UNIVERSITY OF SOUTHAMPTON

FACULTY OF PHYSICAL AND APPLIED SCIENCES

Electronics and Computer Science

Integrating Formal Verification and Simulation of Hybrid Systems

by

Vitaly Savicks

Thesis for the degree of Doctor of Philosophy

May 2016

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF PHYSICAL AND APPLIED SCIENCES
Electronics and Computer Science

Doctor of Philosophy

INTEGRATING FORMAL VERIFICATION AND SIMULATION OF HYBRID SYSTEMS

by Vitaly Savicks

An increasing number of today's systems can be characterised as cyber-physical, or hybrid systems that combine the concurrent continuous environment and discrete computational logic. In order to develop such systems as safe and reliable one needs to be able to model and verify them from the early stages of the development process. Current modelling technologies allow us to specify the abstractions of these systems in terms of the procedural or declarative modelling languages and visual notations, and to simulate their behaviour over a period of time for analysis. Other means of modelling are formal methods, which define systems in terms of logics and enable rigorous analysis of system properties. While the first class of technologies provides a natural notation for describing physical processes, but lacks the formal proof, the second class relies on mathematical abstractions to rationalise and automate the complex task of formal verification. The benefits of both technologies can be significantly enhanced by a collaborative methodology. Due to the complexity of the considered systems and the formal proof process it is critical that such a methodology is based on a top-down development process that fully supports abstraction and refinement. We develop this idea into a tool extension for the state of the art Rodin platform for system-level formal modelling and analysis in the Event-B language. The developed tool enables integration of the physical simulation with refinement-based formal verification in Event-B, thus enhancing the capabilities of Rodin with the simulation-based validation that supports refinement. The tool utilises the Functional Mock-up Interface (FMI) standard for industrial-grade model exchange and co-simulation and is based on a co-simulation principle between the discrete models in Event-B and continuous physical models of FMI. It provides a graphical environment for model import, composition and co-simulation, and implements a generic simulation algorithm for discrete-continuous co-simulation.

Contents

List of Figures

List of Tables

Declaration of Authorship

I, Vitaly Savicks , declare that the thesis entitled *Integrating Formal Verification and Simulation of Hybrid Systems* and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly while in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all main sources of help;
- where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself; savicks2014rodin, savicks2014co, savicks2014cosim and savicksintegrating
- parts of this work have been published as: [129], [127], [126] and [128]

Signed:...

Date:...

Acknowledgements

I would like to express the immense gratitude to my supervisors Prof. Michael Butler and Dr. John Colley who supported me with guidance, help and wisdom of looking at complex problems in a simple way. I would also like to thank all the colleagues of the ESS group for their friendly and cheerful, but also motivating attitude and work environment they have created within the group.

This work would not be possible without the financial support of the ADVANCE project (http://www.advance-ict.eu) and the great effort of its brilliant team. I would like to give a special thanks to researchers behind the ProB tool: Jens Bendisposto and Joy Clark, and our partners from Critical Software: Karim Kanso and Brett Bicknell.

Finally, I would like to express sincere and deep gratitude to my parents for their constant support and encouragement.

Nomenclature

AOA	Angle of Attack
API	Application Programming Interface
C2	Command and Control
CPS	Cyber-Physical System
CT	Continuous-Time
DE	Discrete-Event
DLL	Dynamic-Link Library
DSL	Domain-Specific Language
FCPC	Flight Control Primary Computer
FCS	Flight Control System
FMI	Functional Mock-up Interface
FMU	Functional Mock-up Unit
FOM	Federation Object Model
FSM	Finite State Machine
GUI	Graphical User Interface
HLA	High Level Architecture
IPS	Ideal Physical Model
MoC	Model of Computation
OLTC	On-Load Tap Changer
OMC	Open Modelica Compiler
PO	Proof Obligation
RTI	Runtime Infrastructure
SOM	Simulation Object Model
UML	Unified Modeling Language
VDM-SL	Vienna Development Method Specification Language
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Cyber-Physical Systems

As defined by Lee [91], Hybrid, or Cyber-Physical Systems (CPS) are the integration of the networked and concurrent computing with physical processes. This technology is becoming an evolving trend both in the academic field and industry. The U.S. President's Council of Advisors has placed CPS on top of the national priority list in R&D [78], emphasising on particular importance of CPS. The benefits of this technology are apparent as intelligent embedded computing adds plenty of new capabilities to existing physical systems. Such a merge of computation and communication with physical processes would make systems safer, more efficient, cheaper to produce and operate, and, very importantly, capable of working in collaboration to form larger systems, such as a national power grid or an automated transportation control. The implications are enormous and considered by many as the next computing revolution [120]. In order to achieve this merge a lot of challenges need to be solved, some of which lay down the foundation for this research.

1.2 Design Challenges

The most involving challenges result from the fundamental properties of CPS that are yet barely present in embedded systems on the market. The majority of systems are designed using the same computing abstractions as 25 years ago, when resources were limited and the problem of embedded software was reduced to the problem of optimisation [88]. 'Closed box' architecture and reliance on bench testing for verification of concurrency and timing properties remains a standard practice in industry. Functional behaviour of CPS, however, is emerging through the interaction of computing with the physical world and cannot be examined separately. The plant, computer, network and environment are tightly integrated and dependent on each other.

The key features of CPS are high responsiveness to a real-world environment, time-precision and predictability, networking, concurrency, safety and reliability. The most challenging, but also essential of these properties, as argued by Lee [92], is the notion of time, which is missing on most levels of the current computing abstractions, starting from the microprocessor architectures with sophisticated memory models and speculative execution for performance gain, up to unreliable stochastic networking technologies. The fundamental property of the correctness of execution is defined by a terminating sequence of state transformations and therefore is not related to time, which makes it impossible to guarantee predictability. Likewise, the concurrency abstraction based on threads is extremely nondeterministic and unpredictable [89], which makes threads a constant source of difficulty and misconception among software engineers.

At the same time the increasing scale and omnipresence of today's sociotechnical systems puts reliability and safety engineering at the top of priorities. The historically established engineering practices, however, are not capturing the types of modern systems and associated evolving context, which includes a rapid technological innovation, reduced time to market (and hence reduced available timeframe for verification/validation), changing human-computer interaction and increasing coupling and complexity – not only of individual components, but also of their interaction, system evolution over time, structural and functional inconsistency, and indirect causality [95]. Understanding system safety at the early stages of requirement analysis and understanding the requirements themselves becomes a greater challenge. Software engineering is transforming into an integral part of system engineering, in which thorough theoretical analysis of all requirements and constraints of both the system and environment is essential for safety and reliability assurance. This necessitates new methods and tools for the top-down rigorous analysis and the integration of these tools into a single process.

All the problems mentioned are related and are caused either by currently employed engineering practices that are falling behind the technology or by incorrect abstractions that do not reflect specific properties of CPS, especially, the notion of time. No system can be made predictable unless the time is introduced to as many abstraction layers as possible [92]. For software engineers it means programming languages with temporal semantics, synchronisation capabilities and understandable predictable concurrency models. The gap between two system theories: physical and computational, must be reduced as the current practice, based on the separation of concerns, is not effective when the environment and computation are highly integrated. Therefore we need a new collaborative engineering approach, with top-down methods centred on the model-based design. This leads to a high demand for formal modelling techniques that are capable of verifying timing properties and properties of the environment. Along with developing new technologies it is important to develop sound methods for transforming the existing real-time embedded systems to the ones that would conform with the standards of modern CPS [27].

1.3 Existing Technologies

The solutions for the challenges we face in the development of CPS will span multiple computer science disciplines. From the software engineering perspective several approaches seem promising:

- *Programming languages with timing properties.* A number of attempts were made in the past to introduce time in programming languages and to increase predictability. For instance, synchronous programming languages like Esterel and Lustre, developed in the 80s for reactive systems, are based on a synchronous abstraction similar to digital circuits, where the computation is assumed to be instantaneous within a single clock [61]. The concept of logical time and synchronous concurrency eliminate non-determinism and result in more predictable code. In Ada task delays are possible. The latest version's (Ada 2005) Real-Time Systems Annex enhances this with timing events, execution time monitoring and execution time events [41]. For the Java language, Sun and IBM developed a Real-Time Specification (RTSJ) with a high resolution time, an improved memory management and a concurrency model [26]. Simulation-oriented modelling languages with temporal semantics are particularly promising. The equation-based multi-domain Modelica language for complex system design and simulation has a global time variable and time-based events [11]. Simulink, from MathWorks, provides a graphical environment with a block diagram notation for timed system modelling and simulation, with add-ons for real-time and embedded code generation [102]. The visual programming environment LabVIEW [52], from National Instruments, can be extended with a Real-Time module that allows time constraints to be specified with Timed Loop and Timed Sequence structures, and graphically represent timing relationships. All these solutions, especially model-based languages, are very helpful in development of CPS, but currently require better integration into existing engineering processes.

- *Formal methods.* Formal methods use mathematical formalisms for system specification, design and verification. Many computer science theories have contributed to this field. The Automata theory was extended to the Timed Automata, which provides a way to annotate state-transition graphs with timing constraints using real-valued clocks [8]. Statecharts [63] were extended for real-time and hybrid systems, resulting in the Timed and Hybrid Statecharts and a corresponding formal language with statements for delays, preemption and timeouts [76]. Such formalisms as hybrid automata provide a natural notation for mixed discrete-continuous systems [65]. Some theories have been successfully applied in modelling and simulation tools for real-time systems. For instance, the tool environment UPPAAL models systems as networks of timed automata, extended with data types [19]. In the AnyLogic simulation tool the formalism of hybrid state machines is used to model complex interdependencies between discrete and continuous time behaviour [31]. Moreover, a number of temporal logics [99] have been developed to allow specification of system behaviour in terms of logical formulas, including temporal constraints, events and the relationships between the two [17]. Formal specification and modelling languages with automated verification capabilities are very useful when system reliability and safety are the primary concerns. Although these languages are usually state-based and do not handle temporal dynamics, attempts are made to close this gap. The Z notation was augmented with Real-Time Logic [54]. The VDM-RT extended VDM++ with the time and time invariants [143]. Similar efforts have been observed for the Event-B formal method, where time constraints [46] and continuous behaviour [145] were modelled. Finally, the UML has been extended with a MARTE profile for real-time and embedded system modelling and analysis, adding a range of new language units, including the Time unit [111].

– *Simulation tools.* Simulation tools enable model-based design by replacing prototyping, manufacturing and testing with computer-aided modelling and simulation-based analysis, thus reducing risk of the design errors and lowering cost and time of the design process. The capability of simulating the environment and time is particularly useful for hybrid systems. These tools can be categorised into three classes: block-based, equation-based and based on hybrid state machines [32]. The Simulink environment for MATLAB is a popular example of the block-based approach. In addition to continuous and discrete blocks of the standard library it allows custom blocks to be modelled as state machines via the Stateflow extension [103], making it a suitable choice for designing control systems. In the equation-based approach the continuous dynamics are described by algebraic/differential equations while the discrete part is described by conditional and timed events. Such notation is more natural for physical modelling and is leveraged by tools like 20-sim [35], from Controllab Products, and a number of simulators that utilise the Modelica modelling language, e.g. Wolfram SystemModeler [147], Modelon Optimica [7]. To conveniently model hybrid and reactive systems the StateGraph library extends Modelica with components for deterministic hierarchical state diagrams [114]. Other tools support both causal (data flow) and acausal (equation) modelling. MapleSim [67], from Maplesoft, supports traditional signal flow blocks together with Modelica components. SimulationX [149], from ITI GmbH, in addition to a rich component library provides open comprehensive CAx-interfaces (computer-aided technologies) for external engineering applications and a co-simulation interface to other simulation software. Third class of the tools is similar by means to the StateGraph library, where equations are associated with states of a hybrid state machine. This provides a better integration of the continuous behaviour with the discrete logic. AnyLogic [30], from XJ Technologies, is a good example of this approach.

The advances in programming languages, formal methods, modelling and simulation tools outlined above are undoubtedly important for the development of CPS. Furthermore, mainstream adoption of these tools, integration and incorporation into a unified engineering process promises to build a strong basis for the incremental evolution of technology towards amalgamation of computing and environment. While there exists an argument on the fundamental flaws in the computing abstractions, such as threads and the lack of time, the development and integration of existing tools can be a good transfer step before the new technologies with in-build timing appear.

1.4 Limitations

Major limitations of existing tools relate to the outdated core abstractions of computing. In particular, the lack of timing and inadequate models of concurrency force industrial developers to stick to old and proven engineering practices. New programming languages with temporal semantics are still at the evolution stage and are seldom applied on real-scale problems. Most of the embedded systems are developed in “good old” C [75].

With domain-specific languages (DSL) the situation is less gloomy. The advances in the object-oriented modelling techniques, modelling frameworks, tools and standards, such as UML, as well as processes supporting them are widely used by industry. Mainly popular among the general computing in the past they are getting more attention in embedded real-time systems and are understood by experts as a simple and efficient means of system specification, abstraction and early analysis [86]. Disadvantages of these languages are the lack of constructs to express time-related constraints and their semi-formal semantics, not sufficient for the rigorous analysis of complex safety-critical systems [87]. More recent attempts from the Object Management Group consortium [98] address the lack of time semantics, but the standards are still evolving. Hence, the use of DSLs is not fully integrated into development processes and is often limited to abstract specification and design, leaving the task of analysis of system properties to simulation and extensive testing.

Formal methods solve the problem of the lack of formalism in DSLs by providing precise and sound mathematical specification languages and rigorous verification techniques, such as theorem proving and model checking. Unfortunately, the expressiveness of formal methods can be limited compared to DSLs. For example, modelling of timed physical systems is not always possible without considerable abstraction due to constrained data types and verification capabilities of tools. Broadly speaking, formal methods also have a number of historically established and attached myths, notably identified and challenged by Hall [62]. However, the continued and emerging use of formal methods in recent years, the development of tools that mask the formality behind the automation and abstraction, and evolving language support effectively debunk these myths and “domesticate” formal methods for a wide range of system engineering applications [71]. Myths such as the increased complexity, high cost of development and hindered scalability are addressed by the advancements in tooling, introduction of higher abstractions (DSLs for formal methods) and effective modular and incremental modelling techniques, but the lack of timing properties, in particular for modelling continuous dynamics processes of environment, is going to impede the adoption of formal methods in the development of CPS. Therefore, industry gives favour to pragmatic solutions over rigorous approaches.

Tools for simulation, analysis and implementation were first developed using common sense reasoning, and only afterwards – formalisation [47]. Simulation-centric tools are the most popular among designers, although their semantics are too general for formal analysis. Such simulation tools as Simulink/Stateflow provide an excellent modelling and simulation capability for design and functional verification of embedded systems, but there is a need for rigorous domain-specific analysis and methods for refinement of high-level models towards implementation, ideally with fully-fledged unified process that leverages the advantages of DSLs, formal verification and simulation [14].

1.5 Addressing the Issues

Several approaches that address existing limitations have been mentioned already in Section 1.3. DSLs and formal methods had a number of attempts to introduce continuous semantics and timing properties. In the Event-B, for instance, the discrete time relations: delay, expiry and deadline, have been added via an extension to the Rodin platform [125]. Other work has focused on modelling continuous behaviour using the existing constructs [145], proposing new extensions of the mathematical language [6] or adding theories of real numbers and continuous functions over real intervals using current tools [43]. Another means of facilitating the application of formal languages in the development of hybrid systems is to combine them with other technologies, such as simulation. This was studied, for example, by Su, Abrial and Zhu [140], who have developed patterns and collaborative techniques based on model transformation between Event-B and Simulink. A collaborative approach based on co-simulation was proposed by Tudoret et al. [142], in which the discrete behaviour was modelled using the formalism of clocked data flow in Signal and continuous dynamics done in Simulink. A good example of a co-simulation framework is the DESTECS project, which enables discrete-event models specified in VDM to be co-simulated with continuous-time models of 20-sim [55]. Other co-simulation methods are supported by the tool-independent industrial architectures and interface standards, such as the High Level Architecture [49] and Functional Mock-up Interface [24]. Interactive simulation is another useful technique that links models from various simulation tools (control systems designed in Matlab/Simulink, physical models from Modelica and hybrid-DAE models from Dymola) and allows them to be simulated interactively [100]. The lack of rigorous analysis techniques for simulated models was addressed, for instance, in a methodology called "invisible formal methods" for Simulink [141].

We believe that a collaborative approach between multiple state of the art technologies including formal methods is a practical solution. The design of CPS demands that all effects of the system are considered, including the interaction between the environment, plant and digital controller. This requires methods capable of dealing with heterogeneous components that exhibit different behaviour. The continuous-time environment maps particularly well to modelling constructs of equation-based and block-based simulation frameworks, whilst controllers can be naturally represented as discrete-event systems. High complexity of CPS components is anticipated, therefore powerful abstraction methods, modularity and iterative (refinement-oriented) development are essential. Finally, the technology must be able to mix appropriate mathematical representations and provide the means for overall analysis of system behaviour on different abstraction levels. The major challenge here is in the correct interaction (and mixing) of different domains, which requires formal techniques [101].

The objective of this research is to tackle the problem of hybrid system and CPS development by applying the refinement-based formal methods and physical system simulation technologies in a single framework. The idea is based on a premise that layered formal modelling and analysis of control systems helps to break down system complexity and better understand requirements, hence it reduces the design errors at the early stages of development and provides high level of assurance of design correctness with respect to functional and safety requirements. At the same time state of the art physical simulation tools are capable of emulating and validating extremely complex models of the environment. However, simulation tools do not support formal reasoning about system properties, while the formal modelling is limited in capabilities at describing complex physical processes. We propose a collaborative technique based on the co-simulation between the well-established Rodin toolset for the automated formal modelling and analysis of complex systems, and a Functional Mock-up Interface standard for tool-independent model exchange and co-simulation, which enables integration of the physical simulation models.

The approach for hybrid system development proposed in this work emphasises:

- *Model-driven development*, in a sense that system requirements and behaviour are captured by the model of the system and are verified through the modelling process, which facilitates better understanding of system requirements, constraints and hazards, thus reducing the number of errors in the next stage of system design.
- *Rigorous analysis*, by applying a formal method on the modelling process, which requires the developer to formally specify system behaviour and invariants, and to verify them in a systematic manner. A supporting tool that aids verification with

the automated model-checking and/or theorem proving can significantly facilitate this process.

- Step-wise modelling, in which the modelling process starts from a reduced set of the most essential functional requirements that are specified as an abstract model, which hides away system complexity. This facilitates verification since the abstract properties are easier to formalise and prove. More requirements and functional details are systematically added to the abstract model in small steps called refinements, so that the complexity of the model and required verification effort increase gradually.

- *Physical simulation*, which enables detailed modelling and validation of the dynamic physical behaviour of the system's environment and other continuous-time components that are hard or impossible to specify and verify with a formal method.

- *Co-modelling and co-simulation*, as a technique to combine the application of the rigorous analysis and physical simulation, such that formal modelling and verification is supplemented by the physical simulation. A refined formal model of the discrete system co-exists with a more concrete physical model of the environment and is co-simulated to validate the interaction between them.

Given that the presented method requires simultaneous co-modelling of system subcomponents in a formal specification language and a physical modelling language, we propose a development workflow that includes the following key steps:

1. The initial specification of the system starts from an abstract formal model that captures the basic set of functional requirements. This can be a model that includes both the control system and its (abstract) environment, or just the discrete-event controller component.

2. Specification of the discrete part of the system, which is typically a discrete-event controller, in a formal notation, starting from the basic set of requirements incorporated into an abstract model that is gradually refined by adding more requirements and details.

