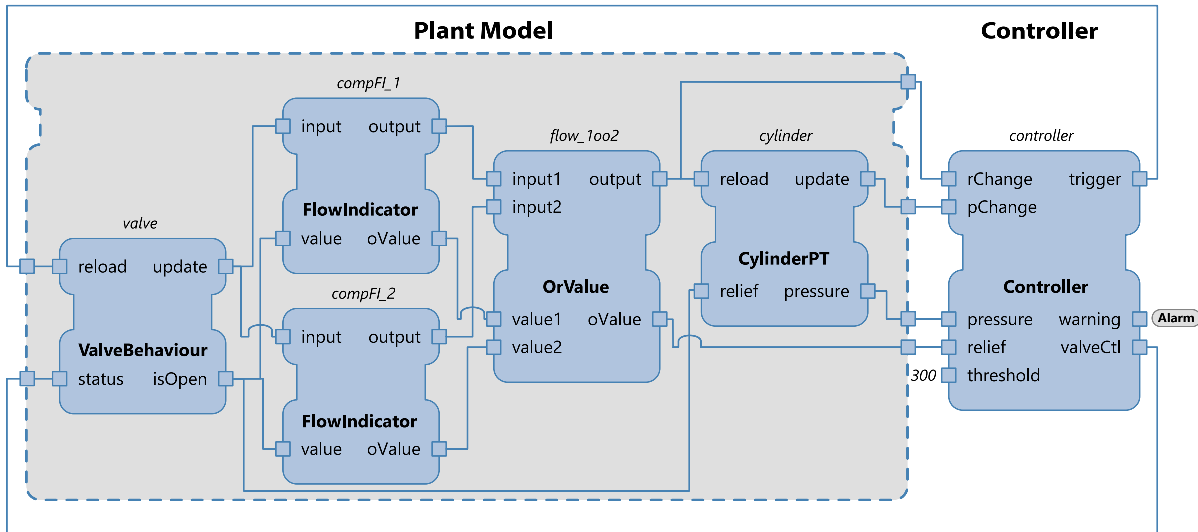


Figure 5.1: IEC 61499 implementation of the boiler control system

that the result of analysis is also sound for the given FBN.

We begin by presenting the definition of IEC 61499 function blocks based on [106], and their execution semantics based on the synchronous execution semantics [105]. Later, we present the formal structure of Prism models along with their respective execution semantics. Having presented the two formalisms, we then present the set of rules to translate IEC 61499 function block networks into equivalent Prism models.

5.1 The Boiler Control System

In this section we revisit the boiler control system presented earlier in Section 2.1 and use it for illustration purposes. The function block network implementation of the boiler system presented in Figure 5.1 uses the *model-view-controller* (MVC) design pattern [10] consisting of a *plant-model* that comprises models of the valve, pressure transmitter and flow indicators, as well as the *controller*. Optionally, a *view* can be connected to the network for visual monitoring and simulation, however it is omitted as it is not necessary for the discussion at hand. The controller reads inputs from the pressure transmitter and based on a threshold value and decides whether to open a control valve for pressure relief. In order to avoid the pressure over-run hazard, the controller must also monitor the flow indicators (relief input variable) and sound an alarm (warning output variable) if the flow indicators do not report expected inputs.

5.2 Formalisation

IEC 61499 provides two types of *function blocks* (FBs) for developing complex control software. *Basic FBs* are the smallest units of execution, whereas *Composite FBs* encapsulate networks of function blocks. All FBs have *interfaces* that expose their respective inputs and outputs as defined below.

Definition 5.2.1 (Function Block Interface)**.** A function block interface $\mathcal{I}$ is a tuple such that, $\mathcal{I} = \langle E_I^{\mathcal{I}}, V_I^{\mathcal{I}}, E_O^{\mathcal{I}}, V_O^{\mathcal{I}} \rangle$ where, $E_I^{\mathcal{I}}, V_I^{\mathcal{I}}, E_O^{\mathcal{I}}$ and $V_O^{\mathcal{I}}$ are finite sets of input events, input variables, output events and output variables respectively. Figure 5.1 shows interface of several function blocks, for example $\mathcal{I}$ corresponding to the Controller FB has the following events and variables.

- $E_I^{\mathcal{I}} = \{\mathsf{rChange},\ \mathsf{pChange}\}$
- $V_I^{\mathcal{I}} = \{\mathsf{pressure},\ \mathsf{relief},\ \mathsf{threshold}\}$
- $E_O^{\mathcal{I}} = \{\mathsf{trigger}\}$
- $V_O^{\mathcal{I}} = \{\mathsf{warning},\ \mathsf{valveCtl}\}$

5.2.1 Basic Function Blocks

Basic function blocks (BFBs) implement their behaviour using Moore-type finite state machines called execution control charts (ECCs). The ECC of a BFB accepts inputs and emits outputs through the encapsulating FB interface. The accepted input variable values can be processed using textual blocks of code called *algorithms*, which may be executed upon entering a state and new values may be generated for the output variables. For the purpose of internal computations and hidden data, *local variables* may be used. Algorithms and local variables are formally defined as the local declaration of a BFB. These definitions are adopted from [106] and are presented as follows.

Definition 5.2.2 (Local Declaration)**.** Local declaration of a BFB over an interface $\mathcal{I}$ is a tuple $L^{\mathcal{I}} = \langle V_L^{\mathcal{I}}, A_L^{\mathcal{I}} \rangle$ where, $V_L^{\mathcal{I}}$ is the set of internal variables and $A_L^{\mathcal{I}}$ is the set of algorithms operating over $V_L^{\mathcal{I}}$, $V_I^{\mathcal{I}}$ and $V_O^{\mathcal{I}}$. An algorithm in a local declaration is a finite sequence of *statements* ($\rho_0,\ \rho_1,\ \ldots,\ \rho_n$) that operate over available variables $V_L^{\mathcal{I}}, V_I^{\mathcal{I}}, V_O^{\mathcal{I}}$ i.e., local variables as well as input and output variables.

BESTPFE.COM List of research project topics and materials

Algorithms in IEC 61499 are implemented in external host languages, such as the C or Java language. In our previous work [108], we chose C language because of its wide applicability on realtime systems. In the scope of the current work, the proposed methodology relies on the Prism model checker for probabilistic verification, which puts some restrictions on the use of variables and possible operations on the values. Primarily, these limitations attempt to avoid state-space explosion problem during the verification process. The limitations are listed as following.

- A variable can either be a `Boolean` type or `Integer` type. These values map to IEC 61499 types `BOOL`, and any of the integer types e.g., `INT`, `DINT`, `USINT`, or `BYTE`.
- Integer type variables must define an initial, a minimum and a maximum value. The execution of Prism model begins with initial variable values and during all possible executions, the value of the variable must not exceed the said range.
- Arithmetic operations are allowed on integer type variables except for division. This restriction is often imposed by model checkers to simplify the verification process.
- Boolean expressions are used as location guards and only assignment and conditional assignment statements are allowed in the command updates. Where locations and commands are akin to states and transitions.

Because of to these limitations, the proposed methodology cannot support the complete C language. Specifically, we restrict the BFB algorithms to contain only increment/-decrement and assignment statements that are built using Boolean/numeric literals and expressions. This simplification also enforces that all branching statements and loops shall be modelled using transitions instead of being modelled inside algorithms. Furthermore, certain read/write limitations are also imposed on variables i.e., multiple write operations are not permitted on a given variable, and due to the delayed-updates of the synchronous execution semantics, these updated values are not available for read operations in an algorithm. The grammar for the restricted BFB algorithms is given in Backus-Naur form in Table 5.1.

Based on the definition of the FB interface and local declaration, we can now proceed to define the structure of a BFB as follows.

Definition 5.2.3 (Basic Function Block)**.** A basic function block is a tuple $\mathtt{BFB} = \langle \mathcal{I}, L^{\mathcal{I}}, \mathtt{ECC}^{\mathcal{I},L^{\mathcal{I}}} \rangle$ where, $L^{\mathcal{I}} = \langle V_L^{\mathcal{I}}, A_L^{\mathcal{I}} \rangle$ is a local declaration over interface $\mathcal{I} = \langle E_I^{\mathcal{I}}, V_I^{\mathcal{I}}, E_O^{\mathcal{I}}, V_O^{\mathcal{I}} \rangle$ and, $\mathtt{ECC}^{\mathcal{I},L^{\mathcal{I}}} = \langle Q, q_0, X, T \rangle$ is the execution control chart where:

- Q is a finite set of *execution control states* (ECStates), with $q_0 \in Q$ as the initial state.

- $X : Q \rightarrow 2^{(A_L^{\mathcal{I}} \cup E_O^{\mathcal{I}})}$ is the action function that assigns a finite set of algorithms and output events to a given state $q \in Q$.

- $T : Q \rightarrow 2^{(E_I^{\mathcal{I}} \,\cup\, \{\texttt{true}\}) \,\times\, \mathcal{B}(\hat{V}) \,\times\, Q}$ is the transition function where $\hat{V} = (V_I^{\mathcal{I}} \cup V_O^{\mathcal{I}} \cup V_L^{\mathcal{I}})$ is the set of input, output and internal variables. Here, the notation $(E_I^{\mathcal{I}} \cup \{\texttt{true}\})$ denotes set of all input events including the always present `true` event, and $\mathcal{B}(\hat{V})$ is the set of all Boolean expressions over all variables. Furthermore, the inputs are read from the previous tick, which results in the flow of events and variables between function blocks in a unit-delayed fashion as proposed by the synchronous execution semantics [105]. The notation $t = (q, e, b, q')$ represents individual transitions $t \in T(q)$ where, q and q' are the predecessor and successor ECStates respectively, $e \in E_I^{\mathcal{I}}$ is an input event, and $b \in \mathcal{B}(\hat{V})$ is a Boolean expression. For every $q \in Q$, $T(q)$ is always an ordered set, i.e., for any two elements $t_1,\ t_2 \in T(q)$ we have $(t_1 > t_2) \vee (t_1 < t_2)$. We capture this order of transitions graphically using the notation `<n>`, where is `n` is the index of an element in the order set.

Figure 5.2 presents the execution control chart of the Controller function block. Using this example for illustration, we have the following elements.

- $A_L^{\mathcal{I}} = \{$incCounter, openValve, resetCounter, setAlarm, closeValve$\}$ is the set of algorithms

- $Q = \{$DO_OPEN, OPENED, DO_CLOSE, CLOSED, ALARM$\}$ is the set of ECStates

- $q_0 =$ CLOSED is the initial ECState

- X is the action function that maps states to respective action sets. For example, it maps the DO_CLOSE state to $\{$closeValve, incCounter, trigger$\}$. Where closeValve and incCounter are the algorithms to be invoked, and trigger is the output event to be emitted.

Table 5.1: BNF for BFB algorithms

```
     <algorithm>  ::= <statement-list>
<statement-list>  ::= <statement-list> <statement>
     <statement>  ::= var <a_op> <expr>
          <expr>  ::= val | var | <expr> <b_op> <expr> | ‘(’ <expr> ‘)’
                      | <bexpr> | <bexpr> ‘?’ <expr> ‘:’ <expr>          Expression
         <bexpr>  ::= <expr> <c_op> <expr> | <bexpr> <l_op> <bexpr>
                      | ‘!’ <bexpr>                                      Boolean expression
          <a_op>  ::= ‘=’ | ‘+=’ | ‘-=’ | ‘*=’                           Arithmetic operators
          <b_op>  ::= ‘+’ | ‘-’ | ‘*’                                    Binary operators
          <c_op>  ::= ‘<’ | ‘>’ | ‘<=’ | ‘>=’ | ‘==’ | ‘!=’              Comparison operators
          <l_op>  ::= ‘&&’ | ‘||’                                        Logical operators
```

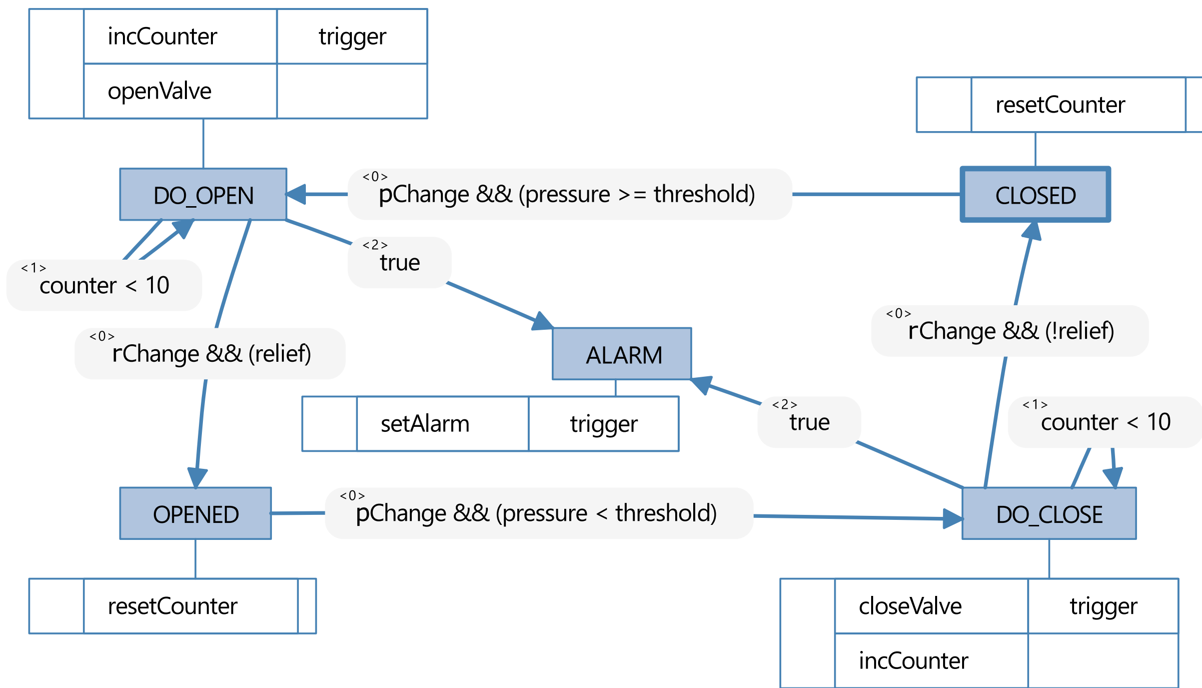


Figure 5.2: Execution Control Chart of the Controller function block

- T is the transition function that maps states to ordered set of respective egress transitions. For example it maps the DO_CLOSE state to (t_0, t_1, t_2) such that:

$$t_0 = (\mathsf{DO_CLOSE}, \mathsf{rChange}, \mathsf{!relief}, \mathsf{CLOSED})$$
$$t_1 = (\mathsf{DO_CLOSE}, \varnothing, \mathsf{counter} < 10, \mathsf{DO_CLOSE})$$
$$t_2 = (\mathsf{DO_CLOSE}, \texttt{true}, \varnothing, \mathsf{ALARM})$$

5.2.2 Synchronous Execution of BFBs

Execution of a BFB using the synchronous execution semantics [105] is performed in a step-by-step manner, where each step executes in logical time called a *tick*. During each tick, a given BFB reads inputs, updates its current state and execute state-entry actions as depicted in Algorithm 5.1. Here, lines $2-3$ ensure that execution begins from the initial state q_0. After the initialisation tick, line 5 loads inputs from the encapsulating function block interface, such that the values are loaded from the previous tick as indicated by the keyword pre. Lines $7-14$ iterate over a sorted set of all transitions from the current state, however, this iteration stops as soon as an enabled transition is located and processed (line 12). The result of this pre-emptive iteration over an ordered set induces a *priority* such that, a higher order transition has a higher priority of getting selected. This processing of the enabled transition entails three steps. Firstly, the current state is updated to point

Algorithm 5.1: Execution of a single *tick* of a Basic Function Block

```
1  Function BFBTick(BFB):
2      if cState = null then
           /* Initialisation tick does not read inputs                              */
3          cState ← BFB.q0
4      else
           /* Load input events and variables from previous tick                    */
5          loadInput(pre E_I^I, pre V_I^I);
6      end
       /* Iterate over the ordered set of transitions                               */
7      for t ∈ getTransitions(cState) do
           /* The first enabled transition will be of the highest priority          */
8          if isEnabled(t) then
9              cState ← getSuccessor(t);
               /* Execute algorithms of the successor state                         */
10             executeAlgos (cState);
               /* Emit output events and variables                                  */
11             emitOutput(E_O^I, V_O^I);
12             return;
13         end
14     end
15     return;
```

towards the successor state of the transition (line 9). Secondly, algorithms in the state-entry actions of the new current state are executed (line 10). Thirdly, any output events in the state-entry actions are emitted through the function block interface (line 11). This emission also updates the value of the output variables reflecting any change that may have been caused by the algorithm executions. Finally, the loop is aborted to make sure that at most one transition is taken (line 12). If no transitions could be enabled, the function returns normally without updating the current state, executing any algorithms, or emitting any events (line 15). This concludes one tick process of a BFB.

5.2.3 Composite Function Blocks

Composite function blocks (CFB), like BFBs, also contain an FB interface. However, unline BFBs, their behaviour is implemented by an encapsulated network of function blocks. This function block network (FBN) may contain several instances of various function blocks including both BFBs as well as CFB. The exposed IOs of these instances are connected through virtual *wire connections* indicating the flow of events and variables from outputs to inputs of the respective blocks. Figure 5.1 presents the FBN of the boiler control system, which contains several function block instances including two instances (compFI_1 and compFI_2) of the FlowIndicator function block type. We formally define FBN

as follows.

Definition 5.2.4 (Function Block Network)**.** A function block network is a tuple, $\texttt{FBNetwork} = \langle \texttt{FBs}, C_{\texttt{e}}, C_{\texttt{v}} \rangle$ where,

- $\texttt{FBs} = \{\texttt{FB}_1, \texttt{FB}_2, \cdots, \texttt{FB}_n\}$ is a finite set of function block instances. A function block instance is a pair $\texttt{FB}_i = \langle name_i, \texttt{FBT}_i \rangle$ where $\texttt{FBT}_i$ is an FB type with an interface $\mathcal{I}_i = \langle E_I^{\mathcal{I}_i}, V_I^{\mathcal{I}_i}, E_O^{\mathcal{I}_i}, V_O^{\mathcal{I}_i} \rangle$ and $name_i$ is a unique identifier within the scope of `FBNetwork` and is called instance name.
- $C_{\texttt{e}} \subseteq (\bigcup_{i=1}^{n} \texttt{FB}_i.E_O^{\mathcal{I}_i}) \times (\bigcup_{j=1}^{n} \texttt{FB}_j.E_I^{\mathcal{I}_j})$ is the set of event connections between the instances of the network.
- $C_{\texttt{v}} \subseteq (\bigcup_{i=1}^{n} \texttt{FB}_i.V_O^{\mathcal{I}_i}) \times (\bigcup_{j=1}^{n} \texttt{FB}_j.V_I^{\mathcal{I}_j})$ is the set of variable connections between the instances of the network. Variable connections are restricted such that for any two distinct variable connections $C_{v1} = (src_1, dest_1), C_{v2} = (src_2, dest_2) \in C_{\texttt{v}}$ $dest_1 \neq dest_2$ i.e., an input variable (destination) cannot read from multiple sources. Furthermore, any event or variable generated by an instance $\texttt{FB}_i$ is only available for reading in the next *tick*. This imposes a unit-delayed communication between FB instances, thus making the order of `FBs` irrelevant as the order of execution of FB instances has no effect on their behaviour. Such unit-delayed composition is often used in synchronous languages such as in SL [109] to ensure causal composition.

The boiler system FBN in Fig. 5.1 contains a set of function block instances and a set of wire connections i.e., the following.

- $\texttt{FBs} = \{$valve, compFI_1, compFI_2, flow_1oo2, cylinder, controller$\}$ is the set of all function block instances
- $C_{\texttt{e}}$ is a set of tuples representing all event-connections for example, $\langle$cylinder::update, controller::pChange$\rangle$ and $\langle$comFI_1::output, flow_1oo2::input1$\rangle$ are two event connections from Figure 5.1
- $C_{\texttt{v}}$ is a set of tuples representing all variable-connections for example, $\langle$cylinder::pressure, controller::pressure$\rangle$ and $\langle$comFI_1::oValue, flow_1oo2::value1$\rangle$ are two variable connections from Figure 5.1

A given FBN can be encapsulated by an FB interface, thus forming a composite function block (CFB). The inputs and outputs of the encapsulated FBN are exposed

through the interface of the CFB through a *proxy* i.e., a variable or event port on the interface is connected on behalf of an encapsulated variable or event. This encapsulation results in a provision for hierarchy in the IEC 61499 system. Thus, a composite function block may become part of a higher level FBN by exposing its encapsulated IO. Figure 5.3 shows an instance of a BFB named bfb1 encapsulated inside a composite function block cfb. The inputs and outputs of the encapsulated block are exposed through proxy connections and connected to in a higher level FBN e.g., bfb1::oVar is connected to bfb2::iVar through a proxy output cfb::oVar.

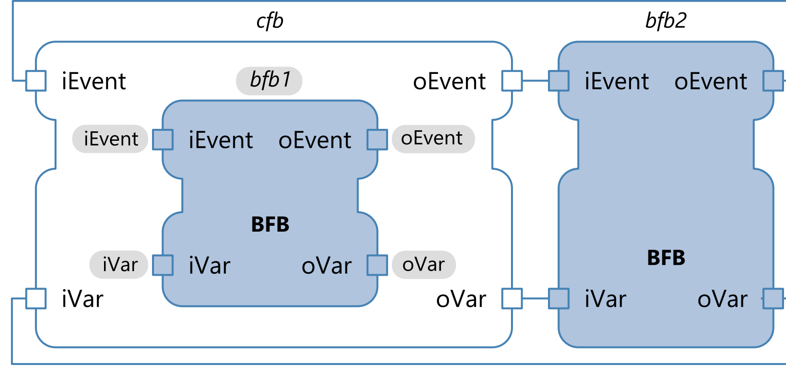


Figure 5.3: A composite function block

Definition 5.2.5 (Composite Function Block)**.** A composite function block is a tuple, $\texttt{CFB} = \langle \mathcal{I}, \texttt{FBNetwork}, P, \rangle$ where, $\mathcal{I} = \langle E_I^{\mathcal{I}}, V_I^{\mathcal{I}}, E_O^{\mathcal{I}}, V_O^{\mathcal{I}} \rangle$ is the interface that encapsulates the function block network $\texttt{FBNetwork} = \langle \texttt{FBs}, C_{\texttt{e}}, C_{\texttt{v}} \rangle$. Also, a set of proxy connections exist between the interface and the encapsulated FBN, namely $P = P_{E_I^{\mathcal{I}}} \cup P_{E_O^{\mathcal{I}}} \cup P_{V_I^{\mathcal{I}}} \cup P_{V_O^{\mathcal{I}}}$ such that,

- $P_{E_I^{\mathcal{I}}} : E_I^{\mathcal{I}} \rightarrow 2^{\left(\bigcup_{i=1}^{n} \texttt{FB}_i.E_I^{\mathcal{I}_i} \right)}$ is the encapsulation function for input events. The tuples in $P_{E_I^{\mathcal{I}}}(e_i)$ indicate proxy connections between the input event e_i on the CFB interface and a subset of input events of the encapsulated FBs.

- $P_{E_O^{\mathcal{I}}} : E_O^{\mathcal{I}} \rightarrow 2^{\left(\bigcup_{i=1}^{n} \texttt{FB}_i.E_O^{\mathcal{I}_i} \right)}$ is the encapsulation function for output events. The tuples in $P_{E_O^{\mathcal{I}}}(e_o)$ indicate proxy connections between a subset of output events of the encapsulated FBs and the output event e_o of the CFB interface.

- $P_{V_I^{\mathcal{I}}} : V_I^{\mathcal{I}} \rightarrow 2^{\left(\bigcup_{i=1}^{n} \texttt{FB}_i.V_I^{\mathcal{I}_i} \right)}$ is the encapsulation function for input variables. The tuples in $P_{V_I^{\mathcal{I}}}(v_i)$ indicate proxy connections between the input variable v_i on the CFB interface and the input variables of the encapsulated FBs. Furthermore, all elements in $P_{V_I^{\mathcal{I}}}(v_i)$ must be *type-compatible* e.g., for any input variable proxy connection $(v_i, v_i') \in P_{V_I^{\mathcal{I}}}(v_i) \implies \mathsf{typeOf}(v_i)\mathsf{=}\mathsf{typeOf}(v_i')$.

- $P_{V_O^{\mathcal{I}}} : V_O^{\mathcal{I}} \rightarrow (\bigcup_{i=1}^{n} \texttt{FB}_i.V_O^{\mathcal{I}_i})$ is the encapsulation function for output variables. For a given output variable $v_o \in V_O^{\mathcal{I}}$ of the interface of the encapsulating CFB, $P_{V_O^{\mathcal{I}}}(v_o)$ only has a single element i.e., the variable v_o can expose a single encapsulated output variable. Furthermore, any element in $P_{V_O^{\mathcal{I}}}(v_o)$ must be *type-compatible* e.g., for any output variable proxy connection $(v_o, v_o') \in P_{V_O^{\mathcal{I}}}(v_o) \implies \mathsf{typeOf}(v_o)=\mathsf{typeOf}(v_o')$.

As evident from the above definition, the event encapsulation functions $P_{E_I^{\mathcal{I}}}$ and $P_{E_O^{\mathcal{I}}}$ allow one-to-many and many-to-one connections between the input and output event encapsulation respectively. Similarly, the input variable encapsulation function $P_{V_I^{\mathcal{I}}}$ also allows a one-to-many connection between input variables of the interface and the encapsulated FBN. However, the output variable encapsulation function P_{ovx} does not have a multiplicity i.e., it only allows a one-to-one connection between the interface output variables and the encapsulated output variables. This added restriction ensures that no value conflicts arise during the emission on output variables. On the other hand, multiplicity of output event encapsulation is handled through a disjunction i.e., if any of the encapsulated output event is *present*, the respective interface output event is also set to *present*.

5.2.4 Synchronous Execution of CFBs

Under the *synchronous execution semantics* [105] execution of a given function block network `FBNetwork` is performed in logical time intervals called *ticks*. During each tick, each FB in the given FBN is executed as described in Algorithm 5.2. Similar to the execution to BFBs, CFBs load inputs from their encapsulating interface from the previous tick (see line 2). Lines $3-9$ iterate over all instances in the function block network and execute a corresponding tick one by one. Here, if the instance is of a BFB type, the execution is performed using Algorithm 5.1 (line 5) whereas, if the instance is of a CFB type, recursion of Algorithm 5.2 is performed (line 7). Please note this algorithm results in a recursion such that, the depth of recursion tree is same as the levels of hierarchy in the given CFB.

Observation 5.2.1. *Without loss of generality we can assume that FBNs only contain instances of BFBs [110].*

Algorithm 5.2: Execution of a single *tick* of a Composite Function Block

```
1  Function CFBTick(CFB):
       /* Load input events and variables from interface                         */
2      loadInput(pre E_I^𝓘, pre V_I^𝓘);
       /* Iterate over the set of encapsulated BFBs and CFBs                     */
3      for FB ∈ FBN do
           /* Execute one tick on each FB                                        */
4          if FB is Basic then
               /* Execute BFBTick function defined previously                    */
5              BFBTick(FB);
6          else
               /* Recursively execute CFBTick function                           */
7              CFBTick(FB);
8          end
9      end
       /* Emit output events and variables                                       */
10     emitOutput(E_O^𝓘, V_O^𝓘);
11     return;
```

5.3 The Prism Language

The Prism language [58] provides syntax for modelling of various type of stochastic systems. We restrict ourselves to Markov decision processes semantics of the Prism language as it is compatible with the discrete time (synchronous) semantics of function blocks adopted in this work. Based on this restriction, we present a formalisation of Prism models, which is a subset of its available feature. We use this formalisation later in the text for the purpose of automatic transformation of function blocks into Prism models.

5.3.1 Markov Decision Processes

Markov decision processes (MDP) combine probabilistic choices similar to that of Markov chains, and non-deterministic choices similar to that of labelled transition systems in a single Markov model. In this manner, MDP can be viewed as extensions of discrete-time Markov chain to allow non-determinism. Formally, MDP is defined as follows (adopted from [5]).

Definition 5.3.1 (Markov Decision Processes)**.** MDP is a tuple $\mathtt{MDP} = \langle S, s_0, Act, \mathsf{Steps}, AP, L\rangle$ where:

- S is the set of states with $s_0 \in S$ as the initial state.

- Act is the set of actions.
- $\mathsf{Steps} : S \rightarrow 2^{Act \times Dist(S)}$ is the transition probability function where, Act is the set of actions and $Dist(S)$ is the set of discrete probability distributions over the set of state S. A probability distribution over a set X is a function $f : X \rightarrow [0, 1]$ such that $\sum_{x \in X} f(x) = 1$.
- AP is the set of atomic propositions.
- $L : S \rightarrow 2^{AP}$ is the labelling function.

In the above definition, the transition probability function Steps returns one or more distributions for a given state. The non-empty nature of this function avoids deadlocks. Figure 5.4 presents an example MDP adopted from [5]. We can describe this MDP using Definition 5.3.1 as follows.

- $S = \{\mathsf{S}_0, \mathsf{S}_1, \mathsf{S}_2, \mathsf{S}_3\}$ is the set of all states with $s_0 = \mathsf{S}_0$ as the initial state.
- $Act = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$ is the set of actions
- Steps is the transition probability function i.e. the following:

$$\mathsf{Steps}(\mathsf{S}_0) = \{\ (\mathsf{a}, \{(1, \mathsf{S}_1)\})\ \}$$
$$\mathsf{Steps}(\mathsf{S}_1) = \{\ (\mathsf{b}, \{(0.3, \mathsf{S}_1), (0.7, \mathsf{S}_0)\}),\ (\mathsf{c}, \{(0.5, \mathsf{S}_2), (0.5, \mathsf{S}_3)\})\ \}$$
$$\mathsf{Steps}(\mathsf{S}_2) = \{\ (\mathsf{a}, \{(1, \mathsf{S}_2)\})\ \}$$
$$\mathsf{Steps}(\mathsf{S}_3) = \{\ (\mathsf{a}, \{(1, \mathsf{S}_3)\})\ \}$$

- $AP = \{S0, S1, S2, S3\}$ i.e., the state-labels used as atomic proposition.
- L is the labelling function, which in this example uses the state labels i.e. $L(\mathsf{S}_0) = \{S0\}$, $L(\mathsf{S}_1) = \{S1\}$, $L(\mathsf{S}_2) = \{S2\}$, and $L(\mathsf{S}_3) = \{S3\}$.

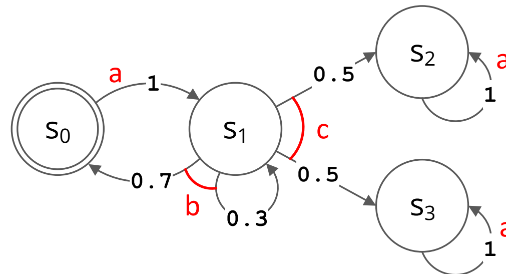


Figure 5.4: An example of Markov decision processes [5]

In the above example, $\mathsf{Steps}(\mathsf{S}_1)$ presents a non-deterministic choice between actions b and c. Based on this choice, the probability distribution $Dist(\mathsf{S}_1)$ then leads to a probabilistic choice e.g., if action b was selected from S_1, it leads to either one of S_0 and S_1 with probabilities 0.7 and 0.3 respectively. On the other hand, if action c was selected from S_1, it leads to one of S_2 and S_3 with a probability of 0.5 each.

5.3.2 Prism Model and Modules

A Prism model comprises one or more modules that interact with each other by means of reading and updating values of variables. A Prism module with two variables, `status` and `count`, is presented in Table 5.2. This module has a Boolean variable named status with an initial value `false`. An integer variable named counter has an initial value of 0, and is bounded by a range of [0, 100]. It contains two commands that operate over these variables to implement an input counter. Both commands wait for an action `t` to evaluate their guard conditions. The first command reads an *external* Boolean variable named input to evaluate if it can be enabled, whereas the second command is enabled if the value of the counter variable becomes greater than or equal to 99. Upon action `t`, one of the enabled commands is selected non-deterministically for execution, which results in execution of the corresponding update statements e.g., if the first command is enabled and selected, the counter is incremented, and if the second command is enabled and selected, the overflow status flag is set. Using this illustration, we formally described a Prism model as follows.

Definition 5.3.2 (Prism Model)**.** A Prism model is tuple, $\mathcal{M} = \langle \mathtt{mdp}, M, \mathcal{G}, \mathtt{t} \rangle$, where `mdp` declares the type of Prism model $\mathcal{G}$ is the set of global constants, M is the composition of the finite set of Prism modules $\{M_0, M_1, ..., M_k\}$ that synchronise using the action label `t` for lock-step execution. A Prism module in model $\mathcal{M}$ is a tuple $M_i = \langle name_i, V_i, C_i \rangle$,

Table 5.2: An example of a Prism module

```
1 module input_counter
2     //Variables
3     counter : [0..100] init 0;
4     status : bool init false;

5     //Commands
6     /*A variable named input is defined in a seperate module*/
7     [t] (input) -> (counter'=(counter+1))&(status'=false);
8     [t] (counter>=99) -> (counter'=0)&(status'=true);
9 endmodule
```

BEST PFE.COM
List of research project topics and materials

where $name_i$ is a distinct identifier of the Prism module M_i, V_i is the set of local variables of the module M_i and C_i is the set of commands in M_i. A Prism command is a tuple $C = \langle \texttt{t}, g, U \rangle$, where:

- $\texttt{t}$ is the action label of the command used for synchronisation with other modules in the Prism model $\mathcal{M}$ in each execution cycle. This corresponds to the *tick* of a synchronous program (refer to Sections 5.2.2 and 5.2.4).
- $g \in \mathcal{B}(\mathcal{V})$ is the Boolean guard of C where, $\mathcal{B}(\mathcal{V})$ is the set of Boolean expressions over variables and global constants in the Prism model $\mathcal{M}$ i.e., $\mathcal{V} = \cup_{i=0}^{k} V_i \cup \mathcal{G}$.
- $U = \{(\lambda_0, u_0), ..., (\lambda_n, u_n)\}$ is the set of probabilistic updates with discrete probability value λ_i, such that for all $(\lambda_i, u_i) \in U$ we have $\lambda_i \geq 0$ and $\sum_{i=0}^{n} \lambda_i = 1$. For a given probabilistic update $(\lambda_i, u_i) \in U$, u is and update sequence and is of the form $(v'_i = expr_i) \wedge (v'_j = expr_j) \wedge \cdots \wedge (v'_n = expr_n)$. In an update sequence u, each element has two components, namely:
  - $v_i \in V_k$ is a variable to be updated such that, v_i being a local variable of Prism module M_k is writeable. Furthermore, in a given command $C \in M_k$, any variable $v \in V_k$ can be updated only once in the scope of u.
  - $expr_i \in Expr(\mathcal{V})$, is an element from the set of all Boolean and arithmetic expressions over all variables in the parent Prism model. Prism language semantics uses the notion of delayed updates i.e., any expression evaluated in an ith execution cycle will use values from updates made in the $(i-1)^{th}$ execution cycle, and the update value is available for evaluation in the $(i+1)^{th}$ execution cycle.

Definition 5.3.3 (Prism Module Composition)**.** The composition of two Prism modules M_i and M_j is also a Prism module i.e., $M = \langle V, C \rangle$, where V is the set of all variables of the Prism modules M_i and M_j, and C is the set of commands created as the result of rule-based composition. The composition of commands is created by the following rule presented as follows.

$$\frac{(\texttt{t}, g, U) \in M_i \quad and \quad (\texttt{t}, g', U') \in M_j}{(\texttt{t}, (g \wedge g'), (U \otimes U')) \in M}$$

Where, $\otimes$ is defined as follows:

$$
\begin{aligned}
U = \; & \{(\lambda_0, u_0), (\lambda_1, u_1), ..., (\lambda_n, u_n)\} \\
U' = \; & \{(\lambda'_0, u'_0), (\lambda'_1, u'_1), ..., (\lambda'_{n'}, u'_{n'})\} \\
U \otimes U' = \; & \{(\lambda_0 * \lambda'_0), (u_0 + u'_0)), ..., (\lambda_n * \lambda'_0), (u_n + u'_0)), \\
& (\lambda_0 * \lambda'_1), (u_0 + u'_1)), ..., (\lambda_n * \lambda'_1), (u_n + u'_1)), \\
& \quad \vdots \qquad\qquad \vdots \qquad\qquad \vdots \\
& (\lambda_0 * \lambda'_{n'}), (u_0 + u'_{n'})), ..., (\lambda_n * \lambda'_{n'}), (u_n + u'_{n'}))\}
\end{aligned}
$$

Here, the commands with `t` action label are composed in parallel by taking the conjunction ('$\wedge$') of the condition guards and concatenation ('$\otimes$') of the probabilistic update sequences. This concatenation entails multiplying the probability values ('*') and appending of update statements ('+'). An illustration of this rule is presented in Table 5.3 where two probabilistic commands from two different modules are combined together using concatenations of the probabilistic updates to form a new probabilistic command in the composition i.e., the following.

$$
\begin{aligned}
C_1 &= (\texttt{t}, \quad g_1 = \texttt{m1A}, \quad U_1 = \{(0.3, \texttt{m1B'=false}), (0.7, \texttt{m1B'=true})\}) \\
C_2 &= (\texttt{t}, \quad g_2 = \texttt{m2A}, \quad U_2 = \{(0.7, \texttt{m2B'=false}), (0.3, \texttt{m2B'=true})\}) \\
C_{12} &= (\texttt{t}, \quad g_1 \wedge g_2, \quad U_1 \otimes U_2)
\end{aligned}
$$

Where, $g_1 \wedge g_2 =$ `m1A` && `m2A` and $U_1 \otimes U_2$ is computed as follows.

$$
\begin{aligned}
U_1 \otimes U_2 = \{ & \\
& ((0.3 * 0.7), \quad (\texttt{m1B'=false}) + (\texttt{m2B'=false})), \\
& ((0.7 * 0.7), \quad (\texttt{m1B'=true}) + (\texttt{m2B'=false})), \\
& ((0.7 * 0.3), \quad (\texttt{m1B'=true}) + (\texttt{m2B'=true})), \\
& ((0.3 * 0.3), \quad (\texttt{m1B'=false}) + (\texttt{m2B'=true})) \\
\} &
\end{aligned}
$$

Table 5.3: Illustration of composition of probabilistic commands of two Prism modules

```
1    [t] (m1A) -> 0.3 : (m1B'=false) + 0.7 : (m1B'=true);
```

```
1    [t] (m2A) -> 0.7 : (m2B'=false) + 0.3 : (m2B'=true);
```

```
1    [t] (m1A && m2A) -> 0.21 : (m1B'=false) & (m2B'=false)
2    + 0.49 : (m1B'=true)  & (m2B'=false)
3    + 0.21 : (m1B'=true)  & (m2B'=true)
4    + 0.09 : (m1B'=false) & (m2B'=true);
```

The execution of resultant compositions of Prism modules is performed using statistical simulation. In such an execution, a random number generator is used to perform non-deterministic and probabilistic selection between the available choice of commands.

5.3.3 Execution of a Prism Module

The execution of a Prism module is cyclic and in every cycle a Prism module performs the steps depicted in Algorithm 5.3. Firstly, lines $2-6$ iterate over all commands of the Prism module and evaluate the Boolean guards to compute a set of enabled commands K. If no command can be enabled, the cycle is completed as instructed by lines $7-9$. Otherwise, a non-deterministic selection is made from the set K (line 10). Next, an update-pair is selected from the command (line 11) in a probabilistic manner with respect to the associated probability values. The update statements of the selected pair are then executed (line 12), which updates the values of one or more variables. Prism uses a delayed composition where all values are read from the previous cycle and all updates are made available in the next cycle.

Algorithm 5.3: Execution of a single *cycle* of a Prism module

```
1  Function PrismCycle(M):
       /* iterate over all commands                                  */
2      for c_i ∈ M.C do
           /* Evaluate all enabled commands                          */
3          if evaluate(c_i.g) then
4              K ← (K ∪ {c_i});
5          end
6      end
7      if isEmpty(K) then
8          return;
9      end
       /* Non-deterministically select an enabled command            */
10     c_k ← nSelect(K);
       /* Probabilistically select an update-pair from c_k           */
11     u_k ← pSelect(c_k.U_k);
       /* Execute update statements in u_k                           */
12     execute(u_k);
13     return;
```

5.4 Converting Function Blocks to Prism

The goal of this conversion is to create a Prism model which is semantically equivalent to the given FBN. The synchronous execution of FBs in an FBN has some similarities to the MDP semantics of Prism language. For example, the step-by-step synchronous execution can be emulated using the cyclic execution of Prism modules. Similarly, both use the unit-delayed composition of corresponding components. These similarities between the two structures make is possible to create a rule-based transformation such that, the resultant Prism model is behaviourally equivalent to the given FBN. We can further simplify this conversion by using Observation 5.2.1 to assume that every function block in the given FBN is a BFB. Thus, the goal of transformation is to map every BFB instance $\texttt{BFB}_i$ in the FBN to a corresponding module $\mathcal{M}_i$ in the resultant Prism model, which is represented as follows.

$$\texttt{BFB}_i \in \texttt{FBNetwork} \Longleftrightarrow M_i \in \mathcal{M} \tag{5.4.1}$$

We know from Definition 5.3.2 that a Prism module consists of a set of variables and commands i.e., $M_i = \langle V_i, C_i \rangle$. Thus, the conversion of each BFB of the given FBN entails systematic generation of variables and commands in the corresponding Prism module such that, the execution behaviour of the Prism model is equivalent to the FBN. The overall process of conversion is presented as a flow chart in Figure 5.5. As presented in the sub-flow labelled Ⓐ, creation of variables is performed iteratively. Firstly, a Boolean variable is created for each output event, which is followed by creation of variables for local and output variables of a given BFB. The detailed process of creating these variables is described in the sub-sections as indicated in the respective stages. It is noteworthy that variables are created only for the output signals of a given BFB i.e., output events, output variables and local variables. Inputs in Prism modules are read directly from the sibling modules, consequently, creating separate variables for inputs becomes unnecessary.

Commands of the resultant Prism module are generated as shown by the sub-flow labelled Ⓑ. As indicated, there are three types of commands that are to be generated: an init-command, a set of command representing transitions of BFBs, and self-loop command to avoid deadlocks. Further details on how to create these commands is available in the sub-sections as indicated in the respective stages of the flow. In the subsequent sections, we shall present the various concepts employed in the construction of variables and commands of the resultant Prism module.

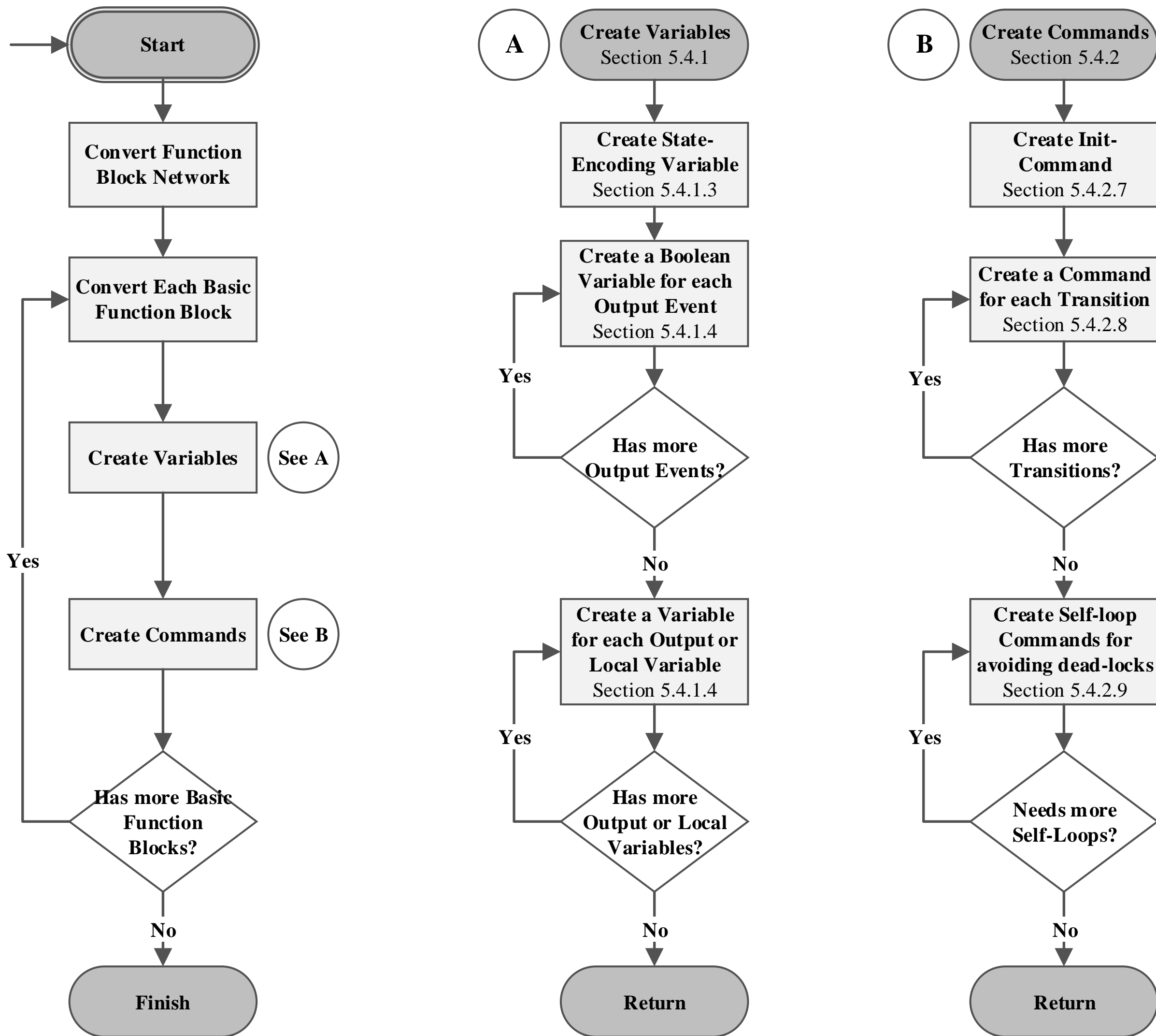


Figure 5.5: The process of converting a FBN to a Prism model presented as a flow chart

We start with an intuitive illustration of the Controller function block is presented in Figure 5.2. The corresponding generated Prism module is presented in Table 5.4. Here, the 8 transitions of the controller BFB are mapped to 8 respective Prism commands i.e., lines $11 - 30$. Additionally, we have an initialisation command on line 10, and three generated self-loop commands on lines $32 - 36$. Since, inputs are read from the source Prism modules directly, this is also reflected in the generated Prism module. For example, the input event rChange is read from the source FB instance flow_1oo2::output, which is reflected in the command on line 14 where an external variable named `output_flow__1oo2` is read. Note that the a fully-qualified scheme of variable names is used to avoid any potential naming conflicts.