3. The abstract specification is elaborated, and the remaining system requirements and constraints are incorporated into the model in small refinement steps, maintaining the lowest possible increase rate of complexity. This process is accompanied by the formal verification and is iterative and non-linear, in a sense that several design choices can be made at each refinement step that may not always lead to a verifiable specification, hence refinements may be redone.

4. A physical model of the environment is developed after the formal model includes sufficient functionality to enable its co-simulation with the physical part. Similarly,

the physical model can be abstract enough to exhibit the behaviour required only for co-simulation. The abstract model is then refined in parallel with the formal model refinement.

To enable this workflow it is necessary to employ a systematic formal modelling, refinement and verification method as well as physical simulation technology, and provide a flexible co-simulation mapping of the formal and physical co-models at each refinement step. In this work we focus on the Event-B method (and its automated Rodin toolset) as the refinement-based formal verification method that enables high-level abstraction of system requirements and incremental development through step-wise refinement. The advantage of this approach over other formal methods for hybrid systems is the ability to distribute system complexity via the abstraction and layered refinement, which reduces the complexity of formal proof whilst maintaining requirement traceability. With respect to simulation we consider the general class of simulation technologies that are open-source and supported by existing tools. The proposed concept is developed into a tool extension for the Rodin platform that supports refinement and provides the capability of combined discrete-continuous verification through co-simulation, thus bridging the gap between the refinement-based formal analysis of discrete-event systems in Event-B and continuous-time simulation technologies (with the emphasis on an open-source Modelica language). Synergy of both technologies should reduce their individual limitations and result in an integrated method, capable of covering the whole development process: from requirements specification, design and analysis to implementation and verification-validation.

In order to validate the output of this project, evaluate its suitability and demonstrate the benefits for the hybrid system development we devise the following assessment criteria:

- The effectiveness of the approach at discovering and validating design errors in the discrete and continuous parts of the hybrid system that is being developed;
- The improvements over existing methods and addressed limitations;
- The level of expertise and resources required for the modelling and verification;
- The level of reusability and extensibility of the developed solution.

1.6 ADVANCE Project

This work was partially funded and driven by the ADVANCE project[1] (Advanced Design and Verification Environment for Cyber-physical System Engineering) whose objective was to develop a unified tool-based framework for automated formal verification and simulation-based validation of CPS. The project tried to build on the existing formal modelling language (Event-B) and associated toolset (Rodin), and augment those with novel approaches for verification, validation and testing, which included application of the system interaction and hazard analysis based on the System-Theoretic Process Analysis [95], multi-simulation via FMI standard and automated test case generation for the Modified Condition/ Decision Coverage [64]. To follow the objectives of the project we took a pragmatic approach that used existing tools and technologies as much as possible rather than re-inventing them.

The developed multi-simulation tool was used by the industrial partners of the project (Critical Software, Selex ES) on one of the main ADVANCE case studies. The case study explored a real problem in energy grids that concerns with automated voltage distribution and control in smart grid networks with domestic micro-generation via solar panels. Our tool was used to validate the developed formal model of the voltage control algorithm and the communication network of sensor interface units. The validation was performed by simulating the formal model against a developed Modelica model of the environment, represented by the low voltage network topology of a real test site. The iterative evaluation lead to constant feedback and significant improvements of the multi-simulation tool. This evaluation process and its outcomes are described in more detail in the project deliverable of the case study [22].

1.7 Work Outline

As a first step we determine the role of formal methods in the system engineering process. We evaluate a number of existing tools and methods of formal analysis (Event-B, Classical B, Z Notation, VDM, TLA) and introduce the theory for hybrid system modelling: hybrid automata and state charts. Next, we evaluate physical modelling and simulation technologies (Modelica, MATLAB Simulink, Ptolemy) and co-simulation frameworks (Functional Mock-up Interface, High Level Architecture, DESTECS). The important aspect at this stage is to identify technologies that are applicable to CPS development, mature (have reliable and maintained tool support) and widely used in both academia and industry on real-life problems. Other important criteria for the right choice of technologies are the openness and portability of code, ease of extension and wide support/interoperability with other tools.

[1]http://www.advance-ict.eu

In the next step we formulate the concept of our integration approach, which is based on the architecture of the FMI for Co-simulation standard and the Rodin platform, and essentially is a signal-based graphical composition of components for combined simulation. Here we formalise the semantics of both discrete and continuous-time steps of simulation and data exchange, and describe the actions of the simulation orchestration algorithm between Event-B and FMI components. The algorithm and the corresponding abstract data structures lay down the design foundation for implementation of our tool.

Following the actual design and implementation we validate the resulting tool on a number of case studies from different domains. The Modelica language and Dymola modelling environment are used for the development of the physical part of each system, while the discrete control part is specified in Event-B with the aid of Rodin tools. Two parts are then composed and simulated using the developed environment. The analysis of the obtained results against the devised assessment criteria indicates that the introduced improvements to the Rodin toolset facilitate verification process of multi-domain hybrid systems. We also highlight the advantages of such approach compared to the traditional simulation and suggest possible further improvements.

As an outcome of this work we provide an improvement to the existing Rodin toolset that addresses limitations of the Event-B language with respect to modelling of complex hybrid systems and facilitates validation of the mixed discreet-continuous system behaviour. The developed MultiSim tool equips Rodin with a simple to use graphical environment for modular composition and generic co-simulation of discrete-continuous components. It is based on the latest Rodin and Eclipse frameworks and supports state of the art physical modelling technologies via an established standard.

The remainder of the document is structured as follows. Second chapter introduces the evaluated methods of formal analysis, physical simulation technologies and co-simulation frameworks, concluding with a comparison analysis and a working plan for the development of proposed solution. Third chapter presents our concept of a generic simulator and explains its semantics with respect to discrete-event and continuous-time components. Fourth chapter describes the implemented tool and technologies behind it. Chapter 5 illustrates the experimental evaluation of the tool on a number of case studies from multiple domains. Final chapter summarises the achieved results and concludes with a plan for future work.

The main contributions of this work are following:

- A tool for the Rodin platform that extends its formal verification capabilities with the simulation-based validation that supports refinement and co-simulation with continuous physical models in the FMI format.

- A formalised discrete step simulation semantics for Event-B components that supports refinement and enables iterative development and simulation of Event-B models.
- A generic simulation orchestration algorithm that can simulate deterministically an arbitrary number of interconnected discrete-event (Event-B) and continuous-time (FMI) subsystems.
- A successful evaluation of the developed tool on a number of case studies from different domains, including the ADVANCE smart grid industrial case study, and comparison of the Rodin-based development process with traditional simulation.

Chapter 2

Background

In this chapter we describe state-of-the-art tools and methods, which we have identified as potential building blocks for the development of our integrated framework. First we briefly review some of commonly used formal methods. Then we give an introduction to a number of accepted formalisms for hybrid systems, such as hybrid automata and hybrid state machines, and the latest tools and languages for hybrid modelling. Finally, we describe co-simulation technologies suitable for integrating the first two aspects and conclude with a summary of the weaknesses of presented tools regarding the design of CPS, and an outline of the working plan to address those weaknesses by extending the existing tools and methods. The specific list of the reviewed technologies is by all means not comprehensive, but is intended to be representative of the domain of formal methods and simulation tools.

2.1 Formal Methods

Formal methods are the mathematical techniques for developing software and hardware systems. Based on mathematical theories they enable rigorous analysis and verification of system models at any stage of the development life-cycle. A good analogy for formal development, as opposed to a common code and test approach, is the use of blueprints in traditional engineering disciplines, where a formal artifact called blueprint is used for reasoning about the future system during its construction [2].

Despite a broad view on formal methods as being applied solely for formal proving, this is only one of their aspects. In fact, the key feature of the formalised development is the precise specification of what a system should do that is a prerequisite for verifying its correctness [62]. This involves a number of activities:

- Writing a specification

- Proving properties of the written specification
- Constructing an implementation from the specification via mathematical manipulations
- Verifying the implementation by mathematical argument

The vital step in the system development process is requirements engineering. Understanding system requirements, constraints and faults is a difficult task, which is often postponed to testing, when the system is already implemented and full coverage is problematic, leading to poor understanding of requirements and increased cost of fixing design errors. Verified modelling of requirements results in better understanding of the system being built, hence better design, quality and reliability. A precise specification acts as a technical contract between the client and developer and provides a common understanding of the purpose of the system [106].

Moreover, the discrete nature of computers and the fact that software is intrinsically unstable – if an implementation is not exact, the system is likely to collapse – makes non-rigorous verification weak [121]. Rigorous design means that a design is subjected to mathematical quantification of the desired system behaviour and through mathematical reasoning is proven to satisfy the requirements. It does not negate testing or ensure that a formally constructed system is correct, but it helps to identify conditions under which a system may fail.

Finally, formal methods and supporting tools enable verified construction from an abstract to a concrete model through refinement and subsequent transformation of the mathematical structures into executable code. Complex specifications can be decomposed into subcomponents and reused (reuse without a rigorous specification mechanism is a disastrous risk [72]).

Nevertheless, formal methods are by no means universal and have a number of limitations [77]. First of all, they cannot fully guarantee correctness and completeness of the specification with respect to the informal requirements. Secondly, complete formality is hard to achieve. It is also generally undecidable whether or not an implementation satisfies its specification. Particularly, in the context of cyber-physical systems a formal description of the environment in which the system is supposed to operate is required. A completely formal and correct specification of the physical environment is difficult to accomplish. In addition, formal methods generally require a higher degree of mathematical skill from software engineers. With respect to tools, formal methods are not integrated into systems that cover the full development process. Few tools support a combination of formal and informal technologies. Finally, formal methods only address certain aspects of software quality, mainly reliability and correctness, and are not always economically sensible.

Despite these limitations formal methods are a valuable tool for the development of complex systems, in which correctness and reliability are critical. In summary, software should be no exception to engineering design, and within the context of complex multi-domain systems such as CPS it is advantageous to use a rigorous development approach. In the following sections we review a list of formal methods that we have selected as a representative of the formal technologies with a record of successful applications in both the industry and academia. We introduce each method and evaluate it based on the assessment criteria, outlined in Section 1.5. The review starts from the Event-B method as our formal method of choice for this work.

2.1.1 Event-B

Event-B [2] is a formal method for system-level specification, modelling and analysis of discrete systems. The design of the Event-B language is greatly influenced by Action Systems [12] and the B Method [4]. Thus system behaviour is modelled in Event-B as a collection of state variables and discrete events that change system state. The employed mathematical notation is based on a simple formalism of set theory and first-order logic with the intent to facilitate formal proof. Key features of Event-B are the refinement mechanism that enables abstraction and incremental construction to overcome system complexity, and formal verification by deductive proof that allows to verify system properties in a rigorous way. Backed by the Rodin platform [3], formal proof can be performed automatically and interactively.

An Event-B model consists of *contexts* and *machines* that can have relationships between them. Contexts describe static parts of the model: *sets*, *constants* and *axioms*. They can *extend* other contexts, introducing new static data. The structure of the context that contains sets s, constants c and axioms $A(s, c)$ is as follows.

CONTEXT *name*
EXTENDS *context*
SETS s
CONSTANTS c
AXIOMS $A(s, c)$
END

Event-B machines describe dynamic part of the system: *variables* that determine system state, *invariants* (constraints on variables) that must hold to satisfy certain system properties and *events* that update variables and model system behaviour. A machine can *see* contexts and *refine* an abstract machine:

MACHINE *name*
SEES *context*
REFINES *machine*

VARIABLES v
INVARIANTS $I(v)$
EVENTS e
END

An event consists of *guards* (conditions, on which the event occurs) and *actions* (state modifiers) that are specified via before-after predicates. It can also declare *parameters*, or bound local variables that may represent inputs from the environment. A machine can have multiple events, but it must contain a special *INITIALISATION* event that executes first and initialises system variables to a valid state. An event has the following syntax.

EVENT *name*
ANY t
WHERE $H(u, v)$
THEN *act*
END

where $H(u, v)$ is a guard predicate on event parameters u and machine variables v, and *act* is an action that consist of a collection of assignments (deterministic or non-deterministic) to state variables, executed simultaneously. A deterministic assignment has the form: $x := E(x, y)$, where E is an expression on some initial values x, y. A nondeterministic assignment can either pick a value from a set: $x :\in Set$, or assign a value that satisfies a *before-after* predicate, with $x\prime$ denoting the value of x after the assignment: $x :\mid P(x, y, x\prime)$. All events in Event-B are atomic, i.e. only one event can execute at any moment. If a number of events are enabled (their guards evaluate to *true*), one is picked nondeterministically. This property, however, should not be considered a limitation of the language since any atomic event can be broken down into multiple events via refinement.

Event-B has a simple type system that consists of basic discrete types, such as boolean, integer or a declared set. Structured types, such as relations and functions, can be created using type constructors: powerset and cartesian product. Types of model elements are inferred using a type inference system, which is implemented in Rodin.

The top-down structured development approach via refinement is a primary feature of Event-B. A specification starts from an abstract model that describes the most essential system properties and elaborates to a concrete specification that covers all the requirements. Each refinement step can introduce new variables and events (horizontal, or superposition refinement), incorporating more requirements into the model. The refinement can also replace abstract variables with more concrete ones (vertical, or data refinement). Here is shown an example of an abstract Event-B model from Abrial [3] that specifies the access register system of a building:

```
MACHINE $m0$
SEES $c0$
VARIABLES $register, in, out$
INVARIANTS
    $register \subseteq USER$ // set of registered users
    $in \subseteq register$ // users inside the building
    $out \subseteq register$ // users outside the building
    $in \cap out = \varnothing$ // cannot be simultaneously in&out
    $registered \subseteq in \cup out$ // reg. users are in or out
EVENTS
    $INITIALISATION$
        BEGIN $register := \varnothing$
            $in := \varnothing$
            $out := \varnothing$
        END
    $Register$
        ANY $u$
        WHERE $u \in USER \setminus register$
        THEN $register := register \cup \{u\}$
            $out := out \cup \{u\}$
        END
    $Enter$
        ANY $u$
        WHERE $u \in out$
        THEN $in := in \cup \{u\}$
            $out := out \setminus \{u\}$
        END
END
```

The abstract machine above models the access register as a set of registered users (*register*), split into two disjoint subsets: *in* and *out*, representing users who are inside and outside the building. The event *Register* models the registration process by taking a parameter u that denotes some (nondeterministically chosen) non-registered user and adding it to a set of registered/outside users. The event *Enter* models a user u, who is registered and located outside, entering the building by moving it from *out* to *in*. The refinement step below represents an implementation decision to replace the abstract variables *in* and *out* with a database-like user record status, modelled as a total function *status*. *Register* and *Enter* events are refined: guards and actions on variables *in* and *out* are replaced to use the *status* function. In order to ensure that the refined machine/events imply the abstract model, concrete variables must be formally related to abstract variables through so called *gluing invariants*. Two gluing invariants therefore are defined in the refined machine that relate the *status* function to the sets *in* and *out*.

```
MACHINE $m1$
REFINES $m0$
SEES $c1$
VARIABLES $register, in, out, status$
INVARIANTS
    $status \in register \rightarrow STATUS$ // user status
    $\forall u \cdot u \in register \wedge status(u) = IN \Rightarrow u \in in$ // a gluing invariant relating user status to indoor users
    $\forall u \cdot u \in register \wedge status(u) = OUT \Rightarrow u \in out$ // a gluing invariant relating status to outdoor users
EVENTS
```

```
    $INITIALISATION$
      BEGIN $register := \varnothing$
        $status := \varnothing$
      END
    $Register$
      REFINES $Register$
      ANY $u$
      WHERE $u \in USER \setminus register$
      THEN $register := register \cup \{u\}$
        $status(u) := OUT$
      END
    $Enter$
      REFINES $Enter$
      ANY $u$
      WHERE $status(u) = OUT$
      THEN $status := IN$
      END
END
```

The semantics of machines and refinement relationships in Event-B is expressed via proof obligations (PO) of various types: well-definedness of expressions, invariant preservation, refinement POs, all of which need to be proven in order to verify model correctness. For example, invariant POs must be proven to guarantee that a model invariant is preserved by all events. The Rodin platform facilitates the verification process by automatically generating all proof obligations and trying to discharge them automatically with built-in and third-party provers.

Compared to other methods described below, and in particular to its predecessor Classical B, the Event-B method offers a very simple general approach to system specification and automated verification, backed up by a well-supported and easily extensible Rodin toolset with an active user and developer community. The latest release of Rodin is dated 22 June 2015, with over 40[1] available third-party plug-ins that enable theory extension, refactoring, decomposition, modular development, UML modelling, animation, code generation, etc. In particular, a very useful extension (and a standalone tool) for validating Event-B models in addition to the automated formal proof capabilities offered by Rodin is the ProB model-checker and animator tool [94]. It enables bounded model checking of Event-B models by initialising declared sets to finite values and exploring the state space by executing events in order to find theorem/invariant violations, deadlocks and other constraints. In addition to the validation, modelling extensions such as State machines [131] enable visual abstraction and high level modelling in Event-B using the graphical state chart notation. Based on these features we have chosen Event-B and Rodin to be the formal technology for this work, which has also extensively utilised both of the mentioned Rodin extensions.

[1]According to the official Event-B Wiki at http://wiki.event-b.org/index.php/Rodin_Plug-ins

2.1.2 B Method

B Method [5], also known as classical B, is a predecessor of the Event-B language. Although the latter is considered an extension of B, the two methods differ in their design and goals. While Event-B is a general formal approach for designing distributed concurrent systems, B Method is targeted at specification, design and implementation of software. It has a long and successful history of industrial applications in the fields of railway transportation, automotive and aeronautics, and is supported by commercial tools such as Atelier B from ClearSy[2] for the development of certified software (according to ClearSy in 2014 more than 25% of automatic metros had safety-critical software developed with Atelier B). In fact, it was one of the first formal methods to be backed up by an industrial strength tool suite that incorporated full development process from the abstract modelling using refinement and automated proof, to implementation and automatic code generation [133].

The specification language of B Method is close to Event-B – it is state-oriented and based on the Abstract Machine Notation (AMN), first-order predicate calculus, Zermelo-Fraenkel set theory for data modelling and General Substitution Language (GSL) for modelling operations, which must perform according to specification if called within a given precondition. Similar to Event-B the key notion of B Method is the top-down stepwise development from a high-level specification towards a concrete implementation using the refinement mechanism accompanied by mathematical proofs. Refinement decreases the complexity of the proof process and improves the traceability of requirements as specification details are introduced in small steps. Proof obligations guarantee that refinements are consistent with their abstractions. The last step in a refinement chain is the implementation, written in a language called B_0 that resembles a programming language.

A system is modelled in B as a collection of related components, where a component is either an *abstract machine*, *refinement* or *implementation*. The modelling process starts from the abstract *machine*, which has a static definition of *sets*, *constants* and *properties* over sets and constants. Compared to Event-B both static and dynamic aspects are defined in a single component. The dynamic aspect comprises *variables*, inductive *invariant* (properties over variables), *assertions* (safety properties), *initialisation* substitution and a list of *operations*. An operation in B consists of a precondition, body (generalised substitutions on variables) and an optional list of input and output parameters. Operations are defined in terms of the weakest precondition semantics, which means that a precondition must be true when one calls an operation (otherwise any state can be reached and the invariant is not guaranteed). Unlike in Event-B the generalised substitution language in B is not limited to three basic substitutions (assignment, relational assignment and non-deterministic choice from a set) and includes guard, choice

[2]http://www.atelierb.eu

and iteration substitutions. An illustrative example of an abstract machine from [122], which specifies an array element search algorithm, is shown below.

```
MACHINE $search$
CONSTANTS $t, n$  // array t of size n
PROPERTIES $n \in \mathbb{N} \land t \in 0..\mathbb{N} \to \mathbb{N}$
OPERATIONS
$b \leftarrow search(v) =$
   PRE $v \in \mathbb{N}$ THEN
      IF $v \in ran(t)$
      THEN $b := true$
      ELSE $b := false$
      END
   END
END
```

The abstract machine represents the concept that must be developed into a program. It does not specify how the search algorithm should be implemented, but rather formalises the main requirement, that is, the algorithm must return *true* if an element v is found in the array t, otherwise it must return *false*. In order to resemble the executable code the abstract specification has to be refined.

During the development process abstract *machines* are refined through a sequence of design steps into *refinements* and subsequently into *implementations*. This process is facilitated by the modular development, supported in B method via different structuring primitives, which include the language clause for refinement *refines*, hierarchy: *promotes* (makes hidden identifiers visible in the component's signature), *includes* (a machine includes another machine and promotes its operations) and *extends* (inclusion without promotion), and sharing, or reuse of the existing machine's constructs: *sees* (read-only sharing of services) and *uses* (read/write sharing). In addition, the *imports* clause allows an implementation to call operations of imported machines. Thanks to modular development B specifications can be refined independently and then composed, which helps to reduce the proof complexity and allows for model re-use. The goal of the development is to obtain a proved implementation that can be translated into code. An example implementation of the *search* algorithm specified above may look as follows:

```
IMPLEMENTATION $searchImlp$
REFINES $search$
OPERATIONS
$b \leftarrow search(v) =$
   VAR $i$ IN
      $i := 0$
      WHILE $i < n$ DO
         IF $t(i) = v$ THEN
            $b := true$
         END
         $i := i + 1$
         INVARIANT $0 \leq i \leq n \land b \in \mathbb{B} \land b = true \leftrightarrow v \in t[0..i-1]$
         VARIANT $n - i$
      END
```

```
    END
END
```

The refined specification modifies the *search* operation by describing concrete steps of the algorithm that implement the abstract requirement. The refinement uses an unbounded choice substitution (VAR x IN Sx END), guarded substitution (IF P THEN T END) and iteration substitution (WHILE P DO S INVARIANT J VARIANT V END) to implement a while loop with a counter variable i that traverses the array t and sets the return variable b to true if the element v is found in t. The loop requires an invariant and variant to be specified, which must be proved. The important step of each refinement is to prove that: 1) the refined specification is correct (proof of correctness), 2) the refinement is a valid replacement of the abstract machine in all possible situations the abstract machine is defined (proof of consistency).

The *implementation* corresponds to the program level, where the GSL is restricted to its subset language B_0 that is close to the target programming language, hence it can be translated directly to executable code, such as C or Ada, using tools like Atelier B. Thus, the resulting code can be proved to be consistent with the original specification and correct. The B method guarantees that on a checked development the generated program terminates and causes no runtime errors [20].

2.1.3 Z Notation

Z Notation [138] is another formal language for specifying computer-based systems that shares roots with B. Similar to B, it supports refinement and is based on the mathematical notation of set theory, lambda calculus and first-order predicate logic. The distinguishing feature of Z is the schema notation for structuring and modularising mathematical specifications, which enables their reuse and composition. Schemas are a named box notation that combines and structures mathematical objects and their properties. A schema can describe a system state in terms of variables and their relations (state schemas) or the way that state can change using functions that modify state variables (operation schemas). Schemas are also used to describe system properties and to reason about refinements [148]. A simple schema of the book database from [139] is shown below:

BookCollection

$collection : \mathbb{P}\ BOOK$

$\#\ collection \leq max\_size$

The state schema *BookCollection* declares a variable *collection* as a set of elements of type *BOOK*. The predicate below expresses a system invariant that the size of the *collection* must not exceed *max_size*.

Similar to B the operations on states are expressed in Z via before-after predicates. However, instead of assignment Z makes use of identifier decorations: a variable with no decoration represents the current (before) state, variables decorated by prime ($'$) represent the new (after) state, input variables are decorated by '?' and output variables are decorated by '!'. An operation schema that lets to add a new book to the book collection can be defined as follows:

$$\begin{array}{|l}
\textit{AddBook} \\ \hline
\Delta \textit{BookCollection} \\
\textit{newbook}? : \textit{BOOK} \\ \hline
\textit{newbook}? \notin \textit{collection} \\
\textit{collection}' = \textit{collection} \cup \{\textit{newbook}?\} \\ \hline
\end{array}$$

The *AddBook* operation schema includes the original schema *BookCollection*, thus adding all its state components and associated predicates. The Δ-inclusion convention denotes that the included schema's state variables may be changed, in other words, the operation modifies its state. A similar Ξ-inclusion leaves the state unchanged, and is often used for specifying error handling operations.

Defined schemas can be further structured using various schema operators, such as logical connectives and quantification, decoration, change of state, renaming, hiding, projection and sequential composition. Z also supports generic (parameterised) schemas.

With some similarities to B the Z notation has important differences. For instance, invariants in Z are incorporated into operation specifications and alter their meaning, whereas in B invariants are checked against the state changes described by operations to ensure consistency. B also makes a careful distinction between the logical properties of preconditions and guards, which is not as clear in Z, where preconditions are implicit in operation definitions [45].

Although Z is a powerful formal specification language, it is not intended for describing timed and concurrent systems. Several works address this limitation by combining it with other formal languages, for example, with Timed Communicating Sequential Processes [97], which is a process-based formalism for modelling concurrency that also supports real-time semantics.

Among still actively maintained tools for development in Z are the ProofPower[3] tool suite for Higher Order Logic, which also supports Z, and the ongoing Community Z Tools[4] project that aims at developing a toolset for editing, type checking and animating Z specifications and its extensions.

2.1.4 TLA

Temporal Logic of Actions is a formal method based on a temporal logic targeted at specifying and reasoning about concurrent algorithms [81]. The central idea of TLA is that reasoning about and verifying correctness of an abstract algorithm expressed in terms of a simple mathematical logic is much easier than verifying a program that implements the algorithm. It offers a single logic for specifying both the algorithm and its properties as formulas. To prove the correctness of the algorithm one must prove that the formula specifying the algorithm implies that of the properties. TLA is designed to be simple yet powerful to express complex concurrent algorithms. It combines two logics: a logic of actions and a standard temporal logic. The semantics is defined in terms of states, where a state is an assignment of values to variables. State functions correspond to expression of an ordinary programming language, while predicates correspond to boolean expressions and assertions to be verified. Actions correspond to atomic operations of a concurrent program.

An action is a boolean expression, such as $x\prime + 1 = y$, formed from unprimed variables (referring to the old state), primed variables (referring to the new state) and constants. Formally, the semantic meaning $[\![\mathcal{A}]\!]$ of an action $\mathcal{A}$ is a function that assigns a boolean $s[\![\mathcal{A}]\!]t$ to a pair of states s, t that denote an old state and a new state, respectively. The meaning of the above action $s[\![y = x\prime + 1]\!]t$, which asserts that the old state value of y is greater than the new state value of x by 1, equals to the boolean $s[\![y]\!] = t[\![x]\!] + 1$ (unprimed variables v are replaced by $s[\![v]\!]$ and primed variables $v\prime$ are replaced by $t[\![v]\!]$). If $s[\![\mathcal{A}]\!]t$ is true, the pair s, t is called $\mathcal{A}$ step, which, in the context of a program, means that executing an operation denoted by action $\mathcal{A}$ in state s produces state t. Action $\mathcal{A}$ is valid if it is true regardless of the value substitutions for all primed and unprimed variables. The validity of an action expresses a theory about values.

Temporal logic [117] is used in TLA for reasoning about algorithms, which requires reasoning about their executions – sequences of states. Temporal formulas are built from elementary formulas with boolean operators and the unary operator $\Box$ (*always*). The semantics of temporal logic is based on behaviours – infinite sequences of states that a computer goes through when executing an algorithm. Thus, a temporal formula is an assertion about a behaviour. Formally, the meaning $[\![F]\!]$ of a formula F is a boolean function on behaviours, and $\sigma[\![F]\!]$ denotes the value that F assigns to behaviour σ. The

[3] http://www.lemma-one.com/ProofPower/getting

[4] http://czt.sourceforge.net

behaviour σ satisfies F if $\sigma[\![F]\!]$ is true. A temporal formula is called valid if it is satisfied by all possible behaviours. For example, $\langle s_0, s_1, s_2, \ldots\rangle[\![\Box F]\!]$ asserts that F is true at all times during the behaviour $\langle s_0, \ldots\rangle$. Other temporal formulas of TLA include:

$\Diamond F$	F is *eventually* true
$\Box \Diamond F$	F is true *infinitely often*
$\Diamond \Box F$	F is *eventually always* true
$F \leadsto G$	F *leads to* G (at any time F is true, G is true then or at some later time)

Algorithms and properties are defined in TLA in terms of temporal formulas. An algorithm specified by a temporal formula F satisfies a property G if every behaviour representing its possible execution satisfies G, or more formally, if $\sigma[\![F \Rightarrow G]\!]$ is true for every behaviour σ. As algorithms are specified using actions, for an action $\mathcal{A}$ to satisfy a behaviour first pair of states in that behaviour must be an $\mathcal{A}$ step: $\langle s_0, s_1, s_2, \ldots\rangle[\![\Box\mathcal{A}]\!] \triangleq s_0[\![\mathcal{A}]\!]s_1$. Consequently, a behaviour satisfies a temporal formula $\Box\mathcal{A}$ if every step of that behaviour is an $\mathcal{A}$ step. Formulas such as $\Box\mathcal{A}$ are called RTLA (Raw Temporal Logic of Actions) – they allow assertions on behaviours that should not be assertable. Hence, TLA is defined as a subset of RTLA by allowing *stuttering* steps that do not modify variable values. As an illustration of how a program can be specified in TLA follows an example from [81] of a simple program with two incrementing counters:

```
**var natural** $x, y = 0$;
**do** $\langle true \rightarrow x := x + 1\rangle$
  $[\!]$
  $\langle true \rightarrow y := y + 1\rangle$ **od**
```

This program uses Dijkstra's **do-od** construct and $[\!]$ (*bar*) separator for unordered alternatives [51]. An RTLA formula Φ representing this program may be defined as follows:

$$
\begin{aligned}
&Init_\Phi \triangleq (x = 0) \wedge (y = 0)\\
&\mathcal{M}_1 \triangleq (x\prime = x + 1) \wedge (y\prime = y)\\
&\mathcal{M}_2 \triangleq (x\prime = x) \wedge (y\prime = y + 1)\\
&\mathcal{M} \triangleq \mathcal{M}_1 \vee \mathcal{M}_2\\
&\Phi \triangleq Init_\Phi \wedge \Box\mathcal{M}
\end{aligned}
$$

Predicate $Init_\Phi$ denotes the initial condition, $\mathcal{M}_1$ and $\mathcal{M}_2$ are the atomic operations of incrementing either x or y, and $\mathcal{M}$ is a step representing an execution of one of operations. To enable stuttering (steps that leave both x and y unchanged) Φ needs to be modified:

$$\Phi \triangleq Init_\Phi \wedge (\Box\mathcal{M} \vee ((x\prime = x) \wedge (y\prime = y)))$$

In the TLA notation actions (unless they are predicates) can only appear in the form $\Box[\mathcal{A}]_f$, where f is a state function and $[\mathcal{A}]_f \triangleq \mathcal{A} \vee (f\prime = f)$. A shorthand notation for ordered pairs with equal components lets one to rewrite a conjunction $(x\prime = x) \wedge (y\prime = y)$

to an equality $\langle x\prime, y\prime\rangle = \langle x, y\rangle$, in which $\langle x\prime, y\prime\rangle$ can be further rewritten as $\langle x, y\rangle\prime$. Using this notation the above formula Φ can be rewritten in TLA as

$$\Phi \triangleq \mathit{Init}_{\Phi} \wedge \Box[\mathcal{M}]_{\langle x,y\rangle}$$

Formula Φ describes a safety property – an assertion than something bad may never happen. In this case, that the program may not start in any state other than Init_{Φ} followed by a step no other than $[\mathcal{M}]_{\langle x,y\rangle}$. To complete the definition of a program one must also add a liveness property – an assertion than something will eventually happen, such as x and y will be incremented infinitely often, and a fairness property – an assertion that a possible operation will eventually be executed, such as both x and y will be incremented infinitely often. Liveness in TLA is expressed in terms of fairness [136], for which two definitions are available:

Weak fairness:	$\mathit{WF}_f(\mathcal{A}) \triangleq (\Box \diamond \langle\mathcal{A}\rangle_f) \vee (\Box \diamond \neg \mathit{Enabled}\langle\mathcal{A}\rangle_f)$
Strong fairness:	$\mathit{SF}_f(\mathcal{A}) \triangleq (\Box \diamond \langle\mathcal{A}\rangle_f) \vee (\diamond\Box\neg \mathit{Enabled}\langle\mathcal{A}\rangle_f)$

The TLA+ specification language [82], which is a syntactic extension to TLA, can be compared to B Method, since it includes ZF set theory, functions and constructs for modular development such as *extends* and *instance*. The semantics of temporal operators is expressed over traces of states, as opposed to the weakest precondition calculus for B actions. Both semantics are equivalent with respect to safety properties, but TLA allows expression of fairness and eventuality properties, not directly supported by B (one can still use a tool such as ProB with LTL for model-checking temporal properties).

The most recent development environment for TLA+ is the TLA Toolbox[5], which combines the PlusCal translator from an algorithmic language to TLA+, TLC model checker and TLAPS mechanised prover (currently limited to non-temporal safety properties). The toolbox is made open-source and is available free of charge under the MIT license. Although not as active and rich in contributions from community as toolsets like Rodin, the TLA Toolbox is still maintained, with the latest release dated 25 February 2014.

2.1.5 VDM

Vienna Development Method (VDM) is one of the oldest model-oriented formal methods for the development of computer-based systems [57]. It comprises a set of mathematically well-defined languages and tools for construction and analysis of systems and helps to identify the areas of incompleteness or ambiguity in informal specifications. Models are expressed in the VDM-SL (specification language), which supports the definition of data – (constructed) structured data types, collections and basic types, and functionality – operations over data types, defined either implicitly by preconditions and postconditions or explicitly by algorithms. An extension of the VDM-SL – VDM++, adds support for

[5] http://research.microsoft.com/en-us/um/people/lamport/tla/toolbox.html

object-oriented and concurrent system modelling. VDM, however, does not prescribe a particular development process or methodology and gives developers the freedom of using VDM components as they fit.

The key principles of VDM are the abstraction from the details that are not relevant to the purpose of the model (hence, the model's purpose must be well-understood), and the rigour, i.e. the capacity to perform a mathematical analysis of the model's properties in order to gain confidence in the validated characteristics of its implementation.

The structure of the models in the VDM-SL consists of the declaration of data and functionality. Data includes inputs/outputs and state (internal) data, while functionality includes operations that can be invoked from the system interface, as well as auxiliary functions. The VDM++ adds capability to structure models into class definitions, where state variables take the role of instance variables and operations represent class methods.

The data is defined in VDM using the basic abstract data types (natural, integer, rational, real, character) and type constructors. The data can be restricted by invariants that represent conditions to be respected by all elements of the data type to which they are attached. For instance, a variable type to represent the latitude of an aircraft can be declared as follows[6]:

```
Latitude = real
inv lat == lat >= -90 and lat <= 90
```

The `Latitude` type can then be used in constructed types to represent the position of an aircraft and its path:

```
Position :: lat  : Latitude
            long : Longitude
            alt  : Altitude
FlightPath = seq of Position
FlightDetails = map AircraftId to FlightPath
```

The functionality is described in terms of functions and operations that accept input values and produce output values. Functions can be implicit or explicit. The explicit function is an expression that denotes the returned result in terms of input parameters, as in the following function declaration that adds a new position to the end of a flight path:

```
AddPos: FlightPath * Position -> FlightPath
AddPos(fp, p) == fp^[p]
```

[6]Code examples are from [57]

Here the first line declares the input parameters and a return type, while the second line defines the body of the function, i.e. adding a position to the end of the path. The definition of this function, and explicit functions in general, is biased towards implementation, therefore an implicit function definition is more suitable for the abstraction of functionality in VDM. Implicit definitions are given in terms of logical expressions (postconditions) that must be satisfied by the return result and do not have a direct reference to a particular computation algorithm. The following example shows an implicit declaration of a function that selects an aircraft for landing:

```
Select(fd:FlightDetails) a:AircraftId
pre dom fd <> {}
post a in set dom fd
```

Here the function only defines that the returned aircraft `a` is present in the domain of flight details `fd`, with no explicit indication of how to compute the result. The latter means that implicit functions are not directly executable. Additionally, both explicit and implicit functions may be not well-defined for all inputs, e.g. a map of flight details `fd` can be empty. Such constraints on inputs are explicitly declared in VDM as preconditions. The second line in the above example declares a precondition on non-emptiness of the input to ensure a correct result can be returned.

Finally, the operations in VDM are special type of functions that, in addition to returning a result, can have side effects on state variables. A variable can be defined as a state variables using the `state` construct:

```
state Airspace of
      fd: FlightDetails
end
```

Similar to functions, operations can be implicit or explicit. For instance, an implicit operation to add a new aircraft with the initial position in its path can be defined as follows:

```
New(a:AircraftId, p:Position)
ext wr fd: FlightDetails
pre a not in set dom fd
post fd = ~fd munion {a |-> [p]}
```

The `ext` clause specifies that the operation has a read/write access `wr` on the variable `fd`. The ~ symbol on the state variable in the postcondition denotes its value before the execution of the operation. The `munion` operator forms the union of two mappings.

Most of the validation in VDM can be performed using proof obligations – static and run-time error checks on the constructed model, and validation conjectures – checks on expected emergent behaviour. A range of POs is common for all models in VDM, such as data type POs (values belong to a specified type) and invariant preservation/satisfiability POs (functionality respects invariants).

Besides proof obligations and mathematical analysis via formal proof, which is more relevant for critical applications, a less rigorous technique of exploring the properties of a model is testing. Explicit operations and functions can be executed directly by the interpreter, provided their expressions are within the executable subset of the language. This may not always be the case due to the expressiveness of the VDM-SL, e.g. a quantification over unbounded data types makes expressions non-executable. Despite these caveats, the models of industrial applications are usually built as executable and validated via testing/coverage analysis.

A much more rigorous and general validation technique compared to testing is the formal proof, which is supported in the VDM-SL semantics by the proof theory [21] and the development of the automated tools, such as the open-source Overture project[7] and the VDMTools commercial tool[8], which also supports code generation.

When compared to Event-B and classical B, VDM follows similar design principles and objectives. For instance, it also uses predicate logic to describe invariants and postconditions of the operations on data (implicit function definitions). Similarly, VDM uses sets and partial functions for modelling data. The latter property can lead to undefined terms if a function is applied outside of its domain [45]. To deal with undefinedness VDM uses the Logic of Partial Functions (LPF) in which, for instance, the value of a conjunction in which either operand is false is defined to be false [56], as opposed to Event-B, which uses classical logic that simplifies proof, but hence requires more care to specify well-definedness conditions[9].

2.2 Hybrid Modelling and Simulation

In this section we describe the hybrid modelling formalisms and simulation technologies that support them. Our development workflow involves co-modelling and co-simulation of the continuous-time physical environment alongside the formal modelling of the discrete control. Hence, it requires a physical modelling and simulation technology that, according to our stipulated assessment criteria, should have the following properties:

[7]http://www.overturetool.org
[8]http://www.vdmtools.jp/en/
[9]The tools for VDM actually use two-valued logic assuming left-to-right evaluation (McCarthy conditionals [104]) and not full LPF.