Table 5.4: Generated Prism module from the Controller function block shown in Figure 5.2. Note that "_controller" postfix is omitted from variables names for readability.

```
1  module controller
2      s : [-1..4] init -1;
3      //s = {0, 1, 2, 3, 4} : {CLOSED, DO_OPEN, OPENED, DO_CLOSE, ALARM}

4      //Generated from output events
5      trigger : bool init false;

6      //Generated from internal and output variables
7      warning : bool init false;
8      valveCtl : bool init false;
9      counter : [0..10] init 0;

10     [t] (s=-1) -> (s'=0) & (trigger'=false) & (counter' = 0);

11     [t] (s=0) & (update_cylinder) & (pressure_cylinder >= 300) ->
12             (s'=1) & (trigger'=true) & (valveCtl' = true) &
13             (counter' = (counter < 10) ? (counter + 1) : counter);

14     [t] (s=1) & (output_flow__1oo2) & (oValue_flow__1oo2) ->
15             (s'=2) & (trigger'=false) & (counter' = 0);

16     [t] (s=2) & (update_cylinder) & (pressure_cylinder < 300) ->
17             (s'=3) & (trigger'=true) & (valveCtl' = false) &
18             (counter' = (counter < 10) ? (counter + 1) : counter);

19     [t] (s=3) & (output_flow__1oo2) & (oValue_flow__1oo2 = false) ->
20             (s'=0) & (trigger'=false) & (counter' = 0);

21     [t] (s=1) & ((output_flow__1oo2 = false) | (oValue_flow__1oo2 = false)) &
22             (counter < 10) -> (s'=1) & (trigger'=true) & (valveCtl' = true) &
23             (counter' = (counter < 10) ? (counter + 1) : counter);

24     [t] (s=1) & ((update_cylinder = false) | (oValue_flow__1oo2 = false))
25             & (counter >= 10) -> (s'=4) & (trigger'=true);

26     [t] (s=3) & ((output_flow__1oo2 = false) | (oValue_flow__1oo2)) &
27             (counter < 10) -> (s'=3) & (trigger'=true) & (valveCtl' = false) &
28             (counter' = (counter < 10) ? (counter + 1) : counter);

29     [t] (s=3) & ((output_flow__1oo2 = false) | (oValue_flow__1oo2)) &
30             (counter >= 10) -> (s'=4) & (trigger'=true);

31     //Generated self-loops for emulating synchronous execution semantics
32     [t] (s=0) & ((update_cylinder = false) | (pressure_cylinder < 300)) ->
33             (s'=0) & (trigger'=false);

34     [t] (s=2) & ((update_cylinder = false) | (pressure_cylinder >= 300)) ->
35             (s'=2) & (trigger'=false);

36     [t] (s=4) -> (s'=4) & (trigger'=false);
37 endmodule
```

5.4.1 Generating Variables

In this section we present discussion on generating variables in the Prism module. In the context of our model transformation, there are tree types of variables that are generated in a Prism module: the state-encoding variable, Boolean variables for encoding output events, integer and Boolean variables representing output and local variables of a given BFB. The process of generating these variables is captured by three respective transformation rule named T1–T3. These rules are supported by a range of macros for the purpose of simplification. The implementation of these macros is discussed in detail in the following subsections.

5.4.1.1 Mapping Variable Names

The generated variables follow a specific naming convention, which is implemented by the `nameOf` macro, which maps a given identifier to its equivalent in the generated Prism module. This is performed by a rename mechanism, which ensures that the identifier names are unique by using the name of the corresponding Prism module as a postfix. Using the `bfb1` as the current Prism module name, we illustrate this rule with the help of the following examples, where suggested variable names are renamed to actual variable names.

- nameOf(oVar1) = `oVar1_bfb1`
- nameOf(oEvent) = `oEvent_bfb1`
- nameOf(s) = `s_bfb1`

Using this mode of fully-qualified names avoid any potential naming conflicts for variables name i.e., in cases where variables with same name are declared in more than one BFBs, or multiple instances of same FBType is used in the given FBN. Note that a simple character-stuffing mechanism is also implemented to encode module names that already contain the underscore character e.g., we generated a variable name `output_flow__1oo2` for the output event names `flow_1oo2::output`. This mechanism helps avoiding additional naming conflicts.

5.4.1.2 The variable macro

The variable macro generates a variable definition in the current Prism module with a specified *name*, *type*, *range*, and *initial* value. The context of invocation determines the current Prism module and uses its name for the purpose of generating a fully-qualified name. We use the following notation:

$\mathsf{variable}(name, type, range, init)$ *where,*

- *name:* a valid identifier for Prism variable names. The uniqueness of this variable is ensured by using the $\mathsf{nameOf}(name)$ macro.
- *type:* a valid Prism variable type i.e., either integer (INT) or Boolean (BOOL).
- *range:* an ordered set of valid values for the given Prism variable type i.e., an upper and lower bound for integer type variables $[l, u]$ where $l, u \in \mathcal{Z}$ and $u \geq l$. Whereas, the range for a Boolean variable is the set of Boolean constants $[\texttt{false}, \texttt{true}]$. Since the range of Boolean variables is statically defined, it is omitted by using the *don't care* symbol $\top$.
- *init:* a valid initial value for the generated Prism variable from the ordered set *range* i.e., $init \in range$.

We use the controller BFB (see Figure 5.2) to extract and present examples generated by the variable macro as follows:

- $\mathsf{variable}(\mathsf{nameOf}(\mathsf{counter}),\ \mathrm{INT}\ ,\ [0,\ 5],\ 1)$
  $= \quad \texttt{counter_controller : [0..5]\ \ init\ \ 1;}$
- $\mathsf{variable}(\mathsf{nameOf}(\mathsf{trigger}),\ \mathrm{BOOL},\ \top,\ \texttt{true})$
  $= \quad \texttt{trigger_controller : bool\ \ init\ \ true;}$

5.4.1.3 Encoding ECStates

The states of a given BFB are encoded using distinct integer values of a state-encoding variable s as shown on line 2 of Table 5.4.

$$\mathsf{variable}(\mathsf{nameOf}(\mathsf{s}), \mathrm{INT}, [-1, \mathsf{len}(Q) - 1], -1) \qquad \textbf{(T1)}$$

Here, we used the nameOf macro to compute a unique name for the state-encoding variable of the current Prism module. This state-encoding variable is defined as an integer type variable with a specified range of $[-1, \mathsf{len}(Q)-1]$ where, $\mathsf{len}(Q)$ is the number of states in the given BFB. It uses -1 as its initial value, representing that the pre-initialisation state of a BFB. The purpose of adding this value is to enable semantically correct translation of Moore-type BFBs to Mealy-type Prism modules. The macro valueOf manages the subsequent value mapping of the state-encoding variable using the declarative index ($\mathsf{indexOf}$) of a given state in the corresponding ECC e.g., a value between 0 and $\mathsf{len}(Q)-1$. Whereas, the value -1 is assigned when a special don't care element is used i.e., $\top$.

$$\mathsf{valueOf}(q) \quad = \quad \begin{cases} \mathsf{indexOf}(q) & q \in Q \\ -1 & \top \end{cases} \qquad s.t.,\ q \in Q \cup \{\top\}$$

Boolean guards over the state-encoding variable induce *locations* in the generated Prism module such that, the initialisation value -1 induces the *init-location* to perform the module initialisation. We use the valueOf macro for generating Boolean guards that induce locations in the generated Prism module against the corresponding ECStates. Similarly, we also use the valueOf macro for updating values of the state-encoding variable against a given successor ECState i.e., the following.

checkState(q) = (s==valueOf(q))

updateState(q) = (s=valueOf(q))

In the above examples, the fragment shown inside single quote marks represent an expression or a statement. The expression generated by the checkState macro evaluates the state-encoding variable s against the encoded value of the given ECState q. Whereas, the updateState macro generates an assignment statement, which assigns the encoded value of the given ECState q to the state-encoding variable s. We illustrate the two macros using the following examples taken from the controller BFB shown in Figure 5.2.

- checkState(CLOSED) = (s==0)
- checkState(OPENED) = (s==2)
- updateState(DO_CLOSE) = (s=3)

5.4.1.4 Encoding the Events and the Variables

Function blocks consist of two types of signals i.e., events and variables, Prism language on the other hand, only contains variables. For the purpose of model transformation, FB events are encoded as Boolean variables in Prism modules as shown in line 5 of Table 5.4. The status of an event (*present* or *absent*) is mapped to the Boolean value of the corresponding variable i.e., `true` = *present* and `false` = *absent*. Based on this mapping scheme, we present Rule T2 for encoding events of BFBs in the corresponding Prism modules. The initial value of events under the synchronous execution semantics is absent, which is depicted in the Rule T2 as well.

$$\mathsf{variable}\Big(\mathsf{nameOf}(e), \text{BOOL}, \top, \texttt{false}\Big) \text{ } \textit{where } e \in E_O^{\mathcal{I}} \quad \textbf{(T2)}$$

Similarly, Rule T3 encodes output and local variables respectively with the corresponding initial values of their BFB counterparts as shown in lines $7-9$ of Table 5.4.

$$\mathsf{variable}\Big(\mathsf{nameOf}(v),\ \mathsf{typeOf}(v),\ \mathsf{rangeOf}(v),\ \mathsf{initOf}(v)\Big),\ \textit{where } v \in V_O^{\mathcal{I}} \cup V_L^{\mathcal{I}} \quad \textbf{(T3)}$$

- **nameOf**: generates a fully-qualified variable name for the given FB element i.e., event or variable, using the current Prism module name.
- **typeOf**: determines the type of a given IEC 61499 variable and maps it to either integer (INT) or Boolean (BOOL).
- **rangeOf**: returns the range of a given IEC 61499 variable, which is statically defined for Boolean variables as [`false`, `true`]. The integer type variables can define their own lower and upper bound values, however in case of missing information, we can use the values for short-integer namely, `SINT_MIN` and `SINT_MAX` constants as defined by ISO-C standard.
- **initOf**: determines the initial value of a given IEC 61499 variable that belongs the range specified above. In case of missing values, we assign 0 to be the initial value of an integer type variable, and `false` to be the initial value of a Boolean variable.

Thus, Rules T1–T3 are used to create variables in the Prism module, where Rule T1 creates an integer typed state-encoding variable, Rule T2 creates Boolean variables for encoding output events, and Rule T3 creates integers and Boolean variables to represent the output and local variables of the respective types.

BEST PFE.COM
List of research project topics and materials

5.4.2 Generating Commands

In this section, we present our approach to bridge the gap between syntax and semantics of IEC 61499 and Prism language. In a Prism module, commands are akin to its execution instructions. In the context of model to model transformation, these commands must emulate the synchronous execution semantics of the source FB structure. For this purpose, we define a set of macros that facilitate the generation of commands in the generated Prism modules that are behaviourally equivalent to the synchronous execution of FBs.

5.4.2.1 Encoding Wire-Connections

In FBNs, events and data flow from the outputs of FB instances to the inputs using explicit wire connections (see Definition 5.2.4]). Because of the delayed-communication, the output emitted from a source FB instance is read by the destination FB instance in the next tick. The new values loaded by these wire connections is used for the purpose of (i) evaluating Boolean guards for enabling/disabling transitions, and (ii) using values in execution of algorithms.

The process of reading new values from the source FB instances is translated in a straight forward manner in Prism because of the global readability scope of its variables i.e., Prism modules can read all variables in their parent Prism model including those that belong to their sibling modules. Therefore, wire-connections in a given FBN can be encoded as direct read operation on the source Prism module. We define **sourceOf** macro that exploits this global readability scope to avoid making redundant copies of output variables, which would have otherwise been required to emulate the wire-connections between of source and target FBs. Consider the example presented in Figure 5.6. Here, we can use the sourceOf macro to map a given input signal to a set of source signals using its wire-connections. This set of source signals may contain multiple elements for event-wire connections, where multiple source events are permitted as inputs. This source events set is then used to construct an expression in Prism syntax as a Boolean disjunction. On the other hand, source sets for variable connections can only contain a single element, as restricted by the definition of wire-connections (see Definition 5.2.4). In this case, the name of the source variable can be directly used as a valid expression.

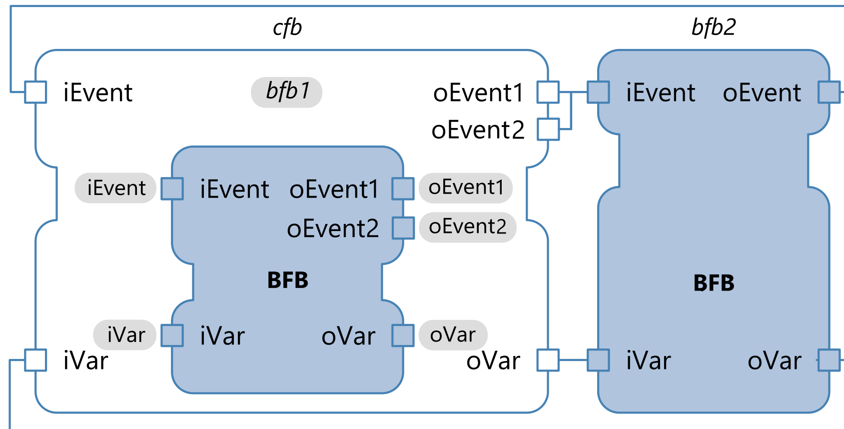


Figure 5.6: Lookup source of wire-connections for Prism syntax generation

Consider the following examples extracted from Figure 5.6.

- sourceOf(cfb.bfb1.iEvent) = (`oEvent_bfb2`)
- sourceOf(cfb.bfb1.iVar) = (`oVar_bfb2`)
- sourceOf(bfb2.iEvent) = (`oEvent1_bfb1_cfb` | `oEvent2_bfb1_cfb`)
- sourceOf(bfb2.iVar) = (`oVar_bfb1_cfb`)

In the first two examples, the source of the input was traced back to output signal of bfb2, which were then renamed using the nameOf macro. The third example presents a case where multiple sources are available for an input event. In this case, the sources are traced and renamed, and then combined to form a Boolean disjunction. Since input variables cannot have multiple sources, a combinator operator (e.g., Boolean disjunction) is not required.

5.4.2.2 Ensuring Life-time of an Event

Synchronous execution semantics define the life-time of an event as one tick. In the generated Prism module, we ensure this by explicitly setting values of the corresponding Boolean variables in each execution cycle. We define a **setStatus** macro, which can be used for this explicit encoding of the current status of output events. It takes two disjoint sets of output events to map their statuses either as present or absent i.e., as follows.

$$\mathsf{setStatus}(\mathcal{E}, \overline{\mathcal{E}}) : \ \mathcal{E} \rightarrow \texttt{true}, \ \overline{\mathcal{E}} \rightarrow \texttt{false} \ \ s.t., \mathcal{E} \uplus \overline{\mathcal{E}} = E_O^{\mathcal{I}}$$

During each execution cycle, this macro is invoked with appropriate parameters to set the status of the Boolean variables corresponding to the output events.

5.4.2.3 Encoding the State-Entry Actions

The state-entry actions in BFBs comprise a sequence of algorithms and emission of output events (see Definition 5.2.3). These actions can be encoded by generating a set of assignment statements in Prism syntax that emulate these actions. For this purpose, we define a macro named stateActions as follows.

$$\mathsf{stateActions}(q) \;=\; \mathsf{setStatus}(\,\mathcal{E}_q, \overline{\mathcal{E}_q}) \cup \Gamma_q \quad \textit{where,}$$

- $q \in Q$ is an ECState of a given BFB
- $\mathcal{E}_q = E_O^{\mathcal{I}} \cap X(q)$ is the set of output events emitted in the context of state-entry action for the ECState q. Where $E_O^{\mathcal{I}}$ is the set of all output events and $X(q)$ is the state-entry action for the ECState $q \in Q$.
- $\overline{\mathcal{E}_q} = E_O^{\mathcal{I}}/X(q)$ is the set of output events that are *not* emitted in the context of state-entry actions for the ECState q.
- $\Gamma_q = \{\rho \mid \rho \in A_L^{\mathcal{I}} \cap X(q)\}$ is the set of statements from the algorithms invoked from the state-entry action of the ECState q.

5.4.2.4 Preserving Transition Priority

Transitions from a given state $q \in Q$ of a BFB $T(q)$ is an ordered set (see Definition 5.2.3). However, commands in Prism modules are unordered (Definition 5.3.2). To preserve the order of transitions of the given BFB, we use negations of Boolean guards of the higher priority transitions for ensuring that a lower priority Prism command can only be enabled when higher priority commands are disabled. We define a macro named ensureOrder to preserve order of a given transition using the said Boolean negation mechanism as follows.

$$\mathsf{ensureOrder}(t_j) \;=\; \widehat{\mathcal{B}}(e_j, b_j) \bigwedge_{t_i \in T(q)}^{t_i > t_j} \neg\Big(\widehat{\mathcal{B}}(e_i, b_i)\Big) \quad \textit{where,}$$

- $t_j = (q, e_j, b_j, q'_j) \in T(q)$ is a given transition to preserve its order.
- $t_i = (q, e_i, b_i, q'_i) \in T(q)\ s.t., t_i > t_j$ is an element from the set of all transitions from q such that t_i is of higher priority than the given transition t_j. Recall that t_0 is the highest priority transition. We use the index of the transitions in a manner where lower index $i < j$ implies higher priority $t_i > t_j$.

- $\widehat{\mathcal{B}}(e,b)$ is the Boolean guard of a given transition $t = (q, e, b, q')$, which comprises the status checking of the event e, and evaluation the Boolean expression b over the set of local and input variables of the given BFB.

The implementation of ensureOrder macro takes the first i elements from the ordered set $T(q)$ such that, $i < j$. This selection results in an ordered subset of all transitions from q that are of higher priority than the given transition t_j. Then, a new Boolean guard is computed through conjunction of negations of selected Boolean guards of all selected transitions t_i with the Boolean guard of the given transition t_j. In case where the given the transition is the highest priority transition, i.e. $t_j = t_0$, it results in an empty set of higher priority transitions. Thus, in this case the resultant Boolean guard comprises only of the Boolean guard of the given transition t_j. Consider the following examples of the DO_CLOSE ECState from the ECC of the controller FB (see Figure 5.2). Here, we have the following transitions.

$$\begin{aligned} T(\mathsf{DO_CLOSE}) = \{\ & t_0 = \mathsf{DO_CLOSE} \xrightarrow{<0>\ \mathsf{pChange\&\&(!relief)}} \mathsf{CLOSED}, \\ & t_1 = \mathsf{DO_CLOSE} \xrightarrow{<1>\ \mathsf{counter<10}} \mathsf{DO_CLOSE}, \\ & t_2 = \mathsf{DO_CLOSE} \xrightarrow{<2>\ \mathsf{true}} \mathsf{ALARM}\ \ \} \end{aligned}$$

Using the ensureOrder macro, we get the following.

- ensureOrder(t_0) = pChange && (!relief)
- ensureOrder(t_1) = (counter<10) && (!pChange || relief)
- ensureOrder(t_2) = (counter>=10) && (!pChange || relief)

5.4.2.5 Converting Expressions and Statements

The **convExpr** macro is defined for converting expressions and statements from IEC 61499 syntax into equivalent Prism syntax. This macro is useful for translating algorithms and condition guards for a given BFB. This translation entails processing *identifiers* and *operators* in the given expression or statement.

$$\mathsf{convIdent(identifier)} = \begin{cases} \mathsf{nameOf(identifier)} & \backslash\backslash \text{ output and local events and variables} \\ \mathsf{sourceOf(identifier)} & \backslash\backslash \text{ input events and variables} \end{cases}$$

$$\mathsf{convOp(operator)} = \text{Perform a look up for an equivalent operator from Table 5.5}$$

Using Figure 5.2 for illustration, we present the following examples of the convExpr macro.

- convExpr({counter == 10}) = `(counter_controller = 10)`
- convExpr({valveCtl = true}) = `(valveCtl_controller' = true)`
- convExpr({rChange && (!relief)})

  = `(output_flow__1oo2) & (!oValue_flow__1oo2)`
- convExpr({pChange && (pressure >= threshold)})

  = `(update_cylinder) & (pressure_cylinder >= 300)`

In the first example, since the local variables counter does not require a source lookup, it is simply renamed using the nameOf macro. In the second example, we used the assignment operator (see Table 5.5) to assign a Boolean *true* value to a variable. Note that for IEC 61499 we assume 'C'-like syntax that uses '=' and '==' operators to differentiate between Boolean equality and assignment operations. In the third example, the input event pChange was looked up using the sourceOf macro to find the source output event i.e., flow_1oo2::output, which was then renamed using the nameOf macro. Next, the input variable relief was replaced with `oValue_flow__1oo2`, which is the renamed source output variable. Similarly, the last example was processed using the sourceOf macro, except that the source of the input threshold was an integer constant value 300. In all of the above examples, operators were translated using Table 5.5.

Table 5.5: Lookup table for translating syntax from IEC 61499 to Prism

Operator Name	IEC 61499 Operator	Prism Operator
disjunction	$\vert\vert$ or $\vert$	$\vert$
conjunction	&& or &	&
equality	== or =	=
inequality	! = or <>	! =
negation	!	!
addition	+	+
subtraction	−	−
multiplication	∗	∗
division	/	not supported
assignment	a = b	a′ = b
increment	a + +	a′ = (a + 1)
decrement	a − −	a′ = (a − 1)
addition w/ assignment	a+ = b	a′ = (a + b)
subtraction w/ assignment	a− = b	a′ = (a − b)
multiplication w/ assignment	a∗ = b	a′ = (a ∗ b)
division w/ assignment	a/ = b	not supported
conditional assignment	a = (b)?c : d	a′ = (b)?c : d

5.4.2.6 The command Macro

We define the **command** macro for creating a command in a given Prism module. Given a set of Boolean expressions B, and a set of assignment statements and event statuses S, the command macro produces a Prism command (t, c, U) as follows.

$$\mathsf{command}(B, S) \;=\; \Big(\mathtt{t},\; g = \bigwedge_{b \in B} \mathsf{convExpr}(b),\; U = \bigcup_{s \in S} \mathsf{convExpr}(s)\Big) \qquad (5.4.2)$$

Recall from Definition 5.3.2, `t` is the synchronization label of Prism which is enforced on all generated Prism commands to perform execution in a lock-step. Next, we take all Boolean expressions from the given set B and convert them to their Prism equivalent syntax using the convExpr macro. The converted expressions are used to construct a Boolean conjunction using the '&' operator (see Table 5.5). Lastly, we take all assignment statements from the given set S and construct a sequence of equivalent assignment statements in Prism syntax using the '&' operator. Consider the following examples derived from the ECC of the controller BFB presented in Figure 5.2.

- command({(counter < 10)}, {(counter = (counter<10)?(counter+1):counter)})

```
= [t] (counter_controller < 10)  ->
    (counter_controller' = (counter_controller < 10) ?
      (counter_controller + 1) :  counter_controller);
```

- command({(pChange && (pressure >= threshold))}, {(valveCtl = true)})

```
= [t] (update_cylinder) & (pressure_cylinder>=300)  ->
    (valveCtl_controller'=true);
```

5.4.2.7 Encoding the Moore-type Initialisation

The ECCs of BFBs are Moore-like state machines, which may have initial state actions. In synchronous execution semantics, these actions are invoked in an initialisation tick. Unlike other ticks, in the initialisation inputs are ignored and the current state is set to the initial ECState $q_0 \in Q$. These initialisation can be encoded in the Mealy-like Prism command structure by generating an *init-command* in generated Prism modules

as follows.

$$\mathsf{command}(\{\mathsf{s==-1}\}, \{\mathsf{updateState}(q_0)\} \cup \mathsf{stateActions}(q_0)) \tag{T4}$$

Using the controller function block presented in Figure 5.2, we generate the following Prism command i.e., line 11 of Table 5.4.

```
[t] (s_contoller=-1) -> (s_controller'=0)
    & (trigger_controller'=false) & (counter_controller'=0);
```

In this generated command, [t] is the action label for the lock-step execution of this command as suggested by the synchronous execution semantics. Secondly, the special *init-value* (-1) is used to induce the *init-location*. This is followed by a set of update statements generated from the action set associated with the initial state q_0. Together, these three components make the *init-command*.

5.4.2.8 Encoding Transitions

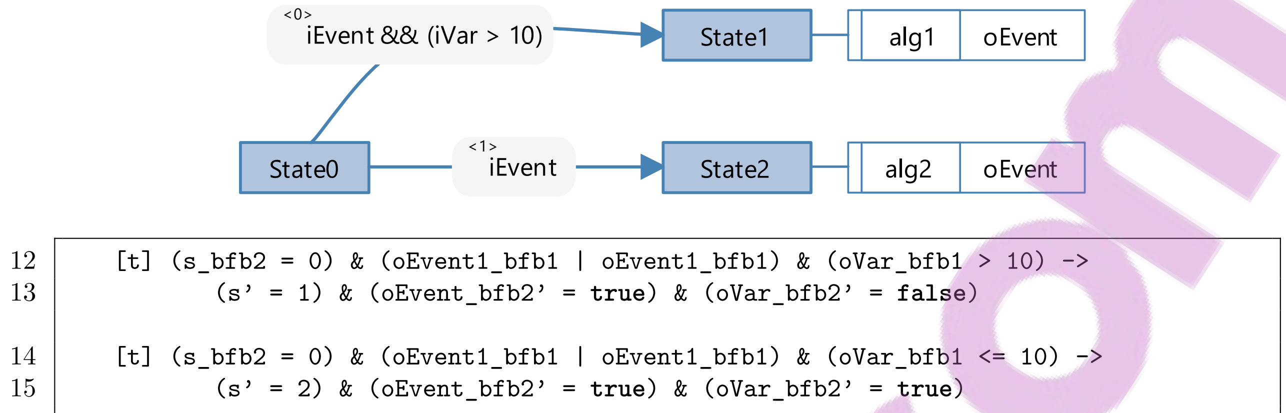
Given a transition $t = (q, e, b, q') \in T(q)$ of a given BFB, we can use the command macro to generate behaviourally equivalent Prism commands in the corresponding Prism module i.e., as follows.

$$\mathsf{command}\Big(\mathsf{checkState}(q) \cup \mathsf{ensureOrder}(t),\ \mathsf{updateState}(q') \cup \mathsf{stateActions}(q')\Big) \tag{T5}$$

The Rule T5 uses the checkState macro to induce a location corresponding to the predecessor state q. Here, the ensureOrder macro is used to generate a condition guard, which ensures that the transition order is preserved. Lastly, the action set of the successor state q' i.e., the sequence of algorithm invocations and emissions of output events is converted using the stateActions macro. Figure 5.7 illustrates the conversion of two transitions into corresponding Prism commands. Here, the wire-connection encoding was performed by the sourceOf macro using the function block network presented in Figure 5.6.

5.4.2.9 Avoiding Deadlocks

ECCs of BFBs have an implicit stay operation when none of the egress transitions are enabled. In order to avoid deadlocks in the corresponding generated Prism module, self-loop commands are added to mimic stay operations using the Rule T6. This rule ensures




```
12    [t] (s_bfb2 = 0) & (oEvent1_bfb1 | oEvent1_bfb1) & (oVar_bfb1 > 10) ->
13         (s' = 1) & (oEvent_bfb2' = true) & (oVar_bfb2' = false)

14    [t] (s_bfb2 = 0) & (oEvent1_bfb1 | oEvent1_bfb1) & (oVar_bfb1 <= 10) ->
15         (s' = 2) & (oEvent_bfb2' = true) & (oVar_bfb2' = true)
```

Figure 5.7: Illustration of Rule T5 for encoding deterministic transitions. The source state-transition structure is shown above and the generated Prism commands are shown below.

that every module has a reaction in every execution cycle, thus ensuring that the composition of Prism modules remains reactive. However, if a state already has an unconditional self-loop, this rule is not needed.

$$\mathsf{command}\Big(\mathsf{checkState}(q) \cup \mathsf{negateAll}(q),\ \mathsf{updateState}(q) \cup \mathsf{setStatus}(\varnothing,\ E_O^{\mathcal{I}})\Big) \qquad \textbf{(T6)}$$

- Here, the invocation of macros $\mathsf{checkState}(q)$ and $\mathsf{updateState}(q)$ induce a self-loop from the given state q to itself.
- The $\mathsf{negateAll}(q)$ macro is used to ensure that the generated self-loop bears the lowest priority among all egress transitions from the given ECState q. This is implemented by taking all Boolean guards from all egress transitions and constructing a Boolean conjunction over their negations. Thus, the generated self-loop can only execute when all other egress transitions are disabled. The implementation of $\mathsf{negateAll}(q)$ is as follows.

$$\mathsf{negateAll}(q) = \bigwedge_{t_i \in T(q)} \neg\Big(\widehat{\mathcal{B}}(e_i, b_i)\Big) \quad \textit{where, } t_i = (q, e_i, b_i, q_i)$$

- Lastly, the macro invocation $\mathsf{setStatus}(\varnothing,\ E_O^{\mathcal{I}})$ generates Boolean assignment statements that set status of all output events to *absent*. Here, the empty set symbol $\varnothing$ depicts that none of the events are present. This ensures that the state-entry actions are not invoked but the life-time of an event (e.g., *1-tick*) is enforced.

Using the controller ECC for illustration (see Figure 5.2), we see that three ECStates CLOSED, OPENED, and ALARM require self-loops. Whereas two ECStates DO_OPEN and

DO_CLOSE do not require self-loops because they already have a un-conditional egress transitions predicated on the always present `true` event. For the said three ECStates, the generated self-loops command are visible as lines $32-36$ of Table 5.4.

5.4.3 Algorithm for Generating the Prism Model

Algorithm 5.4 uses the proposed transformation Rules T1–T6 to converts a given FBN to a Prism model. An illustration of this algorithm is presented in Table 5.7. In this illustration we used the Prism module generated for the Controller function block presented in Figure 5.2. We divided the module into several segments, where each segment is generated by a particular set of lines of this algorithm. This is shown in form of a label visible on the right hand bottom side of each segment. For example, the illustration shows that line 3 of Algorithm 5.4 generated a unique name `controller` for the Prism module using the FB instance name. Line 4 generated the state-encoding variable with a range $[-1, 4]$, such that values: $0, 1, 2, 3, 4$ correspond to ECStates: CLOSED, DO_OPEN, OPENED, DO_CLOSE, ALARM and the value -1 corresponds to an *initialisation command*. Lines $5-7$ and $8-10$ generated variables for output events, and output and local variables respectively. This involved generating unique names for the variables, and assigning them appropriate type and value range. We present the name lookup map in Table 5.6, which uses the sourceOf and nameOf macros to map names of input events and variables to their source variable names in the corresponding Prism modules for the direct-read access (see Section 5.4.2.1).

There are three types of commands in the generated Prism module namely, the init-command, transition-encoding commands, and self-loop commands. Lines $11-13$ generated the said initialisation command. Due to the initial value of the state-encoding variables, this command is always the first command to execute in the generated Prism module. Lines $15-19$ generated 8 commands corresponding to the 8 transitions of the controller BFB. Generating these commands involved converting condition guards while preserving the transition order, converting action sets into update statements and updating the state-encoding variable to emulate successor ECState. Lines $20-24$ generated the self-loop commands to mimic the stay-operation of synchronous execution semantics. All commands (generated by lines $11-25$) ensure the life-time of events by explicitly setting the status of all Boolean variables representing the output events. In this manner, Algorithm 5.4 generates a Prism module for each BFB in the FBN (see line 2), thus constructing the overall Prism model.

Table 5.6: Name lookup table for Controller FB interface IO

Name	Type	sourceOf	nameOf	Remarks
trigger	Output Event	—	`trigger_controller`	Postfix is omitted
rChange	Input Event	flow_1oo2::output	`output_flow__1oo2`	Event source lookup
pChange	Input Event	cylinder::update	`update_cylinder`	Event source lookup
warning	Output Variable	—	`warning_controller`	Postfix is omitted
valveCtl	Output Variable	—	`valveCtl_controller`	Postfix is omitted
pressure	Input Variable	cylinder::pressure	`pressure_cylinder`	Variable source lookup
relief	Input Variable	flow_1oo2::oValue	`oValue_flow__1oo2`	Variable source lookup
threshold	Input Variable	Constant::300	300	Constant value was used

Algorithm 5.4: Generating a Prism model for an FBN

Input: `FBNetwork` –The given function block network
Output: $\mathcal{M}$ –The generated Prism model

```
1  Function Transform(FBNetwork)
2    foreach BFB_i ∈ FBNetwork do
       /* generating name for the new Prism module */
3      M_i.name ← nameOf(BFB_i)
       /* generating state-encoding variable (see rule T1) */
4      M_i.V ← variable(nameOf(s), INT, [−1, len(Q) − 1], −1)
       /* generating variables against output events of BFB_i (see rule T2) */
5      foreach e ∈ BFB_i.E_O^I do
6        M_i.V ← variable(nameOf(e), BOOL, ⊤, false)
7      end
       /* generating variables against local and output variable of BFB_i (see rule
       T3) */
8      foreach v ∈ BFB_i.V_O^I ∪ BFB_i.V_L^I do
9        M_i.V ← variable(nameOf(v), typeOf(v), rangeOf(v), initOf(v))
10     end
       /* generating command for initial state actions of BFB_i (see rule T4) */
11     initCondition ← {'s==-1'}
12     initUpdate ← updateState(q_0) ∪ stateActions(q_0)
13     M_i.C ← command(initCondition, initUpdate)
       /* iterate over all ECStates of BFB_i */
14     foreach q ∈ BFB_i.Q do
         /* generting commands for all transitions in BFB_i (see rule T5) */
15       foreach t = (q, e, b, q′) ∈ T(q) do
16         transCondition ← checkState(q) ∪ ensureOrder(t)
17         transUpdate ← updateState(q′) ∪ stateActions(q′)
18         M_i.C ← command(transCondition, transUpdate)
19       end
         /* generating self-loop commands (see rule T6) */
20       if (q, true, ∅, q′) ∉ T(q) for any q′ ∈ BFB_i.Q then
21         loopCondition ← checkState(q) ∪ negateAll(t)
22         loopUpdate ← updateState(q) ∪ setStatus(∅, E_O^I)
23         M_i.C ← command(loopCondition, loopUpdate)
24       end
25     end
26     𝓜 ← M_i /* adding the new module in the generated Prism model */
27   end
28   return 𝓜
```

BESTPFE.COM
List of research project topics and materials

Table 5.7: Illustration of Algorithm 5.4 on the Prism module generated from the Controller BFB shown in Figure 5.2. Note that `_controller` postfix is omitted for readability.

```
1   module controller                                                  Algorithm 5.4 : 3
2       s : [-1..4] init -1;
3       //s = {0,1,2,3,4} : {CLOSED,DO_OPEN,OPENED,DO_CLOSE,ALARM}     Algorithm 5.4 : 4
4       //Generated from output events
5       trigger : bool init false;                                     Algorithm 5.4 : 5 – 7
6       //Generated from internal and output variables
7       warning : bool init false;
8       valveCtl : bool init false;
9       counter : [0..10] init 0;                                      Algorithm 5.4 : 8 – 10
10      [t] (s=-1) -> (s'=0) & (trigger'=false) & (counter' = 0);      Algorithm 5.4 : 11 – 13
11      [t] (s=0) & (update_cylinder) & (pressure_cylinder >= 300) ->
12             (s'=1) & (trigger'=true) & (valveCtl' = true) &
13             (counter' = (counter < 10) ? (counter + 1) : counter);

14      [t] (s=1) & (output_flow__1oo2) & (oValue_flow__1oo2) ->
15             (s'=2) & (trigger'=false) & (counter' = 0);

16      [t] (s=2) & (update_cylinder) & (pressure_cylinder < 300) ->
17             (s'=3) & (trigger'=true) & (valveCtl' = false) &
18             (counter' = (counter < 10) ? (counter + 1) : counter);

19      [t] (s=3) & (output_flow__1oo2) & (oValue_flow__1oo2 = false) ->
20             (s'=0) & (trigger'=false) & (counter' = 0);

21      [t] (s=1) & ((output_flow__1oo2 = false) | (oValue_flow__1oo2 = false)) &
22             (counter < 10) -> (s'=1) & (trigger'=true) & (valveCtl' = true) &
23             (counter' = (counter < 10) ? (counter + 1) : counter);

24      [t] (s=1) & ((update_cylinder = false) | (oValue_flow__1oo2 = false))
25             & (counter >= 10) -> (s'=4) & (trigger'=true);

26      [t] (s=3) & ((output_flow__1oo2 = false) | (oValue_flow__1oo2)) &
27             (counter < 10) -> (s'=3) & (trigger'=true) & (valveCtl' = false) &
28             (counter' = (counter < 10) ? (counter + 1) : counter);

29      [t] (s=3) & ((output_flow__1oo2 = false) | (oValue_flow__1oo2)) &
30             (counter >= 10) -> (s'=4) & (trigger'=true);            Algorithm 5.4 : 15 – 19
31      //Generated self-loops for emulating synchronous execution semantics
32      [t] (s=0) & ((update_cylinder = false) | (pressure_cylinder < 300)) ->
33             (s'=0) & (trigger'=false);

34      [t] (s=2) & ((update_cylinder = false) | (pressure_cylinder >= 300)) ->
35             (s'=2) & (trigger'=false);

36      [t] (s=4) -> (s'=4) & (trigger'=false);                        Algorithm 5.4 : 20 – 24
37  endmodule
```

5.5 Preserving the Execution Semantics

The application of the transformation rules generates a Prism model that preserves the synchronous execution semantics of the source FBN. From Equation 5.4.1, we know that every BFB in the given FBN has a corresponding Prism module in the generated Prism model. During execution, both the FBN and the Prism model execute their respective components in a cyclic manner (see Algorithms 5.2 and 5.3). In this section we establish that these executions are equivalent i.e., the execution behaviour of the given FBN is preserved by the generated Prism model.

We use the notion of execution states and transitions to represent Prism models and FBs as transitions systems. Later, we use this notion to demonstrate Bisimulation equivalence [107] between a given BFB and the corresponding generated Prism module. This equivalence is then used to demonstrate the equivalence of an FBN and the corresponding generated Prism model i.e., the compositions of individual BFBs and Prism modules.

5.5.1 Execution States

In this section we describe the execution of a BFB using the notion of transition systems. We use the notation $Var_{\mathtt{BFB}}$ to represent *execution-variables* of the BFB namely, the output events, and the output/local variables i.e., for a given BFB, $\mathtt{BFB} = \langle \mathcal{I}, L^{\mathcal{I}}, \mathtt{ECC}^{\mathcal{I},L^{\mathcal{I}}} \rangle$, we have $Var_{\mathtt{BFB}} = \{q_c, e_0, ..., e_m, v_0, ..., v_n\}$ where:

- $q_c \in \{\top\} \cup \mathtt{ECC}^{\mathcal{I},L^{\mathcal{I}}}.Q$ is a state variable
- $e_0, ..., e_m \in \mathcal{I}.E_O^{\mathcal{I}}$ are the output events
- $v_0, ..., v_n \in \mathcal{I}.V_O^{\mathcal{I}} \cup L^{\mathcal{I}}.V_L^{\mathcal{I}}$ are local and output variables

During the synchronous execution of $\mathtt{BFB}$, each tick assigns a vector of values to the execution variables such that, q_c indicates the current ECState of $\mathtt{BFB}$ in the said tick. Let *Eval* denote the set of evaluations that assign values to the execution-variables. A distinct evaluation $\eta \in Eval(Var_{\mathtt{BFB}})$ is a snapshot of a particular tick and is written in the form $\eta = [q_c{=}z_{q_c},\ e_0{=}z_{e_0},\ \cdots,\ e_m{=}z_{e_m},\ v_0{=}z_{v_0},\ \cdots,\ v_n{=}z_{v_n}]$ i.e., pairwise discrete values of the execution-variables $Var_{\mathtt{BFB}}$. Thus, the execution state-space of $\mathtt{BFB}$ is defined by $Eval(Var_{\mathtt{BFB}})$ such that, $Eval(Var_{\mathtt{BFB}}) \subseteq dom(q_c) \times dom(e_0) \times ... \times dom(e_m) \times dom(v_0) \times ... \times dom(v_n)$. Where, $dom(x)$ is the domain of the execution-variable x. For FBs, the domain of an event is $\{\mathtt{absent}, \mathtt{present}\}$ and domain of a Boolean variable is $\{\mathtt{true}, \mathtt{false}\}$. For

integer type variables we can declare minimum and maximum values such that, $\texttt{MIN} \leq x \leq \texttt{MAX}$. We also treat the current ECState q_c of the ECC as an execution variable such that, $dom(q_c) = \{\top\} \cup \texttt{ECC}^{\mathcal{I},L^{\mathcal{I}}}.Q$ where, the symbol $\top$ represents the pre-initialised state of a BFB and $\texttt{ECC}^{\mathcal{I},L^{\mathcal{I}}}.Q$ is the set of ECStates.

Definition 5.5.1 (Execution state of a BFB)**.** The execution state of a given BFB $\texttt{BFB}$ is an evaluation of its execution variables i.e., $\eta \in Eval(Var_{\texttt{BFB}})$. We can represent value of a particular execution variable using notation $\eta(x) \in dom(x)$. The initial execution state of a BFB is given by $\eta_0(q_c) = \top,\ \eta(e_m) = \texttt{absent}$, and $\eta(v_n) = \mathsf{initOf}(v_n)$ where, initOf is the initial value macro for variables.

Similarly, we can define the execution state of a Prism module as an evaluation of its execution variables Var_M as follows.

Definition 5.5.2 (Execution State of a Prism Module)**.** An execution state σ of a given Prism module M is defined as a discrete evaluation of its execution variables Var_M i.e., $\sigma \in Eval(Var_M)$ where, $Var_M = (v_0, \cdots, v_n)$ is the set of execution variables of M. We can represent value of a particular execution variable using notation $\sigma(x) \in dom(x)$.

Let M be a Prism module generated for a given BFB $\texttt{BFB}$. The execution variables M has three partitions namely, the state-encoding variable, Boolean variables encoding output events, and the variables encoding the local and output variables of $\texttt{BFB}$. Therefore, we can represent the execution variables of generated Prism modules as follows.

$$Var_M = \{\mathsf{s}, e_{0_M}, \cdots, e_{m_M}, v_{0_M}, \cdots, v_{n_M}\} \quad \textit{where,} \tag{5.5.1}$$

- Current state indicator $q_c \in Var_{\texttt{BFB}}$ corresponds with the state encoding variable $\mathsf{s} \in Var_M$ (from Rule T1)
- For every output event $e_{0_{\texttt{BFB}}}, \cdots, e_{m_{\texttt{BFB}}} \in Var_{\texttt{BFB}}$, there exists a corresponding Boolean variable $e_{0_M}, \cdots, e_{m_M} \in Var_M$ (from Rule T2)
- For every output/local variable $v_{0_{\texttt{BFB}}}, \cdots, v_{n_{\texttt{BFB}}} \in Var_{\texttt{BFB}}$, there exists a corresponding Boolean variable $v_{0_M}, \cdots, v_{n_M} \in Var_M$ (from Rule T3)

From above, we observe a bijective mapping between the sets of execution variables $Var_{\texttt{BFB}}$ and Var_M. Furthermore, due to Rules T1-T3, the corresponding variables of the two sets also have equivalent domains i.e., $dom(x_{\texttt{BFB}}) = dom(x_M)$ where, $x_{\texttt{BFB}} \in Var_{\texttt{BFB}}$ and $x_M \in Var_M$.

Definition 5.5.3 (Equivalence of execution states)**.** Given an execution state $\eta \in Var_{\texttt{BFB}}$ of a BFB BFB and an execution state $\sigma \in Var_M$ of a corresponding generated Prism module M, σ is equivalent to η iff:

1. The value of the state-encoding variable $\mathsf{s} \in Var_M$ (generated by Rule T1) is equivalent to the current ECState $q_c \in Var_{\texttt{BFB}}$ i.e., $\sigma(\mathsf{s}) = \mathsf{valueOf}(\eta(q_c))$
2. The values of all event-encoding Boolean variables $e_{i_M} \in Var_M$ (generated by Rule T2) are equivalent to the statuses of the corresponding output variables $e_{i_{\texttt{BFB}}} \in Var_{\texttt{BFB}}$ i.e., the following.

$$\sigma(e_{i_M}) = \begin{cases} \texttt{false} & \eta(e_{i_{\texttt{BFB}}}) = \texttt{absent} \\ \texttt{true} & \eta(e_{i_{\texttt{BFB}}}) = \texttt{present} \end{cases}$$

3. The values of all variables $v_{j_M} \in Var_M$ (generated by Rule T3) are equivalent to the values of the corresponding output and local variables $v_{j_{\texttt{BFB}}} \in Var_{\texttt{BFB}}$ i.e., $\sigma(v_{j_M}) = \eta(v_{j_{\texttt{BFB}}})$.

Based on the above definition, we can establish that the initial execution states of a given BFB and its corresponding generated Prism module are equivalent i.e., as follows.