- It supports abstraction, modularity and hierarchical development to enable the top-down co-modelling and validation process, starting from an abstract model of the system towards its realisation.
- It is capable of modelling and simulating complex physical processes of the continuous environment in which the developed hybrid system shall be operating.
- It is a free and open technology that is supported by as many tools as possible to be fully integrated and versatile.
- It supports a free and open standard for the model co-simulation, which can be easily adopted and implemented into a supporting tool for the integrated development.

One of the technologies that is gaining recent popularity in multi-domain simulation field is the Modelica language, described later in this section. Compared to other simulation technologies it matches all the listed criteria. Hence, we chose Modelica as a preferred physical modelling and simulation language. But first, we introduce the discrete-continuous formalisms that are at the foundation of any hybrid system modelling.

2.2.1 Hybrid Formalisms

The concept of automaton is often used to effectively model state-based computation processes. Here we describe two automata-based formalisms particularly designed for hybrid systems. They both derive from the conventional finite-state automaton [60], which can be defined as a 4-tuple (S, S_0, A, T) with a

- finite set of control states S
- subset of initial locations $S_0 \subseteq S$
- finite set of events A, including the internal event $\tau \in A$
- finite set of transitions $T \subseteq S \times A \times S$ of the form $t = (s, a, s\prime) \in T$, also written as $s \xrightarrow{\alpha} s\prime$, where $s \in S$ is the source state, $s\prime \in S$ is target state, and $a \in A$ is transition action.

A transition system of the finite automaton $M = (S, S_0, A, T)$ can be defined using the enabling condition $\mathit{Enabled}(l_0, (l, a, l\prime)) \Leftrightarrow (l_0 = l)$, where l_0 is the current configuration from the set of all possible configurations S, and configuration transition function $\Gamma_{l,a,l\prime}(l) := l\prime$. The resulting action rule is:

$$\frac{Enabled(l,t)}{l \xrightarrow{t} \Gamma_t(l)} action$$

The rule states that if a transition t from the location l is enabled, a transition to a new location, determined by the transition function Γ, can be taken, which produces a specified action.

For the finite behaviour a set of accepting locations $S_f \subseteq S$ can be defined using the valid finite traces $s_0 \xrightarrow{a_0} s_1 \dots s_n$ with $s_0, s_1 \dots s_n \in S^*$ that are given by all finite sequences of transitions starting from the initial location $s_0 \in S_0$ and ending at the accepting location $s_n \in S_f$.

Hybrid Automata

The formalism of hybrid automata [65] has been developed to describe mixed discrete-continuous systems, for example where a digital controller interacts with an analog plant. A hybrid automaton has two kinds of transitions: discrete jumps in the state space, caused by mode switches, and continuous evolution along continuous flows within a mode. Mathematically such an automaton H can be defined by

- continuous state space $\mathbb{R}^n$
- control graph, or finite directed multigraph (Q, E) with control modes (vertices) Q and control switches (edges) E
- initial conditions $init_q \subseteq \mathbb{R}^n$ that are predicates, which initially hold in $q \in Q$
- invariant conditions $inv_q \subseteq \mathbb{R}^n$ that are evolution domain restrictions, which must be true while in $q \in Q$
- flow conditions $flow_q \subseteq \mathbb{R}^n \times \mathbb{R}^n$ that are relationships between continuous state $x \in \mathbb{R}^n$ and its derivative $\dot{x} \in \mathbb{R}^n$ during continuous evolution in $q \in Q$
- jump conditions $jump_e \subseteq \mathbb{R}^n \times \mathbb{R}^n$ that determine a new continuous state value $x\prime \in \mathbb{R}^n$ from its old value $x \in \mathbb{R}^n$ when edge $e \in E$ is followed

The jump condition $jump_e$ is defined as a conjunction of transition guards $guard_e \subseteq \mathbb{R}^n$ that determine conditions on which an edge can be taken, and variable resets $reset_e \subseteq \mathbb{R}^n \times \mathbb{R}^n$, specified by a list of assignment statements $x_1 := \theta_1, \dots x_n := \theta_n$, which update state variables if transition $e \in E$ is followed. In turn, flow conditions are specified by a set of differential equations $\dot{x}_1 := \theta_1, \dots \dot{x}_n := \theta_n$. As an example, a temperature regulation system can be modelled by the following hybrid automaton.

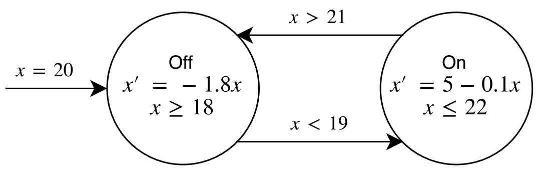


Figure 2.1: Hybrid automaton of a thermostat

In Figure 2.1 the temperature is denoted by x. The controller has two modes: *Off*, in which the heater is off and the temperature falls according to the flow condition $\dot{x} = -1.8x$, and *On*, in which the heater is on and the temperature rises according to $\dot{x} = 5 - 0.1x$. The initial condition is the *Off* mode and the temperature of 20 degrees. Jump conditions specify control dynamics: the heater may go on if the temperature drops below 19 degrees or go on if it rises above 21 degrees. Furthermore, invariants guarantee that at the latest the heater will be turned on if the temperature drops to 18 degrees, or it will go off if the temperature rises to 22 degrees.

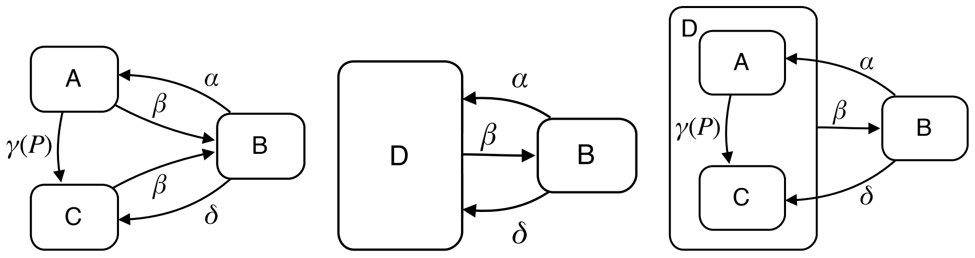
The execution of hybrid automata involves continuous flows and discrete jumps between modes. The automaton can be abstracted to a timed transition system, which retains the information only on the source, target and duration of each flow. It can be abstracted further to a time-abstract transition system that abstracts duration of flows, resulting in a fully discrete system.

Hybrid automata support composition. Two automata H_1 and H_2 can be composed to a parallel composition $H_1 \parallel H_2$, with the timed and time-abstract semantics, defined by interaction via joint events. If H_1 and H_2 have a common event α, they are synchronised on α-transition. If only one automaton has event α, each α-transition is synchronised with 0-duration time transition of the other automaton. For each real $\delta > 0$ a time transition of H_1 with duration δ must synchronise with a time transition of H_2 of the same duration.

Statecharts

Finite state machines and their corresponding transition systems (state diagrams) are well-recognised formalisms for event-based dynamic systems. However, they have a disadvantage of a 'flat' structure that leads to an exponentially growing multitude of states and, as a result, in unstructured and chaotic diagrams. For example, a flat model of N parallel state machines, each containing M states, requires a total number of M^N states instead of $M \times N$ states of the equivalent structured model. This problem was tackled by Harel in his visual notation, called Statecharts [63], designed as extension of conventional state machines with clustering, orthogonality (concurrency) and refinement, and 'zoom' capabilities (moving between levels of abstraction) by introducing AND/OR decomposition of states, inter-level transitions and a broadcast mechanism for communication between concurrent components. In a nutshell *statecharts = state-machines + depth + orthogonality + broadcast-communication.*

The clustering and refinement is achieved in statecharts by allowing encapsulation of states, and transitions that could originate and terminate at any level of depth. A state machine[10] in Figure 2.2(a) can be transformed to two-level statechart in Figure 2.2(c): states A and B can be clustered into a new state D with XOR (exclusive or) semantics, and two transitions on event β replaced by a single transition, outgoing from the superstate. Conversely, the resulting statechart could be achieved via refinement of state D in Figure 2.2(b) by introducing substates A and C, extending α and δ triggered transitions to point to substates and adding the γ transition.



(a) Original (b) Abstract (c) Clustered/refined

Figure 2.2: Statechart clustering and refinement

The term orthogonality describes the AND decomposition of states, which captures the property that if a state is active, the system must be in all of its AND components. This is achieved in statechart notation by splitting a state into components using dashed lines. Orthogonality can exhibit independence or synchronisation behaviour, respectively triggering either a single or multiple transitions in orthogonal regions at the same time.

Finally, functional behaviour of the system is described in statecharts by actions and activities that are allowed on both transitions and states. The transition labelling has the form $\alpha(P)/S$, where α is the triggering event, P is the guard and S is the action to be carried out upon transition. Actions can be also carried out on entering and exiting a state. An activity can be associated with a state, denoting it is carried out continually throughout the system being in that state. As opposed to actions, which are instantaneous, activities take some time. Figure 2.3 shows an example of orthogonality with AND states A and F, state actions S and T and state activity X.

[10]State machine examples are from [63].

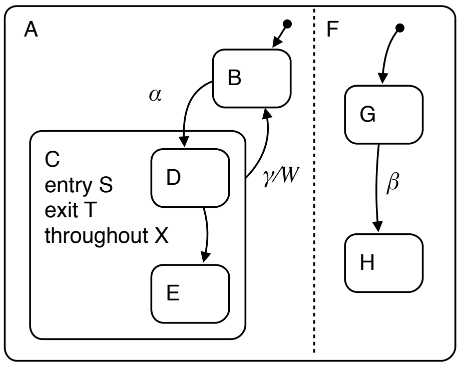


Figure 2.3: Orthogonal states and state actions/activities

Hybrid Statecharts

Although discrete event approach, employed by statecharts, is justified for modelling the discrete reactive systems, it is not always appropriate to model both the system and the environment as discrete processes. Hybrid statecharts [76] are the extension of Harel's statecharts with continuous semantics. They introduce timed transition annotations, given by time intervals $\{l, u\}$ between the lower and upper bound of transition, and state labelling by differential equations, denoting the continuous change that occurs while the state is active. A simple hybrid statechart is shown below.

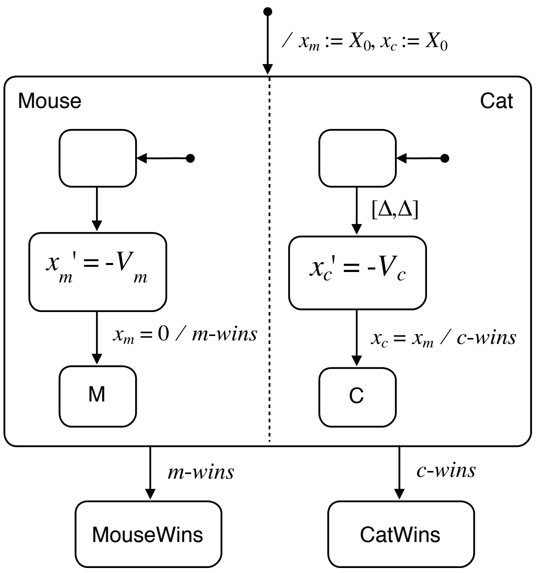


Figure 2.4: Hybrid statechart of the cat and mouse game

Figure 2.4 models a game of cat and mouse from [76]: at time $t = 0$ a mouse starts to run with constant velocity V_m from some initial position towards a hole in the wall at distance X_0. After Δ time units a cat is released from the same initial position. It starts to chase the mouse at velocity V_c along the same path. Depending on the parameters chosen either the mouse reaches the hole, or the cat catches the mouse. The system is modelled by two continuous variables x_m and x_c, denoting the distance of the mouse and cat, respectively, from the wall. Initially both variables are set to X_0. Then each player proceeds in its local behaviour, described by corresponding orthogonal region. The mouse immediately enters its operation state, in which the variable x_m changes according to equation $\dot{x}_m = -V_m$. The cat enters its operation state only after a delay of Δ. The win conditions are modelled by transition guards $x_m = 0$ (mouse wins) and $x_c = x_m$ (cat wins), each generating an event *m-wins* or *c-wins*, respectively, which moves the whole system to the final state *MouseWins* or *CatWins* (the result is nondeterministic if $x_m = x_c = 0$).

2.2.2 Simulink/Stateflow

Simulink is a graphical environment for model-based design and simulation of dynamic and embedded systems that integrates with MATLAB, which is a high-level language and interactive environment for numerical computation, visualisation, and programming, and provides a diagram editor with a library of components for multiple domains, including signal processing, communications and controls [102]. Models are designed using diagrammatic language of hierarchical interconnected blocks. Key features of Simulink are

- Rich and expandable library of customisable blocks, including algorithmic (Sum, Product, Lookup Table), structural (Mux, Switch, Bus Selector), continuous and discrete dynamics (Integrator, Unit Delay)
- Ability to construct new blocks by assembling and grouping the existing ones
- Functions and algorithms written in MATLAB can be incorporated into Simulink as Embedded Function blocks
- Model Explorer, which allows to navigate, create and configure all the signals, parameters and properties of a designed model
- Finished models can be simulated either interpretively (Normal, Accelerator or Rapid Accelerator mode) or as compiled C-code using fixed- or variable-step solvers
- Results of simulation are plotted via Scope blocks and can be examined with a graphical debugger and profiler, which allows to diagnose model performance and unexpected behaviour
- Provided API allows developers to connect Simulink to other simulation software and incorporate custom code

The principle behind the modelling technique in Simulink is similar to actor models, where a continuous-time physical system, described by a differential or integral equation that relates input signals to output signals, can be viewed as a box with an input port and output port, where signal of the input port x and signal of the output port y are functions over time to a signal value: $x, y \in \mathbb{R} \to \mathbb{R}$ [90]. Such a model of a system that can be defined as a function $S \in X \to Y$, where $X = Y = \mathbb{R}^{\mathbb{R}}$, is called an *actor*. Two actors S_1 and S_2 can be then composed by the output/input port, such that $y_1 = x_2$.

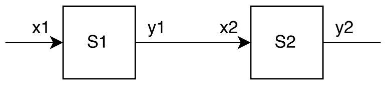


Figure 2.5: Composition of two actors

The same approach is followed in Simulink, where blocks are elementary signal processing units, interconnected by input/output ports. The model is described as a directed graph of blocks that formalises interaction between the individual components. The resulting model is close to the solution algorithm. For instance, a typical control system with feedback that measures an error e, which represents the difference between the desired behaviour r and actual behaviour y, is modelled with a directed loop in Figure 2.6.

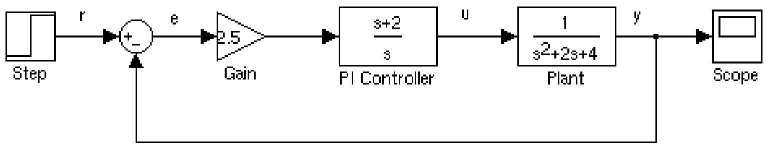


Figure 2.6: Basic model of a control loop in Simulink

Such design is typical to closed-loop feedback systems [105]. Signal u from the controller is sent to the plant where a new output y is obtained and sent back to the sensor to calculate new error value e. The essential part of the feedback system is the proportional-integral-derivative (PID) controller, which is responsible for calculating the desired response to error by applying a proportional gain K_p (reduces rise time), integral gain K_i (eliminates steady-state error) and derivative gain K_d (reduces overshoot):

$$u(t) = K_p e + K_i \int e \, dt + K_d \frac{de}{dt} \tag{2.1}$$

The Simulink model in Figure 2.6 uses two transfer function blocks from the collection of continuous blocks of the standard library to model the controller (*PI Controller*) and the plant. The *Step* block is a source block that generates a step input signal, while the *Scope* block is a sink block that displays the input signal on a plot. The *Sum* and *Gain* are math operation blocks that respectively add two input signals and multiply a signal by a coefficient. When modelling and simulation is completed a corresponding *Scope* block can be opened to display the plot of input signals, such as in Figure 2.7.

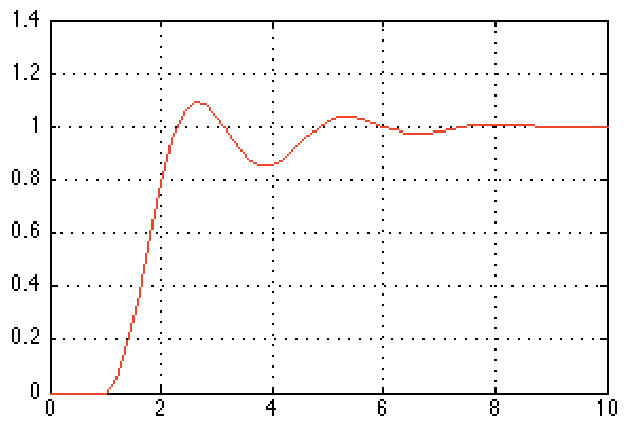


Figure 2.7: PID-controlled signal step from 0 to 1 at time = 1s

All blocks used in the model must be connected by directed lines that define how signals (either scalar or vector) are transmitted. A line can also split the signal, as we have seen in the feedback model. Simulink provides a few general classes of blocks:

- Sources – generate various signals
- Sinks – output or display signals
- Discrete – linear, discrete-time system elements (transfer functions, state-space models, etc.)
- Continuous – linear, continuous-time system elements and connections (summing junctions, gains, etc.)
- Nonlinear – nonlinear operators (arbitrary functions, saturation, delay, etc.)
- Connections – multiplex, demultiplex, system macros, etc.

Additional blocks can be introduced to Simulink either from MATLAB functions, C-code S-functions (system functions) or using the hierarchical modelling capability by grouping a set of linked blocks into a subsystem. Another option is to use a Stateflow block, which can be either *Truth Table* or *Chart*. The last approach is particularly useful when modelling the control logic for hybrid systems, as it is based on a variant of finite state machine notation established by Harel [63], where states and transitions form the basic building blocks of the system. Stateflow charts can represent hierarchy (states having nested states), parallelism (orthogonal states that can be active at the same time) and history (destination of transition depends on historical information). Additionally, Stateflow allows to represent flow graphs (stateless charts). In general, a Stateflow chart is used to specify the discrete controller, while the continuous dynamics of the plant/environment is specified in Simulink [124].

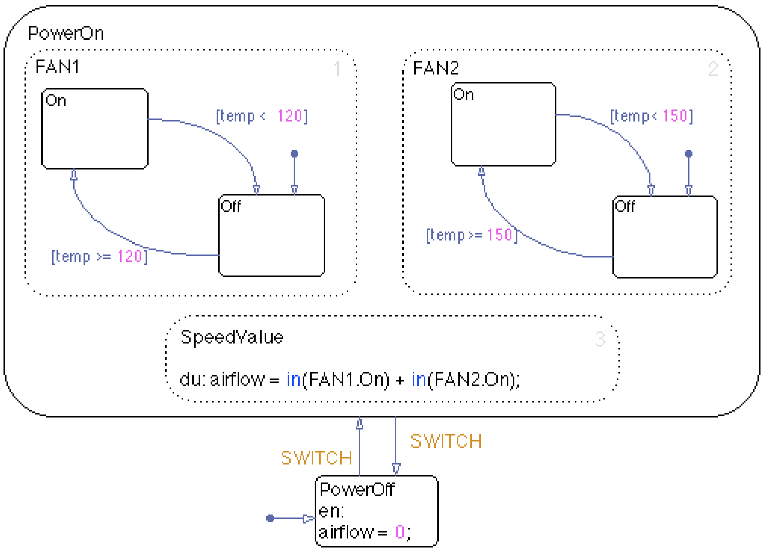


Figure 2.8: Stateflow of air controller

The example state chart in Figure 2.8 is modelling an air control system, consisting of two similar fans that can be switched on/off independently. The system can be switched on/off itself, which is described by the *SWITCH* event that triggers the transitions between two system states: *PowerOff* and *PowerOn*. The latter state contains three parallel states: one for each fan (*FAN1* and *FAN2*) and a helper state *SpeedValue* that calculates the amount of air flow, which equals to the number of working fans. Airflow signal is modified in the chart using two types of state actions: *du* (during) and *en* (entry). Other types supported by Stateflow include *exit*, *bind* and *on_event*.