Lemma 5.5.1 (Equivalence of initial states)**.** The initial execution states η_0 of a given BFB BFB, and σ_0 of the corresponding generated Prism module M, are equivalent $\eta_0 = \sigma_0$

1. The initial values of state variables are equivalent i.e., $\sigma_0(\mathsf{s}) = \mathsf{valueOf}(\eta_0(q_c))$ (from Definition 5.5.1 and Rule T1).
2. The initial values of all output events $e_{i_{\texttt{BFB}}} \in Var_{\texttt{BFB}},\ i \in [0, m]$ are defined by Definition 5.5.1 as $\eta_0(e_{i_{\texttt{BFB}}}) = \texttt{absent}$. The corresponding event-encoding Boolean variables $e_{i_M} \in Var_M,\ i \in [0, m]$ have equivalent value $\sigma_0(e_{i_M}) = \texttt{false}$ (from Rule T2) i.e., $\sigma_0(e_{i_M}) = \eta_0(e_{i_{\texttt{BFB}}})$.
3. The initial values of all variables $v_{j_M} \in Var_M,\ j \in [0, n]$ corresponding to the output and local variables $v_{j_{\texttt{BFB}}} \in Var_{\texttt{BFB}},\ j \in [0, n]$ are generated by Rule T3 using the $\mathsf{initOf}(v_{j_{\texttt{BFB}}})$ macro. Thus we can write, $\sigma_0(v_{j_M}) = \eta_0(v_{j_{\texttt{BFB}}})$.

Thus, from observations $1 - 3$ and Definition 5.5.3, we can say that the given initial execution states $\eta_0,\ \sigma_0$ are equivalent. $\square$

5.5.2 Transitions

The synchronous execution semantics for FBs is characterized by a step-by-step execution i.e., a sequence of steps representing transitions of execution states. The execution of a given BFB starts by performing an *initialisation step*, which takes it from pre-initialising execution state to an initialised state i.e., by performing an action α_0 to take the transition $\eta_0 \xrightarrow{\alpha_0} \eta_1$ such that, $\eta_1(q_c) = q_0$ is the initial ECState state of the given BFB. In this transition, the action α_0 corresponds to the first tick, which invokes the initial ECState actions $X(q_0)$ mapping execution variables their updated values. The initialisation can be also be represented as follows.

$$\eta_0 = [\top, z_{e_0}, ..., z_{e_m}, z_{v_0}, ..., z_{v_n}] \xrightarrow{\alpha_0} \eta_1 = [q_0, z'_{e_0}, ..., z'_{e_m}, z'_{v_0}, ..., z'_{v_n}]$$

The execution of a generic Prism module can also be represented as a step-wise execution. However, due to our scope we are only discuss the execution of generated Prism modules, which are restricted to a subset of Prism language semantics. For example, generated Prism modules execute in a lock-step with other modules and each execution cycle evaluates the state-encoding variable as a part of its Boolean command guard. This execution starts with an initial execution state σ_0 e.g., for the generated Prism module M we have $\sigma_0 = [\mathsf{valueOf}(\top), z_{e_0}, ..., z_{e_m}, z_{v_0}, ..., z_{v_n}]$. An action γ_0 representing the first execution cycle with the synchronisation-label `t` (see Equation 5.4.2) updates the execution state entering the successor $\sigma_1 = (\mathsf{valueOf}(q_0), e'_0, \cdots, e'_m, v'_0, \cdots, v'_m)$. Here, the valueOf macro is used to perform the state-encoding (described in Section 5.4.1.3) e.g., the encoding initial ECState $q_0 \in \mathtt{ECC}^{\mathcal{I},L^{\mathcal{I}}}.Q$ for representing the successor state. This initialisation can be represented as $\sigma_0 \xrightarrow{\gamma_0} \sigma_1$.

Definition 5.5.4 (Equivalence of transition)**.** A pair of given transitions $\eta_i \xrightarrow{\alpha_i} \eta_{i+1}$ and $\sigma_i \xrightarrow{\gamma_i} \sigma_{i+1}$ of a given BFB and a Prism module are equivalent iff the corresponding predecessor and successor states are equivalent i.e., the following.

$$\eta_i \xrightarrow{\alpha_i} \eta_{i+1} = \sigma_i \xrightarrow{\gamma_i} \sigma_{i+1} \iff \eta_i = \sigma_i \wedge \eta_{i+1} = \sigma_{i+1} \quad \textit{where, } \alpha_i \textit{ and } \gamma_i \textit{ represent ticks}$$

Under synchronous execution semantics, BFBs exhibit a so-called *stay* operation when none of the egress ECTransitions can be enabled. In this behaviour, the execution stays at the current ECState until the next tick. This mimics an implicit self-loop over the state variable and is encoded as such in the generated Prism modules (see Rule T6). Due to these self-loops, the execution structures of BFBs and corresponding generated Prism modules do not have any terminal states.

Observation 5.5.1 (No terminal states)**.** *The stay-operation of a BFB is encoded as self-loop command in the corresponding generated Prism module such that, a self-loop can be taken when no other command is enabled. That is, for given self-loop transitions* $\eta_i \xrightarrow{\alpha_i} \eta_{i+1}$ *and* $\sigma_i \xrightarrow{\gamma_i} \sigma_{i+1}$*, the value of state variables* q_c *and* $\mathtt{s}$ *is not modified i.e., we have:* $\eta_i(q_c) = \eta_{i+1}(q_c)$ *and* $\sigma_i(\mathtt{s}) = \sigma_{i+1}(\mathtt{s})$

From the above observation, we can create an infinite sequence of transitions starting from the initialisation step such that, every transition in this sequence represents a successive execution step i.e., a tick. We call this sequence an *execution trace.*

Definition 5.5.5 (Execution trace)**.** An execution trace an infinite sequence of transitions starting from the initial state e.g., $\pi_{\mathtt{BFB}} = \eta_0 \xrightarrow{\alpha_0} \eta_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{i-1}} \eta_i \xrightarrow{\alpha_i} \cdots$ is an execution trace of a given BFB BFB where:

1. η_0 is the initial execution state of BFB
2. $\forall i \geq 0 \ . \ \eta_i \in \mathit{Eval}(\mathit{Var}_{\mathtt{BFB}})$

Definition 5.5.6 (Reachable states)**.** An execution state η_j is a reachable if there exists an execution trace π_j that contains η_j i.e., the following:

$$\eta_j \in \mathit{Reach}(\mathtt{BFB}) \implies \exists \Big(\pi_j = \eta_0 \xrightarrow{\alpha_0} \eta_1 \xrightarrow{\alpha_1} \cdots \xrightarrow{\alpha_{j-1}} \eta_j \xrightarrow{\alpha_j} \cdots\Big)$$

Here, $\mathit{Reach}(\mathtt{BFB})$ is the set of all reachable states of BFB. Trivially, the initial execution state η_0 is a part of every execution trace, thus $\eta_0 \in \mathit{Reach}(\mathtt{BFB})$. Furthermore, every reachable execution state $\eta_j \in \mathit{Reach}(\mathtt{BFB})$, necessarily has a successive transition $\eta_i \xrightarrow{\alpha_i} \eta_{i+1}$ in an execution trace (from Observation 5.5.1). On the other hand if η_i is unreachable ($\eta_i \notin \mathit{Reach}(\mathtt{BFB})$), by Definition 5.5.5 we know that η_i is not contained in any execution trace and therefore, does not have any successive transition.

5.5.3 Bisimulation Equivalence

In the previous sections we presented execution of BFBs and generated Prism modules as execution steps that comprise execution states and transitions. Starting from their initial states, their execution progresses by taking transitions such that, all possible executions form an execution tree. In this section we evaluate *bisimulation equivalence* as an equivalence relation that identifies that the corresponding execution trees have a similar

branching structure [107]. We define the bisimulation equivalence of BFBs and generated Prism modules as follows.

Definition 5.5.7 (Bisimulation Relation)**.** Bisimulation of a BFB `BFB` and a Prism module M, is a binary relation $\mathcal{R} \subseteq \mathit{Eval}(\mathit{Var}_{\mathtt{BFB}}) \times \mathit{Eval}(\mathit{Var}_M)$ such that, for all $(\eta, \sigma) \in \mathcal{R}$ the following holds:

1. The given execution states are equivalent $\eta{=}\sigma$
2. If a transition $\eta \xrightarrow{\alpha} \eta'$ exists, then $\sigma \xrightarrow{\gamma} \sigma'$ also exists, such that $(\eta',\sigma') \in \mathcal{R}$
3. If a transition $\sigma \xrightarrow{\gamma} \sigma'$ exists, then $\eta \xrightarrow{\alpha} \eta'$ also exists, such that $(\eta',\sigma') \in \mathcal{R}$

Definition 5.5.8 (Bisimulation Equivalence)**.** A given BFB `BFB` and a Prism module M are *bisimulation-equivalent*, denoted `BFB` $\sim M$, if their respective initial execution states are bisimilar: $(\eta_0, \sigma_0) \in \mathcal{R}$.

Theorem 5.5.1. Let `BFB` be a given BFB and M its corresponding generated Prism module; `BFB` and M are bisimilar: `BFB` $\sim M$

Proof. The proof of this theorem is provided *by construction* i.e., we shall construct a binary relation and demonstrate that it is a bisimulation. We shall further demonstrate the bisimulation equivalence `BFB` $\sim M$ by showing that the bisimulation relation contains the initial states of `BFB` and M.

1. *Cross product:* Let $\mathcal{R}{=}\mathit{Eval}(\mathit{Var}_{\mathtt{BFB}}){\times}\mathit{Eval}(\mathit{Var}_M)$
2. *Refinement:* Let $\mathcal{R}_e$ be a subset of $\mathcal{R}$ such that it comprises only the pairs of equivalent states i.e., $\mathcal{R}_e{=}\{(\eta_i{,}\sigma_j) \mid \eta_i{=}\sigma_j \wedge (\eta_i{,}\sigma_j){\in}\mathcal{R}\}$ (see Definition 5.5.3).
3. *Bisimulation relation:* $\mathcal{R}_e$ is a bisimulation relation, because for every element $(\eta_i{,}\sigma_i){\in}\mathcal{R}_e$, we know that $\eta_i{=}\sigma_i$ and we can show the following:

   (A) If a transition $\eta_i \xrightarrow{\alpha_i} \eta_{i+1}$ exists, then $\sigma_i \xrightarrow{\gamma_i} \sigma_{i+1}$ also exists, and $(\eta_{i+1}{,}\sigma_{i+1}){\in}\mathcal{R}_e$

   (B) If a transition $\sigma_i \xrightarrow{\gamma_i} \sigma_{i+1}$ exists, then $\eta_i \xrightarrow{\alpha_i} \eta_{i+1}$ also exists, and $(\eta_{i+1}{,}\sigma_{i+1}){\in}\mathcal{R}_e$

   The proofs for (A) and (B) are presented as Lemmas 5.5.2 and 5.5.3 respectively.
4. *Bisimulation equivalence:* The given initial states η_0 and σ_0 of `BFB` and M are equivalent (from Lemma 5.5.1). Therefore, the initial states are contained in the bisimulation relation $\eta_0{=}\sigma_0 \implies (\eta_0{,}\sigma_0){\in}\mathcal{R}_e$ (from *Refinement*). ■

Lemma 5.5.2 (Proof for **(A)**)**.** Let $\eta_i = \sigma_i$ be a pair of equivalent execution states of BFB and M respectively. If η_i is reachable ($\eta_i \in Reach$(BFB)), a transition $\eta_i \xrightarrow{\alpha_i} \eta_{i+1}$ exists in BFB, and we can show that it has a corresponding equivalent transition $\sigma_i \xrightarrow{\gamma_i} \sigma_{i+1}$ in M such that, $\eta_{i+1} = \sigma_{i+1}$. Proving this equivalence would imply that the pair $(\eta_{i+1}, \sigma_{i+1})$ exists in the refined relation $\mathcal{R}_e$ (from Theorem 5.5.1).

Proof. Under synchronous execution semantics, every execution step comprises three sub-steps: *input sampling*, *taking a transition*, and *execution state-entry actions*. These sub-steps are emulated by the generated Prism module M to ensure its equivalence, as demonstrated by the following reasoning.

(a) **Input sampling:** Both BFB and M use unit-delayed updates. Consequently, both α_i and γ_i, which represent an execution step $\alpha_i = \gamma_i$, read their inputs from the previous execution states η_i and σ_i respectively. Since $\eta_i = \sigma_i$, therefore, both structures read equivalent input values.

(b) **Transitions:** Based on inputs read by α_i and the respective guard conditions on egress transitions of $\eta_i(q_c)$, BFB takes transition to a successor ECState $\eta_{i+1}(q_c)$. This step is matched by M with the following three cases:

Case I: *Initialisation step* i.e., $\eta_i(q_c) = \top$. In this case, the state variable is updated to $\eta_{i+1}(q_c) = q_0$. In this case, the initial value of state variable $\sigma_i(\mathsf{s}) = -1$ is assigned by Rule T1. A matching condition guard is generated by Rule T4, thus forcing the initial command to execute in the first execution cycle of M. Rule T4 further uses the macros $\mathsf{updateState}(q_0)$ for generating the update statements. Therefore, both the predecessor and successor execution states have an equivalent value for the state variables: $\sigma_{i+1}(\mathsf{s}) = \mathsf{valueOf}(\eta_{i+1}(q_c))$.

Case II: *No ECTransitions could be enabled from* $\eta_i(q_c)$. In this case, the state variable will not be changed $\eta_i(q_c) = \eta_{i+1}(q_c)$ and no state actions are performed. However, due to the single tick life-duration of events, the output events are reset. This case will be matched by M due to Rules T6 (see lines $20-24$ of Algorithm 5.4) where, macros $\mathsf{checkState}(q)$ and $\mathsf{updateState}(q)$ keep the current value of the state-encoding variable e.g., to emulate the stay operation on the current ECState as $\sigma_i(\mathsf{s}) = \sigma_{i+1}(\mathsf{s}) = \mathsf{valuesOf}(q)$. Here, the macro $\mathsf{negateAll}(q)$ ensures that this self-loop is only taken when no-other transition can be taken. Lastly, the macro $\mathsf{setStatus}(\varnothing,\ E_O^{\mathcal{I}})$ sets all output events to the absent status, thus emulating the event-reset behaviour of BFB. In this case, the execution step

Bestpfe.com

concludes here and no more actions are performed in either of the structures in the current execution step.

Case III: At least one ECTransition can be enabled from $\eta_i(q_c)$. In this case, the highest priority ECTransition shall be taken by BFB. This case will be matched by M by executing an enabled command, which is generated by lines $15 - 19$ of Algorithm 5.4, where Rule T5 matches the predecessor ECState q using the macro checkState(q). Furthermore, this rule uses explicit negations inside the ensureOrder macro to ensure that (i) at most one command is enabled in an iteration, and (ii) the transition order is preserved. Therefore, the command executed by M will necessarily match the highest priority enabled transition as taken by BFB.

(c) **Successor State Actions:** If Case I or III occurs, BFB executes its successor ECState-entry actions $X(q')$ where, $\eta_{i+1}(q_c) = q'$. This step is matched by M because of the macro stateActions in Rules T4-T5 (see lines 12 and 17 of Algorithm 5.4), which ensures that (i) algorithms in $A_L^{\mathcal{I}} \cap X(q')$ are translated into update statements that perform equivalent computation and value assignments for local and output variables, and (ii) the event-encoding Boolean variables are set and with an appropriate Boolean value for event emissions in $X(q')$. Lastly, the macro updateState updates the state-encoding variable s to represent the respective successor ECState q' i.e., $\sigma_{i+1}(\mathsf{s}) = \mathsf{valueOf}(q')$

The equivalence of the three sub-steps presented above, yields equivalent values for the state variable, event-encoding Boolean variables, and variables representing local and output variables. This means that the resultant corresponding execution states are equivalent $\eta_{i+1} = \sigma_{i+1}$ and therefore, $(\eta_{i+1}, \sigma_{i+1}) \in \mathcal{R}_e$. □

Lemma 5.5.3 (Proof for **(B)**). Let $\eta_i = \sigma_i$ be a pair of a equivalent execution states of BFB and M respectively and $\sigma_i \in Reach(M)$ is *reachable*. We can show that a transition $\sigma_i \xrightarrow{\gamma_i} \sigma_{i+1}$ exists in M, and a corresponding transition $\eta_i \xrightarrow{\alpha_i} \eta_{i+1}$ exists in BFB such that $\sigma_{i+1} = \eta_{i+1}$. Proving this equivalence implies that the pair $(\eta_{i+1}, \sigma_{i+1})$ exists in the refined relation $\mathcal{R}_e$ (from Theorem 5.5.1).

Proof. The cyclic execution of M is performed by selecting an enabled command and executing its update statements (see Algorithm 5.3). In each execution cycle, M selects an enabled command $c = (\mathtt{t}, g, U)$ for execution, where g is a Boolean expression over the

set of execution variables Var_M. Since $\eta_i = \sigma_i$, any Boolean expression g satisfied by σ_i is also satisfied by BFB i.e., BFB $\models g \iff \sigma_i \models g$. From Algorithm 5.4, every command generated in M falls under the following three cases.

Case I: The selected command is generated by Rule T4. In this case, the selected command is the initialisation command (see lines $11 - 13$ of Algorithm 5.4). Rule T4 uses the macro updateState(q_0) for representing the initial ECState of BFB i.e., $\sigma_{i+1}(\mathsf{s}) =$ valueOf($\eta_{i+1}(q_c)$) with $\eta_{i+1}(q_c) = q_0$. Also, the macro stateActions(q_0) encodes the state-entry actions $X(q_0)/la$, which assigns a Boolean value to every event encoding variable $e_{0_M}, \cdots, e_{m_M} \in M$ representing statuses of corresponding output events $e_0, \cdots, e_m \in E_O^{\mathcal{I}}$. Similarly, the macro invocation stateActions(q_0) also generates update statements for the variables $v_{0_M}, \cdots, v_{n_M} \in M$, which performs similar updates as the algorithm statements of $A_L^{\mathcal{I}} \cup X(q_0)$

Case II: The selected command is generated by Rule T5. In this case, we know that a corresponding ECTransition $t = (q, e, b, q')$ exists (see lines $15 - 19$ of Algorithm 5.4)), such that $\sigma_i(\mathsf{s}) =$ valueOf(q) and $\sigma_{i+1}(\mathsf{s}) =$ valueOf(q'). The said ECTransition induces a transition in the execution state $\eta_i \xrightarrow{\alpha_i} \eta_{i+1}$ with $\eta_i(q_c) = q$ and $\eta_{i+1}(q_c) = q'$. The said ECTransition can be enabled since $\sigma_i \models g$ and $\sigma_i = \eta_i$. Consequently, (i) the state variable is updated $\sigma_{i+1}(\mathsf{s}) =$ valueOf($\eta_{i+1}(q_c)$) (see macro updateState), and (ii) ECState-entry actions $X(\eta_{i+1}(q_c))$ are performed to update the output events and output/local variables in an equivalent manner (see macro stateActions). This results in an equivalent successor execution state $\eta_{i+1} = \sigma_{i+1}$

Case III: The selected command is generated by Rule T6. This case represents a self-loop command (see lines $20 - 24$ of Algorithm 5.4). This self-loop mimics the stay-operation of BFBs i.e., $\sigma_i(\mathsf{s}) = \sigma_{i+1}(\mathsf{s})$. Here negateAll macro ensures that the it matches a case in BFB where no ECTransition could be enabled i.e., $\eta_i(q_c) = \eta_{i+1}(q_c)$. Furthermore, the specific usage of setStatus macro resets all event-encoding Boolean variables to match the respective statuses of output events in BFB. Also, the values of variables representing output and local variables of BFB remain unchanged similar to the behaviour of BFB. Therefore, the successor execution states are equivalent $\sigma_{i+1} = \eta_{i+1}$

The three cases presented above encompass all generated commands in M, and the successor execution states in these cases are shown to be equivalent i.e., $\sigma_{i+1} = \eta_{i+1}$. Therefore, $(\eta_{i+1}, \sigma_{i+1}) \in \mathcal{R}_e$ □

BESTPFE.COM
List of research project topics and materials

We can represent FBNs and Prism models as compositions of BFBs and Prism modules respectively (see Observation 5.2.1 and Definition 5.3.2). The execution of FBs under synchronous execution semantics, and the execution of generated Prism models are both cyclic. This mode of execution is agnostic to the order of execution of individual components within a given cycle due to the *lock-step* execution of components and *unit-delayed* communication (see Algorithms 5.2 and 5.3). Therefore, without loss of generality, we can assume that the order of execution of BFBs in the given FBN match that of the corresponding modules in the generated Prism model. We define the behaviour equivalence of a given FBN and a Prism model as follows.

Definition 5.5.9 (Behavioural equivalence of FBN and Prism model)**.** A given FBN `FBNetwork` and a Prism model $\mathcal{M}$ are behaviourally equivalent iff:

1. every BFB in FBN $\texttt{BFB}_i \in \texttt{FBNetwork}$ corresponds with a Prism module $M_i \in \mathcal{M}$ such that $\texttt{BFB}_i \sim M_i$ and,
2. the composition operator used for parallel composition of BFBs $\texttt{BFB}_0, \cdots, \texttt{BFB}_n$ and that of the corresponding Prism modules $M_0, \cdots, M_n$ is identical.

Theorem 5.5.2. A given FBN `FBNetwork` and the corresponding generated Prism model $\mathcal{M}$ are behaviourally equivalent

Proof. The behavioural equivalence of `FBNetwork` and $\mathcal{M}$ can be shown as an observation on the transformation rules e.g., as follows.

1. Every BFB $\forall i \in [0, n]$. $\texttt{BFB}_i \in \texttt{FBNetworkFBs}$ corresponds with a generated Prism module $M_i \in \mathcal{M}$ (from Equation 5.4.1) such that $\texttt{BFB}_i \sim M_i$ (from Theorem 5.5.1).
2. Both the BFBs $\texttt{BFB}_0, \cdots, \texttt{BFB}_n$ and the corresponding generated Prism modules $M_0, \cdots, M_n$ execute in lock-step e.g., synchronous ticks and execution cycles synchronised over label `t` respectively (see `command` macro in Rules T4-T6).
3. Both compositions use unit-delayed communication between the individual components $\texttt{BFB}_0, \cdots, \texttt{BFB}_n$ and $M_0, \cdots, M_n$ (see Algorithms 5.1 and 5.3).

■

5.5.4 Model Verification

Formal verification of models often involves processing an abstraction of the original system model. However, this verification is only valid if the abstraction holds a sound equivalence with the system model. In our case, we generate Prism models from the given FBNs through a semantics-preserving transformation. Due to the to bisimulation-equivalence shown in Theorem 5.5.1, the generated Prism models preserve the soundness of linear-time (LTL) properties of the given FBNs [107]. This equivalence relation allows for the use of the generated model for the purpose of formal verification such that it provides the following guarantees.

$$\varphi \models \mathcal{M} \iff \varphi \models \texttt{FBNetwork} \quad s.t., \ \varphi \in \texttt{LTL}$$

The equation presented above states that a given linear property φ is either satisfied by both models or is satisfied by neither. Such verification guarantees are significantly important for model verification where verifying a property on one model gives us information about the other. Specifically, a given property needs only to be verified on the generated Prism model $\mathcal{M}$ to be considered satisfied by the given FBN. A probabilistic extension of linear temporal logic (LTL), called PLTL is used for property specification in Prism model checker [59]. This notation allows quantitative reasoning over non-branching behaviours of the system execution tree. For example, we can specify the query to compute the probability of reaching a failure state during the system execution. However, in our case the generated Prism models do not contain any probabilistic behaviours, therefore the result of the result of quantitative analysis would be either 0 or 1. Consider the property, $P(\boldsymbol{Safe}) := P_{max=0}\ F[\boldsymbol{Alarm}]$. In this above property, we are evaluating if the Alarm state is reachable i.e., if the maximum probability is zero, then the state is not reachable. This naive quantitative analysis is actually equivalent to performing a qualitative analysis of the corresponding system, where the result is always in the form of a Boolean proposition i.e., whether the given property is satisfied.

5.6 Discussion

In this chapter, we presented a formal structure and semantics for IEC 61499 function blocks and Prism models. This formalism was later used to present a set of rules that performed automatic transformation of a given FBN into a behaviourally equivalent Prism model. We presented the arguments and proofs of this equivalence. We further presented

the usefulness of this equivalence for the purpose of formal verification of system properties. This method of verification is on a par with the existing qualitative verification approaches [27, 34, 111].

The semantic preserving model transformation rules provide a basis for a novel model based safety assessment methodology that we shall present in the subsequent chapters. The formalism of FBs and Prism, transformation rules, utility macros and algorithms shall be inherited for developing an extension of IEC 61499 FBs, which allows modelling of stochastic behaviours e.g., probabilistic failures and non-deterministic behaviours. This modelling approach shall be used for integrated reliability modelling of a given system using IEC 61499 that contains both hardware models as well as controlling software. Through model transformation, we shall analyse safety of such systems by employing Prism model checker's probabilistic verification.

6

Stochastic Function Blocks

Model-Based Safety of IEC 61499 Systems

Industrial automation systems are complex real-time systems that consist of hardware and software components. Even though, the traditional safety analysis techniques such as, *reliability block diagrams* and *fault tree analysis* are accepted and well established in the reliability engineering practices, they are largely dependent on practitioners' skills and experience [44]. Model-based safety analysis (MBSA) ameliorates this problem and improves the quality of safety analysis through the use of system models [45]. These models may be manually constructed for the purpose of MBSA, or automatically derived from extensions of models used for system development [46]. The latter approach offers seamless integration of model-based development with the safety analysis practices, where the model of the nominal behaviour of the system is merged with the corresponding failure models [44]. The resultant model may contain failure occurrence and propagation information and the failure effect information. The earlier of the two describe how the failure is generated and propagated through dependent components whereas the latter defines the failure-affected behaviour of the components [47].

In this chapter, we propose a framework for MBSA of industrial automation sys-

tems, which enables modelling of extended plant-models using IEC 61499 using the MVC pattern. This extended plant-model contains model of nominal component behaviours, component failure models, and the respective failure effects on the component behaviours. The primary contribution of this chapter that allows this extended modelling is a novel extension called *Stochastic Function Block* (SFB). SFB offers provision of modelling state based stochastic behaviours. The added expressiveness is leveraged by a proposed rule-based mechanism to automatically derive Markov models directly from the system models, which are later used for safety analysis. The probabilistic and non-deterministic aspects of extended plant-model are verified using the Prism model checker [59]. The result of this verification gives practitioners early feedback on the safety of their system design and implementation. This provides the estimated reliability of the overall systems and indication of any systematic errors in the controller software.

6.1 Overview

In order to illustrate the proposed approach, we use an example of a hazardous gas detection and ventilation system as shown in Figure 6.1. The system consists of a set of hazardous gas (e.g., CO, CO_2, CH_4) sensors installed in parallel to detect and report unusual levels to the control panel. The programmable logic controller (PLC) installed in the control panel activates a ventilation fan upon receiving an indication. Both the sensors and the ventilator are prone to random failures due to their respective failure rates.The emission of hazardous gas is a non-deterministic aspect of the environment with an assumption that the rate of the gas build-up will not exceed the rate of ventilation. An IEC 61499 design for this system is presented in Figure 6.2, which consists of two separations, namely

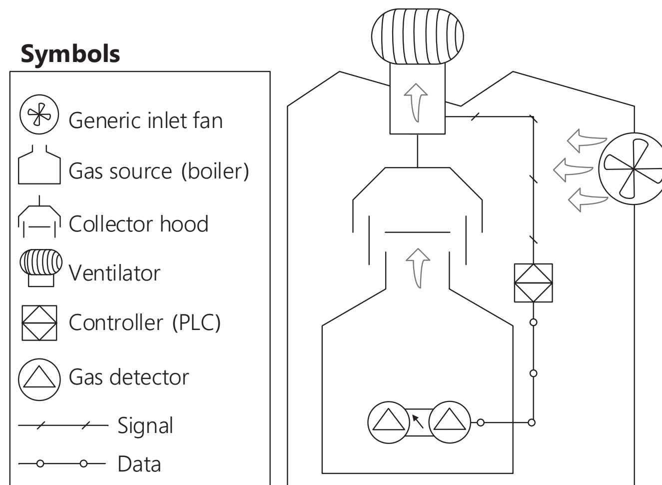


Figure 6.1: A hazardous gas detection and ventilation system

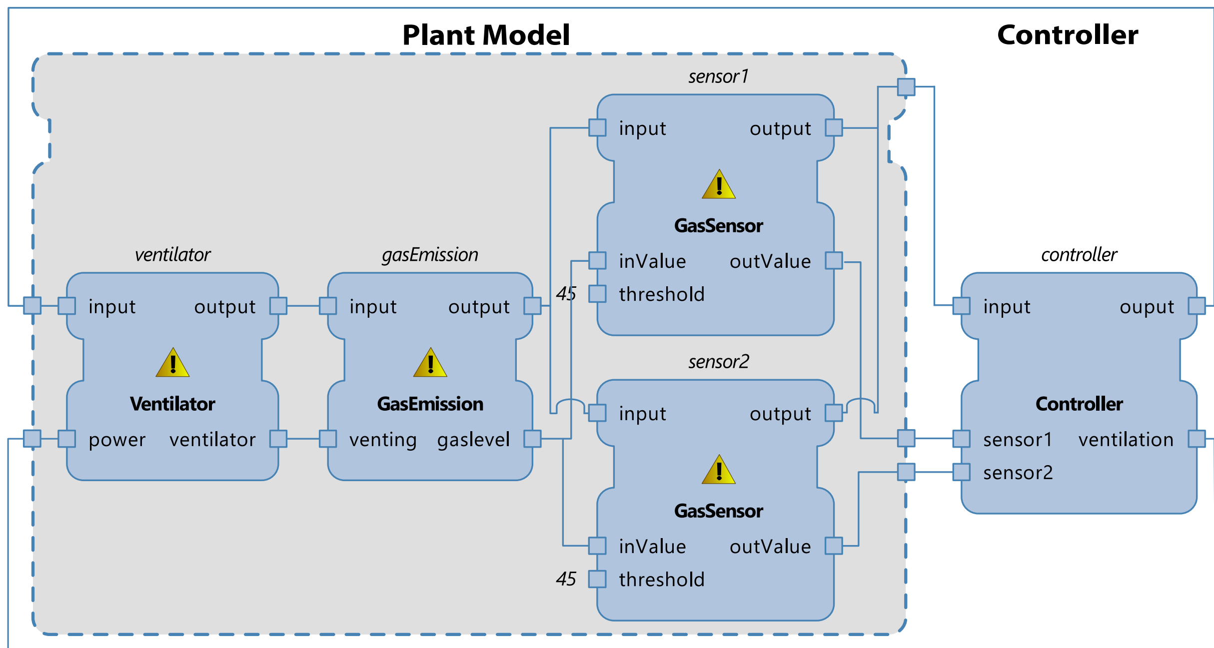


Figure 6.2: IEC 61499 implementation of the hazardous gas detection system using stochastic function blocks

the plant and the controller. The plant-model is created in a component-oriented fashion such that, each component in the plant-model represents the behaviour of a corresponding physical component in the system or the environment. Using stochastic function blocks, these components were modelled to contain their respective stochastic behaviours derived from the type of failure and their respective failure rates.

A process diagram presented in Figure 6.3 that illustrates the proposed approach for model-based safety. The said process begins with the IEC 61499 system model, which uses stochastic function blocks (SFB) for the plant-model. On the other hand, the controller consists of deterministic BFBs. Figure 6.2 shows the plant and controller connected in

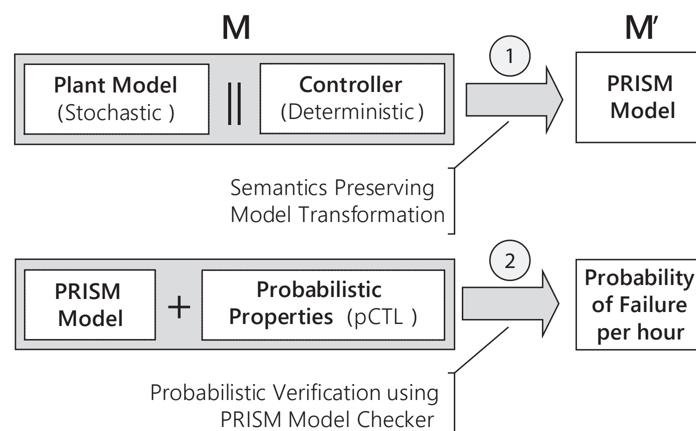


Figure 6.3: Process diagram for the proposed approach

a closed-loop system, which is used to automatically derive Markov decision processes (MDP) using a set of semantics-preserving transformation rules. Thus, a sound representation of the overall system model is obtained, which can be analysed for safety requirements through an automatic verification methodology called probabilistic verification [112]. These safety requirements are expressed as *probabilistic linear temporal logic* using the Prism property specification syntax [113] and the result of this analysis is a quantitative value e.g., estimated probability of satisfying the given requirement.

6.2 Stochastic Function Blocks (SFB)

Stochastic function blocks (SFB) are based on IEC 61499 basic function blocks (BFB). Similar to BFBs, SFBs are also encapsulated by an interface with explicitly declared IO. However, the behaviour of an SFB is implemented using a stochastic ECC, which unlike the ECC of a BFB, can also have stochastic states that have probabilistic or non-deterministic egress transitions. The syntactic sugar for SFBs renders them non-compliant with the IEC 61499 standard, however, SFBs are only used for modelling the failure-affected behaviours in the plant-model and the subsequent safety assessment. The controller on the other hand, is still implemented using the standard IEC 61499 function blocks and is used for the eventual automatic code generation and deployment. This renders the overall model-driven development of the controller software as standard compliant.

SFBs combine probabilistic and non-deterministic behaviours similar to that of MDPs with the nominal behaviours of BFBs. Consider the example of the GasSensor SFB from the hazardous gas detection system presented in Figure 6.4. It represents a *gas sensor* component in the plant model of the hazardous gas detection system presented earlier in this chapter. This SFB models both the nominal, as well as the failure related behaviour of the component. The nominal behaviour of the component is to detect the gas levels and transmit a signal when a threshold value is exceeded e.g., if a hazardous gas exceeds *45 parts per million*. The failure related behaviour comprise two separate types of failures. Firstly, in the *idle* mode of operation, the component can enter a permanent failure due to environmental stress. This failure is probabilistic in nature as represented by the WAITING state. Secondly, a request for input can some times be randomly ignored by the component, which depicts an transient *omission* failure. This failure is modelled using the non-deterministic state ONDEMAND. Using this example, we formally define the structure and semantics of SFBs in the subsequent sections.

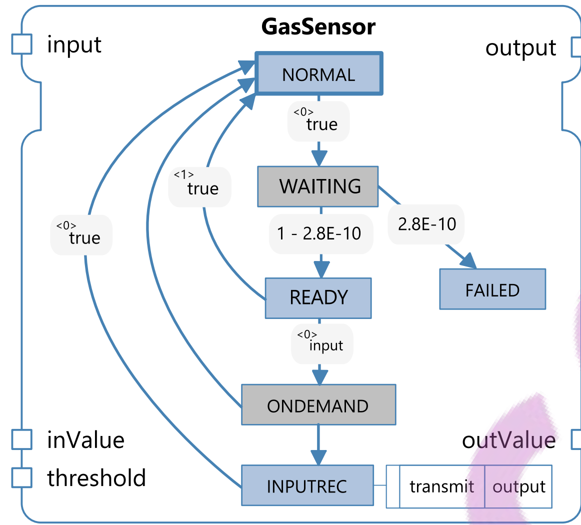


Figure 6.4: *GasSensor* stochastic function block

6.2.1 Formalisation

SFBs extend the definition of BFBs (Definition 5.2.3). Like BFBs, SFBs also have a function block interface ($\mathcal{I}$) with inputs and outputs events/variables, local declaration ($L^{\mathcal{I}}$) containing algorithms and internal variables, and a stochastic ECC ($\texttt{ECC}^{\mathcal{I},L^{\mathcal{I}}}$). Unlike the ECC of BFBs, a stochastic ECC may also contain probabilistic and non-deterministic transitions. Furthermore, we distinguish between three type of states namely, deterministic, probabilistic, and non-deterministic states. The purpose of this partitioning of the set of all states (Q) is to formally arrange probabilistic and non-deterministic transitions. That is, the set of all egress transitions from a probabilistic state is a probability distribution. Similarly, all egress transitions from a non-deterministic state must be non-deterministic. We define the structure of SFB as follows.

Definition 6.2.1 (Stochastic Function Block)**.** Stochastic function block (SFB) is a tuple, $\texttt{SFB} = \langle \mathcal{I}, L^{\mathcal{I}}, \texttt{ECC}^{\mathcal{I},L^{\mathcal{I}}} \rangle$ such that, $L^{\mathcal{I}} = \langle V_L^{\mathcal{I}}, A_L^{\mathcal{I}} \rangle$ is the local declaration over a function block interface $\mathcal{I} = \langle E_I^{\mathcal{I}}, V_I^{\mathcal{I}}, E_O^{\mathcal{I}}, V_O^{\mathcal{I}} \rangle$ and $\texttt{ECC}^{\mathcal{I},L^{\mathcal{I}}} = \langle Q, q_0, X, T, P, R \rangle$ is the execution control chart such that,

- Q is a non-empty finite set of states, partitioned into Q^p the set of probabilistic states, Q^n the set of non-deterministic states and Q^d the set of deterministic states i.e., $Q = Q^d \uplus Q^p \uplus Q^n$.
- $q_0 \in Q^d$ is the initial state, that must be a deterministic state.
- $X : Q^d \to 2^{(A_L^{\mathcal{I}} \cup E_O^{\mathcal{I}})}$ is the action function that assigns a finite set of algorithms and output events to a given state $q \in Q^d$.

- $T : Q^d \rightarrow 2^{(E_I^{\mathcal{I}} \cup \{\texttt{true}\}) \times \mathcal{B}(\hat{V}) \times Q}$ is the transition function for deterministic states where, $\hat{V} = (V_I^{\mathcal{I}} \cup V_O^{\mathcal{I}} \cup V_L^{\mathcal{I}})$ is the set of input, output and internal variable and $\mathcal{B}(\hat{V})$ is the set of all Boolean expressions over set of variables $\hat{V}$. We use the notation (q, e, b, q') to represent an element $t \in T(q)$ such that $q \in Q^d \wedge q' \in Q$. A transition is enabled if the corresponding event $e \in E_I^{\mathcal{I}}$ is present and the Boolean expression $b \in \mathcal{B}(\hat{V})$ evaluates to true. For every $q \in Q^d$, $T(q)$ is always an ordered set, i.e., for any two elements $t_1,\ t_2 \in T(q)$ we have $(t_1 > t_2) \vee (t_1 < t_2)$. We capture this order/priority of transitions graphically using the notation `<n>`, where is `n` is the index of an element in the order set with 0th index being the highest order.

- $P : Q^p \rightarrow 2^{Q^d}$ is the non-empty transition function for probabilistic states such that, for any $q \in Q^p$ the set of probabilistic transitions $P(q)$ is a probability distribution. Where, a probability distribution over a set X is a function $f : X \rightarrow [0, 1]$ such that $\sum_{x \in X} f(x) = 1$. Due to the non-empty nature of P, a probabilistic state cannot be a terminal state.

- $R \subseteq Q^n \times Q^d$ is the transition relation for non-deterministic states.

The transition function T, P and R classify the tree types of transitions: deterministic, probabilistic, and non-deterministic. The definition of transition function T enforces that states in Q^d can only have **deterministic** transitions whereas, P enforces that states in Q^p only have **probabilistic** transitions. Furthermore P imposes a restriction similar to Markov chains [63] that the probabilities on transitions are dependent on the current state only. Similarly, R dictates that states in Q^n can only have **non-deterministic** transitions. These restrictions result in an *alternate model* [63] for the ECCs of an SFB.

Observation 6.2.1 (Alternate Model)**.** *Because of the range of P and R in definition 6.2.1, every stochastic transition must have a deterministic successor state.*

From this observation, we see that in any execution trace of an SFB, any two successive stochastic transitions have one or more deterministic transitions in between. Figure 6.4 shows the execution control chart of the *GasSensor* SFB. This example illustrates the enforced alternate model such that, in all possible execution traces, a stochastic transition is followed by a deterministic transition.

6.2.2 Execution of SFBs

The execution of SFBs obeys the synchronous execution semantics using the logical time intervals named *ticks*. In this execution the *deterministic* states represent the tick-

boundary i.e., starting from the initial state $q_0 \in Q^d$ every tick begins from a deterministic state and ends in a deterministic state. Due to the enforced alternate model, a tick can at most take one stochastic step before stopping at the deterministic successor state. The execution of stochastic steps is performed using statistical simulation. In such an execution, a random number generator is used to perform non-deterministic selection between the available choice of commands. Similarly, Monte Carlo simulation could also be used to perform a probabilistic selection from the available choice of commands. In the context of our current work, SFBs are only used for modelling stochastic behaviours and not for execution of automatic code generation. Therefore, a detailed discussion on these statistical methods is beyond the scope of this thesis.

Algorithm 6.1: Execution of a single *tick* of a Stochastic Function Block

```
1  Function SBTick(SFB):
2      if cState = null then
           /* Initialisation tick does not read inputs                          */
3          cState ← SFB.q_0
4      else
           /* Load input events and variables from previous tick                */
5          loadInput(pre E_I^I, pre V_I^I);
6      end
       /* Iterate over the ordered set of transitions                           */
7      for t ∈ getTransitions(cState) do
           /* The first enabled transition will be of the highest priority      */
8          if isEnabled(t) then
9              sState ← getSuccessor(t);
10             if sState ∈ Q^p then
                   /* The successor is a probabilistic state                    */
11                 P ← getTransitions(sState);
12                 p ← pSelect(P);                   /* probabilistic selection */
13                 cState ← getSuccessor(p);
14             else if sState ∈ Q^n then
                   /* The successor is a nodeterministic state                  */
15                 R ← getTransitions(sState);
16                 r ← ndSelect(R);               /* nodeterministic selection */
17                 cState ← getSuccessor(r);
18             else
                   /* The successor is a deterministic state                    */
19                 cState ← sState;
20             end
               /* Execute algorithms of the successor state                     */
21             executeAlgos (cState);
               /* Emit output events and variables                              */
22             emitOutput(E_O^I, V_O^I);
23             return ;              /* break out of for loop and exit function */
24         end
25     end
26     return;
```

BESTPFE.COM List of research project topics and materials

During each tick, a given SFB executes as depicted in Algorithm 6.1. Lines $2-3$ represent the beginning of execution from the initial state q_0. After the initialisation tick, inputs are loaded in each tick from the encapsulating function block interface as shown in line 5. This loading of inputs is performed in a unit-delayed manner, such that the values are loaded from the previous tick as indicated by the keyword `pre`. Lines $7-24$ iterate over a sorted set of all deterministic transitions from the current state. If an enabled transition is found in this set of egress transitions, it is processed further. At the end of this processing, the loop is broken (see line 23) to ensure that only no more enabled transitions are processed. The processing of the selected enabled transition depends on the type of successor state. If a stochastic transition is found, a next step transition is selected either probabilistically (line 12) or non-deterministically (line 16). This stochastic transition necessarily has a deterministic successor state due to Observation 6.2.1. Regardless of the type of transition loaded, a deterministic successor state `cState` is eventually located (see lines 13, 17, and 19). The action set of this deterministic successor state is executed (see lines $21-22$), and it becomes the current state for the next tick. This concludes one tick process of an SFB.

6.2.3 Composition of Stochastic Function Blocks

Similar to BFBs, SFBs can also be connected with each other using the IO defined on their FB interfaces. The resultant network of SFBs execute in lock-step with each other. This mode of composition allows mixing BFBs, CFBs, and SFBs together in the form of a hierarchy.

Definition 6.2.2 (Composition of Stochastic Function Blocks)**.** Composition of a finite set of SFB instances ($\texttt{FBs} = \{\texttt{SFB}_1, \texttt{SFB}_2, \cdots, \texttt{SFB}_n\}$) forms a function block network $\texttt{FBNetwork} = \langle \texttt{FBs}, C_{\texttt{e}}, C_{\texttt{v}} \rangle$. Where `FBNetwork` has previously been defined in Definition 5.2.4.

The structure of SFB closely resembles that of BFBs, however due to the provision of stochastic transitions, SFBs are more expressive than BFBs. In fact it can be shown that SFBs are a strict superset of BFBs i.e., BFBs are SFB without any stochastic behaviour.

Observation 6.2.2 (BFB is an SFB)**.** *An SFB with an execution control chart* $\texttt{ECC}^{\mathcal{I},L^{\mathcal{I}}} = \langle Q, q_0, X, T, \varnothing, \varnothing \rangle$ *such that,* $Q^n \cup Q^p = \varnothing$*, is a BFB. This essentially means that a BFB is a special case of SFB such that, it only consists of deterministic states.*

Observation 6.2.3 (FBN Comprises of SFBs only)**.** *An IEC 61499 system containing the plant-model and the controller, can be represented as a network of SFBs. This follows*

from the Observations 5.2.1, 6.2.2 i.e., an IEC 61499 system containing CFBs can be flattened to an FBN consisting only SFBs.

6.3 Transformation to PRISM

In Chapter 5, we presented a set of rules (see Section 5.4 Rules T1–T6) to convert BFBs into equivalent Prism modules. A summary of these rules is presented in Tables 6.1. We further demonstrated the soundness of the rules and presented a proof of equivalence of the given FBN with a Prism model that was generated using these rules. In this section, we inherit these transformation rules and extend them for the purpose of converting SFBs into Prism modules. This extension includes further qualification to specify the state partition e.g., Rules T1–T6 only apply to the deterministic states ($q \in Q^d$). Furthermore, we also present additional rules, Rules T7–T8, to transform the stochastic behaviours of SFB into probabilistic and non-deterministic Prism commands in the subsequent sections.

Table 6.1: Rule guide for mapping SFB structure to Prism using transformation rules

FB	**PRISM**	**Rule**	**Lines (Figure 6.6)**
States	State encoding variable	*T1*	2
Output events	Boolean variables	*T2*	5
Output & Local variables	Integer and Boolean variables	*T3*	7
Initial state actions	Initialisation command	*T4*	8
Deterministic transitions	Prism command	*T5*	17 – 19
Implicit stay operation	Prism command self-loop	*T6*	21
Probabilistic transitions	Prism command	*T7*	9 – 11
Non-deterministic transitions	Prism commands	*T8*	12 – 16

6.3.1 Illustration

We begin by presenting an illustration of the transformation process using the GasSensor SFB shown in Figure 6.2. The result of applying Rules T1–T8 on this SFB is presented in Figures 6.5 and 6.6, which shows line numbers generated against the elements of the SFB. For example, the output event is mapped to line 5 because of Rule T2. Similarly, Rule T6 is used to generate line 21, which emulates the implicit stay operation on the FAILED state. Furthermore, the two groups of transitions outlined in green and purple indicate that each of the group was processed together. The purple region shows a deterministic transition followed by two probabilistic transitions, which is processed using Rule T7 to generate a single Prism command shown on lines 9 – 11. Similarly, the green region contains a

non-deterministic choice followed by a deterministic transition, which is processed using Rule T8. Here, each non-deterministic transition inherits its predecessor's Boolean guard to induce non-determinism in the generated commands e.g., lines $12-13$ and $14-16$ both use the same Boolean guard.