2.2.3 Modelica

Modelica is a freely available equation-based object-oriented language for modelling and simulation of complex heterogeneous physical systems that may involve subsystems from multiple domains, including mechanical, electrical, hydraulic, thermodynamic, control and process-oriented applications. The language was designed to allow tools to automatically generate efficient simulation code with the main objective to facilitate exchange of models, model libraries and simulation specifications [59].

Models in Modelica are mathematically described by differential, algebraic and discrete equations. Interaction between models is formalised in terms of connection equations without any specification on causality, i.e. relationship between inputs and outputs[11]. This enables high reusability and readability of declarative (acausal) models, as opposed to context-sensitive procedural approach where causality is fixed (e.g. Simulink). For example, the same equation of a resistor $R \times i = v$ can be used in three ways:

[11]The term causality is used here in a different context from the standard definition of causality accepted in control systems, where it signifies the dependency of output $y(t_0)$ on the values of input $x(t)$ for $t \leq t_0$ [112].

`v := R * i; i := R / v; R := v / i.` On the other hand equation-based models are not oriented to a solution (algorithm) and therefore require more sophisticated symbolic analysis capabilities from the simulation tool.

Every object in Modelica has a class that defines its data and behaviour. A class declaration consists of a list of component declarations and a list of equations after the *equation* keyword. For instance, a low-pass filter can be modelled by the following class[12].

```
class LowPassFilter
  parameter Real T = 1;
  Real u, y(start = 1);
equation
  T * der(y) + y = u;
end LowPassFilter;
```

Three component declarations are used in this class. The component T is prefixed with a specifier *parameter*, indicating that the value of T is constant during simulation runs. The component y is using a modification (parentheses), which allows its attribute *start* to be initialised. In the *equation* block a single equation is declared. Construct *der*() denotes the time derivative of y. After a new class is declared it can be instantiated, i.e. instances (objects) of that class can be created and used by other classes as sub-components:

```
class FiltersInSeries
  LowPassFilter F1(T=2), F2(T=3);
equation
  F1.u = sin(time);
  F2.u = F1.y;
end FiltersInSeries;
```

In the above example two instances of low-pass filter are created with different time constants T and connected by equations. The dot notation used in equations allows components to be referenced within structured components. *time* is a built-in global independent variable that denotes the current time during simulation.

For simplicity and maintainability Modelica provides the following restricted versions of the *class* keyword: *model* (dynamic model that cannot be used as a connector), *connector* (contains only interface ports, no equations), *record* (only data, no equations), *block* (fixed input-output causality), *function* (only public inputs and outputs, one algorithm and no equations), *type* (derived from built-in data types or defined records) and *package* (collection of class declarations).

[12]Modelica examples are from [9].

When created, components can be connected by means of the *connector* class, which declares variables that define a component's communication interface. For example, an electrical resistor component with two pin connectors can be modelled as follows.

```
connector Pin
  Voltage v;
  flow Current i;
end Pin;

model Resistor
  Pin p, n; // "positive" and "negative" pins
  parameter Real R(unit="Ohm");
equation
  R*p.i = p.v - n.v;
  p.i + n.i = 0; // positive currents into component
end Resistor;
```

The *Pin* connector defines an interface in terms of voltage and current. Notice that its variable i is declared with a prefix *flow*, which is used for flow variables. When connectors of different components are connected by equations, non-flow (default) variables are set to equal according to Kirchhoff's first law, whereas flow variables are summed to zero (Kirchhoff's current law). Modelica provides a special operator *connect*, which allows two connectors of the same type to be coupled and generates equations based on the variable kind (flow or non-flow). For example, a simple circuit shown in Figure 2.9 can be modelled by connecting three resistor components together.

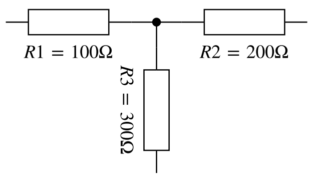


Figure 2.9: Simple electrical circuit

```
model SimpleCircuit
  Resistor R1(R=100), R2(R=200), R3(R=300);
equation
  connect(R1.p, R2.p);
  connect(R1.p, R3.p);
end SimpleCircuit;
```

The equations of the model above are equivalent to:

```
R1.p.v = R2.p.v;
R1.p.v = R3.p.v;
R1.p.i + R2.p.i + R3.p.i = 0;
```

As an object-oriented language Modelica supports encapsulation, aggregation and specialisation, although its type system is influenced by the type theory [1], which separates subclassing (inheritance) from subtyping (structural relationship for type compatibility). Inheritance is used in Modelica as a structuring concept and not for classification or type checking, i.e. a class can be defined as a subtype of another class not only by using an *extends* clause, but also by including all the public components of that class [10]. Additionally, reusable abstract classes can be created using the *partial* specifier. For instance, we might want to have a generic electrical component with two pins, which can be used as a base class for creating a resistor or inductor. This can be achieved as follows.

```
partial model OnePort
  Pin p, n;
  Voltage v "Voltage drop";
equation
  v = p.v - n.v;
  p.i + n.i = 0;
end TwoPin;

model Inductor extends OnePort;
  parameter Real L(unit="H"); // inductance
equation
  L*der(i) = v;
end Inductor;
```

Apart from acausal modelling based on equations Modelica provides the means to model causal systems, such as digital controllers, which can be represented more naturally as algorithms that consist of ordered assignment statements, branches and loops. A Modelica *algorithm* block is designed to hold only assignment statements, denoted by the assignment operator :=, as well as if-then-else expressions and loops.

Finally, Modelica supports the modelling of hybrid systems through unification and concurrent execution of continuous and discrete models. For the discrete part it uses the principle of the synchronous data flow, which states that computation and communication at an event instant does not take time unless the duration is modelled explicitly [58]. The continuous part is described by differential-algebraic equations that follow the single assignment rule, stating that the total number of equations must be equal to the number of unknown variables [115]. Discontinuous dynamics can be modelled by if-then-else expression in equation statements, such as in the following equation of a limiter:

```
y = if u > HighLimit then HighLimit
    else if u < LowLimit then LowLimit else u;
```

This approach can also be applied to model conditional components, whose equations depend on a component's parameters. In addition, an if-then-else statement can be used to replace a set of equations with another set depending on some condition.

Discrete-time and sampled systems are modelled in Modelica by discrete state variables, whose values are changing only at specific points in time, and a *when* clause, which activates equations instantaneously when its condition becomes true. A built-in function $sample(start, interval)$ can be used as a condition of a *when* clause to trigger it when $time = start + n \times interval, n \geq 0$, which is particularly useful for modelling sampling. In equations an operator *pre* is used to denote the left limit of a discrete state variable. An operator $reinit(state, value)$ can be used on a continuous state variable to reassign its value at an event defined by *when* clause. Some of these constructs can be demonstrated in a hybrid model of a classical example of the bouncing ball that involves both continuous motion of the ball and discrete changes in velocity at bounce times:

```
model BouncingBall
  parameter Real g=9.81;
  parameter Real c=0.90; // elasticity constant of ball
  Real height(start=0); // height above ground
  Real v(start=10); // velocity
equation
  der(height) = v;
  der(v) = -g;
  when height<0 then
    reinit(v, -c*v);
  end when;
end BouncingBall;
```

A lot of predefined Modelica components for multiple domains are available as part of the Modelica Standard Library that comes with the language. Besides the textual representation Modelica provides a visual component modelling capability, which can be implemented by graphical editors in modelling/simulation tools. The simulation process itself involves a number of steps, including model translation, analysis, sorting and optimisation of equations, C code generation and compilation to an executable simulation. A number of Modelica compilers are available, including the free OpenModelica Compiler (OMC), provided by the open-source OpenModelica environment[13].

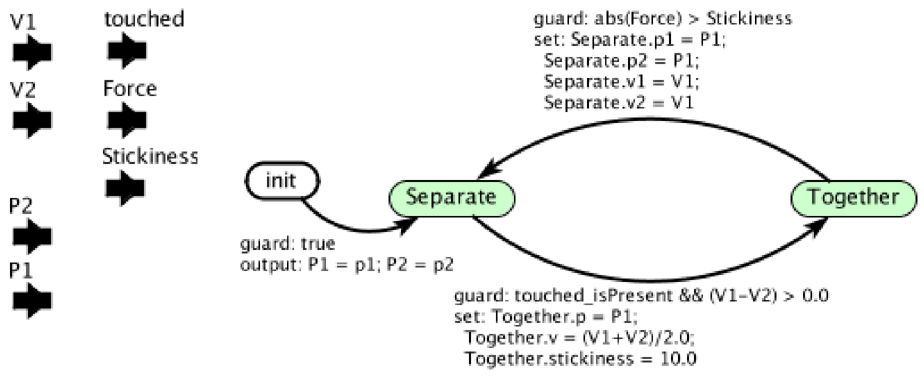
[13] https://www.openmodelica.org

2.2.4 Ptolemy

Ptolemy II is an open-source framework for modelling and simulation of distributed hybrid systems [39]. The framework is developed within the Ptolemy project that studies the design of concurrent, real-time, embedded systems based on the concept of heterogeneous hierarchical assembly of concurrent components and the use of well-known *models of computation* (MoC) that govern their interaction. The models are constructed as a set of interacting components, where the semantics of the interaction is defined by the chosen MoC. The framework has implementations of a number of MoCs including Synchronous Data Flow, Kahn Process Networks, Discrete Event, Continuous Time, Synchronous/Reactive and Modal Models.

The basic element in Ptolemy II is an *actor*, which is a software component that executes concurrently and communicates with other actors using message passing via interconnected ports [96]. Multiple actors and their connections can be aggregated into a *composite actor* that can have its own ports, linked to the ports of nested actors. A hierarchical interconnection of multiple actors constitutes a model.

The semantics of the actor is deliberately incomplete to handle multiple heterogeneous models, required by complex systems. This *abstract semantics* reduces interactions between diverse models to a minimum that achieves a well-defined composition [93]. The concrete semantics of a model is determined in Ptolemy II by a software component called a *director*, which implements a particular MoC that defines the communication mechanism and the execution order of the actors under its control. Each level of the hierarchy in the model can have a distinct director, which allows heterogeneous models to coexist in one system. The emphasis of the Ptolemy project is to explore such combinations of models of computation implemented by various directors. For instance, a hierarchical combination of continuous-time (CT) models with finite state machines (FSM) yields hybrid systems, such as the example system of two sticky point masses from the Ptolemy II demo models, displayed in Figure 2.10.



```
guard: abs(Force) > Stickiness
set: Separate.p1 = P1;
  Separate.p2 = P1;
  Separate.v1 = V1;
  Separate.v2 = V1
```

```
guard: true
output: P1 = p1; P2 = p2
```

```
guard: touched_isPresent && (V1-V2) > 0.0
set: Together.p = P1;
  Together.v = (V1+V2)/2.0;
  Together.stickiness = 10.0
```

Figure 2.10: Hybrid model of the two sticky point masses in Ptolemy II (screenshot from the Ptolemy II environment)

This example models two sticky point masses on springs, oscillating on a flat frictionless table. When the two masses collide, they stick to each other and continue oscillating together. The stickiness decays with time (defined by a stickiness decay rate parameter) and eventually the masses come apart again. This is an example of a modal model, where the system has two modes of operation: *Together* and *Separate*, corresponding to whether the masses are stuck together. Modes are modelled by states in an FSM, governed by a discrete-event (DE) director at the top level (not shown in the figure). Input/output ports are used for data signals: the velocities of the two masses *V1* and *V2*, their positions *P1* and *P2*, *Force*, *Stickiness* and *touched* event indicator. Each state is refined by a continuous-time model of the dynamics in that mode. For instance, the *Together* mode is refined to the following CT model:

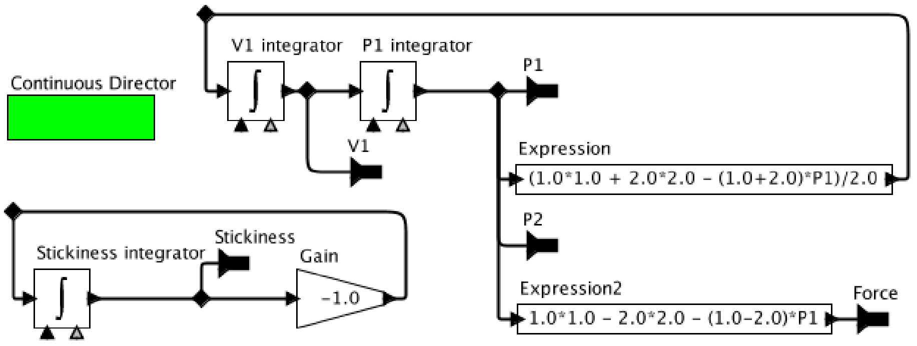


Figure 2.11: Continuous model of *Together* mode from the FSM

Notice that continuous model is governed by a continuous director (CD). The model has an ordinary differential equation for the joined point masses attached to a spring. The second differential equation models the decaying stickiness.

The resulting model can be easily simulated in Ptolemy II. There are built-in signal sink actors for plotting timed signals. Figure 2.12 shows the results of the simulation of the sticky masses model, plotted by the *TimedPlotter* sink actor.

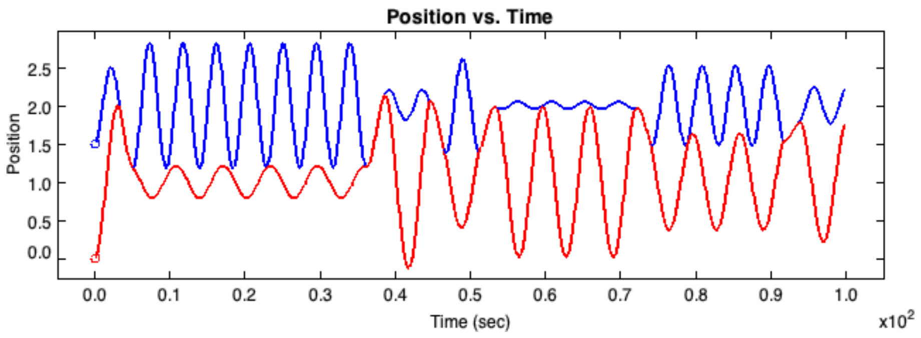


Figure 2.12: Simulation results of the two sticky point masses model

As mentioned earlier, the actual execution semantics of an actor is determined by the associated director, which implements a particular MoC. The execution of an executable actor comprises three main phases:

- *Setup* phase, divided into two steps: *preinitialise* and *initialise*. The preinitialise step is performed only once at the very beginning of the execution and relates to a static analysis of the actor's actions. It is followed by the initialise step, which initialises parameters, resets local state and sends initial messages to output ports.
- *Iterate* phase or sequence of iterations (atomic executions of the actor), divided into three steps: *prefire*, *fire* and *postfire*. The prefire step checks the execution preconditions. The fire step performs the computation of the actor by reading the data from the input ports, processing it and producing the output data on the output ports. The postfire step updates the state, allowing a number of iterations to be executed before a steady state is reached.
- *Wrapup* phase that ensures the execution is properly terminated.

2.3 Co-Simulation Frameworks

2.3.1 Functional Mock-up Interface

FMI is an industrial level tool-independent standard, initially designed by Daimler AG and as of today developed through the participation of 16 companies and research institutes[14]. The standard defines cross-platform API for the exchange and simulation of dynamic models and comes as a set of C header files to be implemented by each individual model, and a `modelDescription.xml` file schema for describing model-specific properties and state variables [24]. The implemented code must be compiled into a dynamic/shared library for the target platform and bundled together with the model description file into a Functional Mock-up Unit (FMU), which is essentially a `.zip` archive that can be used for modelling and simulation.

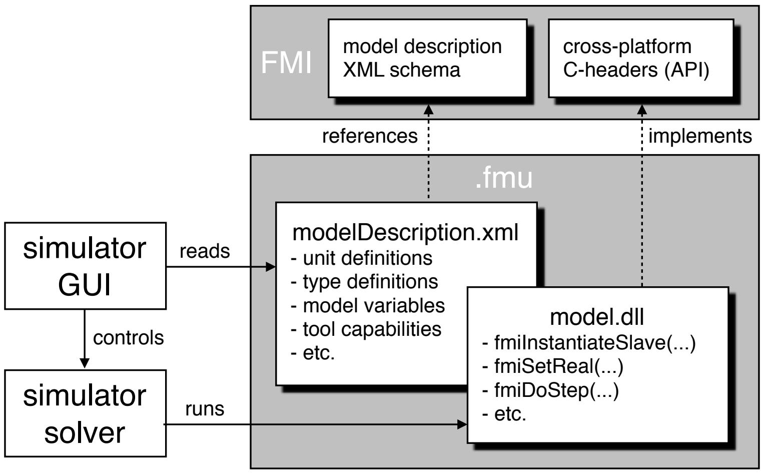


Figure 2.13: The structure of the FMU and its application

[14]As reviewed on 25th August 2015 at https://www.fmi-standard.org/development

The standard is divided into two parts: *Model Exchange*, for generating C-code of a dynamic system model in the form of an input/output block, which can be utilised by other environments; and *Co-Simulation*, for coupling simulators and subsystem models, exported from different environments, in a co-simulation. The key difference between two parts is in the absence of the generated solver in the Model Exchange – dynamic models are described by differential, algebraic or discrete equations that require a solver from the destination environment. The Co-Simulation, on the other hand, is designed for generating self-sufficient FMUs and coupling them with other models in a co-simulation environment, where each subsystem is solved independently by its individual solver and the data exchange happens only at discrete communication points. The latter aspect raises the requirement for a coordination mechanism to run the co-simulation of composed models. The FMI for Co-Simulation calls such a mechanism *Master*, whereas co-simulated models are called *Slaves*. The main goal of the master is to control the data exchange between composed slaves and synchronise their simulation solvers.

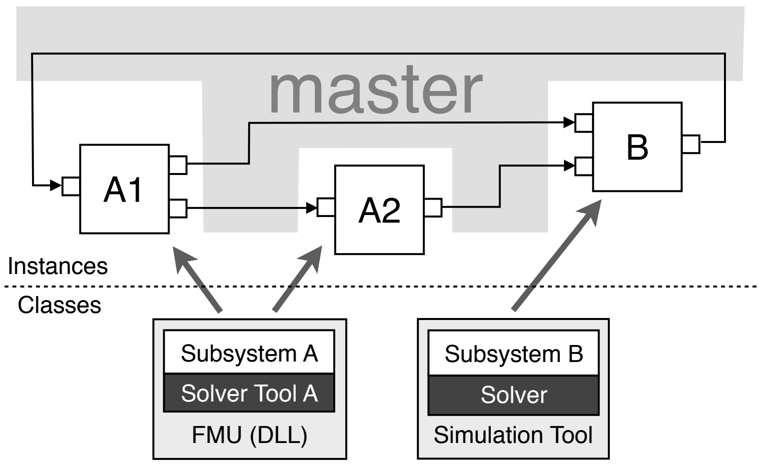


Figure 2.14: FMI master-slave architecture

As seen in Figure 2.14, a slave can represent either an exported DLL (*Subsystem A* in the figure) or a coupled simulation tool that simulates some model (*Subsystem B*), and may be instantiated multiple times, such as *A1* and *A2*. A coupled subsystem can be either continuous in time (described by differential equations) or discrete (difference equations), and can be represented as a block with inputs, outputs and state (internal) variables. Subsystem variables, their causality (input, output, internal) and type (Real, Integer, Boolean, String), along with the information about the model, solver and simulation capabilities are described in the slave-specific description XML-file.

The physical connections between subsystems are represented by mathematical coupling conditions between their inputs and outputs [79]. This information can be effectively encoded in a component connection graph, as in Figure 2.14, and used for the data exchange aspect of the FMI for Co-Simulation. The synchronisation of the simulation from time $tc_0 = t_{start}$ to $tc_N = t_{stop}$ in communication steps $tc_i \rightarrow tc_{i+1}$ is the responsibility of the master's algorithmic component. The FMI for Co-Simulation supports not only fixed-step master algorithms, but also more sophisticated approaches that adapt the step size to the solution behaviour, use higher order signal extrapolation to approximate subsystem inputs, or handle simulation steps sequentially such that the intermediate results from the first subsystems may be used to improve the approximation of subsystem inputs in later stages of the communication step [107, 16, 132]. The standard is designed to support a very general class of algorithms and gives the guidelines on the basic implementation. However, it does not define the master algorithm itself.

2.3.2 High Level Architecture

HLA is a general specification for reuse and interoperability of simulations, first developed by the U.S. Department of Defence [50]. The specification is designed to overcome a common "one size fits all" problem with simulation, i.e. it is practically infeasible to implement a single simulation for all applications. The solution from the HLA perspective is to make specific simulations reusable and (re)composable for each particular application under a so called *federation*, thus reducing the time and cost required to create a synthetic environment for a new purpose [49]. In this regard HLA provides a modular design of simulation components, with a well-defined and separated functionality and interface.

The key principe of federation in HLA is based on the Service Bus topology (Figure 2.15), where each simulation system called a *federate* has a single connection to the service bus called the *Runtime Infrastructure* (RTI) and provides/consumes a set of services it is interested in. This enables flexible replacement of one system for another without affecting other systems, and a scalable addition of new systems and capabilities [108].

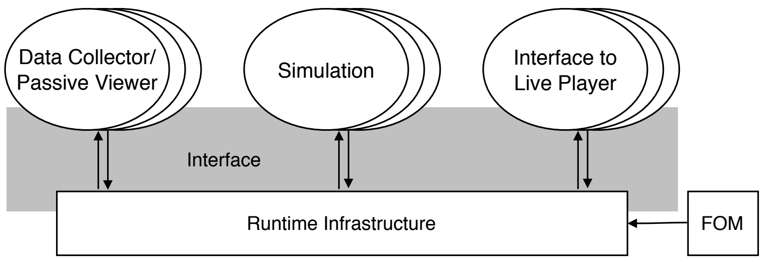


Figure 2.15: Service bus topology of the HLA

The main functional components of the HLA are:

- Federates, which are computer simulations, utilities (simulation data collectors and monitors of simulation activities) or interfaces to live players – instrumented platforms and C2 (command and control) systems that interact with the real world. HLA requires that all federates incorporate a set of agreed capabilities to be able to interact with objects from other federates via the RTI.

- RTI, which is a software component that provides a set of services and functions for the federation of simulations. Services comprise federation management, declaration management, object-level operations, dynamic transfer of attribute/object ownership, synchronisation of the simulation and data routing among federates.
- Runtime interface, an interface between the RTI and federates that is independent of the object model and data exchange requirements.

The standard is defined by the IEEE specification documents: the Federate Interface Specification [68], which describes the services provided by the RTI and federates, the Framework and Rules Specification [69] that summarises the key principles and rules of the federates and their interacting simulations (federations), and the Object Model Template Specification [70] that documents the essential sharable objects in terms of the Federation Object Model (FOM) – a description of the data exchange, and the Simulation Object Model (SOM) – an information on the capabilities of the simulation. The services of the HLA are provided as a C++ and Java API.

2.3.3 DESTECS

The focus of the DESTECS project is on the multidisciplinary collaborative modelling and simulation for the early-stage design space exploration [55]. Within the project a namesake open tools platform has been developed to support co-modelling and co-simulation of discrete and continuous system aspects, with the explicit modelling of faults and fault-tolerance mechanisms. The platform uses continuous-time (CT) models expressed as differential equations in the Bond Graph notation [74], supported by the simulation tool 20-sim [34], and discrete event (DE) models in the VDM, supported by Crescendo [84].

The architecture of the platform is based on a concept of co-model that consist of two component models, one describing a DE computing subsystem and one – a CT environment or plant[15], and a contract that identifies shared design parameters, variables and common events used in the communication between DE and CT subsystems. The fault modelling is achieved by extending a co-model to include a faulty behaviour, which can be triggered during the simulation via a script.

The formal language of the Crescendo/DE part is VDM-RT, which is an extension of VDM++ (see Section 2.1.5) with timing and models of deployment to virtual networked CPUs, so there is at least a theoretical basis for modelling distributed control [143]. The modelling language of the 20-sim tool/CT is the SIDOPS+ [33], which supports model description at three levels of abstraction:

[15]The limitation of the platform is that only one DE and one CT model can be co-simulated. However, the advantage of such limitation is a performance gain for simulation, a simplified maintenance and arguably a better understanding of a co-simulation since only one interaction between the simulators exists [85].

- *Technical component level*, where models are represented by component graphs resembling the actual system of networked devices.
- *Physical concept level*, where physical processes of the devices are modelled using one of the supported representations: bond graphs, block diagrams or ideal physical models (IPMs).
- *Mathematical level*, which captures the quantitative phenomena of the physical processes, either using a set of (acausal) equations or computer code that calculates the output from the input.

The other key features of the SIDOPS+ are: polymorphic modelling that separates the model's *type* (interface) from its *specification* (implementation of the internal behaviour), thus allowing a single model type to be *realised* by different specifications depending on the required behaviour and domain; combined (discrete-continuous) systems modelling through the use of *discrete* variables, linked to the continuous variables via *sample* and *hold* primitives and used in the sequential *statements* (as opposed to the continuous equations); reusability of the models with a compatibility validation mechanism that checks physical type consistency and constraints on the values of variables/parameters; openness of the language to any tool and domain, which is achieved by the tool-independent design of the SIDOPS+.

A promising continuation of the DESTECS project is the INTO-CPS initiative [13], started in 2015. It investigates the development of an integrated tool chain for comprehensive model-based design of cyber-physical systems, and has similar goals to our approach. The project also explores integrated application of the formal modelling (VDM) and physical simulation (Modelica) with the aid of the FMI for co-simulation standard.

2.3.4 Framework Comparison

In order to identify a suitable technology for the co-simulation of discrete and continuous models within our developed tool we have performed a comparison of the reviewed co-simulation frameworks based on the key criteria of tool independence, cross-platform capabilities, the range and popularity of existing tools supporting the technology, availability and openness of the standard, completeness of the documentation and the ease of implementation. An additional important factor regarding the tools is the support of the Modelica language as our language of choice for the physical modelling. Currently it is directly supported only by some of the modelling environments that also support FMI. The comparison has also shown that the FMI standard displays the best balance between the technical capabilities, ease of adoption and compatibility with existing tools. Therefore, our development is based on the FMI standard, more specifically – FMI 1.0, since no full support of the FMI 2.0 was available at the time of implementation of this work. Table 2.1 summarises the compared criteria of each technology.

Table 2.1: Comparison of the co-simulation frameworks

Framework	FMI	HLA	DESTECS
Tool independent	Yes	Yes	No (Crescendo and 20-sim)
Cross-platform	Yes	Yes	No (Windows only)
Number of tools	73[16]	N/A	2 (Crescendo/VDM-RT and 20-sim)
Availability and openness of the standard	Free and open-source	Requires IEEE subscription	Combination of open-source (Crescendo) and commercial (20-sim)
Availability of the documentation	Freely available single specification document with code examples, C header files and a compliance checker	Three specification and two recommended practices documents from the IEEE, free tutorial, C++ and Java API and FOM/SOM samples	Freely available tool-suite manual, examples and a wiki
Ease of implementation	Two platform-independent C header files, open-source implementation in C and Java	Complex	Tied to the DESTECS environment

2.4 Summary

All the listed technologies provide a valuable contribution to the evolutionary development of CPS. State-based formalisms like Event-B and VDM and continuous dynamics simulation tools like Simulink and Modelica are both useful in the design of hybrid systems. However, at the moment they have evident limitations, namely, not fully supported continuous and temporal semantics in the discrete formal methods and the absence of refinement and rigorous analysis capabilities in the simulation tools. Clearly, if combined, they could reduce each other's weaknesses. We think such an integration is achievable not only as an interface solution, but as a methodology and development framework. This requires a formal approach and a unified formalism that combines two

[16]According to the official information from the FMI standard as of 25th August 2015, available at https://www.fmi-standard.org/tools

domains, as the systems we are looking at can be highly complex, real-time and safety-critical. In this view well-established formalisms of hybrid automata and statecharts, supported by the DSL abstractions, can act as a binding material. A good starting point for such an integration is in the development of interfacing tools between discrete and continuous components that are based on the open and widely accepted standards.

2.5 Working Plan

The following chapters describe in detail the concept of our integrated development approach for hybrid systems, its design, implementation and evaluation. We use Event-B method and Rodin platform, Modelica language and FMI for Co-simulation standard as the core technologies of our approach. The working plan of this development comprises the following steps:

1. To establish the architecture of our integrated development approach based on the co-modelling and co-simulation between Event-B and Modelica; the design must support our intended workflow, outlined in Section 1.5.
2. To define the executional semantics of the continuous and discrete co-models that are involved in the development process.
3. To define the semantics of refinement and the compositional semantics of co-simulation of the composed discrete-continuous model.
4. To design and implement a co-simulation master algorithm in accordance with the FMI specification.
5. To implement a tool that supports our workflow, i.e. co-modelling and co-simulation of the step-wise formal modelling in Event-B and physical modelling in Modelica.
6. To validate the developed tool on the case studies representative of the hybrid system domain and compare the obtained results with traditional modelling and validation techniques, drawing the conclusions about the benefits and limitations of our approach.

Chapter 3

Co-simulation

In this chapter we introduce our concept of co-simulation of discrete and continuous models within the Rodin toolset. We first give an overview of our approach and provide a formal description of the simulation step for each type of the simulated component. We then present a generic simulation interface and define its mapping to the formal semantics of two types of components. The chapter concludes with the design of the simulation master algorithm.

3.1 Overview

Cyber-physical and hybrid systems involve multiple components that are typically described in terms of time-continuous differential and algebraic, and discrete-event equations. These components often cannot be modelled and simulated in a single tool, or an individual component is best modelled in a domain specific tool. In these scenarios coupling of different simulators is required. For the cyber-physical systems it is also necessary to couple the simulation with a model of environment. Such a coupling generally means a tailored solution for specific components of the system and therefore requires high development effort.

Co-simulation is an approach for the joint simulation of sub-models of the developed system that are modelled in different tools. Each sub-model is simulated by an individual tool and the intermediate results are exchanged between the tools during simulation at discrete communication points, where within these points each sub-model is numerically solved by its simulator [16]. Coupling of simulators in a co-simulation is performed by an algorithm that takes into account the capabilities of each tool, such as event handling, support of the variable step size and backtracking (rejection of steps). These algorithms differ in the sequence of time integration and data exchange. Classical examples of a numerical model coupling include the Jacobi method (sub-systems are solved in parallel) and Gauss-Seidel method (sub-systems are solved sequentially).

3.2 Concept

Our system development approach relies on the application of the Event-B formal method to modelling of discrete-event control components of hybrid systems and the general class of physical modelling tools for modelling the continuous environment. Hence, it requires a co-simulation between Event-B and physical model simulators. In order to enable this co-simulation we employ the FMI for Co-Simulation standard. The advantage of the FMI standard over proprietary solutions, which are typically tailored to a limited number of simulators and use algorithms that are strongly coupled to a proprietary interface, is that it allows the use of different coupling algorithms within the same interface and does not impose a specific algorithm (the algorithm is not included in the standard) [116]. In addition, FMI has the industrial support [25], integration with well-known physical simulation tools [110, 118], cross-platform capability and availability of the open-source implementations [119, 146]. As explained in Section 2.3.1, the FMI standard is built upon the Master-Slave architecture, where the master is responsible for data exchange and synchronisation of the simulation of all coupled slaves, also called components. We reflect the logic of the FMI master in our simulator and adopt the FMI interface in our co-simulation architecture.

Our concept of generic simulator is designed based on a master algorithm example from the FMI specification document [107]. The evolution of the simulation according to the FMI standard is divided into simulation steps of certain size. The boundaries of these steps are called synchronisation or communication points, at which the master communicates with the slaves and transfers data between them. The data is exchanged between the slaves via connected input and output ports that each slave may have. The simulation step is defined within FMI as a request to a slave to carry out the simulation at a certain time, e.g. a synchronisation point, for an interval that equals the step size. This request is specified in FMI by a function prototype `fmiDoStep(...)`, which takes as an argument an FMI component, current communication point and communication step size. The communication step is normally preceded by a call to `fmiGetXXX(...)` function to retrieve the value of a specific output port, and a corresponding call to `fmiSetXXX(...)` of another slave to set that value of an input port. We combine all operations on an individual slave within a single simulation cycle into a semantic *Step*.

3.3 Formalisation of Co-simulation

Our simulation operates on two types of slaves: continuous FMI models (FMUs) and discrete Event-B machines. Hence, we categorise the Step semantics of all slaves into two groups: continuous and discrete. We refer to the continuous step of FMI models as a *CStep*, to distinguish it from the discrete step of Event-B machines. The semantics of the *CStep* is dependent on the actual implementation of an FMU, but in general it denotes the evolution of a model, specified in terms of continuous equations, over a time period equal to the step size and produces a solution to either a single set of continuous equations or multiple sets of equations applicable to different internal states. The state of a continuous slave can be defined as a time function:

$$F\colon Time \to V \tag{3.1}$$

where $Time$ is the global simulation time and V is a state of the slave's internal variables. Hence, the evolution of continuous slaves over time can be represented on a graph:

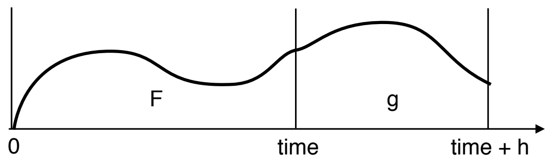


Figure 3.1: The state of a continuous slave over time

where g is a continuous state function defined over interval $[time, time + h]$. The master synchronises simulation at fixed communication points when data is exchanged between components prior to the next simulation step. If the values $time$ and $time + h$ are assumed to be communication points and the value h equals the communication step size, the simulation semantics of a continuous slave at the abstract level can be formally defined using Event-B notation[1] as follows:

```
**machine** $C$
**variables** $F, time$
**event** $CStep$
  **any** $i, o, g, h$
  **where**
    $g \in [time \ldots time + h] \to V$
    $g(time) = F(time)$
    $P(g, i, time, h)$
    $i = g(time)$
    $o = g(time + h)$
  **then**
    $time := time + h$
    $F := F \cup g$
  **end**
```

where parameter i represents slave inputs whose values are defined by other components, o represents slave outputs that depend on the model state at the end of a simulation step and P defines model properties, or properties that g must satisfy given a particular input,

[1]Because we already use Event-B in our modelling approach, for simplicity we formalise the step semantics using Event-B notation.

time and step size. The model of an FMU slave depends on time and therefore is based on the continuous function F, defined over real-valued sets. More on the development of reals and continuous functions in Event-B can be found in [43].

The above specification of *CStep* represents continuous evolution of a simulated submodel at the abstract level. This evolution includes reading inputs, evolving the continuous state and generating outputs. Since continuous (FMI) components and the simulation master execute these operations independently, we refine *CStep* to events that are associated with individual FMI function calls. Because of one to one mapping of events to FMI function calls, this is a trivial refinement step that can be represented graphically using the abstraction of Event Refinement Structures (ERS) – a notation proposed by Fathabadi [53]. The ERS notation gives the means to specify refinement relationships and control ordering between events at different refinement levels.

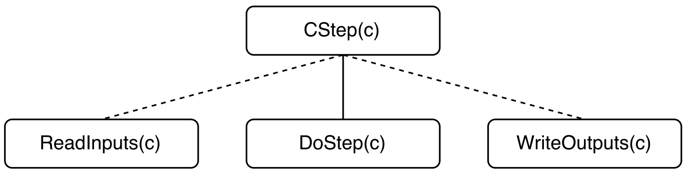


Figure 3.2: Event Refinement Structure of the continuous simulation step semantics

The ERS diagram in Figure 3.2 shows the relationships between the abstract event *CStep*, denoted by the parent node, and concrete events (leaf nodes) that are introduced in the refinement. The *DoStep* event is a direct refinement of *CStep*, which is indicated by the solid connection line (part of the ERS notation). It models the evolution of the continuous component's state over a given time interval. The parameter c of events denotes a component instance, since the same model can be associated with multiple components, i.e. instantiated and simulated by multiple slaves. Events *ReadInputs* and *WriteOutputs* are new events (indicated by the dashed line) that refine *skip*, which is an implicit Event-B event that is always enabled and does nothing. *ReadInputs* sets the values of all input ports to the component's input variables. *WriteOutputs* gets the values of all component's output variables and assigns them to the output ports. Besides the refinement structure the ERS diagram explicitly shows the sequential control flow from left to right among refinement events, hence *ReadInputs* is executed first, followed by *DoStep* and concluded by *WriteOutputs*.

The discrete simulation step of Event-B machines that we refer to as *DStep* represents the evolution of a discrete state in accordance with Event-B semantics, which is defined by state variables modified through the execution of a finite sequence of events. We define discrete state changes as a function over states and event inputs. Therefore the discrete state does not explicitly depend on time, which enables simulation of untimed Event-B models. If required, time can be incorporated as an event parameter and modelled as a state variable in the machine. The following is the abstract formal definition of *DStep*.

```
machine $D$
variables $V$
event $DStep$
    any $i, o, v$
    where
        $i \in T$
        $P(v, i, V)$
        $o = Q(v)$
    then
        $V := v$
    end
```

Variable V represents internal state of the associated discrete model, which is independent of duration of the simulation cycle, in contrast to *CStep*. Parameter i represents the input of some abstract type T. Parameter v represents the new state after a simulation cycle, and is constrained together with the current state V and slave inputs i by a predicate P that defines the model's properties. Slave outputs are specified by the parameter o, whose value is constrained on the new state v by some predicate Q.

Since our semantics of *DStep* does not depend on time directly, we let the modeller decide on the appropriate simulation step size of Event-B components. This step size can be different for different components or instances of the same component. The mechanism of how components with different step sizes are synchronised is described in detail further in this chapter when we talk about the master algorithm.

As the individual simulation (execution) of an Event-B model is equivalent to a finite trace of its events, we interpret the discrete simulation step as a sub-trace that comprises either a single event or a sequence of multiple events. This interpretation enables simulation of the abstract Event-B machines, in which every event models a complete step. If such a machine has multiple events enabled at the time of the simulation step, according to Event-B semantics one of them is chosen non-deterministically. We call this definition an abstract case of *DStep*, which is shown as a refinement structure below.