Figure 6.5: GasSensor SFB annotated with rules and lines numbers (see Figure 6.6)

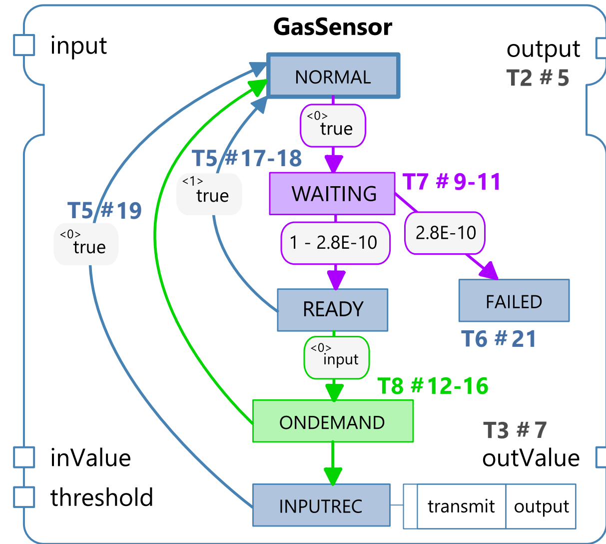


Figure 6.6: Prism modules generated for the GasSensor SFB (see Figure 6.5)

```
1   module sensor1
2       s : [-1..3] init -1;
3       //s = {0, 1, 2, 3} : {NORMAL, READY, FAILED, INPUTREC}

4       //Generated from output events
5       output_sensor1 : bool init false;

6       //Generated from output variables
7       outValue_sensor1 : bool init false;

8       [t] (s_sensor1=-1) -> (s_sensor1'=0) & (output_sensor1'=false);

9       [t] (s_sensor1=0) ->
10              2.8E-10 : (s_sensor1'=2) & (output_sensor1'=false) +
11              1 - 2.8E-10 : (s_sensor1'=1) & (output_sensor1'=false);

12      [t] (s_sensor1=1) & (input_src) -> (s_sensor1'=0)
13              & (output_sensor1'=false);

14      [t] (s_sensor1=1) & (input_src) -> (s_sensor1'=3)
15              & (output_sensor1'=true)
16              & (outValue_sensor1'=(inValvue_src > 45));

17      [t] (s_sensor1=1) & (input_src=false) -> (s_sensor1'=0)
18              & (output_sensor1'=false);

19      [t] (s_sensor1=3) -> (s_sensor1'=0) & (output_sensor1'=false);

20      //Generated self-loops for emulating synchronous execution semantics
21      [t] (s_sensor1=2) -> (s_sensor1'=2) & (output_sensor1'=false);
22  endmodule
```

6.3.2 Semantics Preserving Rule-Based Transformation

The transformation process accepts a given FBN and generates a corresponding PRISM model. We know from our earlier Observation 5.2.1 that any given FBN can be flattened into a network of BFBs. Furthermore, from Observation 6.2.2 we know that every BFB is a special case of SFB. Without loss of generality, we assume in this chapter that an FBN is a network of SFBs. Therefore, the transformation process is reduced to map every SFB instance in the given FBN to a corresponding Prism module using the extended transformation rules i.e., the following.

$$\texttt{SFB}_i \in \texttt{FBNetwork} \Longleftrightarrow M_i \in \mathcal{M} \tag{6.3.1}$$

6.3.3 Modifying Inherited Rules for SFBs

Transformation rules for SFBs are based on the Rules T1–T6 that were presented in Section 5.4. These rules relied on a set of macros for the conversion process. Table 6.2 provides a brief summary of these macros for convenience of the readers.

Table 6.2: Summary of macros used in Rules T1–T6

Name	Purpose	Section
nameOf	Returns the fully-qualified name of the given BFB	Section 5.4.1.1
variable	A macro for definition of a variable in Prism syntax	Section 5.4.1.2
len	Returns the number of items in a given set	
typeOf	Maps the type of a variable in BFB to its Prism equivalent type	Section 5.4.1.2
rangeOf	Determines the range of a given variable in BFB	Section 5.4.1.2
initOf	Determines the initial value of a given variable in BFB	Section 5.4.1.2
valueOf	Assigns a numeric value to a given element e.g., based on its declaration index	Section 5.4.1.3
checkState	Generates an expression to evaluate the state-encoding variable against a given ECState	Section 5.4.1.3
updateState	Generates an assignment statement to update the value of the state-encoding variable to represent a given successor ECState	Section 5.4.1.3
setStatus	Sets the *present* / *absent* status of events by generating assignment statements for the corresponding Boolean variables	Section 5.4.2.2
stateActions	Returns assignment statements generated from state-entry actions of a given ECState	Section 5.4.2.3
ensureOrder	Takes a transition and preserves its order by appending its Boolean condition guard with negated condition guards of higher order transitions	Section 5.4.2.4
command	A macro for declaration of a generic Prism command	Section 5.4.2.6
negateAll	Constructs a Boolean expression that comprises negation of condition guards of all egress transitions from the given ECState. This is required to create a lowest-order self-loop command that only executes when all egress transitions are disabled	Section 5.4.2.9

Rules T1–T6 were developed for transforming BFBs in to Prism modules. For our purpose, we modify these rules to conform to the structure of SFBs e.g., to accommodate the three types of states in SFB. The modifications to the rules are described as follows.

- Rule T1 previously defined a state-encoding mechanism that used distinct integer values to encode ECState of a BFB. We now modify it such that only the deterministic states ($q \in Q^d$) of a given SFB are encoded. The encoding of probabilistic and non-deterministic states is not needed as they get absorbed into generated Prism commands.

$$\mathsf{variable}\Big(\mathsf{nameOf(s)}, \text{INT}, [-1, \mathsf{len}(Q^d) - 1], -1\Big) \quad \textbf{(T1)}$$

- Rules T2–T3 perform event and variable encoding for the given SFB interface, and remain unchanged from Section 5.4.1.

$$\mathsf{variable}\Big(\mathsf{nameOf}(e), \text{BOOL}, \top, \texttt{false}\Big), \quad where\ e \in E_O^{\mathcal{I}} \quad \textbf{(T2)}$$

$$\mathsf{variable}\Big(\mathsf{nameOf}(v),\ \mathsf{typeOf}(v),\ \mathsf{rangeOf}(v),\ \mathsf{initOf}(v)\Big)\ \ where\ v \in V_O^{\mathcal{I}} \cup V_L^{\mathcal{I}} \quad \textbf{(T3)}$$

- Rule T4 encodes the Moore-like initial actions (see Section 5.4.2.7). The rule remains unchanged except that the initial ECState must be a deterministic state ($q_0 \in Q^d$) as dictated by Definition 6.2.1.

$$\mathsf{command}(\{\text{‘s==-1’}\}, \mathsf{updateState}(q_0) \cup \mathsf{stateActions}(q_0))\ \ where\ q_0 \in Q^d \quad \textbf{(T4)}$$

- Rule T5 previously encoded all transitions of BFBs. We modify this rule to encode only the deterministic transitions of SFBs i.e., egress transitions from deterministic states as Prism commands. The modification of this rule qualifies that for a given transition $t = (q, e, b, q') \in T(q)$, both the predecessor q as well as the successor state q' are deterministic states i.e., $q, q' \in Q^d$. This ensures that the rule only processes deterministic transitions. The probabilistic and non-deterministic transitions of SFBs are processed using Rules T7 and T8 introduced in the later sections.

$$\mathsf{command}\Big(\mathsf{checkState}(q) \cup \mathsf{ensureOrder}(t),\ \mathsf{updateState}(q') \cup \mathsf{stateActions}(q')\Big)$$
$$where\ (q, e, b, q') \in T(q),\ and\ q, q' \in Q^d \quad \textbf{(T5)}$$

- Rule T6, as before, helps avoiding deadlocks by simulating implicit stay operation of synchronous semantics. However, this rule is now further qualified to only induce these self-loop commands on deterministic states ($q \in Q^d$). Since probabilistic and

non-deterministic states are absorbed during transformation, this rule is not needed for these states. Furthermore, the modified rule ensures that a self-loop is only created when a deterministic state does not already have a unconditional egress transition $(q, \texttt{true}, \varnothing, q') \notin T(q)$, since otherwise the implicit stay operation is not possible.

$$\mathsf{command}\Big(\mathsf{checkState}(q) \cup \mathsf{negateAll}(q),\ \mathsf{updateState}(q) \cup \mathsf{setStatus}(\varnothing,\ E_O^{\mathcal{I}})\Big)$$
$$\textit{where } q \in Q^d \ \wedge \ (q, \texttt{true}, \varnothing, q') \notin T(q) \qquad \textbf{(T6)}$$

These modifications to Rules T1–T6 are minimal and therefore, these rules are simply inherited from the previous chapter. In this section, we further develop two more rules that are specific to probabilistic and non-deterministic transitions of SFBs. The subsequent sections present the definition and illustrations of these rules.

6.3.4 Encoding Stochastic Transitions

Transitions of a given SFB are converted to commands in the generated Prism module. This conversion is straightforward for *Deterministic transitions*, which are converted using Rule T5. The stochastic transitions of a given SFB i.e., the *Probabilistic* and *Non-deterministic* transitions are also converted into to commands using the two new additional rules that are defined in this section, namely Rules T7 and T8.

6.3.4.1 Probabilistic Update Pairs

A Prism command (see Definition 5.3.2) can be viewed as a probabilistic distribution over update statements. That is, given a Prism command $c = (\texttt{t}, g, U)$, the set of probabilistic update pairs $U = \{(\lambda_0, u_0), (\lambda_1, u_1), ..., (\lambda_n, u_n)\}$ have associated probability values $\lambda_0, ..., \lambda_n$, where $\sum_{i=0}^{n} \lambda_i = 1$. An update pair (λ_i, u_i) represents that the update statements in u_i shall be executed with a probability of λ_i. This probabilistic choice is similar to the probabilistic states of SFBs where a state $q' \in Q^p$ has a set of probabilistic transitions $P(q')$. Elements of this set are of the form (q', p_j, q''_j) that represent a probabilistic transition from q' to q''_j with a probability p_j. Due to the alternate model (see Observation 6.2.1), we know that the successor states q''_j of all probabilistic transition in $P(q')$ are deterministic. The corresponding action sets of these successor states can be converted into a fragment of Prism command i.e., the set of probabilistic update pairs U. We define a utility macro named **probSet** to generate these update pairs from a given probabilistic

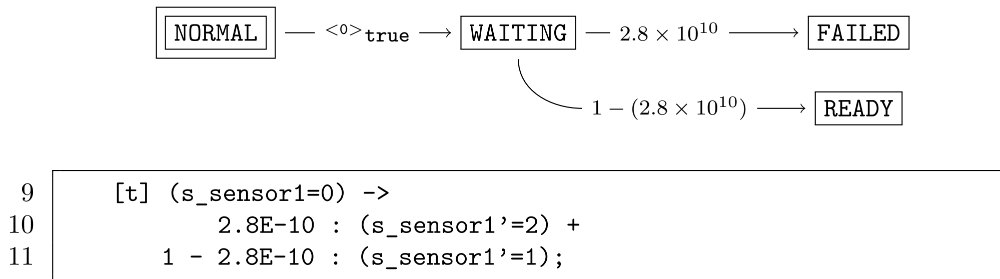
state q' as follows.

$$\text{Given, } P(q') = \{(q', p_0, q_0''), ..., (q', p_n, q_n'')\}$$
$$\textit{where } q' \in Q^p \textit{ and } q_0'', ..., q_n'' \in Q^d \text{ and } \sum_{i=0}^{n} p_i = 1$$
$$\text{We define, } \mathsf{probSet}(q') = \bigcup_{i=0}^{n} \Big(p_i,\ \mathsf{convExpr}\Big(\mathsf{updateState}(q_i'') \cup \mathsf{stateActions}(q_i'')\Big)\Big)$$

In the above definition, each deterministic successor state $q_i'' \in Q^d$ is used to generate a set of assignment statements using macros updateState and stateActions. The first macro generates the assignment statement that updates the value of the state-encoding variable to the encode ECState q_i''. The second macro generates assignment statements that represent the state-entry actions for q_i'' namely, the algorithms invocations and emissions of output events. The generated assignment statements are then subjected to syntax translation using the convExpr macro. Combined together, these assignment statements are associated with a probability value p_i taken from the probabilistic transition (q', p_i, q_i'') to form the set of probabilistic update pairs U.

6.3.4.2 Probabilistic Transitions

Probabilistic transitions are converted using Rule T7 by collapsing a set of probabilistic transitions into a single Prism command. Due to the alternate model (see Observation 6.2.1), it is guaranteed that a probabilistic choice from a given probabilistic state $q' \in Q^p$ is preceded by a deterministic transition e.g., we have a deterministic state $q \in Q^d$ with an egress deterministic transition $t = (q, e, b, q') \in T(q)$, which is followed by a set of probabilistic transitions given by $P(q')$. An example of this arrangement of transitions is shown in Figure 6.7 where, a deterministic transition (NORMAL $\xrightarrow{\texttt{<0>true}}$ WAITING) is followed by two




```
9    [t] (s_sensor1=0) ->
10          2.8E-10 : (s_sensor1'=2) +
11      1 - 2.8E-10 : (s_sensor1'=1);
```

Figure 6.7: Illustration of Rule T7 showing probabilistic SFB transitions above and the equivalent Prism command below

probabilistic transitions ($\texttt{WAITING} \xrightarrow{2.8\times10^{10}} \texttt{FAILED}$) and ($\texttt{WAITING} \xrightarrow{1-(2.8\times10^{10})} \texttt{READY}$). Rule T7 converts such arrangements of transitions by combining them together to form a single probabilistic Prism command. An illustration of this rule is shown in Figure 6.7 where the probabilistic transitions are converted into a single Prism command shown as lines $9-11$. We define this behaviour using the command macro as follows.

$$\begin{aligned}&\text{Given, } t=(q,e,b,q')\in T(q)\ s.t.,\ q\in Q^d \text{ and } q'\in Q^p\\&\text{Yeild a command: } \mathsf{command}\Big(B,\ \mathsf{probSet(q')}\Big)\end{aligned}\qquad\textbf{(T7)}$$

Where, $B = \mathsf{checkState}(q)\ \cup\ \mathsf{ensureOrder}(t)$ is the set of Boolean expressions used for creating the guard conditions of the generated Prism command. These expressions are computed using the two macros checkState and ensureOrder. The macro checkState induces a location in the generated Prism modules using Boolean expression 's==valueOf(q)', which uses the state-encoding variable s (translated to `s_sensor1`) and the encoded value of ECState q. For further detail on encoding ECStates, see Section 5.4.1.3. Additionally, the ensureOrder macro acquires the condition guard from the transition t and preserves its order by using explicit negations (see Section 5.4.2.7). The conjunction of these Boolean expressions gives us the Boolean condition guard g of the generated Prism command. Whereas, the set of assignment statements are generated from the probabilistic successor state q' using the probSet macro as described previously.

The Boolean expressions in B are converted into Prism syntax using the convExpr macro, whereas the set of probabilistic update pairs generated by probSet macro are already converted to Prism syntax. Based on this syntax translation, we yield a Prism command $c = (\mathtt{t}, g, U)$, where $\mathtt{t}$ is the synchronisation label for tick, $g = \bigwedge_{b\in B} \mathsf{convExpr}(b)$, and $U = \mathsf{probSet}(q')$.

6.3.4.3 Non-deterministic Transitions

Non-deterministic transitions are transformed using Rule T8, which creates a set of Prism commands that induce a non-deterministic choice. Similar to the Rule T7, this rule also relies on the alternate model for selecting consecutive pairs of deterministic and non-deterministic transitions. Given a transition $t = (q, e, b, q') \in T(q)$ with $q \in Q^d$ and $q' \in Q^n$, the rule locates all next step transitions from the non-deterministic state q' namely, $R(q')$ and creates a new command for each non-deterministic transition $(q', q''_0), ..., (q', q''_m) \in R(q')$. This rule can be understood from the illustration given in Figure 6.8 where a set of non-deterministic transitions are translated into equivalent Prism

Bestpfe.com

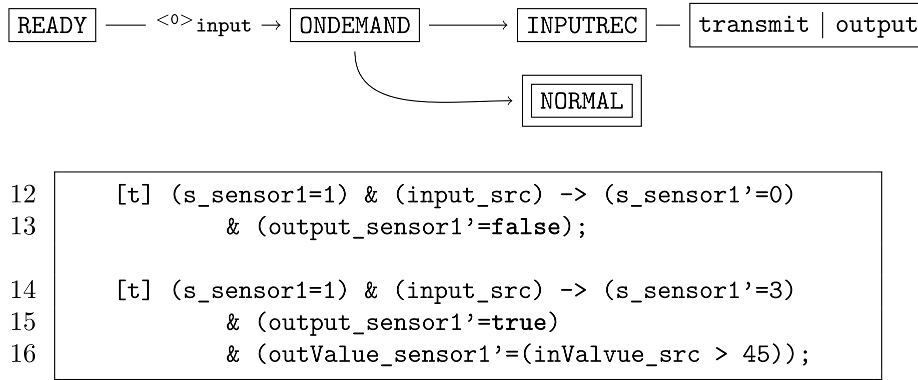


Figure 6.8: Illustration of Rule T8 showing non-deterministic SFB transitions above and the equivalent Prism commands below

commands (lines 12 – 16). This illustration shows that the two non-deterministic transitions ($\texttt{READY} \xrightarrow{\texttt{input}} \texttt{ONDEMAND} \rightarrow \texttt{INPUTREC}$) and ($\texttt{READY} \xrightarrow{\texttt{input}} \texttt{ONDEMAND} \rightarrow \texttt{NORMAL}$) are mapped to two individual Prism commands i.e., commands on lines 12 – 13 and 14 – 16 respectively. Here, both of the generated commands use the same guard condition, thus resulting in a non-deterministic choice during the execution.

We define Rule T8 as follows.

$$\begin{aligned}&\text{Given, } t = (q, e, b, q') \in T(q) \quad \text{and} \quad R(q') = \{(q', q''_0), ..., (q', q''_m)\}\\&\qquad \textit{where } q \in Q^d, q' \in Q^n \text{ and } q''_0, ..., q''_m \in Q^d\\&\text{Yeild a set of commands: } \{C_0, ..., C_m \mid C_i = \mathsf{command}(B, S_i) \quad \textit{s.t.,}\ i \in [0, m]\} \quad (\mathbf{T8})\end{aligned}$$

- $B = \mathsf{checkState}(q) \cup \mathsf{ensureOrder}(t)$ is the set of Boolean expressions used for creating the guard conditions of the generated Prism command as described previously (see definition of Rule T7). It creates a location based on the state encoding variable s translated to `s_sensor1` (see Section 5.4.1.3) and the input event input, which is translated to `input_src` (see Section 5.4.2.7),

- $S_i = \mathsf{updateState}(q''_i) \cup \mathsf{stateActions}(q''_i) \cdot \forall (q', q''_i) \in R(q')$ is the set of assignment statements generated from state-entry actions of a given deterministic state $q''_i \in Q^d$. Here, the macro updateState creates an assignment statement that updates the state-encoding variable to represent the successor ECState q''_i. Whereas, the macro stateActions generates a set of assignment statements that represent the algorithm invocations and output event emissions associated with the successor ECState q''_i.

The generated commands $C_0, ..., C_m$ bear the same condition guard as each other, thus resulting in non-deterministic choice between them. Such that, each non-deterministic

choice is associated with a set of assignment statements generated from a successor state q''_i of $(q', q''_i) \in R(q')$. Based on this syntax translation, we yield Prism commands of the form $c = (\texttt{t}, g, U)$, where $\texttt{t}$ is the synchronisation label for tick, $g = \bigwedge_{b \in B} \mathsf{convExpr}(b)$, and $U = \bigwedge_{s \in S_i} \mathsf{convExpr}(s)$.

6.3.5 Algorithm for Generating the Prism Model

Algorithm 6.2 uses the proposed transformation rules T1–T8 to convert a given FBN to a Prism model. It uses macros that have been defined in the previous sections. A summary of these macros is presented in Table 6.3. An illustration of Algorithm 6.2 is presented in Table 6.4. In this illustration, we used the Prism module generated against the sensor1 instance of the GasSensor SFB presented in Figure 6.6. The said module is divided into several segments, where each segment is generated by a particular set of lines of the said algorithm. This is shown in form of a label visible on the right-hand side.

The illustration in Table 6.4 shows that the line 3 of Algorithm 6.2 generated a

Table 6.3: Summary of the macros used in Algorithm 5.4

Name	Purpose	Section
nameOf	Returns the fully-qualified name of the given BFB, event or variable	Section 5.4.1.1
variable	A macro for definition of a variable in Prism syntax	Section 5.4.1.2
len	Returns the number of items in a given set	
typeOf	Maps the type of a variable in BFB to its Prism equivalent type	Section 5.4.1.2
rangeOf	Determines the range of a given variable in BFB	Section 5.4.1.2
initOf	Determines the initial value of a given variable in BFB	Section 5.4.1.2
valueOf	Assigns a numeric value to a given element e.g., based on its declaration index	Section 5.4.1.3
checkState	Generates an expression to evaluate the state-encoding variable against a given ECState	Section 5.4.1.3
updateState	Generates an assignment statement to update the value of the state-encoding variable to represent a given successor ECState	Section 5.4.1.3
setStatus	Sets the *present* / *absent* status of events by generating assignment statements for the corresponding Boolean variables	Section 5.4.2.2
stateActions	Returns assignment statements generated from state-entry actions of a given ECState	Section 5.4.2.3
ensureOrder	Takes a transition and preserves its order by appending its Boolean condition guard with negated condition guards of higher order transitions	Section 5.4.2.4
command	A macro for declaration of a generic Prism command	Section 5.4.2.6
negateAll	Constructs a Boolean expression that comprises negation of condition guards of all egress transitions from the given ECState. This is required to create a lowest-order self-loop command that only executes when all egress transitions are disabled	Section 5.4.2.9
probSet	Constructs the set of probabilistic update pairs from a given probabilistic ECState	Section 5.4.2.9

BEST PFE.COM List of research project topics and materials

unique name `sensor1` for the Prism module using the FB instance name. Line 4 of Algorithm 6.2 generated the state-encoding variable with a range $[-1, 3]$, such that values: $0, 1, 2, 3$ correspond to ECStates: NORMAL, READY, FAILED, INPUTREC and the value -1 corresponds to an *initialization command*. Lines $5-7$ and $8-10$ of Algorithm 6.2 generated variables for output events and output variables respectively. This involved generating unique names for the variables, and assigning them appropriate type and value range.

The presented Prism module contains five regions containing generated commands. Commands in each of these regions is generated by a different rule. Rules T5, T7 and T8 generate commands against existing transitions in the given SFB, where as rules T4 and T5 generate commands that solely for the purpose of preserving synchronous execution semantics i.e., they do not correspond to an SFB transition. Rule T4 is implemented by

Table 6.4: Illustration of Algorithm 6.2 on the Prism module generated from the sensor1 instance of GasSensor SFB shown in Figure 6.2

Line	Code	Source
1	`module sensor1`	**Algorithm 6.2 :** 4
2	`s : [-1..2] init -1;`	
3	`//s = {0, 1, 2, 3} : {NORMAL, READY, FAILED, INPUTREC}`	**Algorithm 6.2 :** 6
4	`//Generated from output events`	
5	`output_sensor1 : bool init false;`	**Algorithm 6.2 :** 8 – 10
6	`//Generated from internal and output variables`	
7	`outValue_sensor1 : bool init false;`	**Algorithm 6.2 :** 12 – 14
8	`[t] (s_sensor1=-1) -> (s_sensor1'=0) & (output_sensor1'=false);`	**Algorithm 6.2 :** 16 – 18
9	`[t] (s_sensor1=0) ->`	
10	`2.8E-10 : (s_sensor1'=2) & (output_sensor1'=false) +`	
11	`1 - 2.8E-10 : (s_sensor1'=1) & (output_sensor1'=false);`	**Algorithm 6.2 :** 28 – 31
12	`[t] (s_sensor1=1) & (input_src) -> (s_sensor1'=0)`	
13	`& (output_sensor1'=false);`	
14	`[t] (s_sensor1=1) & (input_src) -> (s_sensor1'=3)`	
15	`& (output_sensor1'=true)`	
16	`& (outValue_sensor1'=(inValvue_src > 45));`	**Algorithm 6.2 :** 32 – 38
17	`[t] (s_sensor1=1) & (input_src=false) -> (s_sensor1'=0)`	
18	`& (output_sensor1'=false);`	
19	`[t] (s_sensor1=3) -> (s_sensor1'=0) & (output_sensor1'=true);`	**Algorithm 6.2 :** 24 – 27
20	`//Generated self-loops for emulating synchronous execution semantics`	
21	`[t] (s_sensor1=2) -> (s_sensor1'=2) & (output_sensor1'=false);`	**Algorithm 6.2 :** 41 – 45
22	`endmodule`	

Algorithm 6.2: Generating a Prism model against a given FBN

Input: `FBNetwork` –The given function block network
Output: $\mathcal{M}$ –The generated Prism model

```
1  Function Transform(FBNetwork)
2    foreach SFB_i ∈ FBNetwork do
       /* generating name for the new Prism module                          */
3      M_i.name ← nameOf(SFB_i)
       /* generating state-encoding variable (see Rule T1)                   */
4      M_i.V ← variable(nameOf(s), INT, [−1, len(Q) − 1], −1)
       /* generating variables against output events of SFB_i (see Rule T2)  */
5      foreach e ∈ SFB_i.E_O^I do
6        M_i.V ← variable(nameOf(e), BOOL, ⊤, false)
7      end
       /* generating variables against local and output variable of SFB_i (see
       Rule T3)                                                              */
8      foreach v ∈ SFB_i.V_O^I ∪ SFB_i.V_L^I do
9        M_i.V ← variable(nameOf(v), typeOf(v), rangeOf(v), initOf(v))
10     end
       /* generating command for initial state actions of  SFB_i (see Rule T4) */
11     initCondition ← {'s==-1'}
12     initUpdate ← updateState(q_0) ∪ stateActions(q_0)
13     M_i.C ← command(initCondition, initUpdate)
       /* iterate over all Deterministic ECStates of SFB_i                   */
14     foreach q ∈ SFB_i.Q^d do
         /* generating commands for all transitions in SFB_i                 */
15       foreach t = (q, e, b, q') ∈ T(q) do
16         transCondition ← checkState(q) ∪ ensureOrder(t)
17         if q' ∈ Q^d then
             /* encountered a deterministic sucessor (see Rule T5)           */
18           transUpdate ← updateState(q') ∪ stateActions(q')
19           M_i.C ← command(transCondition, transUpdate)
20         else if q' ∈ Q^p then
             /* encountered a probabilistic sucessor (see Rule T7)           */
21           probUpdate ← probSet(q')
22           M_i.C ← command(transCondition, probUpdate)
23         else if q' ∈ Q^n then
             /* encountered a non-deterministic sucessor (see Rule T8)       */
24           foreach (q', q''_k) ∈ R(q') do
25             kthUpdate ← updateState(q''_k) ∪ stateActions(q''_k)
26             M_i.C ← command(transCondition, kthUpdate)
27           end
28         end
29       end
         /* generating self-loop commands (see Rule T6)                      */
30       if (q, true, ∅, q') ∉ T(q) for any q' ∈ SFB_i.Q then
31         loopCondition ← checkState(q) ∪ negateAll(t)
32         loopUpdate ← updateState(q) ∪ setStatus(∅, E_O^I)
33         M_i.C ← command(loopCondition, loopUpdate)
34       end
35     end
36     M ← M_i /* adding the new module in the generated Prism model         */
37   end
38   return M
```

the lines 11 − 13 of Algorithm 6.2, which generated the init-command on line 8. Rule T6 is implemented by the line 30 − 34 of Algorithm 6.2, which generated a self-loop command on the line 21. Whereas, Rules T5, T7 and T8 generated commands against deterministic, probabilistic, and non-deterministic SFB transitions. Lines 15 − 29 of Algorithm 6.2 iterate over all transitions from a given deterministic state $q \in Q^d$ and processes three conditional blocks on the successor state q' i.e., the following.

1. **Deterministic Successor:** This is a trivial case as it represents a BFB-like transition without any stochastic behaviours. Encoding of the respective transition is performed using Rule T5 (see Section 5.4.2.7). Line 17 − 19 of Algorithm 6.2 handle this case.

2. **Probabilistic Successor:** In this case, all next-step probabilistic transitions are combined together using Rule T7 (see Section 6.3.4.2) to generate a probabilistic command. Line 20 − 22 of Algorithm 6.2 handle this case.

3. **Non-deterministic Successor:** In this case, all next-step non-deterministic transitions are processed using Rule T8 (see Section 6.3.4.3) to generate a set of non-deterministic commands. Line 23 − 27 of Algorithm 6.2 handle this case.

All commands generated by the algorithm ensure the life-time of events by explicitly setting the status of Boolean variables representing the output events e.g., `output_sensor1`. Algorithm 6.2 generates a Prism module for each SFB in the FBN (see lines 2 and 36), thus constructing the overall Prism model.

6.4 Preserving the Execution Semantics

In the previous chapter (see Section 5.5), we presented a proof that demonstrated a trace equivalence relation between a given FBN and the generated Prism model. However, the said proof was limited such that FBNs could only contain BFBs. In this chapter we have proposed SFBs, which can be used instead of BFBs to form FBNs. For this reason, we extend the proofs to address SFBs and the soundness of the proposed approach. Specifically, we aim to show that the application of rules T1–T8 on a given FBN results in a generated Prism model that preserve its synchronous execution semantics.

In this section we present arguments for soundness using the concept of execution states and transitions presented in Chapter 5. The definition of an execution state of a BFB and its equivalence with other execution states applies to SFBs without any change (see Definitions 5.5.1 and 5.5.3). Whereas, the addition of stochastic ECTransitions in

SFBs requires us to adapt the definition of transitions. Later in this section, we develop a proof of a *simulation* relation between a given SFB and the corresponding Prism module, which is sufficient for the soundness of the proposed approach.

6.4.1 Transition

The execution of an SFB passes through intermediate stochastic steps. Algorithm 6.1 represents one execution step/tick of an SFB, where line 19 evaluates a deterministic choice of successors to begin the transition. This transition may pass through intermediate non-deterministic or probabilistic steps using lines $10-13$ and $14-17$ respectively. However, this intermediate step necessarily ends with a deterministic successor state $q \in Q^d$ (from Observation 6.2.1). Due to synchrony hypothesis [114], we assume that FBs execute their ticks instantaneously, thus any intermediate execution states are not "observable" outside the execution of a tick. In synchronous languages e.g., Esterel [115], this is referred to as the so called *micro* and *macro* steps, where a macro-step may contain infinite non-observable micro-steps. Formally, we can represent this as follows.

$$\eta_i \overset{\rho_i}{\dashrightarrow} \eta_{i'} \overset{\tau}{\dashrightarrow} \eta_{i+1} = \eta_i \xrightarrow{\alpha_i} \eta_{i+1} \qquad \textit{where} \;\; \eta_i(q_c), \eta_{i+1}(q_c) \in Q^d \qquad (6.4.1)$$
$$\textit{and} \;\; \eta_{i'}(q_c) \in Q^p \cup Q^n$$

From Definition 6.2.1 the action function X applies to deterministic ECStates, therefore we observe that for any output event $e \in E_O^{\mathcal{I}}$, $\eta(e) = \eta'(e)$ and for any output or local variables $v \in V_O^{\mathcal{I}} \cup V_L^{\mathcal{I}}$, $\eta(v) = \eta'(v)$. On the other hand, a transition of a Prism module M $\sigma_i \xrightarrow{\gamma_i} \sigma_{i+1}$ represents an execution cycle comprising the following: (i) making a non-deterministic choice to select an enabled command, (ii) probabilistically selecting an update pair, and (iii) executing the assignment statements in the selected update pair. In this transition, both the successor and the predecessor states represent an evaluation of execution variables Var_M, which are thus *observable*. Additionally, from Rules T4-T8 we observe that only deterministic ECStates $q \in Q^d$ are encoded by the `updateState` macro. Thus, given two equivalent execution states $\eta = \sigma$ of an SFB and a Prism module respectively, we have $\sigma(\{s\}) = \mathsf{valueOf}(\eta(q_c))$ such that $\eta(q_c) \in Q^d$.

6.4.2 Simulation Relation

In this section we present arguments for simulation relation of a given FBN and the corresponding generated Prism model. This relation forms the basis of soundness of a

safety analysis approach proposed in a later section. We use the definitions of execution states and transitions to provide a by proof of the simulation of the individual SFBs by the corresponding generated Prism modules. This proof is used to further demonstrate that the composition of the individual SFBs and the Prism modules, namely the respective FBN and the Prism model also have a simulation relation.

Definition 6.4.1 (Simulation Relation)**.** Simulation of an SFB `SFB` by a Prism module M, is a binary relation $\mathcal{R} \subseteq Eval(Var_{\texttt{SFB}}) \times Eval(Var_M)$ such that, for all $(\eta, \sigma) \in \mathcal{R}$ the following holds:

1. The given execution states are equivalent $\eta{=}\sigma$
2. If a transition $\eta \xrightarrow{\alpha} \eta'$ exists, then $\sigma \xrightarrow{\gamma} \sigma'$ also exists, such that $(\eta',\sigma') \in \mathcal{R}$

Definition 6.4.2 (Simulation of an SFB)**.** A given SFB `SFB` is simulated by a corresponding generated Prism module M, denoted `SFB` $\preceq M$, if the initial execution states are similar such that $(\eta_0, \sigma_0) \in \mathcal{R}$.

Theorem 6.4.1. A given SFB `SFB` is simulated by the corresponding Prism module M generated by Rules T1-T8: `SFB` $\preceq M$

Proof. The proof of this theorem is provided by constructing a binary relation and demonstrating that it is a simulation. Also, we show that this relation contains the initial states of `SFB` and M and thus, `SFB` $\preceq M$

1. *Cross product:* Let $\mathcal{R}{=}Eval(Var_{\texttt{SFB}}){\times}Eval(Var_M)$
2. *Refinement:* Let $\mathcal{R}_s$ be a subset of $\mathcal{R}$ such that it comprises only the pairs of equivalent states i.e., $\mathcal{R}_s{=}\{(\eta_i,\sigma_j) \mid \eta_i{=}\sigma_j \wedge (\eta_i,\sigma_j){\in}\mathcal{R}\}$ (see Definition 5.5.3). From Rules T4-T8, we know that the `updateState` macro is only applied on deterministic ECStates . Therefore, for any $(\eta_i,\sigma_j){\in}\mathcal{R}_s$, we know that the state variables q_c and `s` represent a deterministic ECState i.e., $\sigma_j(\mathsf{s}){=}\mathsf{valueOf}(\eta_i(q_c))$ and $\eta_i(q_c){\in}Q^d$.
3. *Simulation relation:* $\mathcal{R}_s$ is a simulation relation, because for every element $(\eta_i,\sigma_i){\in}\mathcal{R}_s$, we know that $\eta_i{=}\sigma_i$ and we can show the following:

   (A) The initial states η_0 and σ_0 of `SFB` and M are equivalent (from Lemma 5.5.1). Therefore, the initial states are contained in $\mathcal{R}_s$ (from *Refinement*).

(B) If a transition $\eta_i \xrightarrow{\alpha_i} \eta_{i+1}$ exists, then $\sigma_i \xrightarrow{\gamma_i} \sigma_{i+1}$ also exists, and $(\eta_{i+1},\sigma_{i+1}) \in \mathcal{R}_s$.

Proof. If η_i is reachable ($\eta_i \in Reach(\texttt{SFB})$), a transition $\eta_i \xrightarrow{\alpha_i} \eta_{i+1}$ exists in `SFB` (see Definition 5.5.6). We shall establish that a corresponding equivalent transition $\sigma_i \xrightarrow{\gamma_i} \sigma_{i+1}$ exists in M such that, $\eta_{i+1} = \sigma_{i+1}$. This statement is true due to the following observation on execution semantics of `SFB`:

Every execution step of an `SFB` comprises three sub-steps: *input sampling*, *taking a transition*, and *execution state-entry actions*. These sub-steps can be simulated by the generated Prism module M as demonstrated by the following reasoning.

i. **Input sampling:** Both `SFB` and M use unit-delayed updates. Consequently, both α_i and γ_i read equivalent inputs from the previous execution states η_i and σ_i respectively, since $\eta_i = \sigma_i$.

ii. **Transitions:** Based on inputs read by α_i and the respective guard conditions on egress transitions of $\eta_i(q_c)$, `SFB` makes a transition to a successor ECState $\eta_{i+1}(q_c)$. There are three main possible cases in this step i.e., based on the inputs (Case I) the selected execution step is the initialisation step, (Case II) no ECTransition can be enabled from $\eta_i(q_c)$, or at least one ECTransition can be enabled. We further expand on the latter case where the highest order ECTransition is taken and examine if the successor is (Case III) a deterministic state, (Case IV) a probabilistic state, or (Case V) a non-deterministic state.

Case I: Initialisation step i.e., $\eta_i(q_c) = \top$. In this case, the state variable is updated to $\eta_{i+1}(q_c) = q_0$, which is matched by M. This is due to Rule T4, which uses the macro invocation $\mathsf{updateState}(q_0)$ for generating the update statements. Therefore, both the successor execution states have an equivalent value for the state variables: $\sigma_{i+1}(\mathsf{s}) = \mathsf{valueOf}(q_0)$

Case II: No ECTransitions could be enabled from $\eta_i(q_c)$. In this case, the state variable will not be changed $\eta_i(q_c) = \eta_{i+1}(q_c)$ and no state actions are performed. However, due to the single tick life-duration of events, the output events are reset. This case will be matched by M due to Rules T6 (see lines 30−34 of Algorithm 6.2) where, macros $\mathsf{checkState}(q)$ and $\mathsf{updateState}(q)$ keep the current value of the state-encoding variable e.g., to emulate the stay operation on the current ECState q. Here, the macro $\mathsf{negateAll}(q)$ ensures that this self-loop is only taken when no-other transition can be taken. Lastly, the macro $\mathsf{setStatus}(\varnothing, E_O^{\mathcal{I}})$ sets all output events to the `absent` status, thus emulating the event-reset behaviour of `SFB`. In this case, the execution

step concludes here and no more actions are performed in either of the structures in the current execution step.

Case III: The highest order transition $t{=}(q,e,b,q')$ has a deterministic successor state i.e., $\sigma_{i+1}(q_c){=}q'{\in}Q^d$. In this case Rule T5 shall be used to create a Prism command in M (see lines 17−19 of Algorithm 6.2). The Boolean guard *transCondition* on line 16 ensures that (i) at most one command is enabled in an iteration, and (ii) the transition order is preserved. Therefore, the command executed by M will necessarily matches the highest priority enabled transition as taken by `SFB`, reaching an equivalent execution state as the tick boundary, such that $\sigma_{i+1}(q_c){=}\mathsf{valueOf}(q')$.

Case IV: The highest order transition $q{\to}q'$ has a probabilistic successor state $q'{\in}Q^p$. In this case, the execution semantics of SFB enforce two sub-steps namely, (i) a deterministic transition from q to q', and (ii) a probabilistic selection of a transition from q' to one of the successors q''_i as given by the probabilistic transition function $P(q'){=}\Big\{(q',p_0,q''_0),\ \dots\ ,(q',p_n,q''_n)\Big\}$. Due to the enforced alternate model (see Observation 6.2.1), `SFB` reaches a deterministic ECState $q''_i{\in}Q^d$ as the tick boundary i.e., $\eta_{i+1}(q_c){=}q''_i$.
This execution behaviour of `SFB` can be simulated by M because of *transCondition* and Rules T7 (lines 20−22 of Algorithm 6.2), which (i) ensures that the matching deterministic transition is selected to match the sub-step transition q to q', and (ii) uses the set of egress transitions $P(q'){=}\Big\{(q',p_0,q''_0),\ \dots\ ,(q',p_n,q''_n)\Big\}$ to create probabilistic update statements containing the probabilistic update pairs (see Section 6.3.4.1). Here, the corresponding update statements also update the state-encoding variable using macro $\mathsf{updateState}$. Hence, with the same probability value p_i, both structures can reach an equivalent successor state $\sigma_{i+1}(q_c){=}\mathsf{valueOf}(q''_i)$.

Case V: The highest order transition $t{=}(q,e,b,q')$ has a non-deterministic successor state i.e., $\sigma_{i+1}(q_c){=}q'{\in}Q^n$. Due to the enforced alternate model (see Observation 6.2.1), `SFB` further selects a non-deterministic transition from $R(q'){=}\Big\{(q',q''_0),\ \dots\ ,(q',q''_m)\Big\}$ to reach a deterministic ECState $q''_i{\in}Q^d$. This ECState serves as the tick boundary. This execution can be simulated by M, which can select a matching non-deterministic Prism command. This is ensured because of Rule T8 (lines 23−28 of Algorithm 6.2), which uses the $\mathsf{checkState}$ and $\mathsf{ensureOrder}$ to ensure that

a matching non-deterministic choice is created (see *transCondition* on line 26). The generated non-deterministic commands offer the same non-deterministic choice as given by $R(q')$ and thus, the an equivalent selection of tick-boundary can be matched, namely the deterministic state q_i'' such that, $\sigma_{i+1}(q_c)$=$\mathsf{valueOf}(q_i'')$.

iii. **Successor State Actions:** In all cases except Case II, `SFB` executes its successor ECState-entry actions $X(\eta_{i+1}(q_c))$. This step is matched by M because of the macro stateActions in Rules T4, T5, T7, and T8 (see lines 12,18,21 and 25 of Algorithm 6.2), which ensures that (i) algorithms in $A_L^{\mathcal{I}} \cap X(\eta_{i+1}(q_c))$ are translated into update statements that perform equivalent computation and value assignments for local and output variables, and (ii) the event-encoding Boolean variables are set and with an appropriate Boolean value for event emissions in $X(\eta_{i+1}(q_c))$ (see stateActions macro). Lastly, the macro updateState updates the state-encoding variable s to represent the respective successor ECState $\eta_{i+1}(q_c)$ i.e., $\sigma_{i+1}(\mathsf{s})$= $\mathsf{valueOf}(\eta_{i+1}(q_c))$.

The equivalence of the three sub-steps of a tick results in equivalent values for the state variable, event-encoding Boolean variables, and variables representing local and output variables. Thus, the resultant corresponding execution states are equivalent η'_{i+1}=σ_{i+1}. This implies that the pair (η_{i+1},σ_{i+1}) exists in the simulation relation $\mathcal{R}_s$ (from *Refinement*).

From (1)-(3) and Definition 6.4.1 presented above, we conclude that the given SFB `SFB` is simulated by the corresponding generated Prism module M: `SFB`$\preceq M$ ■

The simulation relation between a given SFB and a corresponding generated Prism model is retained by the containing FBN and Prism models respectively. This is due to the behavioural equivalence of the compositions i.e. both compositions execute in lock-steps and use unit-delayed communication between their components. The arguments presented in Theorem 5.5.2 for this behavioural equivalence also apply to the compositions of SFBs.

6.4.3 Preservation of LTL Properties

The executions of FBNs and corresponding generated Prism models are free from deadlocks such that, their execution traces have no terminal-states (from Observation 5.5.1).

Bestpfe.com

For such branching structures without terminal states, the simulation relation offers trace inclusion [107]: $\texttt{SFB} \preceq M \implies \mathit{Traces}(\texttt{SFB}) \subseteq \mathit{Traces}(M)$. Furthermore, due to the behavioural equivalence of their respective compositions this trace inclusion relation is maintained. This preserves all linear-properties of FBN in the generated Prism model including LTL i.e., for any LTL property φ we have: $\mathcal{M} \models \varphi \implies \texttt{FBNetwork} \models \varphi$

6.5 Safety Analysis

Stochastic function blocks can be used for modelling various modes of failures, as well as the nominal and failure-affected behaviours of respective components. In the following text, we present our approach for modelling failures and their effects as SFB, and later demonstrate an approach for performing quantitative safety analysis using probabilistic model checking.

6.5.1 Probability of Failure

In Chapter 3, we presented an introduction to IEC 61508, which is a functional safety standard for generic electric/electronic/programmable electronic systems [3]. IEC 61508 differentiates between *high-demand* and *low-demand* operations of systems (see Section 3.1.4). A high-demand system is considered to be in contentious operation whereas, a low-demand system is considered operational on demand. The design pattern for modelling these two modes of failures differ significantly because of their respective likelihood measures. In high-demand systems, the likelihood of component failures is measured as *failure rates* (λ), which is the probability of a failure over a time period. An example of such failures is available from the *gas detection system*, where the `GasSensor` bears 1 *failure per* $10^4\,hr$. On the other hand, the likelihood of failure in low-demand systems is measured as instantaneous probability.

Failure rates are estimated by considering failures as periodic events that occur at predictable rates (λ) over a continuous time period (T). This type of measure is suitable for *continuous-time* Markov analysis whereas, SFBs execute in *discrete-time* steps and cannot use such values. Therefore, the given value must be discretised. We use the approach presented in [116] where a small time resolution δ is used to obtain a *per-step* probability value of λ_δ. For example, in order to use $\lambda = 1/10^4\,hr$ in the `GasSensor` SFB, we can discretise it using a time resolution $\delta = 10\,ms$ to obtain a per-step probability

value of $\lambda_\delta = 2.8 \times 10^{-10}$. It is further shown in [116] that if this time resolution is much smaller than the given time period of λ i.e., $\delta \ll T$, then the cumulative error is negligible. This discretisation is not required for on-demand failures since they are already instantaneous probability values.