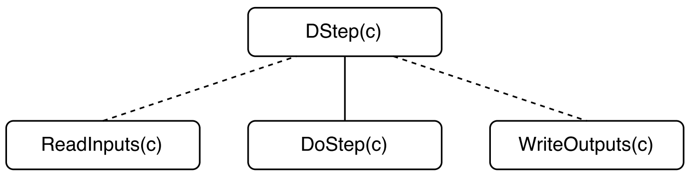


Figure 3.3: ERS diagram of the discrete step semantics of abstract Event-B machines

The event *DoStep* represents execution of one Event-B event, selected randomly from the set of all enabled events of the abstract Event-B machine, associated with the component instance c. *ReadInputs* and *WriteOutputs* are not involved in execution of Event-B events – they accordingly read/write the values from/to ports of the component. Input values (if present) can be read by the event executed in *DoStep* via input parameters, which entails that abstract machine events must be associated with component's input ports and match them by data type.

The discrete step semantics presented in Figure 3.3 enables simple refinement process, consistent with the standard Event-B refinement approach. The abstract machine events associated with *DoStep* can be decomposed (refined) to a number of events. The order of execution of refinement events within a step is not enforced by the simulation. We leave this order to be determined by the Event-B semantics: an event is executed if it is enabled (its guards evaluate to true), unless multiple events are enabled, in which case one of them is chosen non-deterministically.

To indicate the start and end of the discrete simulation step that involves multiple events we introduce two sets of machine events:

- *StartStep*, a set of start step events. An Event-B machine event is a *StartStep* event if it is the first event to be executed at the synchronisation point with another model that is coupled with the current Event-B model via input-output variables. We do not restrict the modeller to have a single *StartStep* event since different events can be enabled at different synchronisation points.
- *EndStep*, a set of end step events. An Event-B machine event is an *EndStep* event if it is the last event to be executed before the next synchronisation point. Fundamentally, an *EndStep* event is required to indicate the end of the simulation step and, if the model is timed, to produce the value of the duration of the executed step. Having an *EndStep* event is also necessary for the refinement of abstract control events, in which case it becomes the refinement of the abstract event.

The modeller is required to determine which events of the Event-B machine constitute each set. The simulation step commences with the execution of one event $e \in \mathit{StartStep}$, followed by a finite sequence of intermediate events $e \notin \mathit{StartStep} \cup \mathit{EndStep}$ and concluded with one event $e \in \mathit{EndStep}$. Consistency of this semantics with the abstract case of *DStep* is ensured by allowing events satisfying $\exists\, e \cdot e \in \mathit{StartStep} \cap \mathit{EndStep}$, in which case we impose a rational constraint that there are no intermediate events and the event e is executed only once within a step. This semantics of *DStep* is illustrated in Figure 3.4 as a refinement structure of the abstract *DStep* from Figure 3.3.

The *DoStep* event is refined to a sequential execution of one *StartStep* event, zero or more *Intermediate* events (represented by a star node called *loop constructor* in ERS) and one *EndStep* event on a component instance c. The *StartStep* event denotes nondeterministic execution of one enabled Event-B event from the previously defined *StartStep* set, while the *EndStep* event denotes a similar semantics with respect to the *EndStep* set, unless the same event is in the *StartStep* set and therefore not executed again. The Intermediate

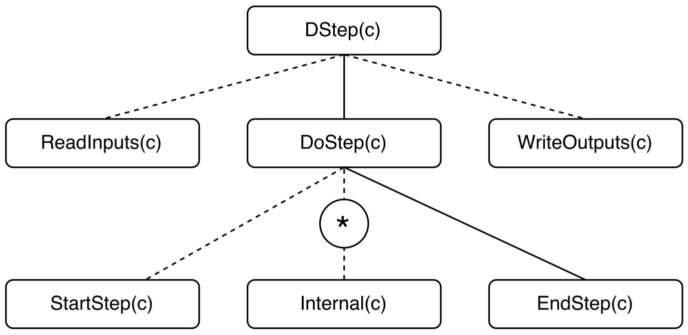


Figure 3.4: ERS diagram of the discrete step semantics of concrete Event-B machines

event denotes nondeterministic execution of any enabled Event-B event $e \notin \mathit{StartStep} \cup \mathit{EndStep}$. As in the abstract case of *DStep*, input values from the component's input ports are read at the beginning of the *DoStep* operation – in this case by Event-B events from *StartStep*, hence input parameters of all events in the *StartStep* set must be associated with input ports of the component and should match those by data type. This concludes our formal definition of the discrete simulation step semantics of Event-B components.

To summarise, the key aspects of Event-B components for co-simulation are:

- Event-B machines can be simulated without explicitly modelling time in Event-B. In a scenario where simulation time is required for the logic of the model, it can be provided via input parameters of the executed simulation step events.
- The modeller decides on the duration of the simulation step for each Event-B component. Different components or instances of the same component can have different simulation step sizes. Simulation step size involves the modeller and is not the same as the model sampling size.
- The modeller defines *StartStep* and *EndStep* sets of machine events to indicate start and end of simulation steps, accordingly. Both sets can have multiple events enabled upon simulation, in which case one of them is selected non-deterministically. The sequence of executed internal events (within the step) allows for nondeterminism in line with the Event-B semantics.
- Components that expect input signals from other components, either discrete or continuous, must have their input ports associated by the modeller with the input parameters of events in StartStep, as those events are responsible for reading the values of input ports upon execution.
- Co-simulation supports refinement of Event-B components. An abstract machine can have events that model full simulation step. Such an event needs to be specified by the modeller as both *StartStep* and *EndStep* event. In the refinement of the

abstract machine the latter event can be refined into multiple StartStep events, intermediate events and EndStep events. Both *StartStep* and *EndStep* events, as well as intermediate events can be refined further in the next refinement step. This flexibility of indicating multiple and/or same events as *StartStep* and *EndStep* events enables the refinement of control events and, most importantly, verification and co-simulation of Event-B components from the early stage of development, which is crucial to ensuring system correctness.

3.4 Combined View of Co-Simulation

The combined semantics of the simulator that includes continuous and discrete components is presented on the ERS diagram in Figure 3.5. The simulator is denoted by the oval root node *MultiSim*. Oval nodes in ERS notation denote a grouping of events, as opposed to the rectangular nodes that designate individual events. As indicated by the loop constructor (star node) the co-simulation involves zero or more simulation *Cycles*, each of which comprises parallel execution of an abstract simulation *Step* on all of the component instances c, where C represents the set of components to be co-simulated. Component instances are synchronised at fixed time intervals depending on their specified step size. Simulation steps are followed by the smallest possible increment in simulation time, performed by the master (event *IncrementTime*). The oval node *all(c:C)* is called *replicator constructor* in ERS. It introduces a new parameter for its sub-event. The *all* constructor specifies execution of its sub-event for all instance values of its parameter. The abstract *Step* involves exclusive execution, indicated by a logical constructor *xor*, of either *CStep* or *DStep*, depending on the type of instance c, indicated by the guard of each event in square brackets.

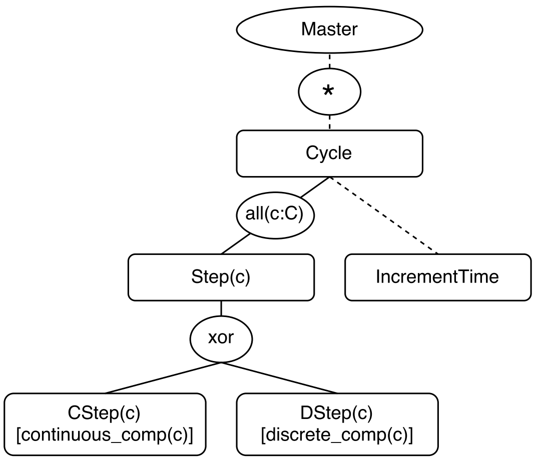


Figure 3.5: ERS diagram of the combined discrete-continuous co-simulation semantics

CStep represents evolution of continuous variables of an FMU over the step interval. *DStep* represents evolution of a discrete system, modelled in Event-B, through the execution of some finite sequence of events according to semantics we defined earlier in Figure 3.4. The *CStep* and the *DStep* evolve independently during the simulation interval. In between each interval the *CStep* and *DStep* components synchronise by exchanging information about their new state. What information is exchanged between components is determined by the modeller by connecting components via input and output ports. Simulation master algorithm is responsible for driving both the *CStep* and the *DStep* of components by executing their semantic operations (*ReadInputs*, *WriteOutputs* or *DoStep*) from the formal definition in Section 3.3 and exchanging data through connected ports on synchronisation points, which are determined by the specified simulation step size.

3.5 Example Mapping

In this section we demonstrate on the representative example of a hybrid system how models are mapped to co-simulation components based on the semantics we defined earlier. We leave out the trivial case of a direct mapping of FMI procedures to FMI components and present the mapping of an Event-B model. We also show how simulation caters for refinement.

As an example we look at the simplified hybrid model of the water tank control system shown in Figure 3.6. The model consists of the continuous environment (and plant), represented by the water tank with a valve, and the digital controller. The water tank is characterised by a constant water outflow (leakage) and a controlled water input that can be turned on or off via a discrete valve. The controller is responsible for maintaining the water level between two thresholds by controlling the opening of the valve and thus adjusting the water input to the tank. We assume that the valve has two discrete states: *open* and *closed*, and operates without delays.

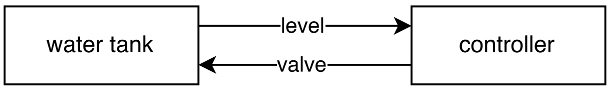


Figure 3.6: Controlled water tank model

The water tank and controller can be modelled as a continuous and discrete component, accordingly, that exchange the water *level* reading from a sensor and the *valve* control order from the controller. At the abstract level we can model the controller as an Event-B machine *TankController*, which contains three control events that operate the valve depending on the current water level reading: *CloseValve*, which closes the valve if the level is above the high threshold *HT*, *OpenValve*, which opens the valve if the level is below the low threshold *LT*, and *KeepValve*, which maintains the current state of the valve when the level is within two thresholds. Each event models a single control cycle, where the duration of the cycle is determined by the modeller.

```
machine $TankController$
variables $valve$
events
    $CloseValve \mathrel{\widehat{=}}$ any $level$ where $level > HT$
        then $valve := closed$ end                                  StartStep
    $OpenValve \mathrel{\widehat{=}}$ any $level$ where $level < LT$                  &
        then $valve := open$ end                                    EndStep
    $KeepValve \mathrel{\widehat{=}}$ any $level$ where $level \geq LT \land level \leq HT$
        then $skip$ end
end
```

The machine *TankController* represents the abstract case of *DStep*, where execution of one of the three events models the complete simulation step with no intermediate events. Using our notion of *StartStep* and *EndStep* events we can assign the simulation step semantics to the machine *TankController* by specifying each event as both the *StartStep* and *EndStep* event. This ensures that at the start of the simulation step one of these events is chosen non-deterministically by the simulator and executed once per simulation cycle. Furthermore, since events take the water level reading as an input parameter *level*, the component needs to have an input port associated with this parameter and connected to the *level* signal from the water tank component. Accordingly, an output port needs to be associated with the control variable *valve*. Associations of the ports to Event-B elements are graphically represented in Figure 3.7.

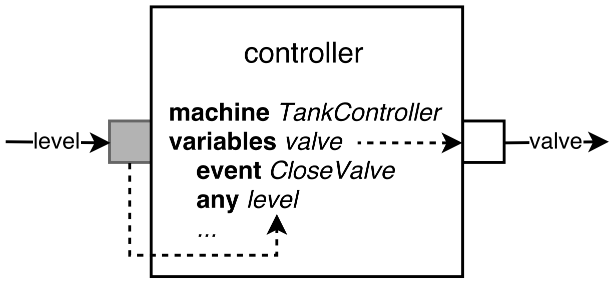


Figure 3.7: Controller component and its mapping to the Event-B machine elements

As a possible refinement of the *TankController* we can imagine modelling a state machine that describes the state of the water tank as seen by the controller. The refinement may introduce a dedicated event *ReadLevel* for sensing the controller's inputs, in this case the *level* signal, that is written to a new state variable *clevel*, representing the controller view on the water level in the tank. The state machine can be modelled by a variable *state* and new events *DecideClose*, *DecideOpen* and *DecideKeep* that represent state transitions, dependent on the value of *clevel*. The abstract events are refined to use the new *state* data to decide on the issued control order to the valve.

machine *RefinedTankController* **refines** *TankController*
variables *valve*, *clevel*, *state*
events

$$\left.\begin{array}{l} \mathit{ReadLevel} \mathrel{\widehat{=}}\ \textbf{any}\ \mathit{level}\ \textbf{where}\ \mathit{state} = 0 \\ \quad \textbf{then}\ \mathit{clevel} := \mathit{level} \parallel \mathit{state} := 1\ \textbf{end} \end{array}\right]\ \mathit{StartStep}$$

$$\begin{array}{l} \mathit{DecideOpen} \mathrel{\widehat{=}}\ \textbf{where}\ \mathit{state} = 1 \land \mathit{clevel} < \mathit{LT} \\ \quad \textbf{then}\ \mathit{state} := 2\ \textbf{end} \\ \mathit{DecideKeep} \mathrel{\widehat{=}}\ \textbf{where}\ \mathit{state} = 1 \land \mathit{clevel} \geq \mathit{LT} \land \mathit{clevel} \leq \mathit{HT} \\ \quad \textbf{then}\ \mathit{state} := 3\ \textbf{end} \\ \mathit{DecideClose} \mathrel{\widehat{=}}\ \textbf{where}\ \mathit{state} = 1 \land \mathit{clevel} > \mathit{HT} \\ \quad \textbf{then}\ \mathit{state} := 4\ \textbf{end} \end{array}$$

$$\left.\begin{array}{l} \mathit{OpenValve}\ \textbf{refines}\ \mathit{OpenValve} \mathrel{\widehat{=}} \textbf{witness}\ \mathit{level} = \mathit{clevel} \\ \quad \textbf{where}\ \mathit{state} = 2\ \textbf{then}\ \mathit{valve} := \mathit{open} \parallel \mathit{state} := 0\ \textbf{end} \\ \mathit{KeepValve}\ \textbf{refines}\ \mathit{KeepValve} \mathrel{\widehat{=}} \textbf{witness}\ \mathit{level} = \mathit{clevel} \\ \quad \textbf{where}\ \mathit{state} = 3\ \textbf{then}\ \mathit{state} := 0\ \textbf{end} \\ \mathit{CloseValve}\ \textbf{refines}\ \mathit{CloseValve} \mathrel{\widehat{=}} \textbf{witness}\ \mathit{level} = \mathit{clevel} \\ \quad \textbf{where}\ \mathit{state} = 4\ \textbf{then}\ \mathit{valve} := \mathit{closed} \parallel \mathit{state} := 0\ \textbf{end} \end{array}\right]\ \mathit{EndStep}$$

end

The refinement *RefinedTankController* decomposes atomic events of the abstract machine *TankController* into multiple events. This is covered by our concrete definition of *DStep*. The event *ReadLevel*, which is executed first within a step, becomes a StartStep event. The refined events *CloseValve*, *OpenValve* and *KeepValve* become the EndStep events. The input argument of the abstract counterparts of these events is now replaced with the *witness* construct, which links the abstract event's input argument *level* with a new state variable *clevel* via an equality predicate. New events *DecideClose*, *DecideOpen* and *DecideKeep* are intermediate events. Since *ReadLevel* is now responsible for reading the water level, parameter *rl* should be associated with the input port of the component that receives the *level* signal.

3.6 Simulation Interface

Before we specify our master algorithm that coordinates co-simulation we must devise a simulation interface, which will be used by the master to command components. Here we explain the interface and its methods. We also give a semantic interpretation of each method depending on the component type (FMI or Event-B).

Since we have identified the key events that constitute both the discrete and continuous step semantics of components (see *ReadInputs*, *WriteOutputs* and *DoStep* in Section 3.3), we derive the simulation interface based on the aforementioned events that operates on an abstract component. In the design of the interface we adopt the FMI standard. The interface consists of the following functions:

- `instantiate()` creates an instance of the simulation component. For Event-B components it is loading an Event-B machine for simulation. For FMI components it is a call of a dedicated function prototype `fmiInstantiateSlave(...)`.
- `initialise(c, start, stop)` initialises a component instance *c* with the simulation *start* and *end* time. For Event-B components it initialises the machine by executing the `initialisation` event. For FMI components it calls the FMI function `fmiInitializeSlave(...)`.
- `readInputs(c)` reads values from all connected input ports of the instance *c*. For Event-B components it constructs an input predicate of the parameter values to be passed to a *StartStep* event that reads the model's inputs. This method does not modify the state of Event-B models. For FMI components it calls the function `fmiSet(...)` for each input variable of the model. This can potentially modify system state.
- `writeOutputs(c)` write values to all output ports of the instance *c*. For Event-B components it reads the values of model variables associated with output ports. For FMI components it calls the function `fmiGet(...)` for each output variable of the model.
- `doStep(c, time, step)` executes a simulation step of the size *step* on instance *c* provided the current simulation *time*. For Event-B components it executes a finite sequence of events according to *DStep::DoStep* semantics, i.e. executes one of the *StartStep* events, followed by a nondeterministic execution of zero or more intermediate events, concluded with one of the *EndStep* events. For FMI components it calls the function `fmiDoStep(...)`.
- `terminate(c)` terminates the simulation of instance *c*. For FMI components it calls the function `fmiTerminate(...)`. Event-B components do not require termination handling. The model simulation resources are handled and freed by ProB.

3.7 Master Algorithm

Our co-simulation can involve Event-B components with a different step size. To handle different step sizes of components we adopt the two-list evaluation technique from [83], which enables evaluation of connected components based on their individual evaluation time. The idea is to have two lists: an *update list* that records the simulated components and their future evaluation time, and an *evaluation list* that stores components, which must be evaluated at current time. As the simulation time progresses the components from the update list are moved to the evaluation list. If a component issues a value to another component, the latter component is added to the evaluation list. Components from the evaluation list get evaluated (executed) and removed from the list. Future evaluations are added to the update list and the simulation time gets incremented.

Another critical aspect of the simulation is the right order of execution of the data exchange operations, such as `fmiSetXXX(...)` and `fmiGetXXX(...)`. The outputs of a model may have algebraic dependencies on the inputs, in which case it is necessary that the inputs are set before the outputs can be read and exchanged with another component. According to the FMI standard this dependency can be explicitly indicated via the `DirectDependency` attribute of a variable definition of the FMU model description. However, the standard leaves this attribute as optional (the default assumption is that each output has a dependency on all inputs). Having direct dependencies of outputs on inputs for two or more connected components can lead to unsolvable algebraic loops, as explained in [36]. For this reason we disallow direct connection of FMI components, that is, if multiple FMI components are co-simulated they are not connected directly, but only via Event-B components.

At the same time we assume no direct dependencies in Event-B components, which is justified by the fact that setting inputs of an Event-B component does not modify its state according to our semantics of *DStep::ReadInputs* (no events are executed), and the output is not updated immediately after the input is set, but at the next communication point $t_{input} + h_c$, where $h_c > 0$ is the component's step size. Since Event-B components have no direct dependencies between their inputs and outputs at the start of a cycle, they can be connected directly, and the data exchange can be performed in any order.