6.5.2 Failure Mode and Effect Modelling

Probabilistic on-demand and per-time failures can be modelled using the probabilistic states ($q \in Q^p$) of SFB such that, the probability of failure is used on transitions leading to deterministic states ($q' \in Q^d$) representing failures. Figure 6.9 shows an example of modelling per-time and on-demand probabilistic failures such that, the notation "`<0>`" represents the highest transition selection order. Here, the per-time failure model depicts an unconditional probabilistic path that can lead to the `FAILED` state at any time. Whereas, the on-demand failure model in Figure 6.9 waits in the `NORMAL` state until the demand occurs (represented with event `input`), then a probabilistic choice can be made to enter the `FAILED` state depicting a failure on-demand.

Non-deterministic behaviours often arise from the unpredictable nature of the external environment. Such behaviours can be modelled using the non-deterministic states ($q \in Q^n$). One example of such a behaviour is the gas emission in the hazardous gas detection system, shown in Figure 6.10. Here, the emissions may build up non-deterministically and may only subside when the ventilator is on as represented by the `VENTING` state.

Figure 6.9: Modelling (a) *per-time* and (b) *on-demand* probabilistic failures

BESTPFE.COM
List of research project topics and materials

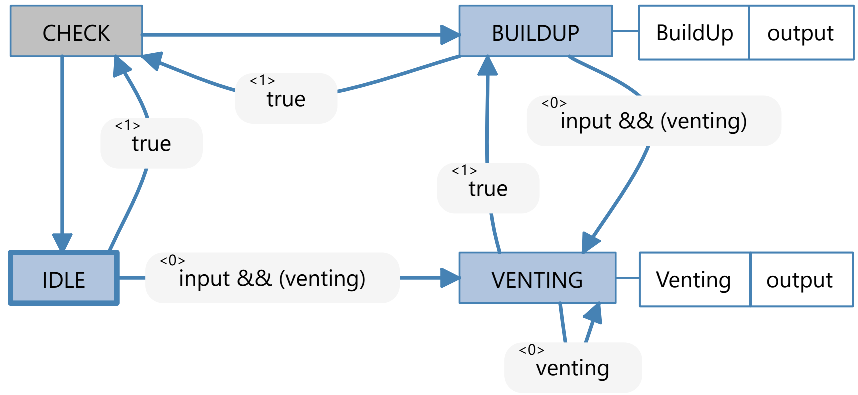


Figure 6.10: Modelling non-deterministic behaviours

6.5.3 Probabilistic Model Checking

Model checking is a technique for automatic verification of system models against a given specification called properties [117]. System properties are sometimes represented using qualitative temporal logic for reasoning over branching and non-branching behaviours of system execution. The non-branching/linear fragment of this temporal logic is referred to as *linear-temporal logic* (LTL), which can be used to specify non-branching/linear properties over the execution behaviours of a system. The result of this verification yields either *true* or *false* and is denoted as $\mathcal{M} \models \varphi$ i.e., the system model $\mathcal{M}$ satisfies the property φ. In probabilistic systems, a discrete probability can be computed for a linear property [107]. This enables quantitative reasoning over probabilistic behaviours of system models. Unlike qualitative verification, the result of probabilistic verification returns a quantitative value, for example, the probabilistic verification of the property $P_{=?}[\phi]$ using Prism model checker [112] determines the probability of satisfying the property ϕ.

The proposed approach uses probabilistic model checking for performing quantitative reliability assessment and safety analysis. This process entails translating the given FBN into a Prism model that contains all execution behaviours of the FBN (see Theorems 6.4.1 and 5.5.2). This behavioural containment ensures the soundness of verification of linear properties such that, any quantitative result obtained from the Prism model also holds for the given FBN (see Section 6.4.3). The generated Prism model is a Markov decision process (MDP) that contains non-deterministic behaviours, therefore probabilities must be ascertained using the maximising or minimising adversaries [118]. Thus, the probability of occurrence of a certain hazard can be computed using the following property specified in Prism property syntax.

$$P(\boldsymbol{Hazard}) := P_{max=?}\ F^{\leq k}[\boldsymbol{Hazard}] \qquad (6.5.1)$$

In the property presented above, $P_{max=?}$ represents the query for calculating the maximum probability of the given property namely, *reaching a Hazard state*. The superscript $\leq k$ represents the number of steps to consider in the bounded model checking, which sets the limit on the execution of the system to occur in a finite time. This value is thus used as the finite number of iterations to be performed by the underlying numerical solver. Therefore, choosing a correct value for this bound improves the scalability of the proposed approach. A reasonable value for this bound can be calculated from the time resolution used for discretisation (δ). For instance, the duration of $1\,hr$ in terms of $\delta = 10\,ms$ gives $k = 360000$, which will yield us the probability of the hazard in *failures per hour*.

6.6 Results

The proposed approach enables probabilistic analysis that can be performed on the generated Prism models. In addition to the hazardous gas detection system, we analysed a set of systems to demonstrate its practicability. We briefly describe the system purpose and configuration as following.

1. The **Gas Detection and Ventilation System** configured to maintain a safe level of CO_2 i.e., below 600 parts per million. Sensors and ventilator are susceptible to per-time failures, whereas the gas emission builds up non-deterministically. An assumption is made that ventilator is capable of ventilating at a faster rate than the maximum rate of emission.

2. A **Fire Detection and Suppression System** that uses three combustion sensors to detect fire and immediately sounds an alarm. Multiple detections are required for opening an electric valve to start the sprinklers. A manual reset switch may be triggered non-deterministically. The source of fire is modelled as non-deterministic SFB, which models the unpredictable occurrences of fire incidents.

3. A **Boiler Control System** with a pressure transmitter, pressure relief control valve, and an assembly of $1oo2$ redundant flow indicators [15]. When the pressure exceeds a threshold value (e.g., 300 PSI), the pressure relief is turned on, and when the pressure falls below the threshold the pressure relief is turned off. The pressure relief is confirmed using the flow indicators.

The result of this analysis is presented in Table 6.5. The time resolution used for discretisation of failure rates is $10\,ms$, resulting in a total number of iterations performed

Table 6.5: Probabilistic verification of example systems.

System	Quantitative specification in Prism property syntax		Result (p/hr)
1	*[P1]*	Probability of gas level exceeding 600 parts per million	$= 3.3598 \times 10^{-5}$
		$P_{max=?}$ [$F^{\leq 360000}$ (`gasLevel` > `600`)]	
	[P2]	Probability of ventilator failing before sensors	$= 3.3598 \times 10^{-5}$
		$P_{max=?}$ [$\overline{(\texttt{s1Fail} \mid \texttt{s2Fail})}$ $U^{\leq 360000}$ (`vFail`)]	
2	*[P3]*	Probability that sprinklers will turn on before sounding alarm	$= 6.342 \times 10^{-10}$
		$1 - P_{max=?}$ [(`s_sprinkler`! = `2`) $U^{\leq 360000}$ (`s_alarm` = `3`)]	
	[P4]	Probability of alarm failure when needed	$= 2.2828 \times 10^{-4}$
		$P_{max=?}$ [$F^{\leq 360000}$ (`s_alarm` = `2`)]	
	[P5]	Probability of sprinklers failure when needed	$= 7.2683 \times 10^{-6}$
		$P_{max=?}$ [$F^{\leq 360000}$ (`s_sprinkler` = `3`)]	
3	*[P6]*	Probability of exceeding the pressure threshold by more than 10%	$= 6.0492 \times 10^{-5}$
		$P_{max=?}$ [$F^{\leq 360000}$ (`pressure` > `330`)]	
	[P7]	Probability of both flow indicators failing	$= 7.3733 \times 10^{-9}$
		$P_{max=?}$ [$F^{\leq 360000}$ (`fi1_Fail` & `fi2_Fail`)]	

$k = 360000$. The result of *[P1]* shows that the probability of hazard in the gas detection and ventilation system i.e., the concentration of CO_2 rising above 600 parts per million is 3.3598×10^{-5} per hour or roughly three times in a decade. This value represents a critical cut set that leads to the hazard [119], namely the failure of the ventilator. Further confirmed with the property *[P2]*, the probability of ventilator failing before sensor assembly is shown to be identical to the discovered cut set probability. While these properties utilise Prism `labels` construct to simplify the property specification, direct use of the state variable values is also valid as demonstrated by properties *[P3]*–*[P5]*. The property *[P3]* uses the negation of a sub-property by subtracting its probability from 1. The result gives us the probability of all scenarios where sprinkler was turned on without sounding the alarm first i.e., where multiple detectors reported smoke and the alarm failed, but the sprinklers did not fail. Similarly, *[P6]* and *[P7]* calculate the probability of pressure overload hazard, and the probability of both flow indicators failing respectively.

Table 6.6 presents the analysis time for each for the presented system. Here, the time spent on calculating the probabilities depends on the size of the model i.e., number of states and transitions in the generated Prism model. As expected, bigger models take more time for their subsequent analysis. Please note that these are not the number of

Table 6.6: Analysis times for probabilistic verification

System	**States**	**Iterations**	**Analysis Time**
Gas Detection and Ventilation System	11173	22034	39 s
Fire Detection and Suppression System	15201	258571	123 s
Boiler Control System	7649	14035	27 s

ECStates and ECTransitions, instead these are the number of states in the overall system when composed together. This composition is generated by the Prism model checker before the analysis beings, and is performed as described in Definition 5.3.3.

6.6.1 Scalability

In order to evaluate the scalability of the proposed unified analysis, a distributed drilling station system was used [15]. The system consists of pneumatic, mechanical and electric components to implement an arm and a pusher assembly that places work-pieces on a series of conveyor belts. These work-pieces are then transferred to a processing station that accepts the work-pieces on a spinning work table. This multi-stage work table uses a series of sensors to verify the presence of a work-piece and uses a drill machine to drill a hole in each work-piece. The processed work-pieces are eventually ejected on to an adjoining conveyor belt for any further processing. The corresponding IEC 61499 system consists of 13 FBs, where 9 FBs relate to the control software. For scalability, conveyor models were added in an incremental manner to create larger systems while keeping the number of iterations of the bounded analysis constant at 360000 ($\delta_t = 10\,ms$). Analysis times of some of the configurations are shown in Table 6.7 that shown an exponential increment. Such an increase is expected due to the combinatorial state space explosion caused by increasing number of components (FBs) used in the analysis. This combinatorial state space explosion is unavoidable because the analysis spans the entire reachable state space of the system. However, the experiment shows that the proposed approach can handle medium to large scale systems/sub-systems ($\approx 10^6$ states). As presented earlier, this scalability can be improved at the cost of accuracy by using a coarser time resolution.

Table 6.7: Analysis times for the scalability experiment.

States	**Transitions**	**Analysis Time**
42680 ($\approx 10^4$)	103284	67 s
178130 ($\approx 10^5$)	401274	131 s
1532630 ($\approx 10^6$)	3381174	804 s
15077630 ($\approx 10^7$)	33180174	19633 s

6.7 Discussion

In this chapter, we presented a model-based safety assessment approach for industrial automation systems. The presented approach uses the system development models and thus seamlessly integrates with model-based development processes. We proposed a novel structure named *stochastic function blocks* based on IEC 61499, which allows modelling probabilistic and non-deterministic behaviours in the *plant-model*. On the other hand, the *controller* is constructed using *deterministic function blocks*, which can later be used to generate the controller software. The rule-based conversion of the overall system model into Prism models enables quantitative assessment using probabilistic LTL properties. The soundness of this result is justified by demonstrating a simulation relation. The quantitative result of this analysis can be evaluated against the reliability targets. Furthermore, any systematic errors in the system will also manifest in the result i.e., by giving a probability of failure ≈ 1 (accounting for numerical errors), which indicates that the specified failure will occur in a repeatable systematic manner.

7

BlokIDE

An IDE for Model-Based Design and Safety

IEC 61508 follows a lifecycle approach for achieving safety [3]. Starting from planning until decommissioning, every phase of the process follows a set of recommendations to mitigate risks to human lives, as much as *reasonably* possible. This includes the realisation phase where system software and hardware are designed, developed and validated. Part 2 and 3 of the standard provides requirements for hardware and software respectively, as well as any support tools used, such as development and design tools, language translators, testing and debugging tools, configuration management tools etc. The standard does not necessitate any particular design methodology i.e., it is entirely possible to use any programming language to implement an IEC 61508 certifiable system. However, use of low-level languages and ad-hoc design practices for implementing complex systems often result in low maintainability and high design complexity. This calls for a tool-chain that is not only built with developer-friendliness in mind but also conforms to the said safety requirements and is compatible with the overall system safety lifecycle.

In this chapter, we present a tool-chain named BlokIDE for the *model-driven development* of industrial control and automation systems. BlokIDE is implemented as

an extension to Microsoft® Visual Studio™ [120]. It implements IEC 61499 [11] as a domain specific visual language and offers *model editors* for editing and managing the various design artefacts. A custom model editor is also implemented for *stochastic function blocks* (SFB) presented in this thesis. The model export feature allows exporting the IEC 61499 models into various other formats e.g., the standard IEC 61499 Xml format [11], SAML format [55], as well as the Prism language [58]. This enables the model-based safety approach proposed in Chapter 6.

7.1 Contributions

An earlier version of BlokIDE existed prior to the commencement of this work [6]. This earlier version primarily had focus on qualitative aspects of safety. Apart from the usual enhancements and maintenance, novel contributions were made to BlokIDE to widen its applicability to quantitative safety aspects. The contributions made in the scope of the current work are summarised as follows.

- Enhancements to BlokIDE to implement the editing capability for SFBs. This implementation uses Visual Studio's visualisation capabilities and represents SFBs and IEC 61499 as a domain specific language (see Section 7.2.1). This also includes a set of design level restrictions to ensure the integrity of models e.g., the enforce the *alternate-model* and state partitioning as prescribed by Definition 6.2.1.
- Implementation of Algorithm 6.2 (see Chapter 6) for automatic semantics preserving conversion of function block networks (FBN) to equivalent Prism models. This implementation is capable or converting FBNs that comprise of both basic and stochastic function blocks.
- In Section 7.3.2 we present a proposed design and implementation process using BlokIDE. This proposed process is derived from the V-Model, which is a well-known and a highly recommended design and implementation process for safety related systems. This proposed design process leverages the design, implementation, validation and verification capabilities of BlokIDE and its companion tools. Resultantly, this makes the BlokIDE-based development amenable to the software realisation requirements of IEC 61508-3 specifically clause 7 [4].
- In Section 7.3.3 we present a gap-analysis of BlokIDE against the various software requirements presented in IEC 61508 Annex. B [4]. Each of the said requirements has an associated recommendation for a target SIL e.g., using finite state machines

is recommended for SIL2 and highly recommended for SIL3. Through the presented gap-analysis, we reach to a conclusion is that BlokIDE is highly suitable for safety related systems targeting SIL2, however additional tools and techniques may be required for design and implementation of systems targeting SIL3.

In the following sections, we present the detailed insight into the design an implementation of BlokIDE and its suitability of design and implementation of safety related systems.

7.2 BlokIDE – Design and Implementation

BlokIDE is implemented as a collection of components for the Visual Studio using its extensibility framework. These components and their inter-dependencies are presented as a UML component diagram in Figure 7.1. The core component of BlokIDE is the diagramming capability of *model editors* for designing function blocks (FB) and their networks (FBN). These models are saved on the file system and are collated together to form a pro-

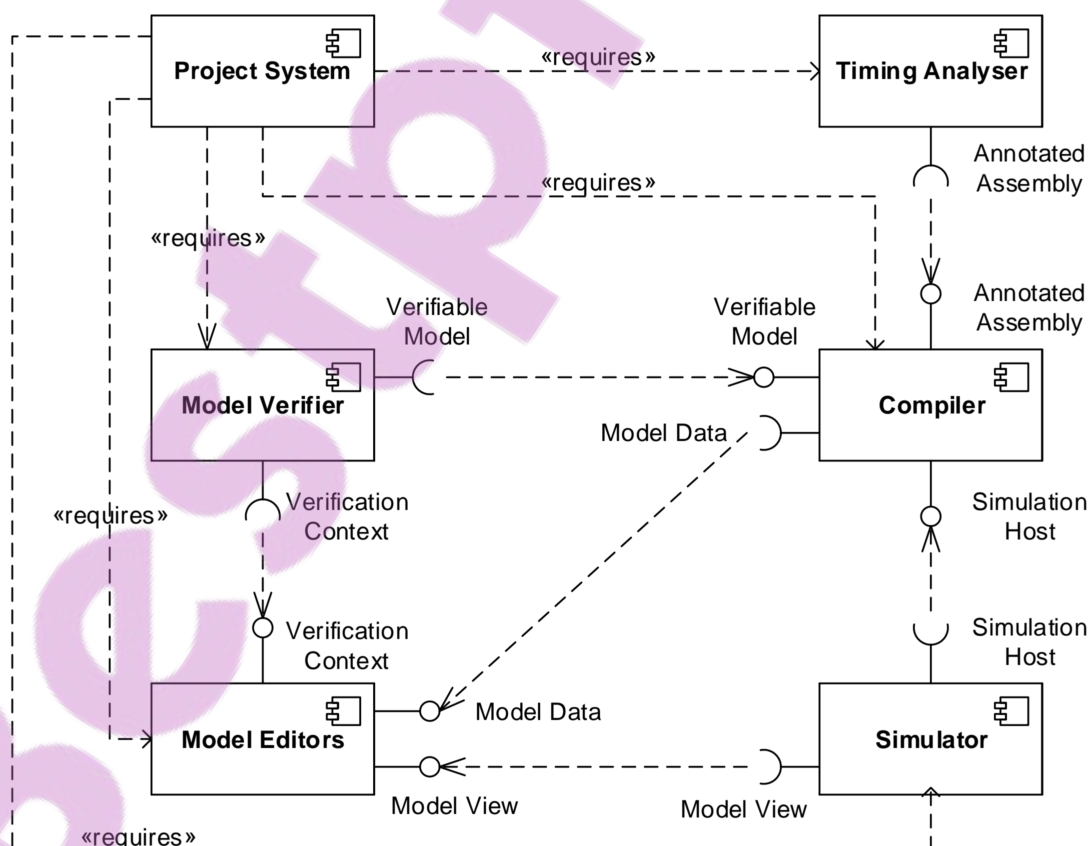


Figure 7.1: UML component diagram of BlokIDE [6]

Bestpfe.com

ject. This *project system* is developed using the managed project framework (MPF), which provides extension points for compilation, simulation and appropriate tool-windows. The synchronous *compiler* [108] is integrated within BlokIDE, which allows automatic code generation from the IEC 61499 models. A custom debug engine and a *simulator* are implemented to allow step-by-step design validation through visual debugging. Furthermore, static analysis tools such as, the *static timing analyser* [30] and the observer-based *model verifier* [34] are also integrated into BlokIDE.

7.2.1 IEC 61499 as a Domain Specific Language

IEC 61499 can be viewed as a domain specific language (DSL). The visual artefacts defined in the standard can be viewed as meta-model definition for the various types of FBs. Chapter 4 provides more details about structure and semantics of IEC 61499. The said meta-model of IEC 61499 essentially, is a set of domain classes and their relationships that are mapped to shapes. Based on this meta-model, model classes and digram editors are automatically generated by Visual Studio's visualisation and modelling framework (VMSDK). This generated code is further customised to implement custom shapes and placement rules. In this section we present the meta-model of IEC 61499 as implemented in BlokIDE.

7.2.1.1 Named Elements

The meta-model for IEC 61499 consists of many entities that carry the *name* attribute. This attribute is a developer friendly moniker used to identify elements in their respective scope. For example, name of a function block type (FBType) is unique in the scope of a project, whereas name of an FB instance is unique within the scope of a given FBN. An abstract base class named **NamedElement** represents all such elements. Other named elements in the meta-model are signals, algorithms, and states. A sub-class of NamedElement is *location-aware* i.e., the location of the element on the corresponding diagram. This is required to implement the visualisation rules imposed by IEC 61499. For example, all input signals appear on the left-hand side of a given FB interface. This makes its necessary to store the location-information. For this purpose, **LocationAwareNamedElement** class is defined in the model, which serves as the base class for all such elements that need location-information. Figure 7.2 presents the domain classes representing the said elements. Here, DisplayText and FullName are derived properties that are calculated from the fully qualified names of the given elements. The properties X and Y of the

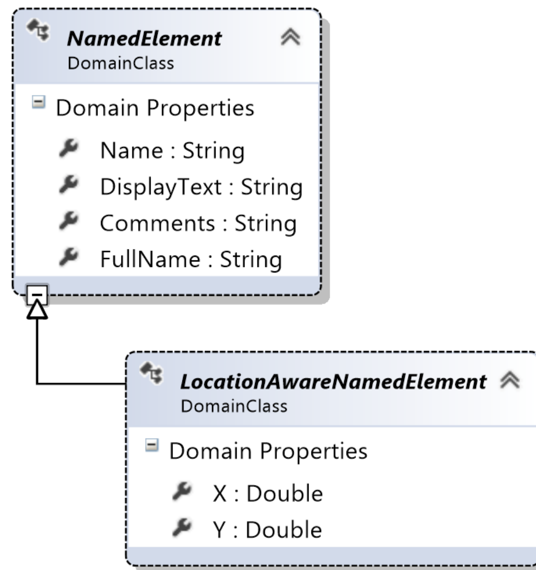


Figure 7.2: Meta-model of the DSL implementing IEC 61499

LocationAwareNamedElement class represent the location of the element on the 2D canvas.

7.2.1.2 Function Block Models

The meta-model for the various types of FBs share a few common attributes. This common aspect of their models has been captured as an abstract class named **BusinessModel**. The information in this domain class pertains to the business domain e.g., who is the author/owner of the model, its creation date and version information. The Header property is used as text literal to include in the generated code e.g., for copyright notice, preprocessor macros, and header includes. Figure 7.3 presents the said domain class.

7.2.1.3 Function Block Interface and References

The interface of an FB presents its input and output (IO) signals. This interface is defined alongside the definition of an FBType to allow accepting inputs and emitting outputs. A reference of a given FBType is used to generate its instances in an FBN, where its interface is used to consume its IOs. The visual similarities between an interface and its reference gives rise to concept of an **AbstractFunctionBlock** that has input and output events/variables. Based on the context, an appropriate child class i.e., **FunctionBlock-Interface** or **FunctionBlockReference** is instantiated. Figure 7.4 presents the relationship between the said domain classes and their attributes. Here, the ModelType property stores what type of function block is being used e.g., Basic or Composite. The property Type-

BESTPFE.COM
List of research project topics and materials

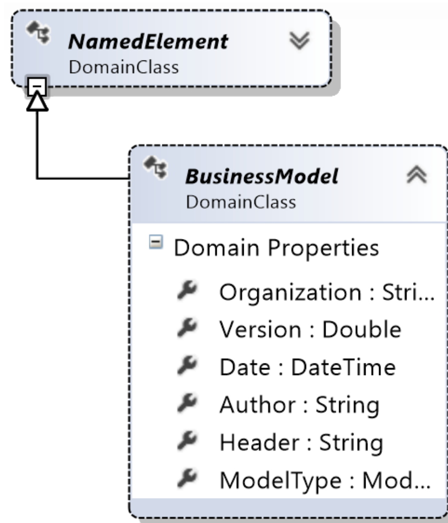


Figure 7.3: Metal-model for the BusinessModel class

Name stores the name of the FBType being referred, whereas the Reference is a direct link to the source model for keeping the reference up-to-date with the FBType definition.

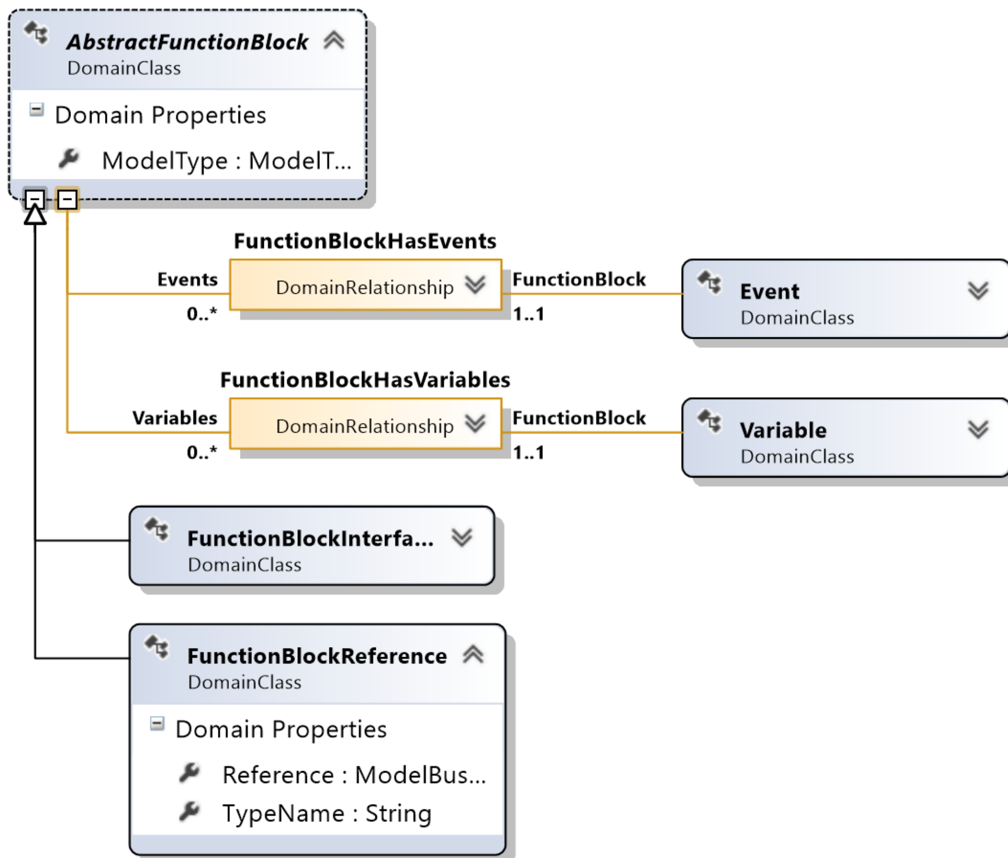


Figure 7.4: Meta-model for FunctionBlockInterface and FunctionBlockReference

7.2.1.4 Signal and Signal References

IEC 61499 describes **Variable** and **Event** as the two types of signals. The meta-model defines these as named-elements that have the attributes of Scope and Type with enumerated values e.g., scope of an variable can be input, output or local, whereas the type of the variable can be one of the variables types defined by IEC 61499 e.g., INT, SINT, DINT etc. Extending the notion of reference, the domain classes **EventRef** and **VariableRef** describe the events and variables of the respective FunctionBlockReference in a given FBN. Figure 7.5 presents the inheritance hierarchy of said meta-model classes.

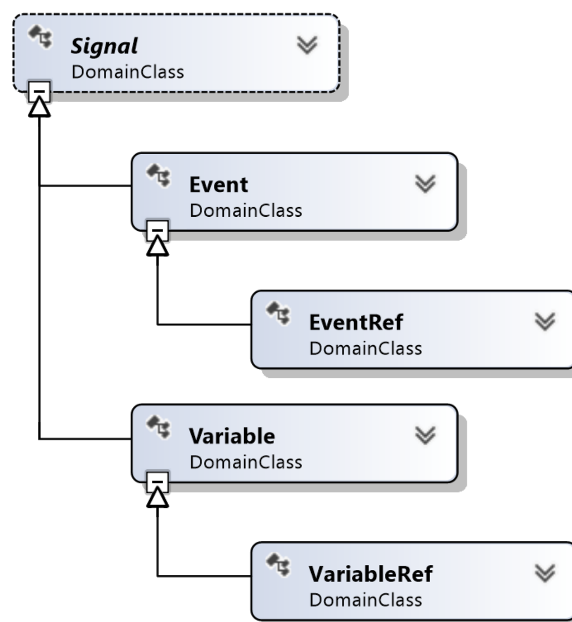


Figure 7.5: Various meta-model classes for Signals

7.2.1.5 Execution Control Charts

Both the basic function blocks (BFB) and stochastic function blocks (SFB) comprise of an execution control chart (ECC). As presented previously, a BFB is a simplified form of an SFB (see Observation 6.2.2). We use this observation to simplify the meta-model by implementing ECCs for both BFBs and SFBs as shown in Figure 7.6. In this model, a **State** can have at most one **ActionSet** i.e., its state-entry actions. Also, a State can connect to zero or more **AbstractTransitionNodes**. This relationship is named StateConnectsToTransitionsNodes and it represents the egress transitions from a given state. A complementary relationship named **TransitionsNodeConnectsToStates** describes the ingress transitions on states. Thus, starting from a SourceState to a TargetState, an AbstractTransitionNode represents an ECC transition. The AbstractTransitionNode, as the name suggests, is an abstract class. Its

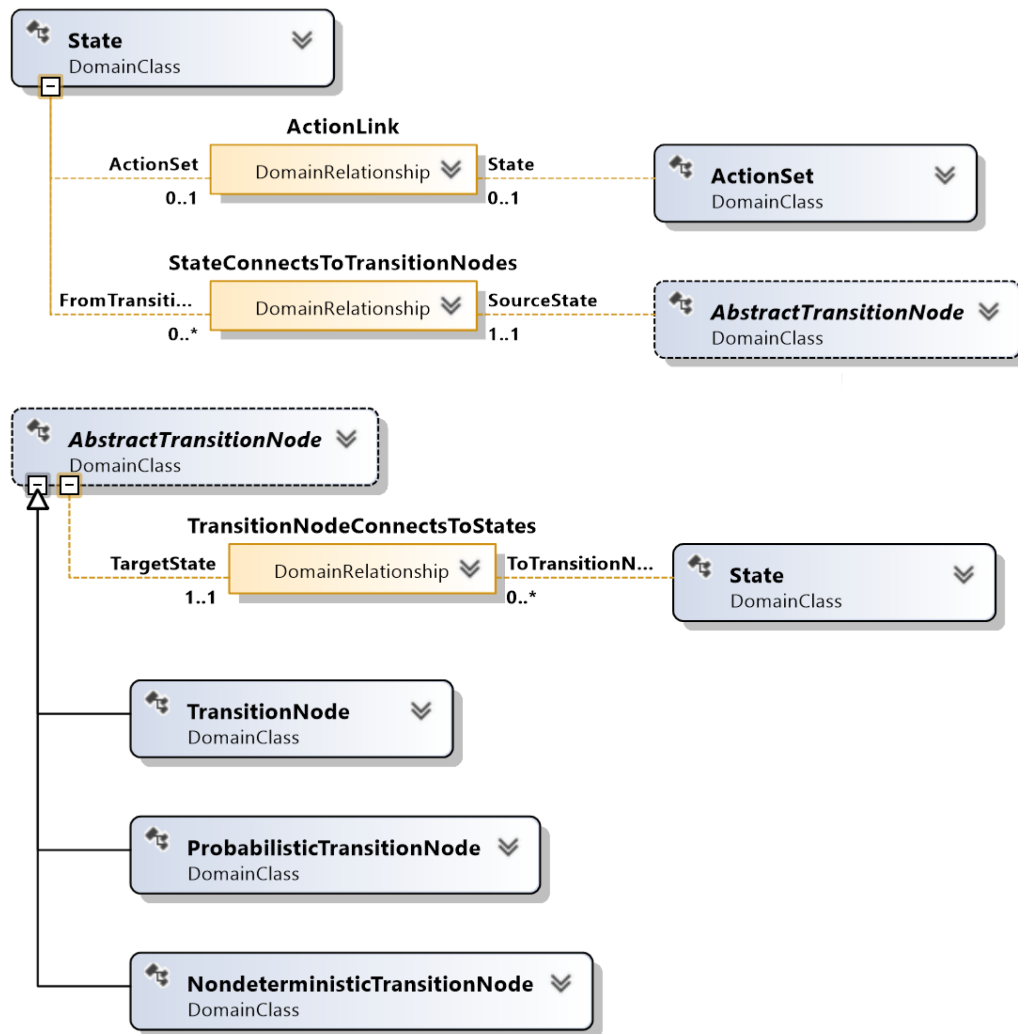


Figure 7.6: Meta-model for ECCs of BFB and SFB

concrete implementations **TransitionNode**, **ProbabilisticTransitionNode** and **NondeterministicTransitionNode** are instantiated depending on user selection, thus representing the three types of transitions in the ECCs of SFBs (see Chapter 6). Following design-time restrictions are implemented to maintain the structural integrity of ECCs.

- An ECC must have exactly one initial state based on the definitions of BFBs and SFBs (see Definitions 5.2.3 and 6.2.1).
- The name of a state must be unique within the ECC.
- A state can only have one type of egress transitions i.e., either using TransitionNode, ProbabilisticTransitionNode, or NondeterministicTransitionNode. This mutual exclusion induces the partition of states into deterministic, probabilistic and non-deterministic states as presented in Definition 6.2.1.

- Sum of probabilities on all egress probabilistic transitions from a given state should be 1. This restriction ensure that transitions from a given probabilistic state is a probability distribution.

- The TargetState of a stochastic transition i.e., created using the relationship TransitionsNodeConnectsToStates, must be a deterministic state. This restriction enforces the alternate model of SFBs (see Observation 6.2.1).

Based on the presented meta-model and the design-time restrictions, we derive two separate templates for BFBs and SFB i.e., if the user selected to create a BFB, stochastic transitions are disabled in the model editor. This eliminates the chances of structural inconsistency in ECCs.

7.2.1.6 Function Block Networks

FBNs of composite function blocks (CFB) are constructed using FunctionBlockReferences encapsulated by a FunctionBlockInterface. The references carry the IO declaration of the respective FBType, which are logical entities. We introduce the concept of **FunctionBlockPort** to represent the visual anchor for making connections between these IOs. The four concrete types **InputEventPort**, **InputVariablePort**, **OutputEventPort** and **OutputVariablePort** allow making rule based decisions on the two types of connection i.e., **WireConnection** and **ProxyConnection**. Furthermore, we also define a set of design-time rules extracted from the definition of FBN (see Definition 5.2.4) to ensure the structural integrity of CFBs and the encapsulated FBNs. The described meta-model is presented in Figure 7.7, whereas the design rules are presented as follows.

- The instance name of a FB reference must be unique within the FBN.

- An event WireConnection can only be made from an OutputEventPort to one or more InputEventPorts.

- A variable WireConnection can only be made from an OutputVariablePort to one or more InputVariablePorts. However a connection is only allowed if the types of corresponding output and input variables are compatible e.g., an integer type output can only be connected to an integer type input.

- An InputVariablePort can be target of at most one WireConnection.

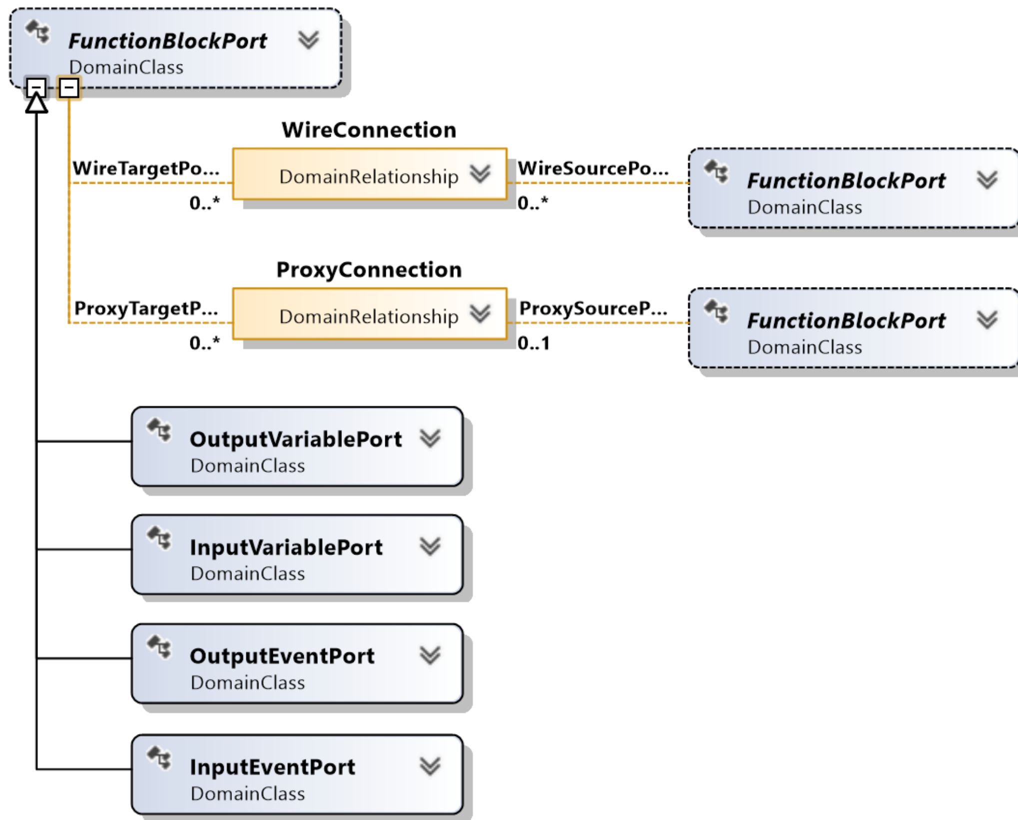


Figure 7.7: WireConnection and ProxyConnection relationships between FunctionBlockPorts

- An event ProxyConnection only be made between ports of similar scope between the encapsulating CFB interface and the encapsulated FB references. For example InputEventPorts of the CFB interface can be have proxy connections with the InputEventPorts of the encapsulated FB references. Similarly, ProxyConnections can be made from OuputEventPorts of FB references to OuputEventPorts of the encapsulating CFB interface.

- A variable ProxyConnection can be made similar to an event ProxyConnection e.g., between ports of similar scope. However, both the source and target of a variable ProxyConnection has to be type compatible as explained earlier. Furthermore, an OutputVariablePort on the encapsulating CFB interface cannot have multiple proxy connections.

- An FBN cannot contain an FB reference of the encapsulating CFB. This ensures that we do not have any self-reference.

The meta-model defined for FB interface, ECC and FBN provides a basic outline of the IEC 61499 as a DSL. A detailed meta-model is presented in Figure 7.8. The implementation of a DSL requires additional effort e.g., for mapping domain classes to shapes,

creating tool items for editor, and implementing various types of rules e.g., addition and deletion rules, layout rules, and implementing custom shapes and connectors. Discussion on these aspect of DSL design is beyond the scope of this thesis.

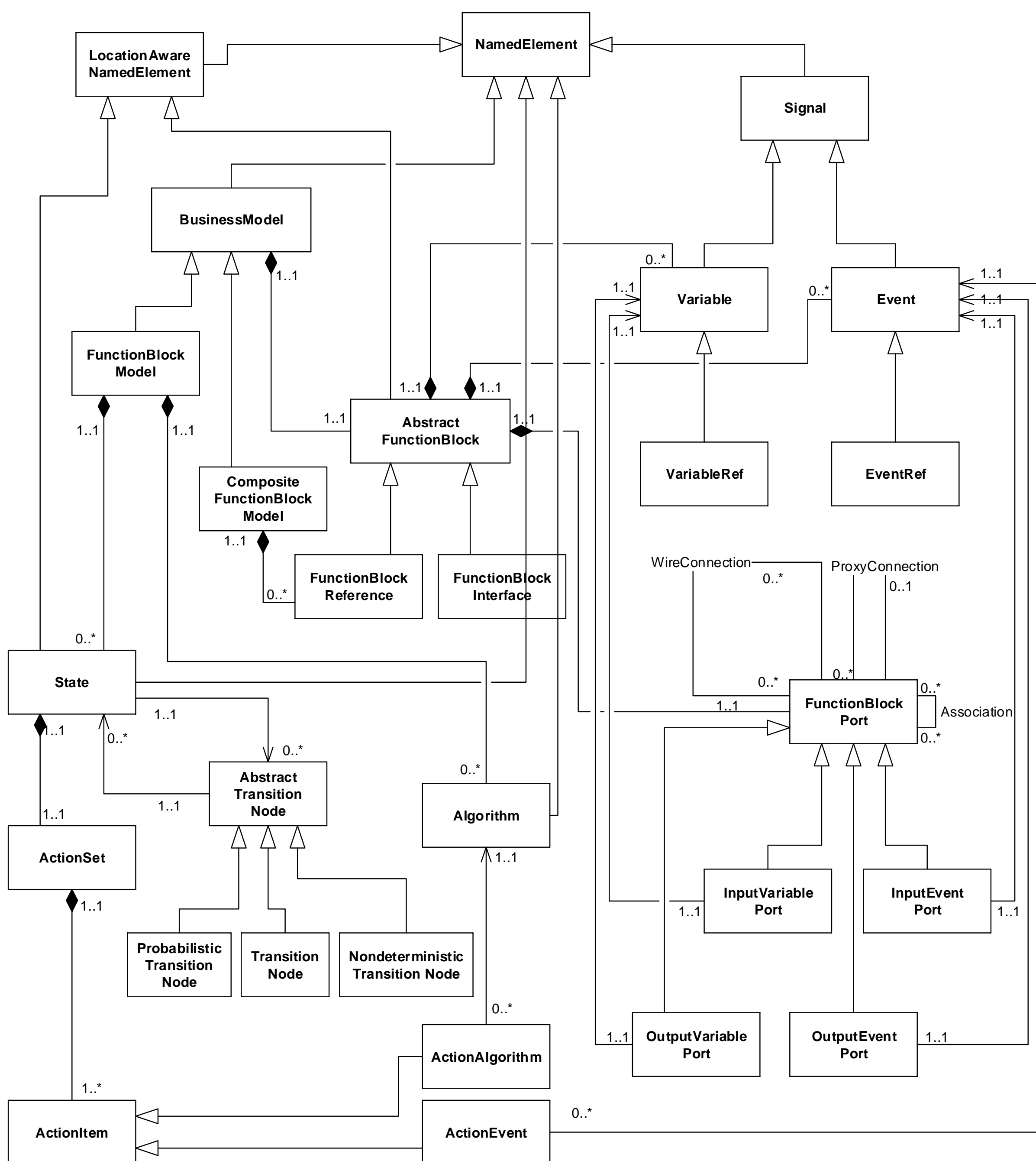


Figure 7.8: Meta-model of IEC 61499 implemented as a domain specific language

7.2.2 Compilation

The *project system* component of BlokIDE uses an XML based project file to store the list of models. This project file also refers to a library of function blocks e.g., for the various service interface function blocks (SIFB) for device specific functionalities. These models are also stored as XML files in the project directory and can be exported as the IEC 61499 standard FBT files. The exported FBT files conform to the standard schema and can be used by any IEC 61499 compliant tool e.g., the synchronous FB compiler [108]. The synchronous compiler is invoked by BlokIDE to perform automatic code generation. The generated ISO-C code implements the synchronous execution of the given FBs and can be compiled using the GNU C compiler (GCC).

Consider an FBN shown in Figure 7.9 that comprises a BFB and a CFB. Even though the top level FBN is abstracted as another CFB, for simplicity we just consider its FBN. Given this structure, the compiler generates a pair of header and C implementation file per FB such that, the C `struct` construct is used to represent an FBType. An additional file called `FBNrun.c` is also generated that contains the `main` function. The structure and outline of the generated code is presented in Figure 7.10. Here, the top level FBN creates two variables representing FB instances `cfb` and `bfb2`, invokes their `init` methods, and enters the reactive loop. Inside the reactive loop, both FB instances are invoked in a predetermined order. The two `run` functions `CFBrun` and `BFBrun` execute their corresponding ticks such that, one iteration of the reactive loop represents one tick. The execution of CFB and BFB have previously been presented in Algorithms 5.1 and 5.2 respectively.

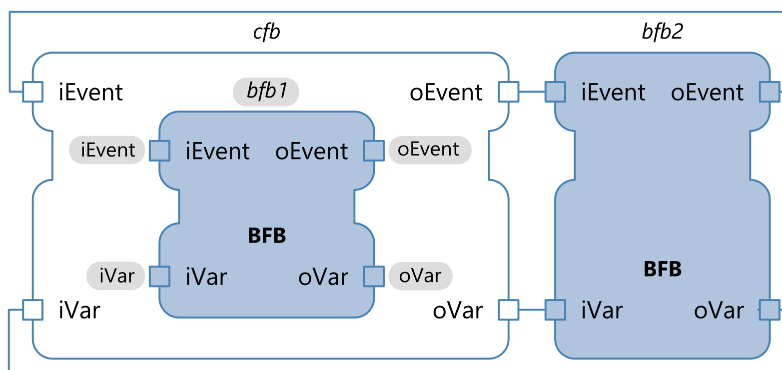


Figure 7.9: A composite function block

FBNrun.c

```
int main(int argc, char* argv[]){
  //FB instances in the top level FBN
  CFB cfb;
  BFB bfb2;

  //set memory to reference vars
  memset(&cfb, 0 sizeof(CFB));
  memset(&bfb2, 0 sizeof(BFB));

  //initialisation
  CFBinit(&cfb);
  BFBinit(&bfb2);

  for(;;){
     CFBrun(&cfb);
     BFBrun(&bfb2);
  }
}
```

CFB.h

```
typedef struct { … } CFB;
void CFBinit(CFB* me);
void CFBrun(CFB* me);
```

CFB.c

```
void CFBinit(CFB* me){ … }
void CFBrun(CFB* me){ … }
```

BFB.h

```
typedef struct { … } BFB;
void BFBinit(BFB* me);
void BFBrun(BFB* me);
void BFBAlgo1(BFB* me);
```

BFB.c

```
void BFBinit(BFB* me){ … }
void BFBrun(BFB* me){ … }
void BFBAlgo1(BFB* me){ … }
```

Figure 7.10: Structure and outline of the generated code

7.2.3 Simulation

We use the distribution system example previously introduced in Chapter 4 for illustration. The system comprises of a pusher and an arm assembly that work in coordination to pick and place work piece onto a conveyor belt. Two basic function blocks (BFBs) named DistStnPusher and DistStnArm are connected with each other through wire connections as shown in Figure 7.11. Under simulation, the events and variable values flow through these wire connections between the two corresponding FB instances as represented by the highlighted color. Developers can observe the behaviour of the system under simulation both visually as well as using the various tools provided in BlokIDE. In this section we discuss the simulation process and the said tools.

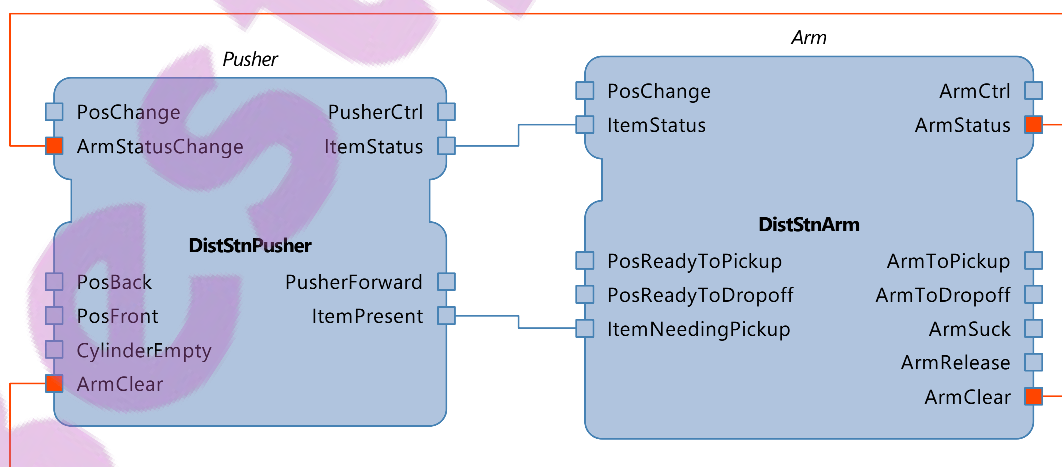


Figure 7.11: Application model of the *Distribution Station* system

Bestore.com

7.2.3.1 Protocol Used By The Simulator

The simulation capability of BlokIDE is implemented as a custom debug engine. It relies on an external process to connect over TCP/IP and communicate simulation data for accepting inputs and delivering execution state information. This communication is based on an XML based simulation protocol. The synchronous FB compiler [108] is extended to implement this protocol such that, when specified, the compiler generates a simulation-enabled version of the code, that not only contains the nominal behaviour of the given FB models, but also acts as a simulation host.

The simulation-enabled generated code is compiled to create the simulation host application, which is then executed. This compilation and execution can be performed on the desktop machine due to the portable nature of generated ISO-C code. However, if device-specific resources are used e.g., IO bus of a PLC, the compilation and execution must be performed on the target devices. When the host application starts executing, it connects to the BlokIDE instance through a TCP/IP connection on port 61495. When connected, the host application executes its initialisation tick and reports the current execution state to BlokIDE as a simulation data packet in XML. Figure 7.12 presents the sequence of actions performed during simulation.

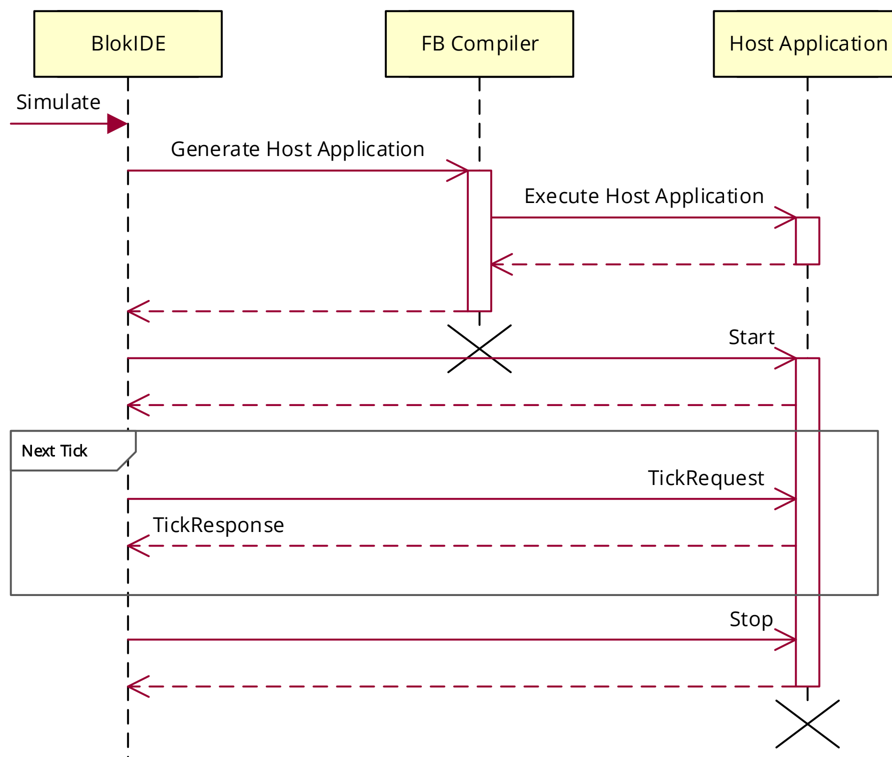


Figure 7.12: Sequence diagram for simulation

The simulation starts by sending a `<Start />` packet to the host application, which in return send the initial execution state of the FB system. From this point onwards, BlokIDE can send inputs the host application as a TickRequest and receive the a Tick-Response that contains the resultant tick execution. The simulation may continue for a number of ticks, after which the developer may choose to stop. The simulation is stopped by sending a `<Stop />` packet to the host application, which causes it to terminate its execution.

Tick Request Packet: The tick request packet provides the value of all input signals for the top level FB, which may be a CFB or a BFB. An instance of this FBType is created by the simulation host, named `root`. The input values for the signals on root instance are provided through the simulator window (see Figure 7.14). These values are arranged in a tick request packet and sent to the host application over the established connection e.g., as shown in Table 7.1. In this particular example the tick request represents a situation where the *pusher* is has been retracted (`PusherBack` = `true`) to load the next work piece from the stacking cylinder. Similarly the *arm* is reporting that it is currently at the pickup location to pick a work piece (`ArmAtPickup` = `true`). In this situation the expected behaviour from the simulation is to respond placing action e.g., by pushing the work piece to the pickup location.

Table 7.1: Tick request Xml packet

```
1 <Tick>
2   <Event Name="InputsChange" Value="PRESENT" />
3   <Var Name="PusherBack" Value="TRUE" />
4   <Var Name="PusherFront" Value="FALSE" />
5   <Var Name="ArmAtPickup" Value="TRUE" />
6   <Var Name="ArmAtDropoff" Value="FALSE" />
7   <Var Name="CylinderEmpty" Value="FALSE" />
8 </Tick>
```

Tick Response Packet: The tick response packet contains the complete snapshot of the system including signal values of each instance of all FB Types. This information is arranged in an XML packet containing the nesting of function blocks in the form of a hierarchy. The purpose of sending the whole system snapshot is to inform BlokIDE about the complete details of current state of the models and avoid the requirement of any further communication until the next tick. An example response packet with nested FBs is presented in Table 7.2. In this example, the simulation host is reporting that the ECC of *pusher* instance took a transition from an ECState named `Back` to another ECState named `GoingForward`. This transition implements the behaviour that a work piece is being placed. Whereas the ECC of the *arm* instance is waiting for the work piece at the pickup location as depicted by the ECState `Waiting`. Similarly, all events and variables of the two BFB instance are also represented in the Xml packet.

BESTPFE.COM List of research project topics and materials

Table 7.2: Tick response Xml packet

```
1  <Tick>
2    <Instance Name="root" Type="DistributionStation">
3      <Events>
4        <Event Name="InputsChange" Scope="Input" Type="EVENT" Value="Present" />
5        <Event Name="OutputsChange" Scope="Output" Type="EVENT" Value="Absent" />
6      </Events>
7      <Vars>
8        <Var Name="PusherBack" Scope="Input" Type="BOOL" Value="1" />
9        <Var Name="PusherFront" Scope="Input" Type="BOOL" Value="0" />
10       <Var Name="ArmAtPickup" Scope="Input" Type="BOOL" Value="1" />
11       <Var Name="ArmAtDropoff" Scope="Input" Type="BOOL" Value="0" />
12       <Var Name="CylinderEmpty" Scope="Input" Type="BOOL" Value="0" />
13       <Var Name="PusherForward" Scope="Output" Type="BOOL" Value="1" />
14       <Var Name="ArmSuck" Scope="Output" Type="BOOL" Value="0" />
15       <Var Name="ArmDrop" Scope="Output" Type="BOOL" Value="0" />
16       <Var Name="ArmToPickup" Scope="Output" Type="BOOL" Value="0" />
17       <Var Name="ArmToDropoff" Scope="Output" Type="BOOL" Value="0" />
18     </Vars>
19     <Instances>
20       <Instance Name="Pusher" Type="DistStnPusher">
21         <Events>
22           <Event Name="PosChange" Scope="Input" Type="EVENT" Value="Present" />
23           <Event Name="ArmStatusChange" Scope="Input" Type="EVENT" Value="Absent" />
24           <Event Name="PusherCtrl" Scope="Output" Type="EVENT" Value="Absent" />
25           <Event Name="ItemStatus" Scope="Output" Type="EVENT" Value="Absent" />
26         </Events>
27         <Vars>
28           <Var Name="PosBack" Scope="Input" Type="BOOL" Value="1" />
29           <Var Name="PosFront" Scope="Input" Type="BOOL" Value="0" />
30           <Var Name="CylinderEmpty" Scope="Input" Type="BOOL" Value="0" />
31           <Var Name="ArmClear" Scope="Input" Type="BOOL" Value="1" />
32           <Var Name="PusherForward" Scope="Output" Type="BOOL" Value="1" />
33           <Var Name="ItemPresent" Scope="Output" Type="BOOL" Value="0" />
34           <Var Name="timeout" Scope="Internal" Type="BOOL" Value="0" />
35           <Var Name="pusherTimer" Scope="Internal" Type="DATE AND TIME" Value="00" />
36         </Vars>
37         <ECState Name="GoingForward" />
38         <ECTransition Source="Back" Destination="GoingForward"
39         Condition="(PosChange) &amp; (!CylinderEmpty &amp; ArmClear)" />
40       </Instance>
41       <Instance Name="Arm" Type="DistStnArm">
42         <Events>
43           <Event Name="PosChange" Scope="Input" Type="EVENT" Value="Present" />
44           <Event Name="ItemStatus" Scope="Input" Type="EVENT" Value="Absent" />
45           <Event Name="ArmCtrl" Scope="Output" Type="EVENT" Value="Absent" />
46           <Event Name="ArmStatus" Scope="Output" Type="EVENT" Value="Absent" />
47         </Events>
48         <Vars>
49           <Var Name="PosReadyToPickup" Scope="Input" Type="BOOL" Value="1" />
50           <Var Name="PosReadyToDropoff" Scope="Input" Type="BOOL" Value="0" />
51           <Var Name="ItemNeedingPickup" Scope="Input" Type="BOOL" Value="0" />
52           <Var Name="ArmToPickup" Scope="Output" Type="BOOL" Value="0" />
53           <Var Name="ArmToDropoff" Scope="Output" Type="BOOL" Value="0" />
54           <Var Name="ArmSuck" Scope="Output" Type="BOOL" Value="0" />
55           <Var Name="ArmRelease" Scope="Output" Type="BOOL" Value="0" />
56           <Var Name="ArmClear" Scope="Output" Type="BOOL" Value="1" />
57           <Var Name="ReleaseDelay" Scope="Internal" Type="INT" Value="0" />
58         </Vars>
59         <ECState Name="Waiting" />
60       </Instance>
61     </Instances>
62     <Connections />
63   </Instance>
64 </Tick>
```

7.2.3.2 Custom Debug Engine

BlokIDE implements a custom debug engine that interacts with the host application to simulate the debugging. Each tick response behaves like a virtual breakpoint and stepping the debugger issues a tick request. Hence, we use the terms debugging and simulating alternatively in BlokIDE context. Simulation can be started by putting Visual Studio in *Simulation* configuration mode and invoking the *Start Debugging* option of Visual Studio. Unlike code debugging, BlokIDE simulator gives the user a visual sense of the current tick with the help of shape highlighting. For instance, the simulator visually highlights the shape that represents the current state of a BFB. Similar highlighting is also performed for transition nodes in an ECC and wire-connections in an FBN. Screenshots of BlokIDE presented in Appendix A, specifically Figures A.11 and A.12 show an ECC and an FBN under simulation respectively.

7.2.3.3 Simulation Tools

In this following text, we present the tools that are implemented for enhancing the developer experience for simulation of IEC 61499 FBs.

Controlling the Simulation: The user can control the simulation by using the toolbar and menu commands, as well as using the keyboard shortcuts available in Visual Studio. Figure 7.13 shows the toolbar and menu commands for starting and controlling the simulation, whereas Table 7.3 presents the keyboard shortcuts for controlling the simulation. Here, *start* of simulation invokes the sequence of actions presented in Figure 7.12. Each tick emulates a breakpoint and puts the IDE in suspended mode. The user may choose to *continue* debugging, or *stopping* after any particular tick.

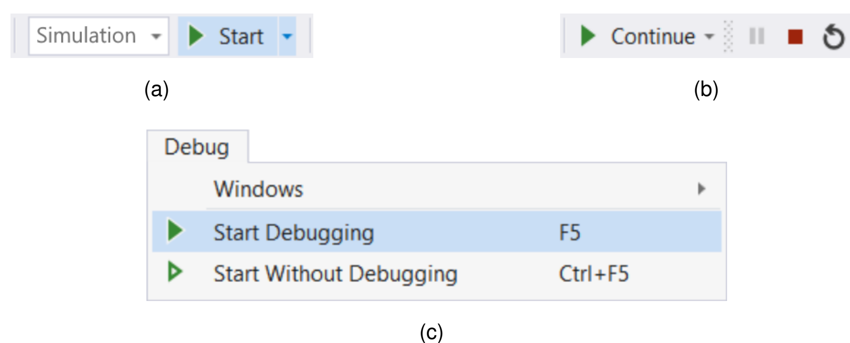


Figure 7.13: Toolbar commands to (a) Start, or (b) Continue, Stop and Restart simulation. Simulation can also be started from the Debug menu (c)

Shortcut Key	Control Feature
F5	Next Tick
Shift + F5	Stop Simulation
Ctrl + Shift + F5	Restart Simulation

Table 7.3: Keyboard shortcuts for controlling the simulation

The Simulator Window: The tick based simulation can accept user inputs for the input events and input variables of the FB being simulated. For this purpose, we have implemented a Simulation Data tool that allows users to specify that whether an input event is *present* or *absent* and provides values for the input variables. This information is used to create and send the tick request packet of the simulation protocol. User input is aided by providing *checkboxes* for Boolean values and *dropdowns* for enumerated values as shown in Figure 7.14.

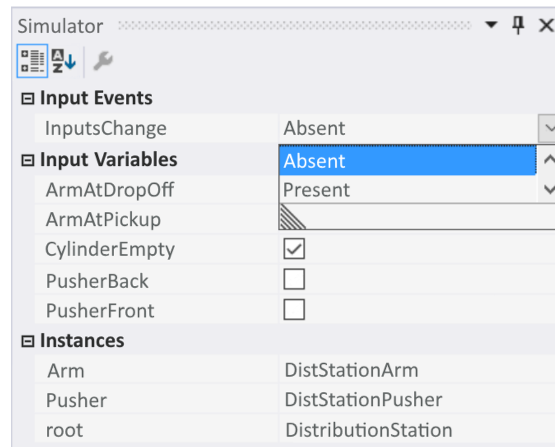


Figure 7.14: Simulation Data tool for taking user inputs

Locals and Stack Trace: BlokIDE simulates FBs in a *tick-by-tick* manner, where each tick represents an execution state of the given system. A history of ticks is maintained in a tool called Call Stack. This tool window maintains the tick history in reverse chronological order as shown in Figure 7.15. The caption of this window is changed to Tick Stack for user friendliness.

Figure 7.15: Tick Stack tool window shows tick history in reverse chronological order

During simulation, each function block type may have several instances, each of which retains a separate copy of variables. Developers need this information to debug their models. For this purpose, we define a concept of tick snapshot, which is a hierarchical collection of all instances along with their variable values. An existing Visual Studio tool known as Locals, is reused to present a tick snapshot. The caption text of this tool window is renamed to Tick Snapshot to make it developer friendly. Tick Snapshot is grid-tree-view with ability to collapse and expand its nodes as shown in Figure 7.16. Furthermore, it works in conjunction with the Tick Snapshot tool such that, based on tick selected in the earlier, the latter presents its details. These tools can thus be used together to examine a given simulation scenario.

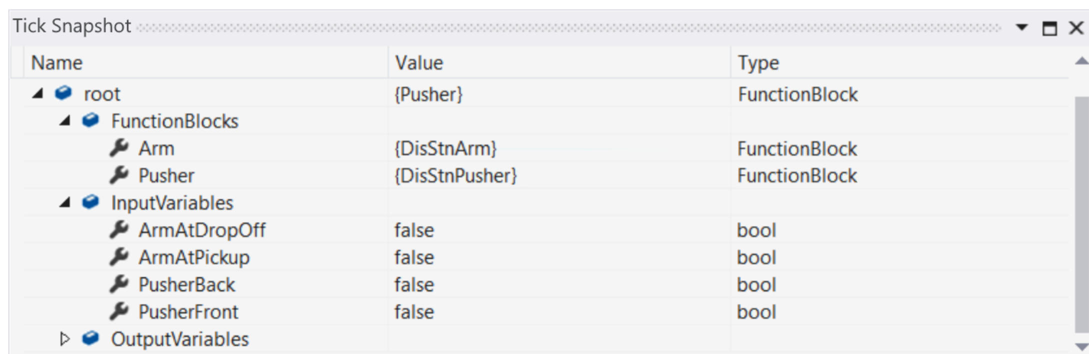


Figure 7.16: Tick Snapshot tool window with hierarchical view of instances and signals

7.2.4 Interfacing with Static Analysers

A set of companion tools are integrated with BlokIDE containing a model verifier and a timing analyser. These static analysis tools rely on the model conversion capability of the FB compiler and process the exported intermediate format. The result of this analysis is reported to the user in BlokIDE living up to the true sense of the word *integrated development environment*. In the following text, we further discuss this integration and the developer experience.

7.2.4.1 Model Verifier

An observer-based formal verification tool [34] is integrated with BlokIDE. The said verification approach relies on a specialised BFB called *observer*. Similar to BFBs, observers comprise of an FB interface and an ECC. The interface of an observer declares inputs to read events and variables such that, all undesired sequences of inputs lead to a specified

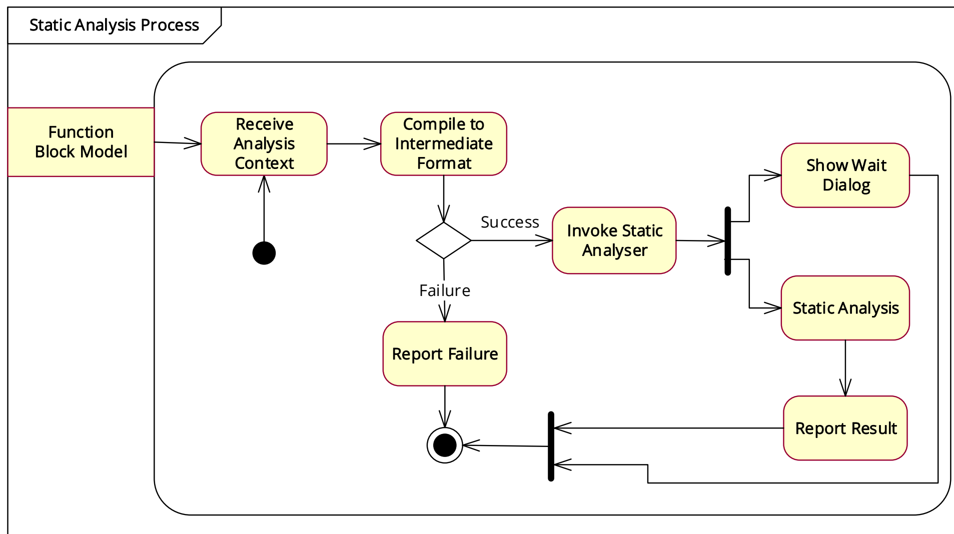


Figure 7.17: Activity diagram for integration with static analysers

violation state. In this manner, a static reachability analysis can be performed on a given FB model to determine if an undesired execution trace is present.

The verification process is invoked using a context menu available on the interface of the observer FB. During the verification process, BlokIDE remains in a waiting mode. This process can take several minutes or even hours depending on the size of the given FB model. Developers are shown a wait dialog with an elapsed time counter as shown in Figure 7.18. The verification can either yield a success outcome or a failure. Success in this context essentially means that the violation state is unreachable. Whereas, the failure result generates a counter example i.e., the discovered execution trace leading to the violation state.

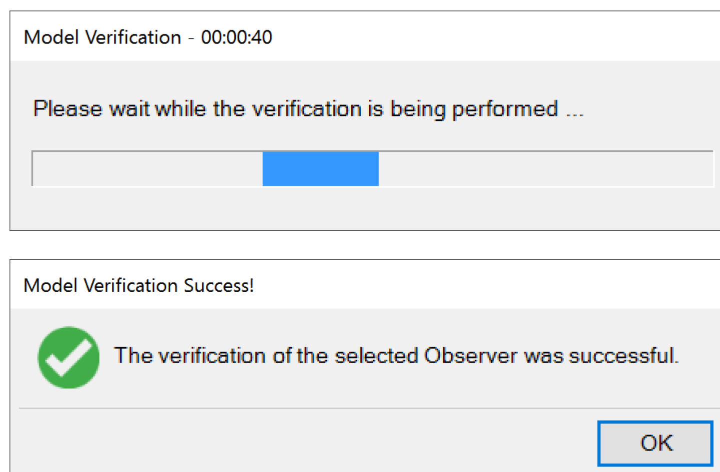


Figure 7.18: Model verification process showing *wait* and *success* dialogs

7.2.4.2 Timing Analysis

BlokIDE uses the synchronous timing analyser [121] for static timing analysis of FB models. The selected FB is converted into an intermediate format for this analysis by the FB compiler, which generates code with timing annotations. These annotations are used by the timing analyser to determine the worst case reaction path and estimate the worst case reaction time (WCRT). WCRT is defined as the number of clock cycles required to produce a reaction in the longest reaction path (i.e., the worst case). This calculated WCRT value provides is a safe over-approximation of the actual WCRT of the system. Developers use this value to determine the guarantees on timeliness requirements of the system and validate the synchrony hypothesis [114]. Depending on the size of the function block, the process of analysis may take up to several minutes. We use the wait dialog as shown earlier to inform the user of the elapsed time. The result of the analysis is a WCRT value presented on the result dialog as shown in Figure 7.19.

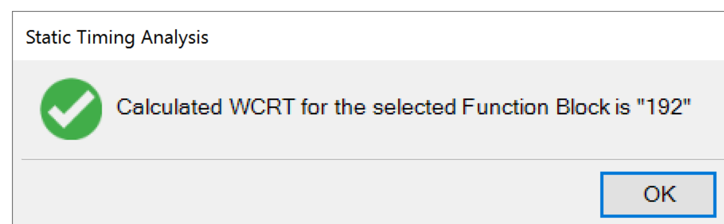


Figure 7.19: Timing analysis *success* dialog

7.3 BlokIDE and Functional Safety

The adopted synchronous execution semantics [105] and the integrated static analysis capabilities are well-suited for safety critical systems. However, compliance with a functional safety standard such as IEC 61508 is also often required. Compliance to such a standard requires establishing evidence and justification against the various requirements and recommendation. In the subsequent text, we illustrate how BlokIDE could be used to implement such a system and offer various justifications on why particular requirements should be deemed satisfied. The presented justification and reasoning should be considered advisory in nature. The actual certification process is performed by independent evaluators as suggested in the standard.

For illustration, we have adapted the *Distribution Station* [95] example previously presented in Section 4.1 as a safety-critical system. Figure 7.20 shows the labelled diagram of the extended distribution station. Due to close proximity with human operators,

additional *safety cameras* (⑰ and ⑱) are installed such that, the control system can halt all mechanical operations upon detecting a hazardous situation. Being a safety-related system, it must conform the guidelines of a functional safety standard (e.g., IEC 61508).

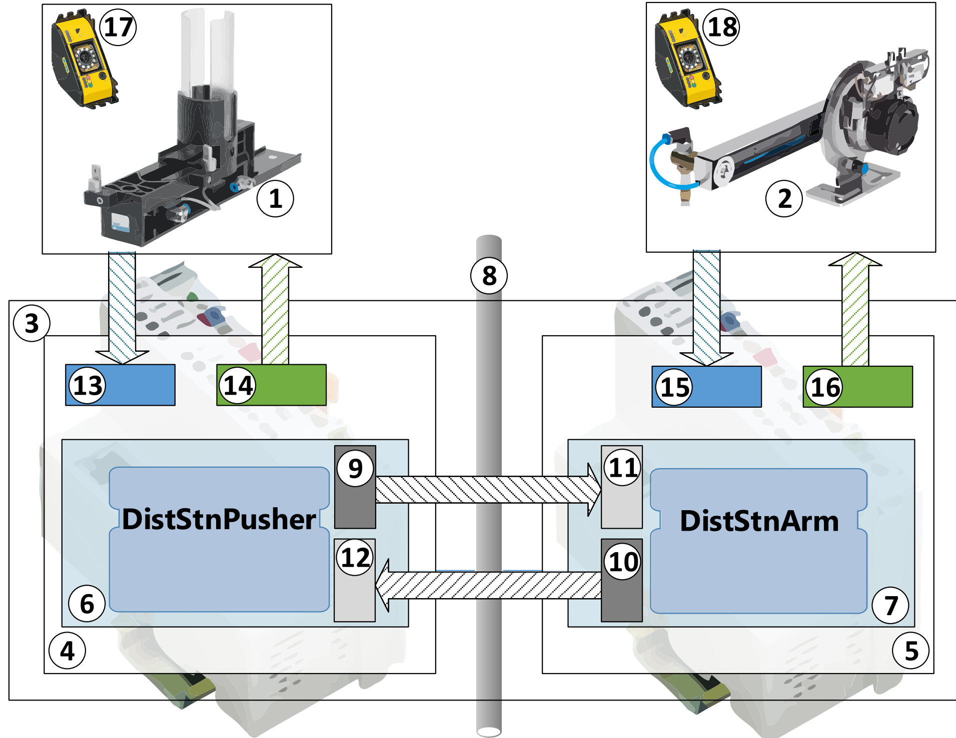


Figure 7.20: Safety critical distribution station system

IEC 61508-2 [94] presents the requirements for E/E/PE systems that are associated with the safety lifecycle governing the design, development, validation, deployment and maintenance activities of the safety-related system. A brief summary of these requirements is available in Section 3.3. In this chapter, we specifically address the E/E/PE design and development requirements, which can be fulfilled by demonstrating conformance in any of the permissible *routes* (see Chapter 3 Section 3.3.3). BlokIDE is suitable for demonstrating compliance of software by avoidance of systematic faults e.g., by following recommendations from IEC 61508-3 Clauses 7.4 – 7.8 [4] and adopting a customised V-Model (route 1_S). Whereas, compliance of hardware can be demonstrated using reliability modelling based on feedback reliability data and hardware fault tolerance (route 2_H). The subsequent sections present a proposal for a BlokIDE based approach for design and developing of E/E/PE system and discuss its suitability towards meeting the said requirements.

7.3.1 Reliability Modelling and Fault Tolerance

BlokIDE implements the model-based safety approach proposed in Chapter 6. This approach enables developers to design both the plant-model as well as the controller software using the model-view-controller (MVC) design pattern. The overall system model is then subjected to probabilistic verification to discovering probability of undesired outcomes. This approach also proposes a way for modelling various types of random failures in the hardware components, which can be applied to the reliability data acquired from sources permitted by IEC 61508-2 Clause 7.4.4.3.3 [94]. The result of this quantitative analysis can then be validated against the requirements of the target SIL as presented in Table 3.4.

The overall process for the proposed validation is presented Figure 7.21. For example, the distribution station can be validated against the requirement for SIL2 i.e., by assuring that $10^{-7} \leq P_{max} < 10^{-6}$ such that, the probability of failure per hour is calculated using Prism model checker [59] with a quantitative specification i.e., a probabilistic property: $P_{max=?}$ [$F^{\leq 360000}$ (`pusher_Fail` || `arm_Fail`)].

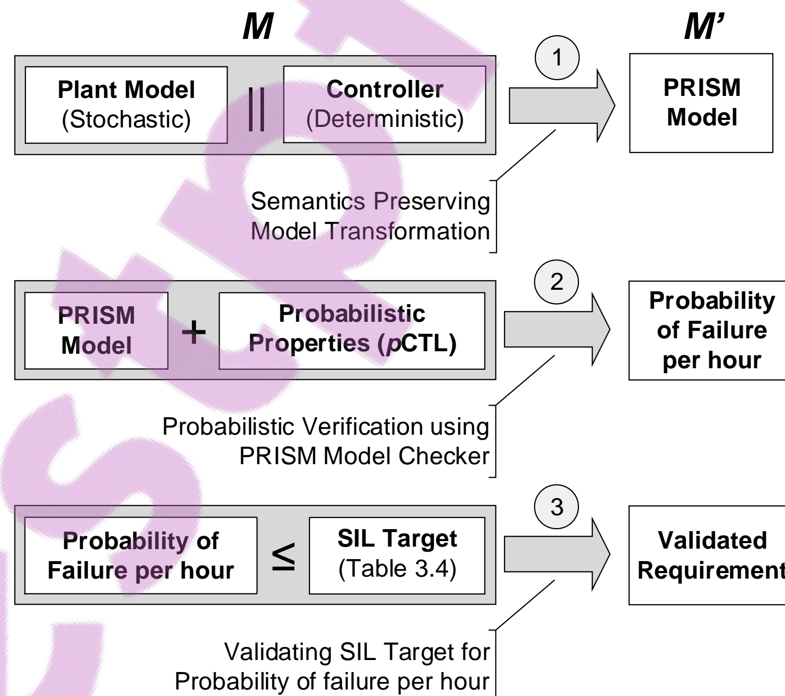


Figure 7.21: Validating SIL target using the proposed approach

7.3.2 Software Realisation

The realisation phase of IEC 61508 software safety lifecycle requires adopting the V-Model for software realisation (IEC 61508-3 Clause 7 [4]). The standard prescribed the purpose and requirements for each of the stages in the V-Model. However, it does not necessitate any specific tools or techniques for performing these activities. Furthermore, customization of the V-Model itself is allowed, provided that certain requirements are met (IEC 61508-3 Table 1 [4]). In this section, we present a BlokIDE based V-Model that conforms to the said requirements. The proposed V-Model is presented in Figure 7.22 such that, the development stages shaded in grey represent the activities that are performed using the BlokIDE and its companion tool-chain. We briefly describe the individual stages of the proposed model and the tooling support offered by BlokIDE as following.

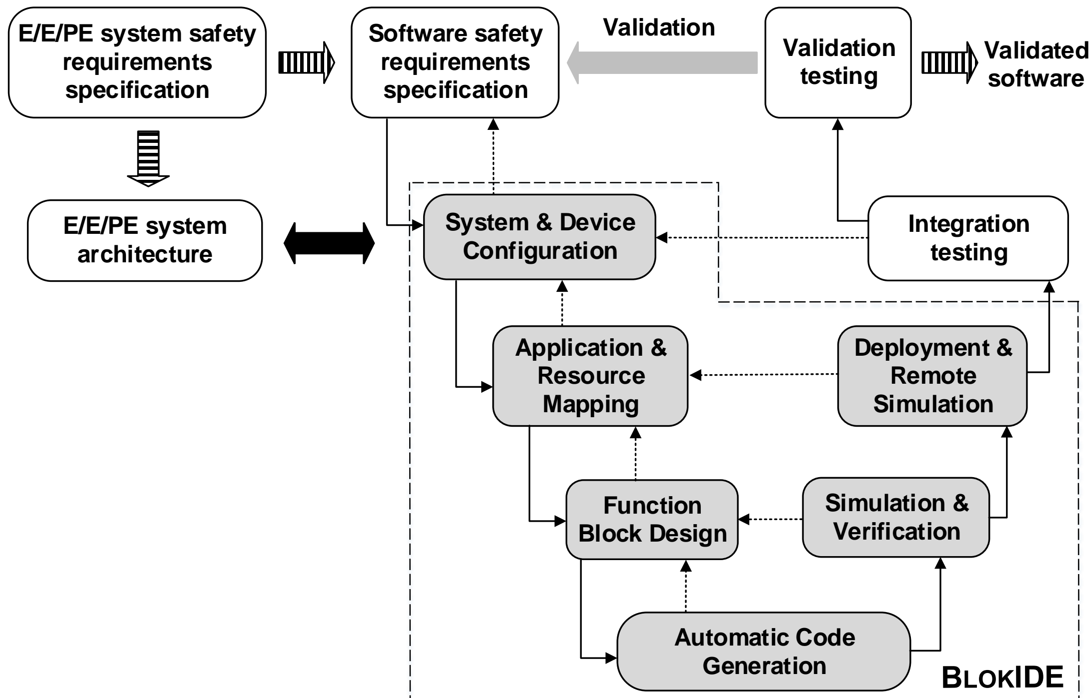


Figure 7.22: V-Model for software realisation using BlokIDE and companion tools

7.3.2.1 System and Device Configuration

The aim of this stage is to create a software architecture that satisfies the requirement specification as derived from the safety requirements and is compatible with the overall system architecture. The activity in this stage involves creating logical counterparts of the physical components in the system. For example, the distribution station consists of a set of sensors, actuators and PLCs, which are directly represented in the corresponding

IEC 61499 System and Devices model. Using Fig. 7.20 as a reference, the activities in this stage would result in the creation of the *System* model (③) and *Device* models (④-⑤) accompanied with their device-specific features e.g., *Resource* models (⑥-⑦) and the respective SIFBs (⑬-⑯). In general, every physical device in the E/E/PE system must be represented as a *Device model* in the *IEC 61499 System*. Commonly used devices, such as some well-known brands of PLC and micro-controllers and their respective device-specific resources, are already available in the BlokIDE *model library*, while others can be designed using the provided model editors and published in the library for reuse.

7.3.2.2 Application and Resource Mapping

The physical layout of the system e.g., connections between sensors, actuators and controllers, imposes requirements for the *Application mapping*. At this stage, the highest level of FBN is developed such that any participating function block only needs to define its interface and the respective IO ports. The behaviour of the said function blocks is implemented later, allowing the practitioners to focus on the high-level application requirements. The physical requirements on the application mapping are preserved e.g., by mapping a *sensor SIFB* to a *Device model* that represents the *physical device* with IO pins connected with the *physical sensor*. However, the function blocks that implement the control logic should be mapped where appropriate e.g., where the require IO is available. The discussion on how to create an effective distributed system using BlokIDE is beyond the scope of this thesis. Further details on this topic are presented in [15].

Using Fig. 7.20 for illustration, this stage involves creating the *Application* and mapping it to the respective *Devices* and *Resources* models. Implementation of the *Application* entails defining interfaces of *DistStnPusher*, *DistStnArm* and instantiating them to implement a function block network. The said function block instances are then mapped to the two *Resource* models (⑥-⑦) in the *System*. This mapping of the instances on different *Device* models induces communication dependencies, which are implemented by the communication function blocks (⑨-⑫) using the Ethernet link (⑧). While Ethernet specific TCP/UDP communication blocks are already available in the BlokIDE *model library*, other types of communication blocks can be created and added to the library.

7.3.2.3 Function Block Design

The top level FBN in the application model consists of one or more function blocks with well-defined interfaces. Whereas, the behaviours of the said function blocks are imple-

BESTPFE.COM
List of research project topics and materials

mented in this stage. During the behaviour implementation of an application function block, additional function blocks might also be created e.g., to implement the FBN of an application level CFB. Using Fig. 7.20 for illustration, this stage involves creating the ECC of the *DistStnPusher* and *DistStnArm* function blocks and coding the respective textual algorithms.

7.3.2.4 Automatic Code Generation

At this stage, all models in the system are already implemented and mapped to corresponding devices. Integration with the synchronous compiler [108] enables automatic code generation, which emits a set of human-readable ISO-C files and corresponding compilation script (*Makefile*). The compiler strictly implements synchronous execution semantics and adopts a correct-by-construction strategy to produce deterministic, dead-lock free code, which is a true representation of the respective models.

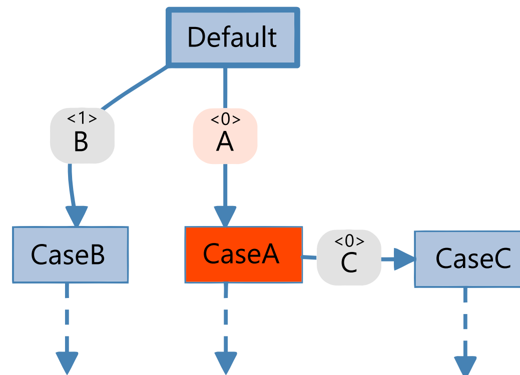


Figure 7.23: Illustration of simulation of a basic function block

7.3.2.5 Simulation and Verification

BlokIDE provides features of *design simulation* and *formal verification*. Together, they provide a means for validating the function block designs i.e., their behaviours implemented as function block networks and ECCs. The design simulator allows step-by-step execution of a given function block by accepting inputs from a simulation input panel and highlighting the changes visually i.e., changes in current states of ECCs and signals flowing through the wire connections. This simulation can be performed both on a network of function blocks, as well as on individual basic function blocks. The simulator is further complemented by an inspection tool to track the individual variable values and event statuses during the simulation. This allows fine-grained control on the simulation

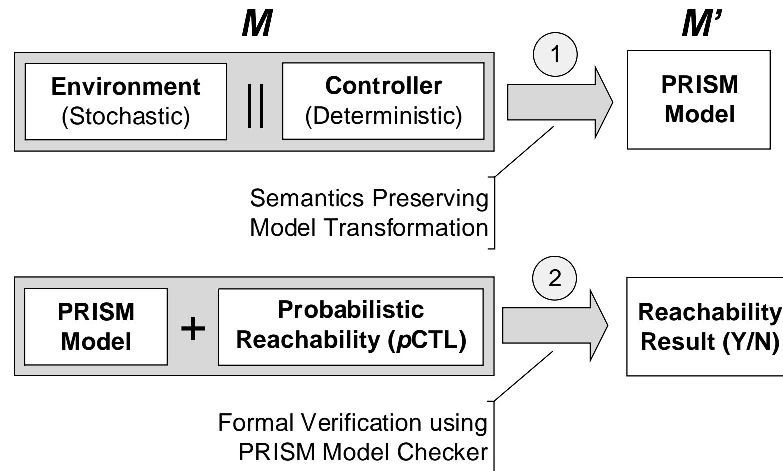


Figure 7.24: Validating qaualitative safety property using the proposed approach

and observability of the models in simulation, and enhance the testability of the overall system. Fig. 7.23 illustrates an ECC under simulation, where the current state of the ECC and the ingress transition are highlighted.

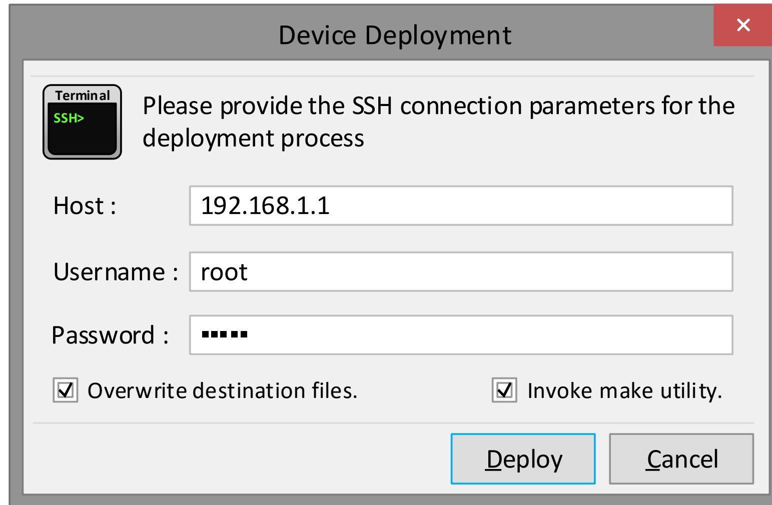
Automatic conversion of function block designs to Prism models not only allows quantitative analysis, it also provides a means to perform qualitative analysis. A typical qualitative verification is performed using a closed-loop arrangement [27, 28] where the inputs the controller are connected to a model of the environment. This model usually represents just the nominal behaviours and can be modelled using the non-determinism transitions of the stochastic function blocks. When connected with the controller, the overall function block network can be converted to Prism language and analysed by the Prism model checker for the reachability of bad states or undesired paths. For example we can verify a qualitative property *"Arm should only attempt to pick up after a workpiece is placed"*, using PCTL specification e.g., $P_{max}=0$ [$\overline{(\texttt{PickingUp})}$ U $(\texttt{ReadyToPickup})$]. Figure 7.24 shows the proposed process for performing the said qualitative verification.

7.3.2.6 Deployment and Remote Simulation

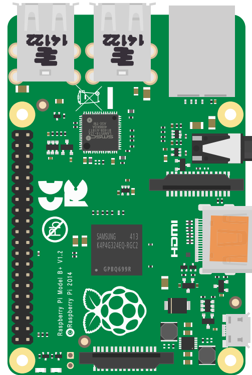
The code generated from basic and composite function blocks is platform-independent. Also, several SIFBs that utilise generic low-level resources e.g., use of timers and communication blocks, also contribute platform-independent code by using the standard GNU system calls. Such systems can, therefore, be compiled into native x86 executables and run on the development machines, as well as on the target devices that support similar GNU standard library. However, in some cases the generated code may be platform-specific e.g., use of IO pins of a PLC bus. For such cases, the code must be compiled

by a platform-specific GNU C Compiler (GCC). BlokIDE supports cross-compliers (e.g., *avr-gcc)* as well as onboard GCC for such platform-specific compilation.

Figure 7.25: Onboard compilation and device deployment over SSH



1. Connect SSH and prepare directory
2. Transfer source files over SSH FTP
3. Invoke makefile for compilation
4. Invoke executable if successful



The onboard compilation uses SSH connections for file transfer (SFTP) and remote invocation of Make utility, which not only removes the need for installing cross-compilers on the development machines but also enables the feature of remote-simulation. This mode of simulation is capable of including 3rd-party code and platform-specific SIFBs and is, therefore, closer to the actual implementation. The process of remote simulation entails the deployment of simulation-ready code on an Ethernet-enabled device, and invocation of resultant executable. The said simulation-ready code contains the simulation protocol layer, which connects to BlokIDE for achieving a visual *hardware-in-loop* simulation. Also, the inspection tool is able to read values from the physical sensors and display the complete snapshot of the current *tick*. This detailed information is useful for validating behaviours

on a single *Device* level e.g., how the various mapped function blocks and hosted resources integrate, communicate and coordinate with each other.

7.3.3 Recommendations in IEC 61508-3 Annex. B

In addition to following the V-Model for software realisation, IEC 61508 provides detailed recommendations on the tools and techniques adopted for safety-related software design and development activities and the supporting tools (IEC 61508-3 Annex. B [4]). Depending on the target SIL, these requirements may be offered as highly recommended (HR), recommended (R), no recommendation (—), or not recommended (NR). Based on this recommendation scale, we evaluate the proposed BlokIDE based approach for its suitability for the realisation of safety-related software. The summary of this evaluation is presented as following.

7.3.3.1 Design and Coding Standard

IEC 61508-3 Table B.1 [4] lists a set of requirements on design and coding standard for the safety-related software. The proposed MDD approach helps the practitioners achieve fulfilment of many of the said requirements. A summary of the fulfilled requirements and related comments are presented as follows.

Table 7.4: Recommendations on Design and Coding Standard

	Technique/Measure	**SIL1**	**SIL2**	**SIL3**	**SIL4**	BlokIDE
1	Use of coding standard to reduce likelihood of errors	HR	HR	HR	HR	✓
2	No dynamic objects	R	HR	HR	HR	✓
3a	No dynamic variables	—	R	HR	HR	✓
3b	Online checking of the installation of dynamic variables	—	R	HR	HR	—
4	Limited use of interrupts	R	R	HR	HR	✓
5	Limited use of pointers	—	R	HR	HR	×
6	Limited use of recursion	—	R	HR	HR	✓
7	No unstructured control flow in programs in higher level languages	R	HR	HR	HR	✓
8	No automatic type conversion	R	HR	HR	HR	✓

1. The automatic code generator uses the same naming convention as used by the software developer for function block types, instances, events and variables. The

remaining structure of the code i.e., encoded state machine, and logical wire connections between IO ports is generated in a mechanised way that ensures uniformity. All comments provided in during the design phase, are emitted in the corresponding code elements, which ensures that the generated code is well-annotated.

2. Code generation in BlokIDE does not generate dynamic objects.

3. Code generation in BlokIDE does not generate dynamic variables and does not use dynamic memory allocation. Therefore, on-line checking of dynamic variables is not required.

4. Due to reactive nature of synchronous execution semantics, the generated code does not utilise interrupts. The generated code relies on synchrony hypothesis [114] to ensure that events are not missed.

5. The generated code uses pointers for referencing function block instances, which means this requirement is not completely fulfilled. However, much of the code that uses pointers is automatically generated through a correct-by-construction strategy and is, therefore, suitable for safety-related applications. However, pointers must be used with care in the ECC algorithms.

6. Due to the iterative nature of reactive design, the generated code does not use recursion.

7. The control flow in the proposed approach is modelled using states and transitions of IEC 61499 ECC, which is unambiguous and a formally defined structure under synchronous execution semantics[122].

8. Restrictions on wire connections ensure that only compatible inputs and outputs are connected. The *ANY* type variables, which is a special data type in IEC 61499, are statically resolved to concrete types during the code generation phase. Together, this ensures that there are no automatic type conversions

7.3.3.2 Modelling

IEC 61508-3 Table B.5 [4] provides the recommendations on modelling techniques that may be used for designing safety-related software. BlokIDE facilitates the fulfilment of these recommendations as justified below.

Table 7.5: Recommendations for Modelling

	Technique/Measure	SIL1	SIL2	SIL3	SIL4	BlokIDE
1	Data flow diagrams	R	R	R	R	×
2a	Finite state machines	—	R	HR	HR	✓
2b	Formal methods	—	R	R	HR	✓
2c	Time Petri nets	—	R	HR	HR	×
3	Performance modelling	R	HR	HR	HR	✓
4	Prototyping/animation	R	R	R	R	✓
5	Structure diagrams	R	R	R	HR	✓

1. IEC 61499 uses textual algorithms for data processing and, therefore, do not meet the requirement of using data flow diagrams. However, this is not a highly-recommended technique even for higher SILs.

2. The ECCs of basic function blocks are Moore-type finite state machines and thus fulfil the corresponding requirement.

3. The synchronous execution semantics is a set of mathematical rules for executing the IEC 61499 function block networks and ECCs. The unambiguous nature of these semantics and their strict application on the automatic code generation is sufficient to justify the fulfilment of the requirement to using formal methods (IEC 61508-7 Sec. C.2.4.1 [123]).

4. Timed Petri nets are used for verification of timing related aspects of the system. Although the proposed is well-suited for such analyses, however BlokIDE does not currently support this feature.

5. The purpose of performance modelling is stated in IEC 61508-7 as *"To ensure that the working capacity of the system is sufficient to meet the specified requirements"* [123]. Due to the mechanised manner of code generation, analytical methods can be used to estimate several performance metrics e.g., memory footprint, stack size. In the context of real-time system, one of the most important performance requirement is to assure reaction time. BlokIDE meets this requirement by using computing worst-case reaction time using the BlokIDE static timing analyser [121].

6. The function block design simulator serves the purpose of prototyping, where a high-level design in simulated on a development machine in the absence of low-level implementation details.

7. The proposed MDD approach uses visual diagrams for the system design at various stages of development. The deliverables at each stage include structural diagrams

such as *Application* model, *Device* and *Resources* configuration, and *Function Block* models.

7.3.3.3 Modular Approach

Using a modular approach for modelling safety-related software not only improves the system testability but also allows module reuse. IEC 61508-3 Table B.9 [4] provides a set of relevant recommendations, which are listed in Table 7.6. The corresponding comments and justification are presented as following.

Table 7.6: Recommendations on Modular Approach

	Technique/Measure	**SIL1**	**SIL2**	**SIL3**	**SIL4**	BlokIDE
1	Software module size limit	HR	HR	HR	HR	✓
2	Software complexity control	R	R	HR	HR	×
3	Information hiding / encapsulation	R	HR	HR	HR	✓
4	Parameter number limit / fixed number of subprogram parameters	R	R	R	R	✓
5	One entry / one exit point in subroutines and functions	HR	HR	HR	HR	✓
6	Fully defined interface	HR	HR	HR	HR	✓

1. Module sizes are directly dependent on the size of a function block network i.e., the number of instances in the network. The modular nature of IEC 61499 allows refactoring of large function block networks to extract sub-modules, which allows developers to adjust the sizes of corresponding modules.

2. MDD approaches are amenable to the techniques for estimating and managing the software complexity. However, BlokIDE currently does not support this functionality.

3. Function block interfaces are highly suited for encapsulation of information and data. The concept of internal variables and clear scope of inputs and outputs ensure that the data is isolated, which minimizes the possibility of coding errors.

4. The explicit definition of input/output events and variables on the function block interfaces, ensures that a fixed number of parameters flow between modules and function blocks.

5. ECCs of function blocks have a single initial state, ensuring single entry point. Execution of ECCs commences from the said initial state and invokes the action-

sets of subsequent states in the ECC in a pre-defined sequential order. This manner of executing algorithms, therefore, have a single entry and exit point.

6. Modules in the proposed approach are composite function blocks that encapsulate a function block network with explicitly defined interfaces. All inputs and outputs that flow in and out of a module are explicitly exposed on the respective interface.

7.4 Conclusion

In this chapter, we presented an MDD approach for design and implementation of safety-related software, which is compatible with the recommendation of IEC 61508 functional safety standard. The proposed V-Model uses BlokIDE for establishing a developer-friendly process of realisation of such software. Several requirements and recommendations for such software and the respective support tools are shown to be satisfied. In its current state, BlokIDE appears to be sufficient for targeting SIL1 and SIL2 since all *highly-recommended* as well as most *recommended* techniques for applications targeting SIL1 and SIL2 are fulfilled. However, the proposed V-Model should only be considered as a proposal that requires much effort for its validation. A possible direction for future research and development is to enhance the companion tool-chain to add support for planning and specification phases of the safety lifecycle as well as to device mechanisms to automatically generate evidence of conformance to the standard. A free academic version of BlokIDE is freely available for download at `http://timeme.io/`

Bestpfe.com

8

Conclusion

In this thesis, we presented a model-based approach for functional safety of complex control systems, specifically industrial control systems that are developed using IEC 61499. The are four key enablers this approach: a novel structure called *stochastic function block* (SFB), a set of *transformation rules* to convert function blocks to Prism language in semantics preserving fashion, an IDE named BlokIDE that implements SFBs and model transformation, and proposal for conforming to IEC 61508 recommendations based on a customised V-Model based on BlokIDE. In the following sections, we summarise these contributions and present some future research directions.

8.1 Model-Based Safety

8.1.1 Summary and Contributions

Industrial automation systems are inherently safety critical and must be analysed for their safety-assured operation using various qualitative and quantitative techniques. Unfortunately, all existing approaches for analysing IEC 61499 function blocks are qualitative in

BESTPFE.COM
List of research project topics and materials

nature. The root cause for this is the lack of provision for the quantitative modelling of stochastic aspects of the system. These aspects arise from the random failures in the hardware components and the environmental non-determinism. In this thesis, we filled this lacuna by proposing a novel structure named stochastic function blocks (SFB) in Chapter 6. Using this structure, practitioners can model *probabilistic* failures, as well as *non-deterministic* behaviours and their respective effects.

SFBs extend the synchronous execution semantics with the semantics of Markov decision processes. Under these semantics, SFBs execute in discrete time in a step-by-step manner, each step is akin to a *tick* in the synchronous execution [105]. furthermore, SFBs also use the delayed communication model for loading inputs and emitting outputs. This renders the semantics of SFBs as a strict superset of that of BFBs such that, a BFB is a trivial SFB that contains no stochastic behaviours. This containment of semantics allows composition of SFBs and BFBs together to form function block networks (FBN). Consequently, co-design the plant-model and the controller in a closed-loop, which is a typical design pattern for validation and verification activities [26, 27].

The resultant overall system model can be used for qualitative and quantitative analysis e.g., by modelling component failure rates and their failure affected behaviours we can analyse the system reliability. Furthermore, probabilistic temporal logic can be used to estimate quantitative risk. This result can be used to discover relationships between component reliabilities and the overall system hazards. The main contributions of SFBs are as following.

1. We presented a formal definition of the structure of SFBs, which describes its various elements in an unambiguous manner. This definition partitions the states of encapsulated *execution control chart* (ECC) in three partitions namely deterministic states, probabilistic states, and non-deterministic states such that the deterministic states are the tick-boundaries.

2. We presented execution semantics for SFBs that are compatible the synchronous semantics of BFBs. This permits the composition of the two models. The enforced *alternate model* of SFB ensures that each execution tick evaluates at most 1 deterministic transition followed by at most 1 stochastic transition. Whereas, the synchronous execution semantics of BFBs allow at most 1 deterministic transitions. This ensures that execution of both types of function blocks starts and finishes on a deterministic state in every tick.

3. We also provided guidelines on how to model the occurrence and the effects of the two types of probabilistic failures i.e., *per-time* and *on-demand* failures. This enables

a sound approximation for the quantitative assessment of the overall system. We further demonstrated how to perform this assessment using probabilistic verification e.g., by using PCTL properties and a probabilistic verifier like the Prism model checker.

8.1.2 Future Work

A large set of systems can benefits from the structure and semantics of SFBs for the purpose of quantitative assessment. However, SFBs are discrete-time models and require discrete-time approximation of real-time behaviours. Such an approximation may not always be possible e.g., in cases where failures occur in a continuous value paradigm like zero-crossings and instantaneous fluctuations. A continuous-time SFB can be devised that can model the continuous dynamics of various aspects of the hardware and the environment. The application of such a model would not just be limited to safety analysis.

8.2 Function Blocks to Prism Conversion

8.2.1 Summary and Contributions

The quantitative assessment of IEC 61499 function blocks and SFBs relies on the underlying semantics-preserving transformation to Prism models, which can be analysed using the Prism model checker. In Chapter 5 we presented a set of transformation rules for converting function blocks into Prism language. These rules are sound such that the generated Prism model preserved the execution semantics of the source FBN.

The transformation rules use a formal definition of IEC 61499 function blocks [106] to encoded their structure and semantics as Prism language [58]. This conversion bridges the syntactic gap between the two notations by using encoding mechanisms i.ee. the following.

1. The states of a given SFB are encoded using valuations of a state-encoding variable v_s.
2. Events are encoded using Boolean variables. The value of such a variable is mapped to the corresponding event status i.e., *true* value means that the event is *present*

and *false* means that the is *absent*.

3. Wire connections of FBN are resolved as direct read access as permitted in the Prism models. Names of the output variables are fully qualified for this purpose e.g., `variableName_moduleName`. A simple source lookup operation resolves the name the source variable, which is used to replace the reference to the input variables.

4. Transitions in SFB are ordered such that the highest order enabled transition is taken during execution of a tick. This order ensures that the behaviour is an SFB is deterministic. However, Prism commands do not have an explicit order. We bridge this gap by using explicit-negations such that a lower order transition is mapped to a command that carries the negation of all higher order transitions Boolean guards. This ensures that a lower order transition can only be enabled when none of the higher order commands can be taken.

5. During execution, if an SFB cannot take any transitions it remains at its current state without executing any actions. This implicit stay operation is simulated by using self-loops with the lowest order i.e., it can only be taken when no other transitions can be enabled from the current state.

The semantics of Prism language are more expressive than SFB semantics, which allows a sound mapping between the two structures. The transformation rules are applied on individual SFBs ($\texttt{SFB}_i$) in a given FBN ($\texttt{FBNetwork}$) to generate respective Prism models ($\mathcal{M}_i$) in the generated Prism model ($\mathcal{M}$) as presented in Equation 8.2.1. A summary of these rules is presented as following.

$$\texttt{SFB}_i \in \texttt{FBNetwork} \iff M_i \in \mathcal{M} \tag{8.2.1}$$

$T1$: This rule performs state-encoding using an integer variable such that, each state in an SFB is mapped to a distinct value using the state-encoding function $\mathcal{V}_s$.

$T2$: This rule creates Boolean variables to represent output events of an SFB.

$T3$: Theses rules create variables in the generated Prism modules to map the output and local variables respectively.

$T4$: Since SFBs are Moore-type structures, they may optionally contain initial state actions, which must be encoded correctly in the Mealy-type structure of Prism

modules. This rule is used for ensuring the equivalence of the initialisation step i.e., by generating an additional command for the initialisation.

$T5$: All SFB transitions are encoded as Prism commands. Starting from the initial state, this rule converts SFB transitions that have both its predecessor and successor states in the deterministic states partition $q' \in Q^d$. The result of this conversion is a Prism command.

$T6$: This rule mimics the implicit-stay operation of SFBs, where the current state remains unchanged if none of the egress transitions could be enabled. The result of this rule is a self-loop command, which can only be taken if none of the transitions are enabled.

$T7$: This rule encodes SFB transitions that have a probabilistic successor state $q' \in Q^p$. In this case, this transformation rule loads all further successors to produce a single Prism command i.e., all elements of the form $q \xrightarrow{(e,b)} q' \xrightarrow{p_0} q''_0,\ \ldots, q \xrightarrow{(e,b)} q' \xrightarrow{p_k} q''_k$ are combined to form a single command.

$T8$: This rule encodes SFB transitions that have a non-deterministic successor state $q' \in Q^n$. In this case, this transformation rule loads all further successors to produce a one Prism command per non-deterministic choice i.e., all elements of the form $q \xrightarrow{(e,b)} q' \rightarrow q''_0,\ \ldots, q \xrightarrow{(e,b)} q' \rightarrow q''_k$ are mapped to individual Prism commands.

Here, rules $T1 - T3$ generate variables and rules $T4 - T8$ generate commands in the resultant Prism module. These rules are applied to every SFB instance in a given FBN to produce a Prism model.

8.2.2 Limitations and Future Work

Due to the syntactic differences in the SFB and the Prism language, some limitations have been applied.

1. Event-data association has been simplified such that, we assume that all variables are associated with a special event `true`, which is *present* in every tick. This results in variables values to be updated in every tick.
2. Algorithms in the ECCs are simplified to contain only assignment statements. Branching statements and loops are assumed to be lifted to the transition guards.

3. Only Boolean and Integer variables are supported. This limitation is due to Prism model checker that only supports the said variable types.
4. Only addition, subtraction, and multiplication are allowed as valid arithmetic operations.

Some of the limitation presented above can be removed by extended the transformation rules presented above i.e., use of additional variables to store input variables can solve the first limitation. The second limitation can be removed by lifting control flow graphs of algorithms and generating additional Prism commands. The last two limitations are due to Prism model checker itself, which only allows simple arithmetic operations on integer variables. This limitation is common in most model checkers and, therefore, cannot be resolved by choosing a different model checker.

8.3 Meeting IEC 61508-3 Requirements with BlokIDE

8.3.1 Summary and Contributions

We presented BlokIDE in Chapter 7, which is a model-driven development tool-chain for industrial automation systems. It implements IEC 614199 as well as the proposed automatic model transformation to Prism language. BlokIDE provides model editors for IEC 61499 for creating function blocks and other design artefacts, which are later used for automatic code generation under synchronous execution semantics. Various attributes of BlokIDE such as, its execution semantics, the companion tools, and the quality of the generated code are discussed for their amenability towards recommendation from IEC 61508-3. In this chapter, we also presented a proposal for the model-driven development of safety critical industrial automation systems using a customised V-Model. This V-Model uses IEC 61499 design artefacts for the purpose of various architecture and design activities and uses BlokIDE companion tools for supporting various types of analyses. The main contributions are as following.

1. BlokIDE a model-driven development tool-chain that implements model editors for IEC 61499 and provides support for compilation, simulation, formal verification, and static timing analysis.
2. BlokIDE implements SFB model editor and automatic conversion to Prism language using an implementation of the transformation rules.

3. A customised V-Model for design and implementation of safety critical systems. Each phase of the V-Model implements the stated purpose of the respective phase of the IEC 61508 prescribed V-Model.

4. A detailed gap analysis to find the level of conformance that can be achieved towards the requirements of IEC 61508-3.

Using this gap analysis, we believe that BlokIDE is useful for implementation of systems that target SIL2 for conformance.

8.3.2 Future Work

BlokIDE puts a lot of emphasis on developer friendliness and, therefore, provides a lot of functionality out of the box. The built-in library of function blocks can, however, be extended. Furthermore, the suitability of BlokIDE for IEC 61508-3 requirements is limited due to several gaps as described in Chapter 7. An academic version of this tool-chain is available for download from `http://timeme.io`. We welcome any requests for access to the source code for future extensions of BlokIDE.

Appendices

Bestpfe.com

A

A.1 BlokIDE Screenshots

BlokIDE offers a project based solution with pre-existing project templates. User can

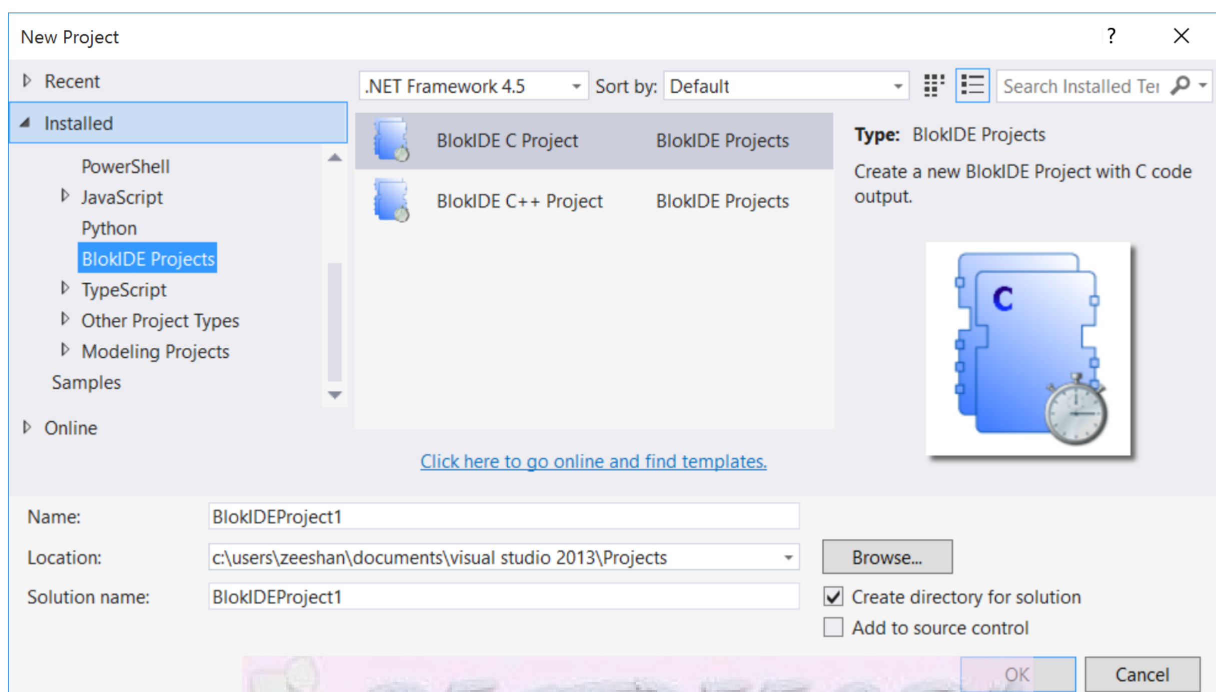


Figure A.1: Creating a new BlokIDE project

BESTPFE.COM
List of research project topics and materials

select a template for creating a new project, which also loads a preset of system settings e.g., compiler path and the library path. In the context of current work, we only use the BlokIDE C Project template as shown in Figure A.1. The created project is initially empty as shown in Figure A.2 ①. The user can add models e.g., basic and composite function blocks into the created project by bringing up the context menu through right click ②, and choosing the option to add a new item ③. At this stage, the user is given an option to select an item template e.g., he may choose to create a basic function block.

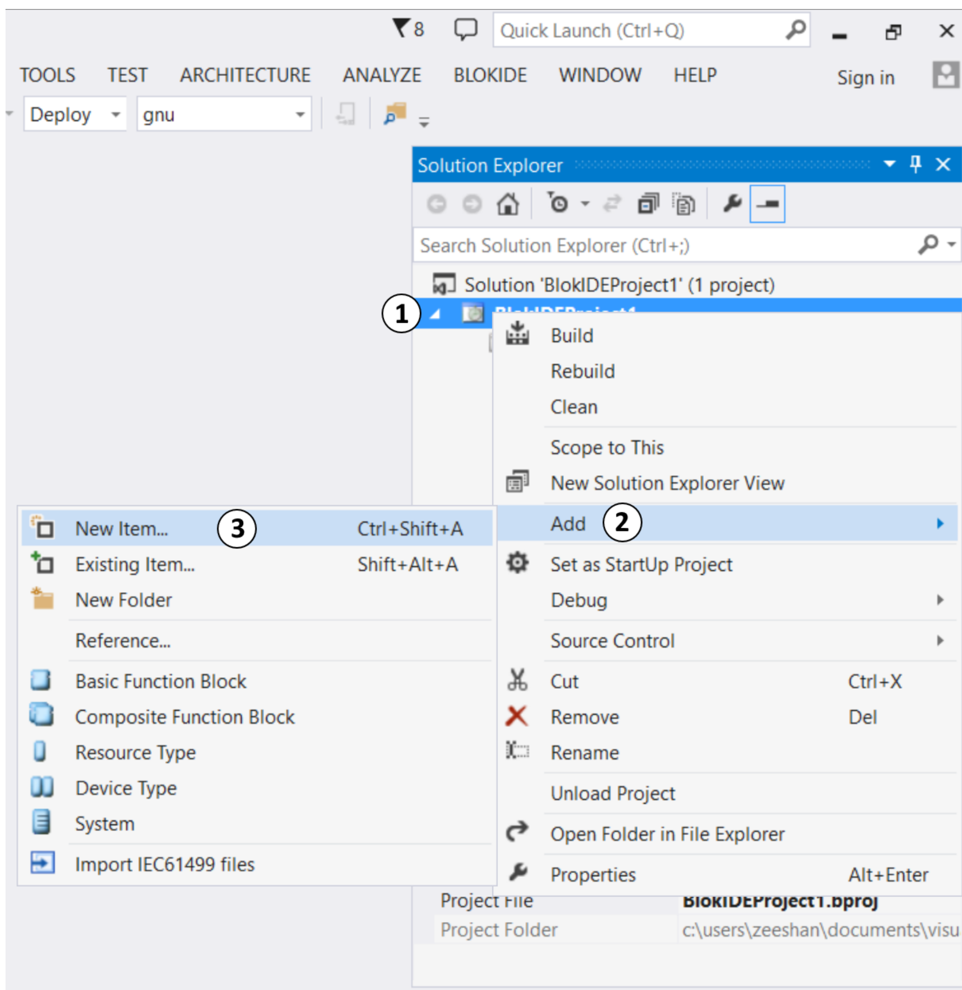


Figure A.2: Creating a new Basic Function Block

A range of templates is available for creating models of different IEC 61499 types as shown in Figure A.3. A new basic function block (BFB) created from the template comes with a blank function block interface and an execution control chart (ECC) with a single state marked as initial state.

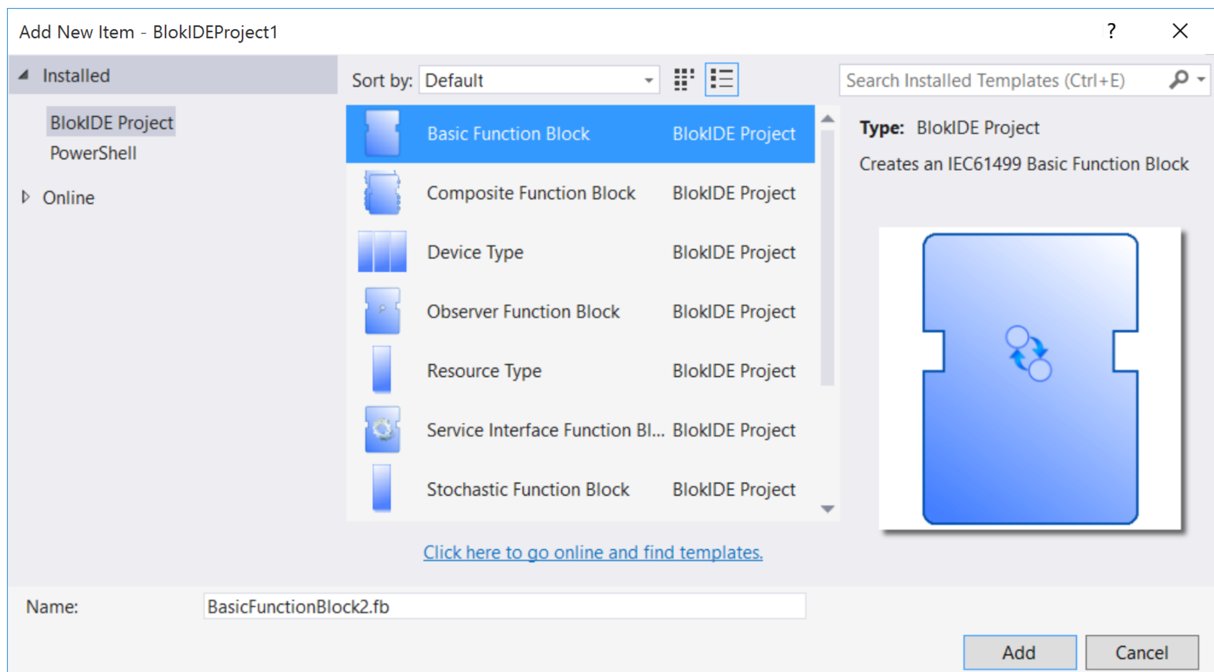


Figure A.3: Function Block Interface Editor

The user can add input and output events and variables using the tools provided in the Toolbox pane as shown in Figure A.4. Optionally, the user can create associations between events and variables as prescribed by the IEC 61499 standard.

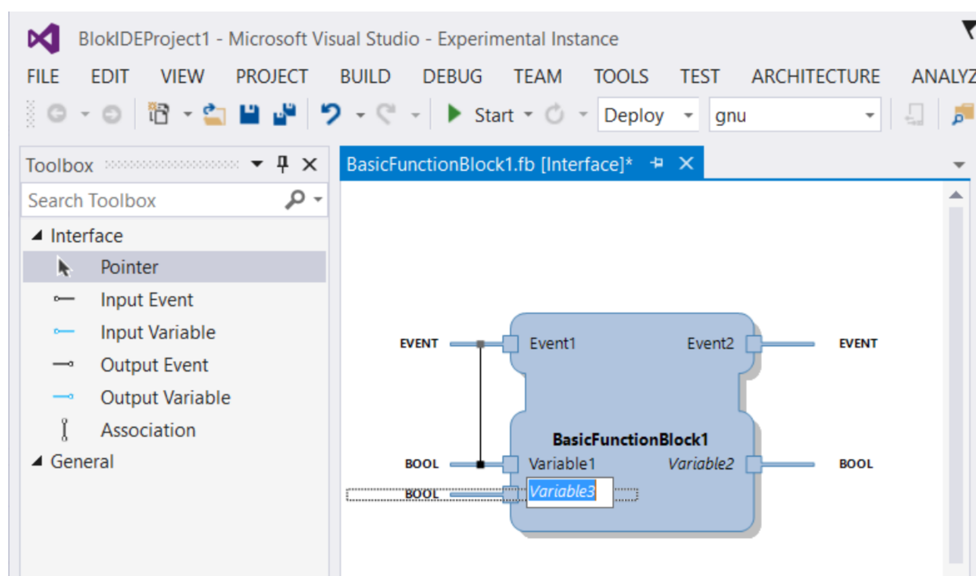


Figure A.4: Creating a new Composite Function Block

Initially, the ECC of the created BFB only contain a single state. User can add more states and connect them using transition by using the tools provided in the Toolbox pane. With the help of the Properties window (see Figure A.5), user can assign condition guards to transitions e.g., an event and/or a Boolean expression over the input and local variables.

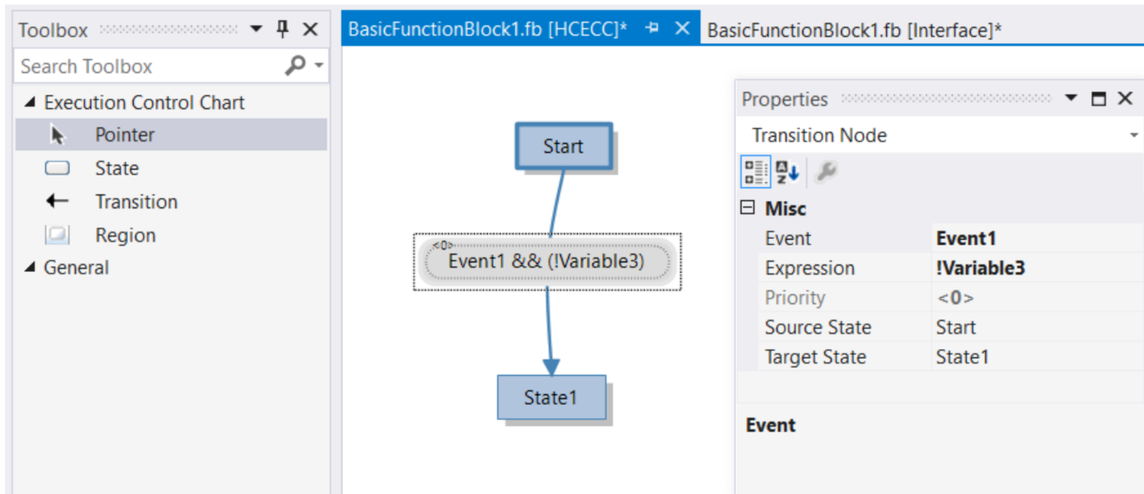


Figure A.5: Creating states and transitions

Recall synchronous execution semantics from Chapter 5, ECC transitions are strictly ordered with their declaration index as priority. That is, a transition created first, has a higher priority than a transition created later. However, the user may choose to edit the priority of a transition, which can be done through the context menu of the transition node as shown in Figure A.6.

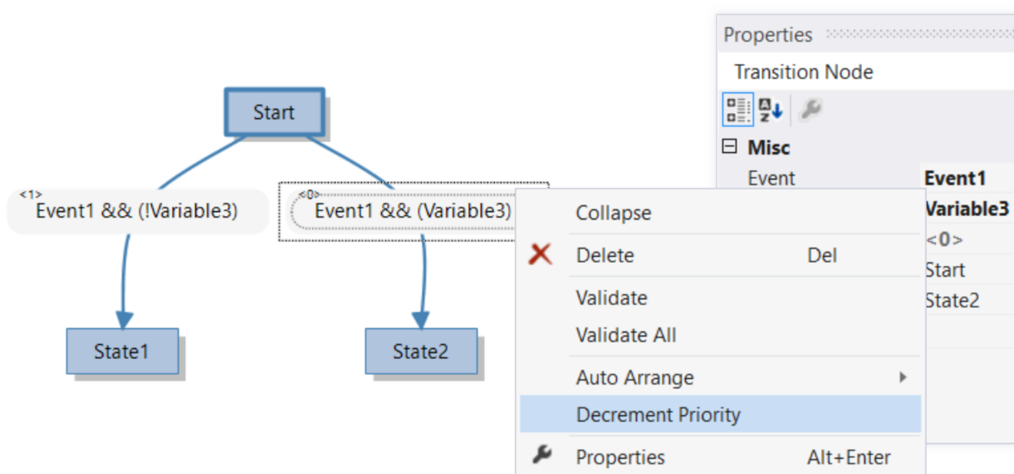
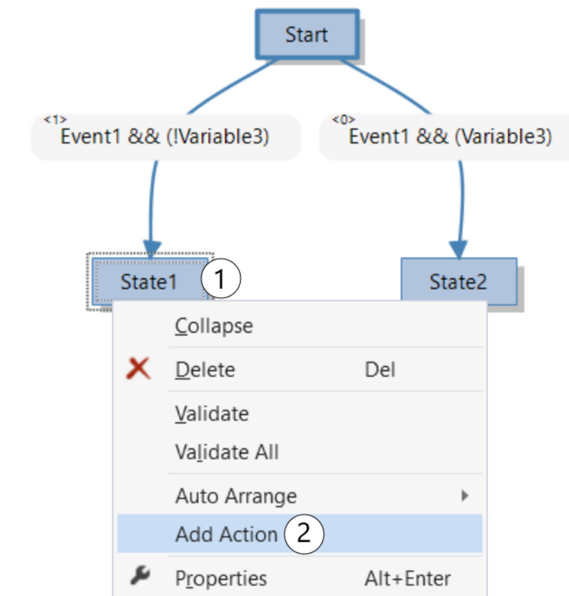


Figure A.6: Edit transition priority

User can assign state-entry actions to a state by choosing to right click on the state ① and using the context menu to choose Add Action ② menu item. This bring up the

Action Item Dialog with *dropdowns* for algorithms ③ and output events ④ lists. Once selected, user clicks on the OK button ⑤. By completing this process, new actions are assigned to the selected state ⑥ as shown in Figure A.7.

Before:



After:

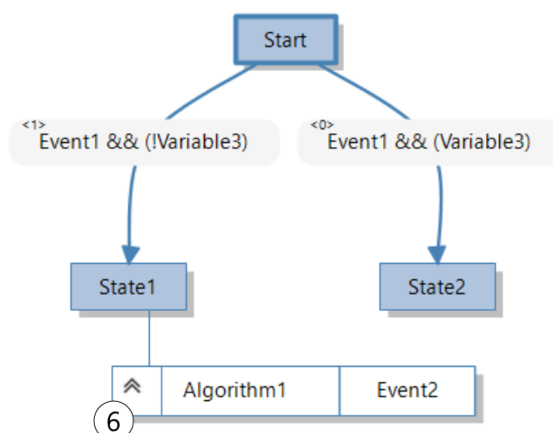


Figure A.7: Adding state-entry actions to states

Similar to the BFBs, composite function blocks (CFB) can also be created using item templates. As before, this creates a new function block model with an empty function block interface. Unlike BFBs, CFBs encapsulate a function block network (FBN). This

FBN is initially blank, but can be populated with references of other blocks by a simple drag-drop operation e.g., dragging BasicFuntionBlock1 model file on to the network diagram creates a new instance of the said function block type. These references or instances can be connected with each other using the Connection tool in the Toolbox pane. Using this tool, user can click on an output port of a function block reference, and drag to connect with a compatible input port of another function block reference.

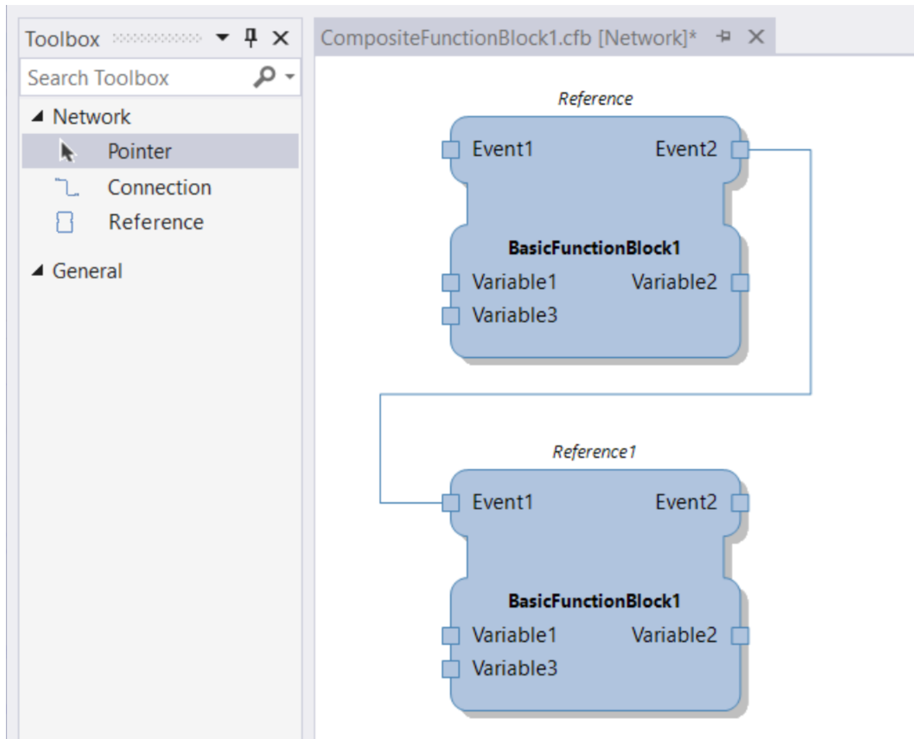


Figure A.8: Function Block Network Editor

Given an FBN, the instance of a function block may become outdated as new changes are made to existing function blocks. Resultantly, some existing connections may no longer

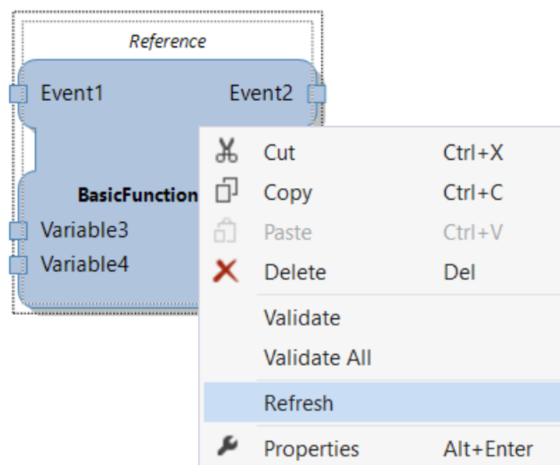


Figure A.9: Refreshing the reference to load changes to referred function block interface

be possible. Furthermore, new IO ports declared on existing function blocks will also not be reflected in the given FBN. To fix this problem, BlokIDE provides a context menu option named Refresh (see Figure A.9). This looks up the reference source and reloads the function block interface.

BlokIDE also provides a Model Explorer to examine and edit properties of function blocks. In a BFB, model explorer shows all states, algorithms, events and variables, whereas in a CFB, model explorer shows encapsulated function block references. Figure A.10 shows the two views of the model explorer side by side.

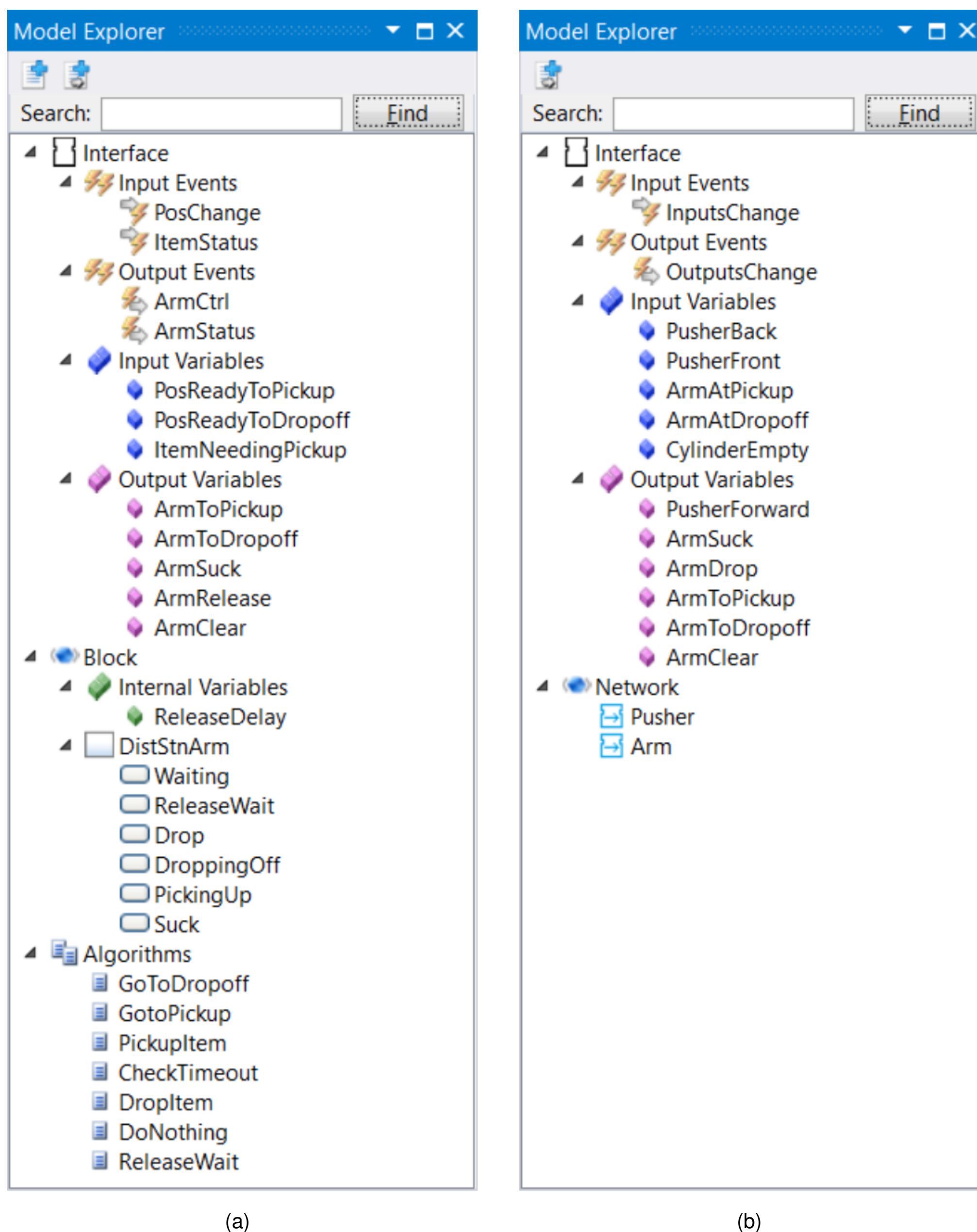


Figure A.10: Model Explorer tool window showing a BFB (a) and CFB (b)

Individual BFBs can be simulated in BlokIDE as shown in Figure A.11. The ECC of the given BFB is highlighted for current state. Furthermore, users can inspect the values of all events and variables using the Tick Snapshot tool. Inputs are provided using the Simulator tool window for the next tick.

Figure A.11: Simulation of an ECC of a BFB

BlokIDE can also simulate CFBs and their encapsulated FBNs as in Figure A.11. The flow of events and variables are highlighted by changing the colours of the corresponding wire connections. The instance hierarchy (shown on bottom left side) can be used to select a particular instance in the simulation for further inspection.

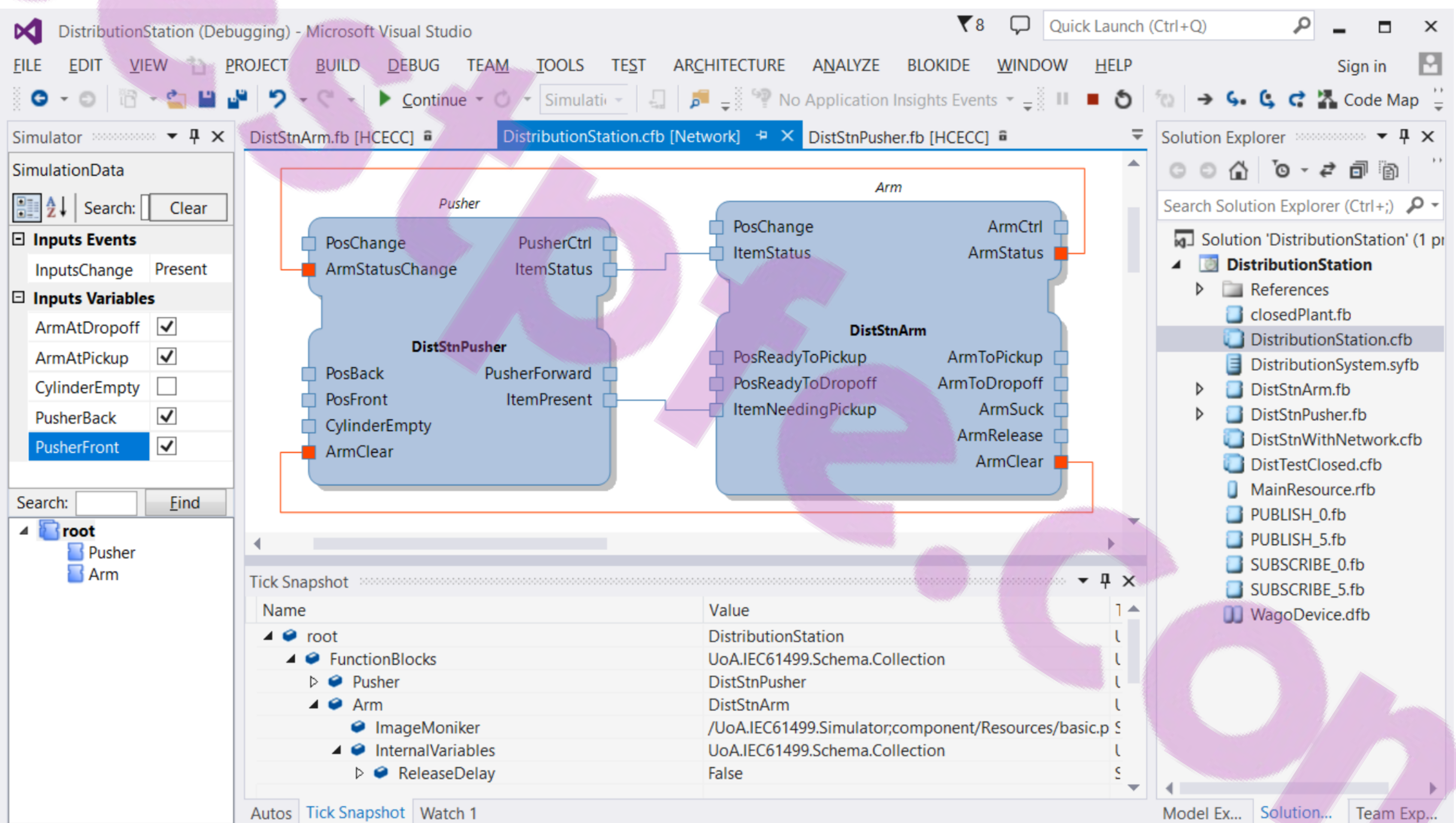


Figure A.12: Simulation of the encapsulated FBN of a given CFB

Integration with the FB compiler [108] allows BlokIDE to perform automatic code generation. The project properties dialog can be used to set the path to the compiler as shown in Figure A.13.

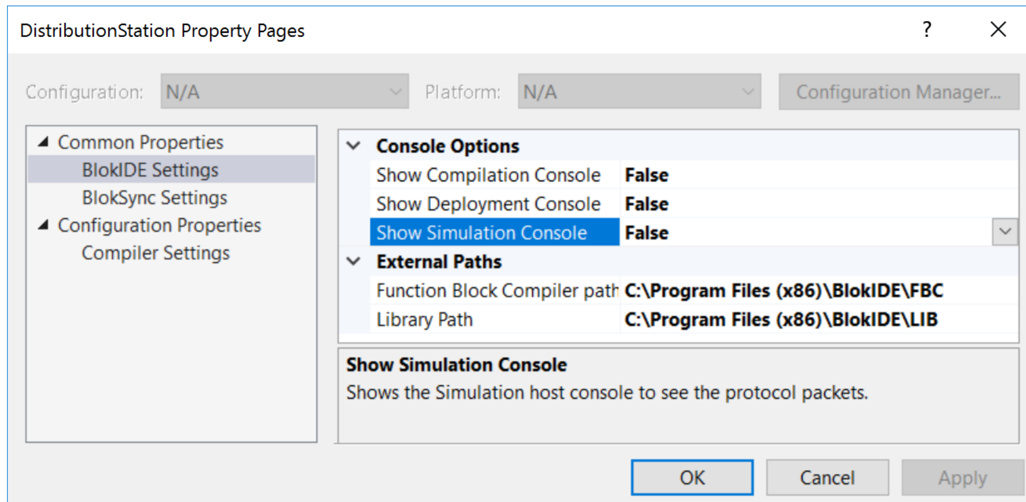


Figure A.13: Project properties dialog for setting compiler path

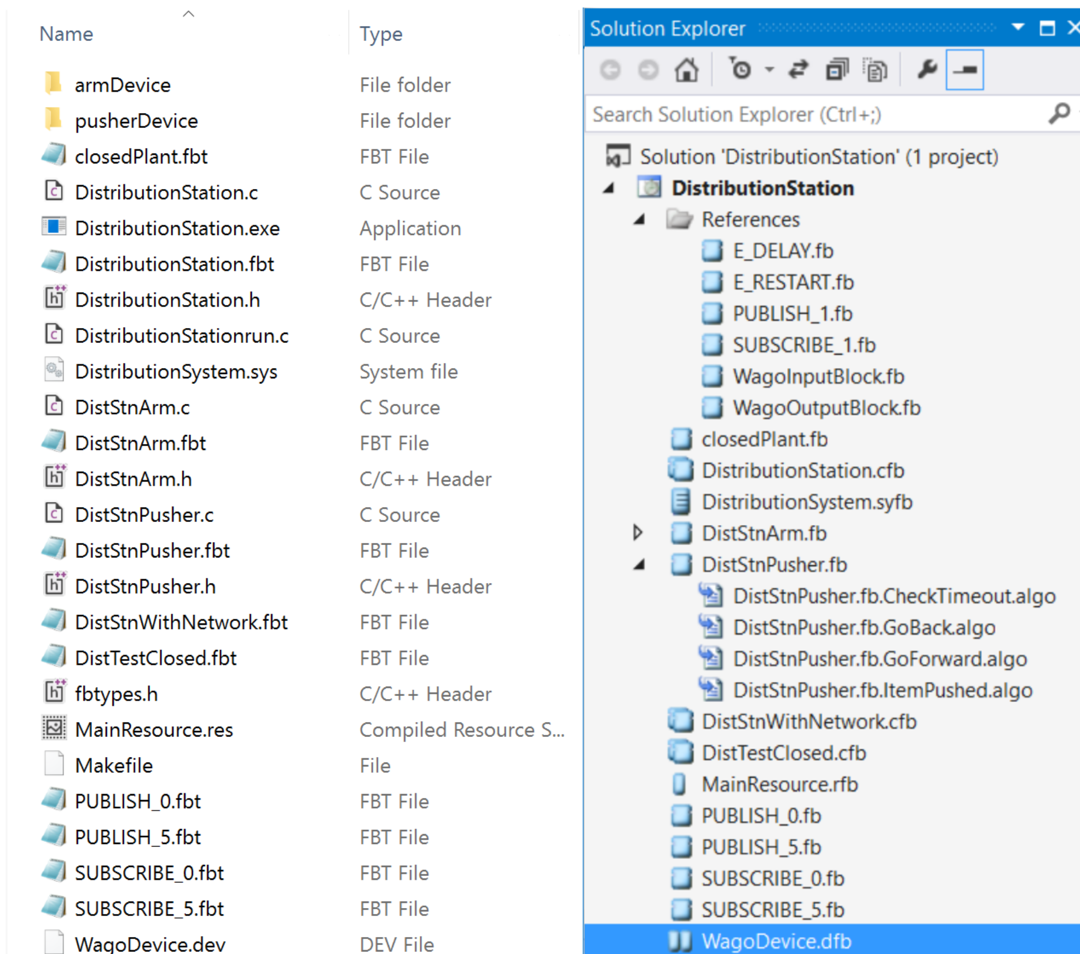


Figure A.14: IEC 61499 models on the right and the generated code on the left

Using visual studio extensibility, the build process is extended to invoke the FB compiler. This process exports the IEC 61499 FBT format files and execute the compiler executable. Resultantly, the generated code is emitted in the output folder, which contains ISO-C source files as well as the Makefile for compilation as shown in Figure A.14. The generated code can be compiled on the PC, however if device specific service interface function blocks (SIFB) are used, the compilation must be performed on the target device. This process is supported by a feature named *device deployment*. In this process, the generated code and the compilation scripts are deployed to the target device and compilation is initiated on-board. For this purpose, an SSH connection is made and files are transferred over SFTP, followed by issuing the make command. Figure A.15 shows the device deployment dialog.

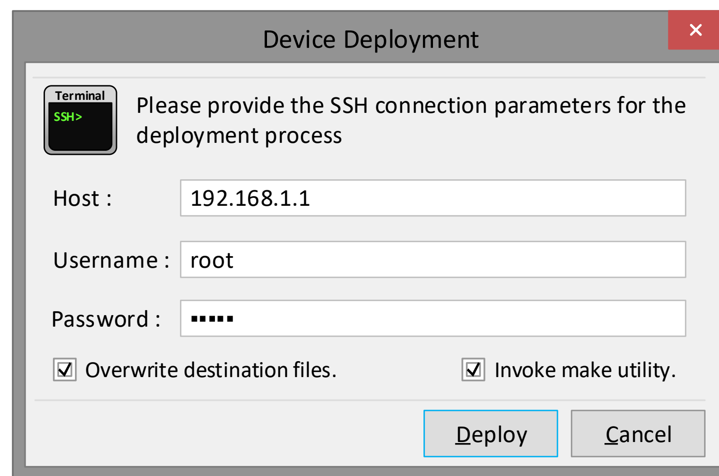


Figure A.15: Device Deployment over SSH

A.2 Prism Screenshots

BlokIDE allows exporting a given FBN to semantically equivalent Prism model. The exported Prism model can be verified using the Prism model checker. Figure A.16 shows the Model view of the model checker where a Prism model can be seen in the main window. The size of the model i.e., number of states and transitions is shown on bottom left hand side. The tab-view pane on the bottom left side can be used to switch to the Properties view. In this view, verification properties can be defined using the Property Editor dialog shown in Figure A.17. The context menu on the properties can be used to start the verification process as shown in Figure A.18. The result of this verification is display in the Property Details dialog as shown in Figure A.19.

BEST PFE.COM
List of research project topics and materials

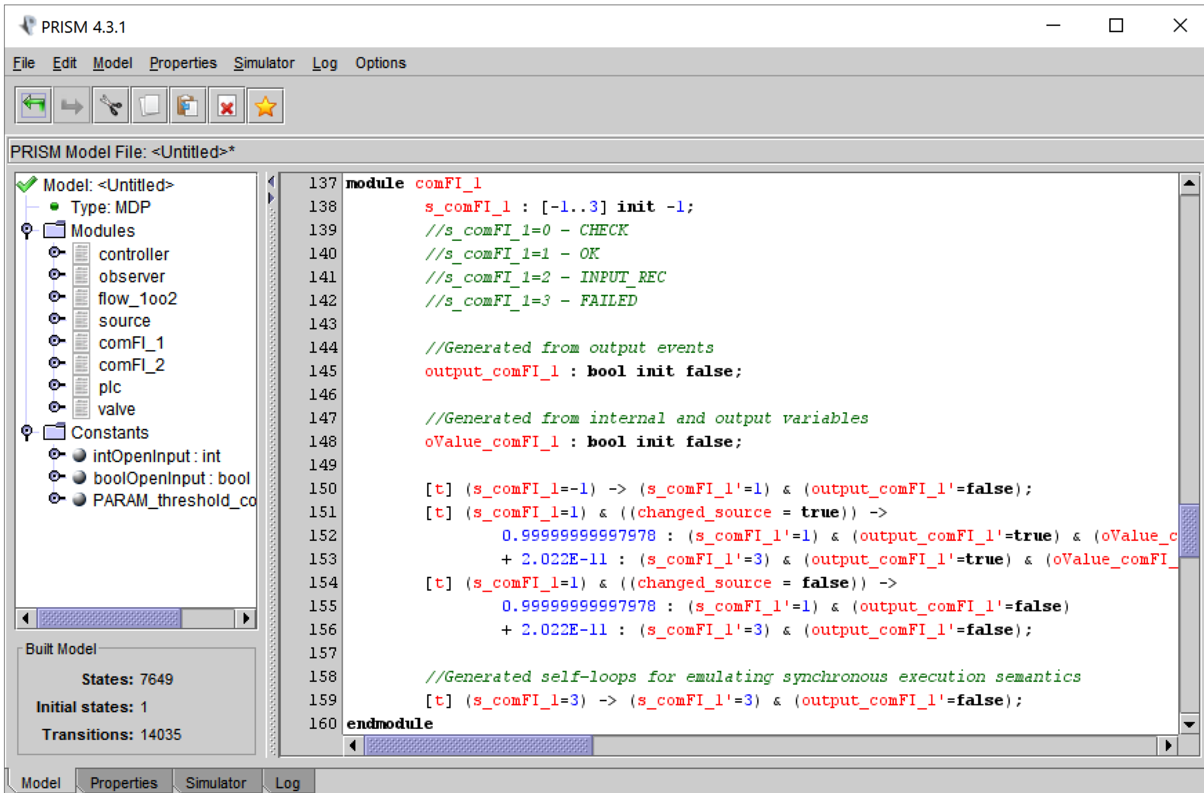


Figure A.16: Prism model checker with a generated Prism model

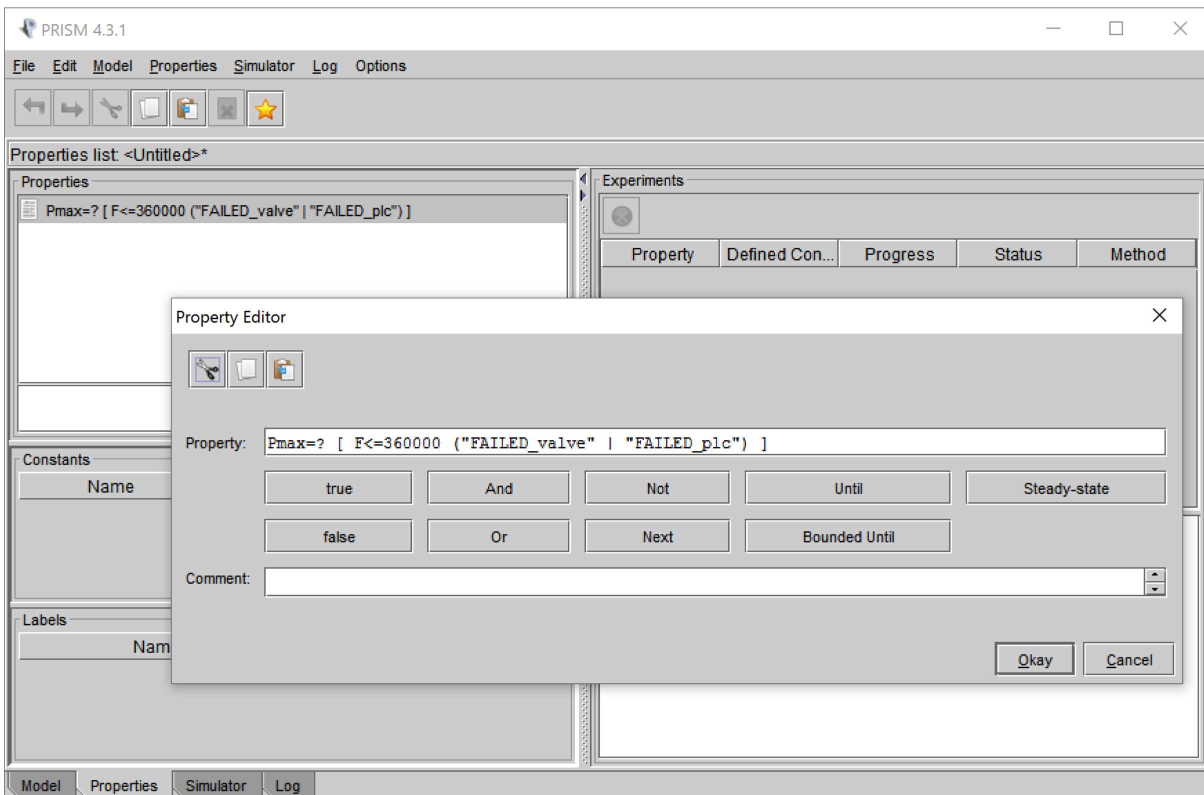


Figure A.17: Defining verification properties in Prism model checker

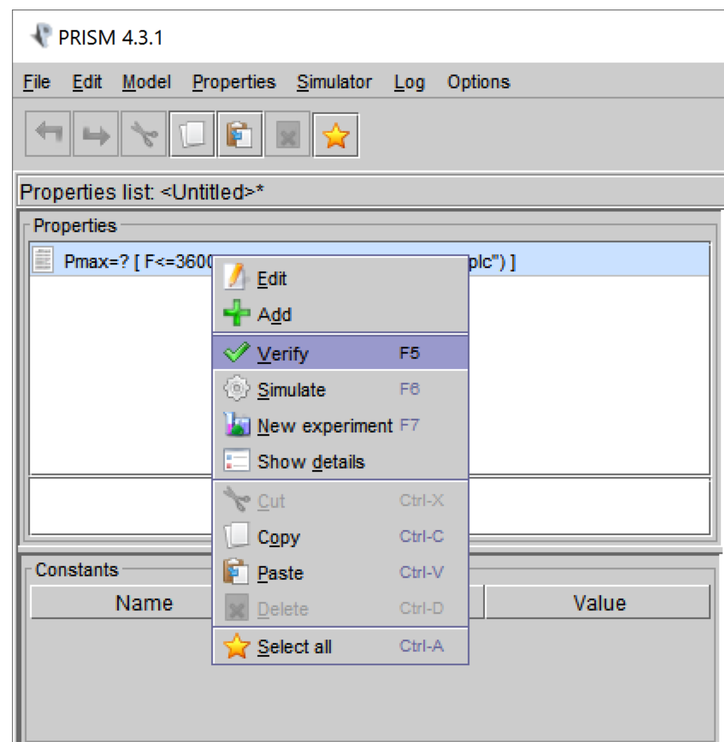


Figure A.18: Start the verification process in Prism model checker

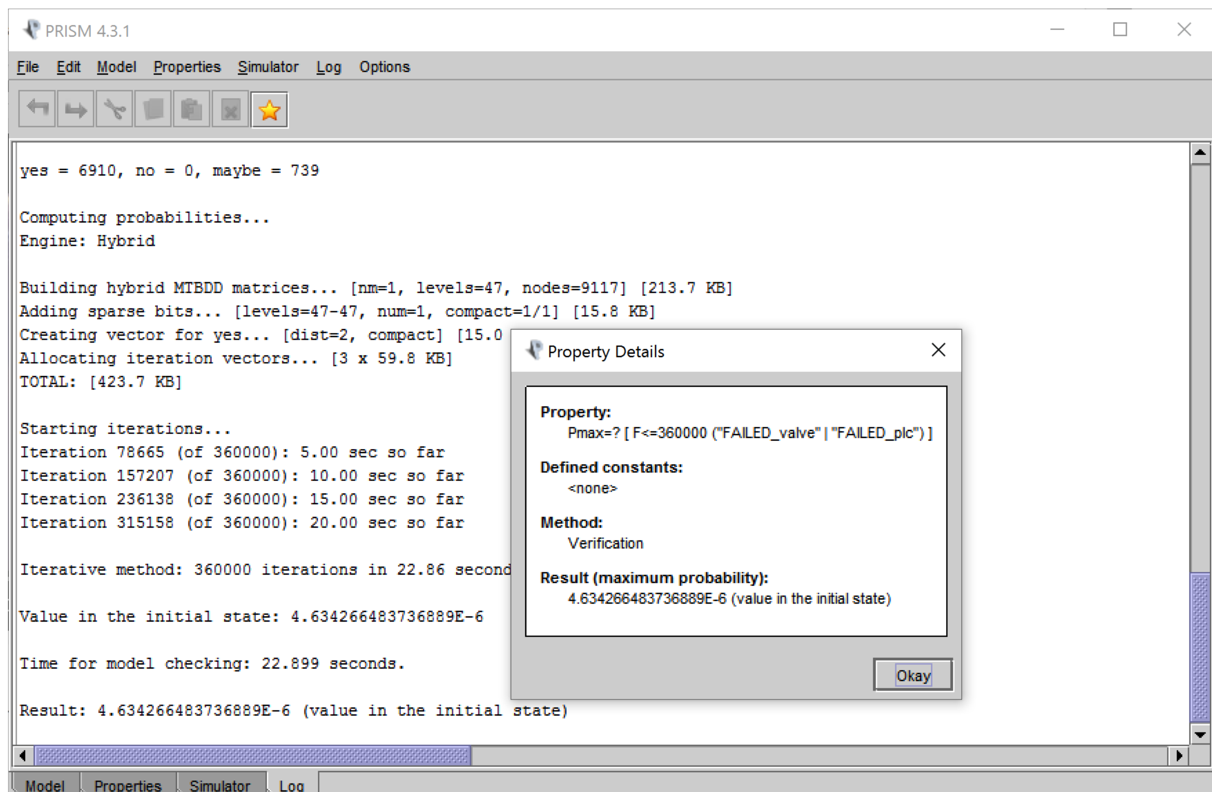


Figure A.19: Result of probabilistic verification in Prism model checker

References

[1] *IEC 61508 Part 5: Examples of methods for the determination of safety integrity levels, Functional safety of electrical / electronic / programmable electronic safety-related systems.* Geneva, Switzerland: International Electrotechnical Commission, 2010, no. BS EN 61508-5:2010.

[2] G. B. Health and S. Executive, *Out of Control: Why Control Systems Go Wrong and how to Prevent Failure*, 2nd ed., ser. Topic reports. HSE Books, 2003.

[3] *IEC 61508 Part 1: General requirements, Functional safety of electrical / electronic / programmable electronic safety-related systems.* Geneva, Switzerland: International Electrotechnical Commission, 2010, no. BS EN 61508-1:2010.

[4] *IEC 61508 Part 3: Software requirements, Functional safety of electrical / electronic / programmable electronic safety-related systems.* Geneva, Switzerland: International Electrotechnical Commission, 2010, no. BS EN 61508-3:2010.

[5] M. Kwiatkowska, G. Norman, and D. Parker, "Probabilistic Model Checking - Part 4 Markov Decision Processes," http://www.prismmodelchecker.org/lectures/biss07/04-mdps.pdf, University of Oxford, 2016, Accessed 14 September 2016.

[6] Z. E. Bhatti, "A model-driven approach for safety critical systems," Master's thesis, The University of Auckland, 2011.

[7] *IEC 61508 Part 4: Definitions and abbreviations, Functional safety of electrical / electronic / programmable electronic safety-related systems.* Geneva, Switzerland: International Electrotechnical Commission, 2010, no. BS EN 61508-4:2010.

[8] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages 12 years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, Jan 2003.

[9] P. Goupil, "AIRBUS state of the art and practices on FDI and FTC in flight control system," *Control Engineering Practice*, vol. 19, no. 6, pp. 524 – 539, 2011, SAFEPROCESS 2009Special Section: Fault Diagnosis Systems (7th IFAC Symposium on Fault Detection, Supervision and Safety of Technical Processes (SAFEPROCESS) in Barcelona, 30th June to 3rd July 2009). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0967066110002704

[10] V. Vyatkin, "IEC 61499 as enabler of distributed and intelligent automation: State-of-the-art review," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 768–781, Nov 2011.

[11] *IEC 61499-1:2012 Function blocks - Part 1: Architecture.* Geneva, Switzerland: International Electrotechnical Commission, 2012, no. IEC 61499.

[12] C. Pang, S. Patil, C.-W. Yang, V. Vyatkin, and A. Shalyto, "A portability study of IEC 61499: Semantics and tools," in *Industrial Informatics (INDIN), 2014 12th IEEE International Conference on*, July 2014, pp. 440–445.

[13] L. Ferrarini and C. Veber, "Implementation approaches for the execution model of IEC 61499 applications," in *Industrial Informatics, 2004. INDIN '04. 2004 2nd IEEE International Conference on*, June 2004, pp. 612 –617.

[14] V. Vyatkin, V. Dubinin, C. Veber, and L. Ferrarini, "Alternatives for execution semantics of iec61499," in *Industrial Informatics, 2007 5th IEEE International Conference on*, vol. 2, June 2007, pp. 1151–1156.

[15] L. Yoong, P. Roop, Z. Bhatti, and M. Kuo, *Model-Driven Design Using IEC 61499 : A Synchronous Approach for Embedded and Automation Systems.* Springer, 2015.

[16] J. Christensen, "Figure: International Language Standardization," in PLCopen Standard Presentation VI.0, 1998.

[17] S. David John and S. Kenneth G. L., *Safety Critical Systems Handbook : A Straightforward Guide to Functional Safety: IEC 61508 (2010 edition) and Related Standards : including: Process IEC 61511, Machinery IEC 62061 and ISO 13849*, 3rd ed. Elsevier/Butterworth-Heinemann, 2011, ch. 1, p. 8.

[18] *IEC 61513 Nuclear power plants - Instrumentation and control important to safety - General requirements for systems.* Geneva, Switzerland: International Electrotechnical Commission, 2011, no. IEC 61513.

[19] *IEC 62061 Safety of machinery - Functional safety of safety-related electrical, electronic and programmable electronic control systems.* Geneva, Switzerland: International Electrotechnical Commission, 2015, no. IEC 62061, iEC 62061:2005+AMD1:2012+AMD2:2015 CSV Consolidated version.

[20] *IEC Functional safety - Safety instrumented systems for the process industry sector.* Geneva, Switzerland: International Electrotechnical Commission, 2004, no. IEC 61511, iEC 61511:2004 SER Series.

[21] G. Berry, "Synchronous Design and Verification of Critical Embedded Systems Using SCADE and Esterel," in *Formal Methods for Industrial Critical Systems*, ser. Lecture Notes in Computer Science, S. Leue and P. Merino, Eds. Springer Berlin Heidelberg, 2008, vol. 4916, pp. 2–2.

[22] G. S. Mirko Conrad, "A verification and validation workflow for iec 61508 applications," The MathWorks, Inc., Natick, MA, USA, Tech. Rep., 2009. [Online]. Available: http://www.mathworks.com/tagteam/59056_2009-01-0271.New.pdf

[23] M. Conrad, "Verification and validation according to iso 26262: A workflow to facilitate the development of high-integrity software," The MathWorks, Inc., Natick, MA, USA, Tech. Rep., 2012. [Online]. Available: http://www.mathworks.com/tagteam/71300_1D-4.pdf

[24] "Efficient Development of Safe Applications Software with IEC 61508 Objectives Using SCADE Suite," Esterel Technologies SA, Elancourt, France, Tech. Rep., January 2012. [Online]. Available: http://www.esterel.com.cn/technology/handbooks/

[25] B. Selic, "The pragmatics of model-driven development," *IEEE Software*, vol. 20, no. 5, pp. 19–25, September -October 2003.

[26] V. Vyatkin, H.-M. Hanisch, C. Pang, and C.-H. Yang, "Closed-loop modeling in future automation system engineering and validation," *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, vol. 39, no. 1, pp. 17–28, January 2009.

[27] S. Patil, S. Bhadra, and V. Vyatkin, "Closed-loop formal verification framework with non-determinism, configurable by meta-modelling," in *Proceedings of 37th Annual Conference on IEEE Industrial Electronics Society (IECON).* IEEE, 2011, pp. 3770–3775.

[28] S. Patil, V. Vyatkin, and M. Sorouri, "Formal verification of intelligent mechatronic systems with decentralized control logic," in *Proceedings of IEEE 17th Conference*

on Emerging Technologies Factory Automation (ETFA'12), September 2012, pp. 1–7.

[29] S. Patil, S. Bhadra, and V. Vyatkin, "Closed-loop formal verification framework with non-determinism, configurable by meta-modelling," in *IECON 2011 - 37th Annual Conference on IEEE Industrial Electronics Society*, Nov 2011, pp. 3770–3775.

[30] M. Kuo, L. H. Yoong, S. Andalam, and P. Roop, "Determining the worst-case reaction time of IEC 61499 function blocks," in *Industrial Informatics (INDIN), 2010 8th IEEE International Conference on*, July 2010, pp. 1104 –1109.

[31] L. Lednicki, J. Carlson, and K. Sandstrom, "Device utilization analysis for IEC 61499 systems in early stages of development," in *Emerging Technologies Factory Automation (ETFA), 2013 IEEE 18th Conference on*, Sept 2013, pp. 1–8.

[32] R. D. Alexander and T. P. Kelly, "Escaping the non-quantitative trap," in *Proceedings of 27th International System Safety Conference (ISSC'09)*, 2009, pp. 69–95.

[33] L. H. Yoong and P. Roop, "Verifying IEC 61499 function blocks using Esterel," *IEEE Embedded Systems Letters*, vol. 2, no. 1, pp. 1 –4, March 2010.

[34] Z. Bhatti, R. Sinha, and P. Roop, "Observer based verification of IEC 61499 function blocks," in *Industrial Informatics (INDIN), 2011 9th IEEE International Conference on*, july 2011, pp. 609 –614.

[35] V. Vyatkin and H.-M. Hanisch, "Formal modeling and verification in the software engineering framework of iec 61499: a way to self-verifying systems," in *Emerging Technologies and Factory Automation, 2001. Proceedings. 2001 8th IEEE International Conference on*, vol. 2, Oct 2001, pp. 113–118 vol.2.

[36] S. Patil, V. Dubinin, and V. Vyatkin, "Formal modelling and verification of IEC 61499 function blocks with abstract state machines and smv - execution semantics," in *Dependable Software Engineering: Theories, Tools, and Applications*, ser. Lecture Notes in Computer Science, X. Li, Z. Liu, and W. Yi, Eds. Springer International Publishing, 2015, vol. 9409, pp. 300–315. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-25942-0_20

[37] E. Carpanzano, L. Ferrucci, D. Mandrioli, M. Mazzolini, A. Morzenti, and M. Rossi, "Automated formal verification for flexible manufacturing systems," *Journal of Intelligent Manufacturing*, vol. 25, no. 5, pp. 1181–1195, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10845-013-0760-z

[38] V. Dubinin, V. Vyatkin, and H.-M. Hanisch, "Modelling and verification of IEC 61499 applications using prolog," in *Emerging Technologies and Factory Automation, 2006. ETFA '06. IEEE Conference on*, September 2006, pp. 774 –781.

[39] C. Schnakenbourg, J.-M. Faure, and J.-J. Lesage, "Towards IEC 61499 function blocks diagrams verification," in *Systems, Man and Cybernetics, 2002 IEEE International Conference on*, vol. 3, October 2002, pp. 6 –11.

[40] H. Prahofer and A. Zoitl, "Verification of hierarchical iec 61499 component systems with behavioral event contracts," in *Industrial Informatics (INDIN), 2013 11th IEEE International Conference on*, July 2013, pp. 578–585.

[41] L. Allen, K. Goh, and D. Tilbury, "Closed-loop determinism for non-deterministic environments: Verification for IEC 61499 logic controllers," in *Automation Science and Engineering, 2009. CASE 2009. IEEE International Conference on*, Aug 2009, pp. 1–6.

[42] H. Lapp, C. Gerber, and H.-M. Hanisch, "Improving verification and reliability of distributed control systems design according to IEC 61499," in *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, Sept 2010, pp. 1–8.

[43] E. Zio, P. Baraldi, and F. Cadini, *Basics of Reliability and Risk Analysis: Worked Out Problems and Solutions (Series on Quality Reliability and Engineering Statistics)*. World Scientific Publishing Company, 2011.

[44] A. Joshi, S. Miller, M. Whalen, and M. Heimdahl, "A proposal for model-based safety analysis," in *Proceedings of the 24th Digital Avionics Systems Conference (DASC'05)*, vol. 2, Oct 2005, pp. 13 pp. Vol. 2–.

[45] A. Rauzy and C. Bleriot-Fabre, "Model-based safety assessment: Rational and trends," in *Proceedings of 10th France-Japan/ 8th Europe-Asia Congress on Mecatronics*, Nov 2014, pp. 1–10.

[46] O. Lisagor, T. Kelly, and R. Niu, "Model-based safety assessment: Review of the discipline and its challenges," in *Proceedings of 9th International Conference on Reliability, Maintainability and Safety (ICRMS'11)*, June 2011, pp. 625–632.

[47] P. Braun, J. Philipps, B. SchÃďtz, and S. Wagner, "Model-Based Safety-Cases for Software-Intensive Systems," *Electronic Notes in Theoretical Computer Science*, vol. 238, no. 4, pp. 71 – 77, 2009.

Bestpfe.com

[48] F. Mhenni, N. Nguyen, and J.-Y. Choley, “Automatic fault tree generation from sysml system models,” in *Proceedings of IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM’14)*, July 2014, pp. 715–720.

[49] E. BÃűde, T. Peikenkamp, J. Rakow, and S. Wischmeyer, “Model based importance analysis for minimal cut sets,” in *Automated Technology for Verification and Analysis*, ser. Lecture Notes in Computer Science, S. Cha, J.-Y. Choi, M. Kim, I. Lee, and M. Viswanathan, Eds. Springer Berlin Heidelberg, 2008, vol. 5311, pp. 303–317. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88387-6_27

[50] A. Gomes, A. Mota, A. Sampaio, F. Ferri, and J. Buzzi, “Systematic model-based safety assessment via probabilistic model checking,” in *Leveraging Applications of Formal Methods, Verification, and Validation*, ser. Lecture Notes in Computer Science, T. Margaria and B. Steffen, Eds. Springer Berlin Heidelberg, 2010, vol. 6415, pp. 625–639.

[51] M. Esteve, J. Katoen, V. Y. Nguyen, B. Postma, and Y. Yushtein, “Formal correctness, safety, dependability, and performance analysis of a satellite,” in *Proceedings of 34th International Conference on Software Engineering (ICSE)*, June 2012, pp. 1022–1031.

[52] P. Feiler, B. Lewis, S. Vestal, and E. Colbert, “An Overview of the SAE Architecture Analysis & Design Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering,” in *Architecture Description Languages*, ser. IFIP The International Federation for Information Processing, P. Dissaux, M. Filali-Amine, P. Michel, and F. Vernadat, Eds. Springer US, 2005, vol. 176, pp. 3–15.

[53] H. Mortada, T. Prosvirnova, and A. Rauzy, “Safety assessment of an electrical system with altarica 3.0,” in *Model-Based Safety and Assessment*, ser. Lecture Notes in Computer Science, F. Ortmeier and A. Rauzy, Eds. Springer International Publishing, 2014, vol. 8822, pp. 181–194. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-12214-4_14

[54] A. Arnold, G. Point, A. Griffault, and A. Rauzy, “The altarica formalism for describing concurrent systems,” *Fundam. Inf.*, vol. 40, no. 2,3, pp. 109–124, Aug. 1999. [Online]. Available: http://dl.acm.org/citation.cfm?id=2378994.2378996

[55] M. Güdemann, M. Lipaczewski, S. Struck, and F. Ortmeier, “Unifying Probabilistic and Traditional Formal Model Based Analysis,” in *8. Dagstuhl-Workshop MBEES 2012 - Model-Based Development of Embedded Systems*, Dagstuhl, Germany, February 2012.

[56] A. Gomes, A. Mota, A. Sampaio, F. Ferri, and E. Watanabe, “Constructive model-based analysis for safety assessment,” *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 6, pp. 673–702, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10009-012-0238-x

[57] A. Beer, T. Georgiev, F. Leitner-Fischer, and S. Leue, “Model-Based Quantitative Safety Analysis of Matlab Simulink / Stateflow Models,” in *Proceedings of Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme IX, Schloss Dagstuhl, Germany, April 24-26, 2013, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, 2013, pp. 60–69.

[58] Department of Computer Science, University of Oxford, “The PRISM Language - Semantics,” http://www.prismmodelchecker.org/doc/semantics.pdf, Tech. Rep., march 2015.

[59] M. Kwiatkowska, G. Norman, and D. Parker, “PRISM 4.0: Verification of Probabilistic Real-time Systems,” in *Proceeding of 23rd International Conference on Computer Aided Verification (CAV’11)*, ser. Lecture Notes on Computer Science, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585 –591.

[60] P. O’ Connor and A. Kleyner, *Practical Reliability Engineering*, 5th ed. John Wiley & Sons, Ltd, 2012, ch. 1, pp. 1–17.

[61] P. O’ Connor and A. Kleyner, *Practical Reliability Engineering*, 5th ed. John Wiley & Sons, Ltd, 2012, ch. 6, pp. 134–176.

[62] C. Carlson, *Effective FMEAs : Achieving Safe, Reliable, and Economical Products and Processes using Failure Mode and Effects Analysis*, 2nd ed. Hoboken: John Wiley & Sons, 2012.

[63] H. Hermanns, *Interactive Markov Chains and the Quest for Quantified Quality*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2002, vol. 2428.

[64] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 1999.

[65] M. Huth and M. Ryan, *Logic in Computer Science - Modelling and Reasoning about Systems*, 2nd ed. Cambridge University Press, 2004, ch. 3, pp. 175–221.

[66] B. Alpern and F. Schneider, “Recognizing safety and liveness,” *Distributed Computing*, vol. 2, no. 3, pp. 117–126, 1987. [Online]. Available: http://dx.doi.org/10.1007/BF01782772

BEST PFE.COM
List of research project topics and materials

[67] K. Goseva-Popstojanova, a.P. Mathur, and K. Trivedi, "Comparison of architecture-based software reliability models," in *Proceedings of 12th International Symposium on Software Reliability Engineering (ISSRE'01)*, 2001.

[68] M. Brito, J. May, J. Gallardo, and E. Fergus, "Use of graphical probabilistic models to build SIL claims based on software safety standards such as IEC61508-3," in *Proceedings of the 14th Safety-Critical Systems Symposium Developments in Risk-Based Approaches to Safety*, 2006, pp. 241–259.

[69] P. Feiler, B. Lewis, S. Vestal, and E. Colbert, "An Overview of the SAE Architecture Analysis & Design Language (AADL) Standard: A Basis for Model-Based Architecture-Driven Embedded Systems Engineering," in *Architecture Description Languages*, ser. IFIP The International Federation for Information Processing, P. Dissaux, M. Filali-Amine, P. Michel, and F. Vernadat, Eds. Springer US, 2005, vol. 176, pp. 3–15.

[70] "SCADE tools," http://www.esterel-technologies.com, Esterel Technologies, 2010, Accessed 6 November 2015.

[71] "Simulink: Simulation and Model-Based Design," http://www.mathworks.com.au/products/simulink/, The MathWorks Inc., Natick, Massachusetts, 2004, Accessed 6 November 2015.

[72] A. Arnold, G. Point, A. Griffault, and A. Rauzy, "The AltaRica Formalism for Describing Concurrent Systems," *Fundam. Inf.*, vol. 40, no. 2,3, pp. 109–124, Aug. 1999.

[73] *SAE Architecture Analysis and Design Language (AADL).* Warrendale, PA, USA: International Society of Automatize Engineers, June 2006, no. SAE-AS5506.

[74] *SAE Architecture Analysis and Design Language (AADL) - Annex E: Error Model Annex.* Warrendale, PA, USA: International Society of Automatize Engineers, June 2006, vol. 1, no. SAE-AS5506/1.

[75] A. Joshi, P. Binns, and S. Vestal, "Automatic generation of Fault Trees from AADL Models," in *Proceedings of the ICSE Workshop on Aerospace Software Engineering*, Minneapolis, USA, 2007.

[76] A.-E. Rugina, K. Kanoun, and M. Kaâniche, "The adapt tool: From aadl architectural models to stochastic petri nets through model transformation," in *Proceedings of the 7th European Dependable Computing Conference (EDC'08).* Washington, DC, USA: IEEE Computer Society, 2008, pp. 85–90.

[77] G. Chiola, M. Ajmone Marsan, G. Balbo, and G. Conte, "Generalized stochastic petri nets: a definition at the net level and its implications," *IEEE Transactions on Software Engineering*, vol. 19, no. 2, pp. 89–107, Feb 1993.

[78] M. Güdemann and F. Ortmeier, "A framework for qualitative and quantitative formal model-based safety analysis," in *High-Assurance Systems Engineering (HASE), 2010 IEEE 12th International Symposium on*, Nov 2010, pp. 132–141.

[79] M. Lipaczewski, S. Struck, and F. Ortmeier, "SAML Goes Eclipse: Combining Model-based Safety Analysis and High-level Editor Support," in *Proceedings of the Second International Workshop on Developing Tools As Plug-Ins (TOPI'12).* Zurich, Switzerland: IEEE Press, 2012, pp. 67–72.

[80] M. Bouissou, H. Bouhadana, M. Bannelier, and N. Villatte, "Knowledge modelling and reliability process - presentation of the figaro language and associated tools," in *Proceedings of International Conference on Computer Safety, Reliability and Security (SAFECOMP'91)*, Trondheim, Norway, 1991.

[81] Y. Papadopoulos and J. McDermid, "Hierarchically Performed Hazard Origin and Propagation Studies," in *Computer Safety, Reliability and Security*, ser. Lecture Notes in Computer Science, M. Felici and K. Kanoun, Eds. Springer Berlin Heidelberg, 1999, vol. 1698, pp. 139–152.

[82] M. Walker and Y. Papadopoulos, "Qualitative temporal analysis: Towards a full implementation of the Fault Tree Handbook," *Control Engineering Practice*, vol. 17, no. 10, pp. 1115 – 1125, 2009.

[83] D. Cancila, F. Terrier, F. Belmonte, H. Dubois, H. Espinoza, S. Gérard, and A. Cuccuru, "SOPHIA: a modeling language for model-based safety engineering," in *Proceedings of 2nd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES-MB'09).* Academic Press, 2009, pp. 11–26.

[84] L. Wang, S. Okon, and N. Server, "Modeling failure modes with SysML," NASA, Tech. Rep., 2013. [Online]. Available: https://books.google.co.nz/books?id=SOL2ngEACAAJ

[85] J. Xiang, K. Yanoo, Y. Maeno, and K. Tadano, "Automatic synthesis of static fault trees from system models," in *Proceedings of 5th International Conference on Secure Software Integration and Reliability Improvement (SSIRI'11)*, June 2011, pp. 127–136.

[86] J. françois Pétin, D. Evrot, G. Morel, and P. Lamy, "Combining SysML and formal methods for safety requirements verification," in *Proceedings of 22nd International Conference on Software & Systems Engineering and their Applications (ICSSEA'10)*, no. hal-00533311, 2010.

[87] A. Baouya, D. Bennouar, O. A. Mohamed, and S. Ouchani, "A quantitative verification framework of SysML activity diagrams under time constraints," *Expert Systems with Applications*, vol. 42, no. 21, pp. 7493 – 7510, 2015.

[88] Y. Jarraya and M. Debbabi, "Quantitative and qualitative analysis of SysML activity diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 4, pp. 399–419, 2014.

[89] S. Li and S. Duo, "A Practicable MBSA Modeling Process Using AltaRica," in *Model-Based Safety and Assessment*, ser. Lecture Notes in Computer Science, F. Ortmeier and A. Rauzy, Eds. Springer International Publishing, 2014, vol. 8822, pp. 1–13. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-12214-4_1

[90] F. Cassez, C. Pagetti, and O. Roux, "A Timed Extension for AltaRica," *Fundam. Inf.*, vol. 62, no. 3-4, pp. 291–332, Mar. 2004.

[91] A. Rauzy, "Mode automata and their compilation into fault trees," *Reliability Engineering & System Safety*, vol. 78, no. 1, pp. 1 – 12, 2002.

[92] A. Griffault and A. Vincent, "The MecÂă5 Model-Checker," in *Computer Aided Verification (CAV)*, ser. Lecture Notes in Computer Science, R. Alur and D. Peled, Eds. Springer Berlin Heidelberg, 2004, vol. 3114, pp. 488–491.

[93] MathWorks Inc. (2013) Simulink: UK Aerospace and Defense - High-Integrity Systems. http://www.mathworks.com.au/aerospace-defense/standards/do-178b.html. Natik.

[94] *IEC 61508 Part 2: Requirements for electrical / electronic / programmable electronic safety-related systems, Functional safety of electrical / electronic / programmable electronic safety-related systems.* Geneva, Switzerland: International Electrotechnical Commission, 2010, no. BS EN 61508-2:2010.

[95] "Distributing station: Getting started with MPS," http://www.festo-didactic.com/my-en/learning-systems/mps-the-modular-production-system/stations/distributing-station-getting-started-with-mps.htm, Festo Didactic, 2010, Accessed 6 November 2015.

[96] V. Vyatkin and J. Chouinard, "On Comparisons of the ISaGRAF implementation of IEC 61499 with FBDK and other implementations," in *Proceedings of 6th IEEE International Conference on Industrial Informatics (INDIN'08)*, Daejeon, July 2008, pp. 289–294.

[97] "ISaGRAF," http://www.isagraf.com, ICS Triplex ISaGRAF Inc., 2008, Accessed 6 November 2015.

[98] "Function Block Development Kit," http://www.holobloc.com, Holobloc Inc., 2008, Accessed 6 November 2015.

[99] "4DIAC-RTE (FORTE): IEC 61499 Compliant Runtime Environment," http://www.fordiac.org, PROFACTOR Produktionsforschungs GmbH, 2010, Accessed 6 November 2015.

[100] L. H. Yoong, P. S. Roop, and Z. Salcic, "Implementing Constrained Cyber-Physical Systems with IEC," *ACM Transactions on Embedded Computing*, vol. 11, no. 4, pp. 78:1–78:22, December 2012.

[101] International Organization for Standardization/International Electrotechnical Commission, *Information technology - Open Systems Interconnection - Basic Reference Model - Conventions for the definition of OSI services*, 1994.

[102] G. Čengić, "Function Block Execution Runtime (Fuber)," http://fuber.sourceforge.net, 2009, Accessed 6 November 2015.

[103] L. Ferrarini and C. Veber, "Implementation approaches for the execution model of IEC 61499 applications," in *Proceedings of 2nd IEEE International Conference on Industrial Informatics (INDIN'04)*, Berlin, June 2004, pp. 612–617.

[104] L. H. Yoong, "Modelling and Synthesis of Safety-critical Software with IEC 61499," Ph.D. dissertation, University of Auckland, 2011.

[105] L. Yoong, P. Roop, Z. Bhatti, and M. Kuo, *Model-Driven Design Using IEC 61499 : A Synchronous Approach for Embedded and Automation Systems.* Springer, 2015, ch. 4, pp. 65–89.

[106] R. Sinha, P. Roop, G. Shaw, Z. Salcic, and M. Kuo, "Hierarchical and concurrent ECCs for IEC 61499 function blocks," *Industrial Informatics, IEEE Transactions on*, vol. PP, no. 99, pp. 1–1, 2015.

[107] C. Baier and J.-P. Katoen, *Principles of Model Checking.* Cambridge, Massachusetts, USA.: The MIT Press, 2008, ch. 3,7, pp. 104–120,496–500.

[108] L. H. Yoong, P. Roop, V. Vyatkin, and Z. Salcic, “A synchronous approach for IEC 61499 function block implementation,” *IEEE Transactions on Computers*, vol. 58, no. 12, pp. 1599 –1614, December 2009.

[109] F. Boussinot and R. De Simone, “The SL Synchronous Language,” INRIA, Research Report RR-2510, Mar. 1995. [Online]. Available: https://hal.inria.fr/inria-00074168

[110] G. D. Shaw, “Reliable model-driven engineering using IEC 61499,” Ph.D. dissertation, Department of Electrical and Computer Engineering, The University of Auckland, 2013.

[111] C. Sünder, V. Vyatkin, and A. Zoitl, “Formal verification of downtimeless system evolution in embedded automation controllers,” *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 1, pp. 17:1–17:17, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2406336.2406353

[112] C. Courcoubetis and M. Yannakakis, “Verifying temporal properties of finite-state probabilistic programs,” in *Proceedings of 29th Annual Symposium on Foundations of Computer Science (FOCS’88)*, 1988, pp. 338–345.

[113] M. Kwiatkowska, G. Norman, and D. Parker, “PRISM Manual - Property Specification,” http://www.prismmodelchecker.org/manual/PropertySpecification/Introduction, University of Oxford, 2016, Accessed 22 January 2017.

[114] A. Benveniste and G. Berry, “The synchronous approach to reactive and real-time systems,” *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1270 –1282, Sep. 1991.

[115] “The Esterel v7 Reference Manual: Version v7_30 for Esterel Studio,” Esterel Technologies S.A.S, Villeneuve-Loubet, December 2005.

[116] M. Güdemann and F. Ortmeier, “Probabilistic model-based safety analysis,” in *Proceedings of the 8th Workshop on Quantitative Aspects of Programming Languages (QAPL’10)*, EPTCS, 2010.

[117] E. Clarke, “The birth of model checking,” in *25 Years of Model Checking*, ser. Lecture Notes in Computer Science, O. Grumberg and H. Veith, Eds. Springer Berlin / Heidelberg, 2008, vol. 5000, pp. 1–26.

[118] M. Kwiatkowska, G. Norman, and D. Parker, “Stochastic Games for Verification of Probabilistic Timed Automata,” in *Proceedings of 7th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS’09)*, ser. LNCS, J. Ouaknine and F. Vaandrager, Eds., vol. 5813. Springer, 2009, pp. 212–227.

[119] M. Güdemann, F. Ortmeier, and W. Reif, "Using Deductive Cause-Consequence Analysis (DCCA) with SCADE," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, F. Saglietti and N. Oster, Eds. Springer Berlin Heidelberg, 2007, vol. 4680, pp. 465–478.

[120] "Visual Studio - Microsoft Developer Tools," https://www.visualstudio.com/, Microsoft, 2015.

[121] M. M. Y. Kuo, L. H. Yoong, S. Andalam, and P. Roop, "Determining the worst-case reaction time of IEC 61499 function blocks," in *8th IEEE International Conference on Industrial Informatics (INDIN'10)*, July 2010, pp. 1104 –1109.

[122] L. H. Yoong, P. Roop, V. Vyatkin, and Z. Salcic, "Synchronous execution of IEC 61499 function blocks using Esterel," in *Proceedings of 5th IEEE International Conference on Industrial Informatics (INDIN'07)*, vol. 2, June 2007, pp. 1189 –1194.

[123] *IEC 61508 Part 7: Overview of techniques and measures, Functional safety of electrical / electronic / programmable electronic safety-related systems.* Geneva, Switzerland: International Electrotechnical Commission, 2010, no. BS EN 61508-7:2010.