Before we specify the master algorithm for co-simulation we give a formal definition of the sets and functions involved:

Set of component instances	C
Set of Event-B component instances	$C_B \subseteq C$
Set of FMI component instances	$C_F \subseteq C$, where $C_B \cap C_F = \varnothing$
Set of input port variables for instance c	U_c
Set of output port variables for instance c	Y_c
Port mapping function	$P \in U \to Y$
Update list as a mutable function of evaluation time over components	$list_u \in C \to Time$
Evaluation list as a mutable set	$list_e \subseteq C$

To simplify the master we specify a separate procedure for exchanging data between components that is shown in the Algorithm 1. As an input the procedure takes a set of interconnected components that need to exchange some data, denoted by C_{dx}, a set of existing Event-B components C_B and a set of existing FMI components C_F. The input set C_{dx} is determined by the current evaluation list, as will be explained further. This procedure executes our simulation interface routines for reading inputs and writing outputs of Event-B and FMI components from C_{dx}. The order of read and write operations is defined by our assumptions on the input-output dependencies for each component type, i.e. we assume that all FMI components have direct input-output dependencies while no such dependencies exist for Event-B components. As a result, the inputs of each FMI component are read before the outputs are written.

```
Input: Set of components for data exchange $C_{dx}$, set of all Event-B components
       $C_B$, set of all FMI components $C_F$.
Result: Exchanged input/output data between components from $C_{dx}$.
foreach $c \in C_{dx} \cap C_B$ do
    writeOutputs($c$)
end
foreach $c \in C_{dx} \cap C_F$ do
    readInputs($c$)
    writeOutputs($c$)
end
foreach $c \in C_{dx} \cap C_B$ do
    readInputs($c$)
end
```

Algorithm 1: Input/Output Data Exchange

The master algorithm is shown in the Algorithm 2. It starts by instantiating and initialising each co-simulated component (lines 2-4). The update list is initialised with a proper evaluation time of each component instance, based on the simulation start time t_{start} and component's step size h_c. For Event-B components $h_c \in \mathbb{R}_{>0}$. For FMI components $h_c = 0$ because the actual step size is determined dynamically at the time of the `doStep(...)` call. After the initialisation the master performs data exchange between all components at line 6 by executing `readInputs(...)` and `writeOuputs(...)` operations according to the Algorithm 1.

The simulation loop begins at line 7, where the simulation time t_{sim} is incremented from t_{start} to t_{end}. At each time increment the update list $list_u$ is checked for components to be evaluated (line 8). A due component is added to the evaluation list $list_e$ at line 9. Besides the components from the update list all FMI components c_f that are connected to either inputs or outputs of an evaluated component c are also added to the evaluation list (lines 10-12). This is where the master synchronises FMUs that either receive discrete signals from evaluated Event-B components or produce continuous signals for Event-B components.

The actual simulation step of each component from the evaluation list is performed at line 18. Notice that the call to `doStep(...)` executes the simulation step of component $c \in list_e$ at time t_{eval} for a step size $\mid t_{sim} - t_{eval} \mid$, where t_{eval} is the last evaluation time of component c. In this rendition of the algorithm all Event-B components are untimed, hence the time and step size passed to `doStep(...)` are not being used. Consequently, we can use $list_u$ to store and retrieve the last evaluation time t_{eval} of FMI components

```
   Input: Sets of components $C, C_B, C_F$ and their step size $h_c$, component inputs
          $U$, outputs $Y$ and input/output mapping $P$, simulation start time $t_{start}$
          and end time $t_{end}$.
   Result: Component instances $c \in C$ get simulated from $t_{start}$ to $t_{end}$.
 1 foreach $c \in C$ do
 2     instantiate($c$)
 3     initialise($c, t_{start}, t_{end}$)
 4     $list_u(c) := t_{start} + h_c$
 5 end
 6 DataExchange($C$)
 7 for $t_{sim} := t_{start}$ to $t_{end}$ do
 8     foreach $c \in C$ and $list_u(c) = t_{sim}$ do
 9         $list_e := list_e \cup c$
10         foreach $c_f \in C_F$ and $list_u(c_f) \neq t_{sim}$ do
11             if $P[U_c] \cap Y_{c_f} \neq \varnothing$ or $P[U_{c_f}] \cap Y_c \neq \varnothing$ then
12                 $list_e := list_e \cup c$
13             end
14         end
15     end
16     foreach $c \in list_e$ do
17         $t_{eval} := list_u(c)$
18         doStep($c, t_{eval}, t_{sim} - t_{eval}$)
19         $list_u(c) := t_{sim} + h_c$
20     end
21     DataExchange($list_e$)
22     $list_e := \varnothing$
23 end
24 foreach $c \in C$ do
25     terminate($c$)
26 end
```

Algorithm 2: Master

(line 17). This time is always equal to the last synchronisation point with Event-B components. For an Event-B component $t_{eval} = t_{sim} - h_c$, since its evaluation time $t_{eval} = t_{start} + n \times h_c$ for all $n \in \mathbb{N}$. To handle timed Event-B models the only necessary alteration of the master is to add a dedicated variable that would store t_{eval} and update it on each evaluation. Finally, the next evaluation time is scheduled at line 19.

After the components are evaluated the master exchanges the data at line 21 (only for the elements of $list_e$) and clears the evaluation list at line 22. Since we disallow direct connections between FMI components that have direct input-output dependencies, and since Event-B components update their outputs in a discrete fashion after a certain interval $h_c > 0$ from the time inputs get updated, the algorithm avoids the problem with potential algebraic loops. Hence, no re-evaluation of components is required after the data is exchanged. The algorithm proceeds with the simulation time increment and the next simulation cycle from line 8.

To aid the understanding of the evaluation order of components and the data exchange that is performed at discrete communication points by the master we visualise the evaluation sequence of a sample component connection graph, shown in Figure 3.8. Components are represented on the graph as blocks with input (grey) and output (white) ports. Non-zero step sizes of components are indicated by the variable h. The direction of the data exchange from one component's output port to another component's input port is indicated by an arrow. The graph consists of two continuous (FMI) components C_1 and C_2, and two discrete (Event-B) components D_1 and D_2, which have a step size of 3 and 4 time units, accordingly.

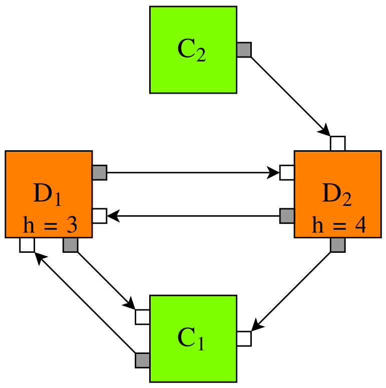


Figure 3.8: An example of the component connection graph

According to the master algorithm and the order of evaluation, which is based on each component's predefined step size, components of the graph Figure 3.8 are evaluated as illustrated on the timeline in Figure 3.9. Initially all components are evaluated at $time = 0$. The next evaluation time is determined by the shortest step size among all components $c \in C$ where $h_c > 0$. Hence, the component D_1 is evaluated at $time = 3$. Because D_1 has an input and an output port connected to C_1, the step size of C_1 is set dynamically to $h_{C_1} = 3$. As a result C_1 is evaluated at $time = 3$. Furthermore, D_2 is also connected to D_1 by an input and an output port. However, because of its step size $h_{D_2} = 4$ it cannot be evaluated at this point, but rather is evaluated at $time = 4$. Because D_2 expects an input from C_2 and produces and output to C_1, both continuous components are also being evaluated at $time = 4$. The co-simulation proceeds in the same fashion by incrementing the time until one of the fixed step-size components needs to be evaluated and also by evaluating every connected component with a dynamic step size.

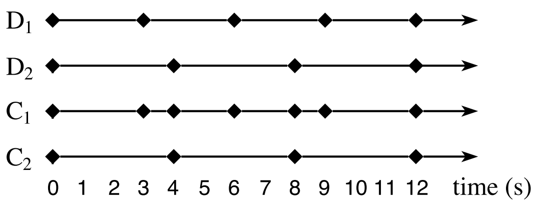


Figure 3.9: Evaluation time of each component

The specified algorithm is capable of simulating an arbitrary graph of connected components in a time-based fashion. The termination of the algorithm is ensured by these properties: 1) we do not re-evaluate components within a simulation cycle, 2) each evaluation of component c progresses the time by the value of its step size $h_c \geq 0$ and 3) simulation terminates if a step gets rejected. As demonstrated on the example in Figure 3.9 continuous components with $h = 0$ (dynamic step size) are only being evaluated if they are connected to some discrete component d with $h_d > 0$ that is being evaluated at the current simulation time, hence their evaluation does not affect the passage of time.

The determinacy of the algorithm is ensured by our assumptions on the existing input-output dependencies of each component type and the limitations on the allowed connectivity between components. All continuous components are assumed to have a potential direct dependency between an input and an output, hence the output cannot be read before the input is provided. This order is maintained by the Algorithm Figure 1. Also, no direct input-output dependencies are assumed to exist in discrete components, i.e. the input does not change the output until the simulation step of size $h > 0$ is executed on the discrete component. This is ensured by our semantics of the *DStep* and its mapping to Event-B machines. On top of that, two or more continuous components are not allowed to be connected directly to each other, but rather via a discrete component, thus breaking potential algebraic loops. The drawback of this limitation is that the continuity of a continuous-time signal may be disrupted. In a scenario where multiple continuous components need to be connected to exchange this type of signal they can be combined into a single component. This coupling can be done via the FMI for Model Exchange standard [113]. Alternatively, continuous models can be combined directly at the model level, for example, in Modelica. For best simulation results this is in fact the recommended approach [116].

3.8 Summary

The proposed co-simulation design provides the simulation of discrete-event models of control systems, specified formally in Event-B, against continuous-time physical models via FMI. The simulation step semantics of continuous components complies with the FMI 1.0 standard in terms of the use of the API, which ensures the valid simulation results of the physical subsystems.

The discrete step semantics of Event-B components that is based on the finite traces of events, commencing with the *StartStep* events and concluded by the *EndStep* events, is designed to be consistent with the Event-B language semantics and supports the refinement-based development. Hence, a discrete model can be simulated and validated against a physical model starting from the abstract level and down the refinement chain, which allows the modeller to break down system complexity by introducing the system details gradually. In addition, this makes it easier for the modeller to formally prove system safety and other properties, as they can be introduced and elaborated at different refinement steps. Since Event-B aspect of the simulation is intended to be used for developing discrete control systems, the discrete step is defined to have a fixed step size, which can be interpreted as a control cycle interval. The step size is a configurable parameter of the component and does not need to be modelled in the associated Event-B machine. Simulation supports untimed Event-B models. Multiple co-simulated Event-B components can have different step sizes that will be evaluated by the master algorithm accordingly.

The designed algorithm is easily implementable in a Java-based environment like Rodin. The execution of Event-B traces within a simulation step can be delegated to the ProB animator tool, whilst the simulation of FMI components is to be performed directly by the FMU, since Co-simulation FMUs must have an integrated solver. The only additional requirement for implementing the simulation is a Java-based interface to interact with an FMU, as they come compiled as a native (C code) shared library. For that purpose we are utilising an open-source solution, as will be described in the following chapter.

Chapter 4

MultiSim Tool

As part of this work we have developed a co-simulation framework, which is an extension of the Rodin platform. In this chapter we give details of its design and implementation and describe the capabilities of the tool, its features and elements of the user interface.

4.1 Overview

The outlined concepts from the previous chapter were developed into a single tool called MultiSim. The tool is designed as a plug-in extension for the Rodin platform and provides a generic multi-simulation capability for engineers of hybrid systems. Our main goal is to allow engineers to validate their formal Event-B models against the realistic physical models of environment by simulating them in a single tool. The idea and design of the extension was proposed for and funded by the ADVANCE project and developed in collaboration with researchers of the University of Dusseldorf (developers of the ProB model-checker [94]). The initial prototype implementation was refined and updated, and is now based on the latest version of the Rodin platform and its frameworks. It has been successfully validated on a number of hybrid system models, illustrated in the next chapter. In addition, it was validated on a smart grid case study by Critical Software [23] in the final year of the ADVANCE project.

4.2 Tool Requirements

In order to support the proposed development workflow within the context of selected modelling and simulation technologies, i.e. Rodin, Modelica and FMI, the tool must meet the following requirements:

- The tool must provide facilities for mapping discrete (Event-B) and continuous (Modelica) models to simulation components, executable under co-simulation via the FMI master according to the semantics we defined in Section 3.3.
- The mapping of Event-B models to simulation components must handle Event-B refinements in order to support a step-wise modelling and validation process that is the core of the workflow: a formal model should be mappable to a simulation component that can be co-simulated with a physical model; a refinement of the model should be also mappable to a component that can be co-simulated with the same physical model or its refinement.
- The tool must implement a generic FMI master algorithm that is capable of co-simulating an arbitrary number of interconnected discrete and continuous components (see Section 3.7); the master must provide an adequate performance level on a typical machine and be scalable, e.g. maintain linear memory consumption with the increase in simulation time.
- The tool should offer a visual notation and a graphical environment for easy composition of co-simulated models. That includes:
  - A block diagram editor for composing co-simulated models into a component graph: each co-simulated model is represented as a component block with a set of input and output port variables that can be connected to the ports of other components via connectors
  - A facility for importing Event-B and Modelica models onto the diagram
  - A capability to instantiate components (create multiple instances of the same model) and configure parameters of each instance individually
  - Configurable parameters of each co-simulated model that can be modified and persisted between simulations
  - Configurable number and properties of the input and output ports of a component block that can be modified and persisted between simulations
- The tool should be able to run co-simulation of the component connection graph and display the simulation state with time, including the exchanged signal values between the input/output ports of components.
- The results of the simulation should be available at the end of a simulation execution and be persisted in a readable text format, which includes a record of the values of input/output variables of each component over simulation time and a trace of executed events of each co-simulated Event-B model.

4.3 Characteristics

4.3.1 Physical Models and FMI

The objective of this work is to fill the gap between existing formal modelling and physical simulation methods, with the focus on the application of Event-B formal method and Modelica modelling language in the development of cyber-physical systems. The Event-B method and accompanying Rodin toolset is a powerful technology for assuring system correctness by construction from the early stages of requirements specification. But the model of a hybrid system has to be abstracted to discrete terms due to the discrete-event semantics of the Event-B language. The intrinsically continuous and unpredictable environment is more naturally expressed in physical terms. Modelica[1] is a free and open-source physical modelling language, with a large number of supported tools and a rich library of domain-specific components, making it a compelling candidate for modelling the physical aspect of hybrid systems. Integration of this technology into Rodin makes it possible to validate formal specifications in conjunction with the models of environment that are expressed in a physical language.

Composition of the formal discrete models with continuous physical models of environment from Modelica is implemented in the MultiSim extension using the Functional Mock-up Interface for Co-Simulation standard. The advantage of this standard over other co-simulation frameworks is its openness, tool and platform-independence and high level of adoption by industry, which enables integration of the Rodin toolset with the state of the art physical modelling and simulation technologies. For the purpose of this work and the goals of the ADVANCE project an interface to FMI 1.0 slave import has been added to the ProB 2.0 package via the JFMI library from Berkeley[2]. MultiSim utilises this interface for the physical model import and implements a generic simulation master algorithm based on a two-list evaluation concept [83] hence enabling deterministic and scalable simulation of multiple Event-B and FMI components in Rodin.

4.3.2 Visual Notation and Extensibility

In order to provide convenient facilities for composing Event-B and FMI components the plug-in is designed as a graphical block-based editor, similar to the familiar tools such as the MATLAB Simulink and Modelica-based environments. This makes it extremely simple for the users to import components on a diagram canvas as blocks and connect them via input and output port signals. The editor performs automatic validation of composed graphs before the simulation.

[1]http://www.modelica.org
[2]http://ptolemy.eecs.berkeley.edu/java/jfmi

The plug-in is designed to be extensible, which is facilitated by employing the latest model-based Eclipse technologies: Eclipse Modelling Framework (EMF) and Graphical Modelling Framework (GMF). EMF and GMF are the standard frameworks of Eclipse platform that provide domain-specific language modelling and code generation capabilities for developing specialised model-based tools and graphical editors. Using the UML class diagram notation a model of the domain-specific language can be designed in EMF and automatically translated into an executable boilerplate code of the structural editor. GMF allows the generated code to be further developed into a graphical editor. Generated editors can be easily customised and extended to accommodate the desired logic and user interface capabilities. We have developed an EMF model of the component connection graph notation and a corresponding GMF editor. Simulation is performed on a generic EMF component class, which hides implemented logic from the master algorithm. Hence, the plug-in can be easily extended to support additional simulation blocks besides Event-B and FMI without redefining the master.

4.3.3 Dependencies and Features

The plug-in uses the latest version of the Event-B EMF framework, which is an extension for the Rodin platform that provides developers with a complete EMF model of the Event-B language and the mechanisms for its extension and serialisation. This makes MultiSim compatible with other tools that are based on the Event-B EMF framework, such as the iUMLB State-Machines for modelling systems using visual abstraction of state charts. In fact, we have used state machines extensively in this work to model the control aspects of hybrid systems in Event-B.

MultiSim makes use of the ProB API for simulating Event-B machines, as well as the new features of ProB 2.0 [18], in particular, the linear temporal logic (LTL) execution at the Prolog client level and the optimised state-space data structure for improved simulation time and memory consumption. The ProB kernel is written in Prolog and works as a client application, which communicates to the Java front end in Rodin. LTL execution allows logical predicates on a model state to be sent and executed on the ProB client side without involving the front end, which substantially reduces memory footprint and improves performance. A schematic view of the MultiSim dependencies on ProB and other tools is shown in Figure 4.1.

The initial version of MultiSim operated the model state space at the ProB front end level, executing and examining each event directly. This approach had performance problems, identified by one of the ADVANCE project partners, Critical Software, through a large-scale model of the smart grid voltage control system. The desired simulation duration (1440 steps, each involving more than 150 events) could not be achieved due to large memory consumption and poor performance, which resulted in the simulation machine running out of resources after reaching 46% progress mark in 10 hours. We addressed the problem in collaboration with the ProB developers team, who have introduced the LTL execution feature and considerably reduced memory footprint of the state space data structure. Moving the implementation to the use of LTL allowed us to delegate event execution to the ProB client, hence minimising the number of control switches between the latter and the simulation tool. As a result, simulation time of a control cycle in MultiSim has decreased on average tenfold compared to the first version of the tool (see Figure 4.2). To help clarify the significance of the performance improvement: simulation of 24 hours of the simulated network operation took 10 hours (stopped at 46%) before the fix and 1 hour afterwards.

Another requirement, identified during the evaluation of the tool on the smart grid case study, was the ability to record a trace of executed events of individual Event-B components. This feature is implemented in MultiSim by serialising executed events to a file that can be loaded by the ProB model-checker, which allows an execution of the

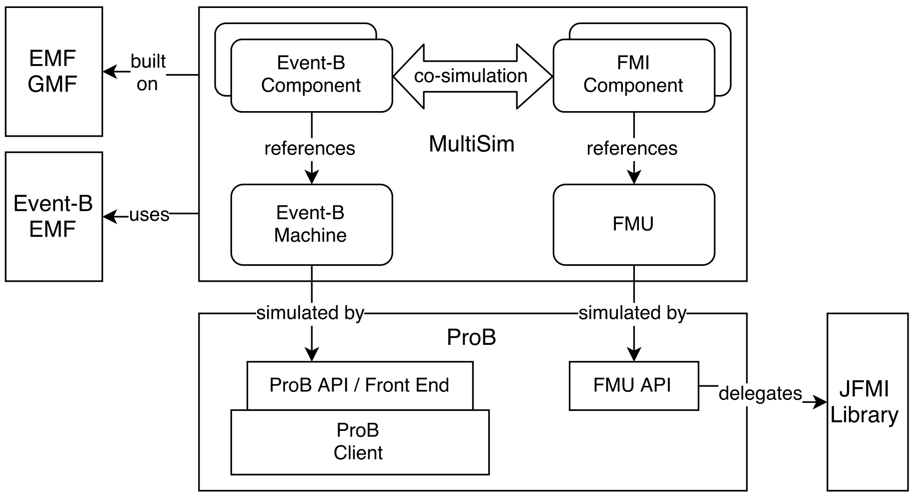
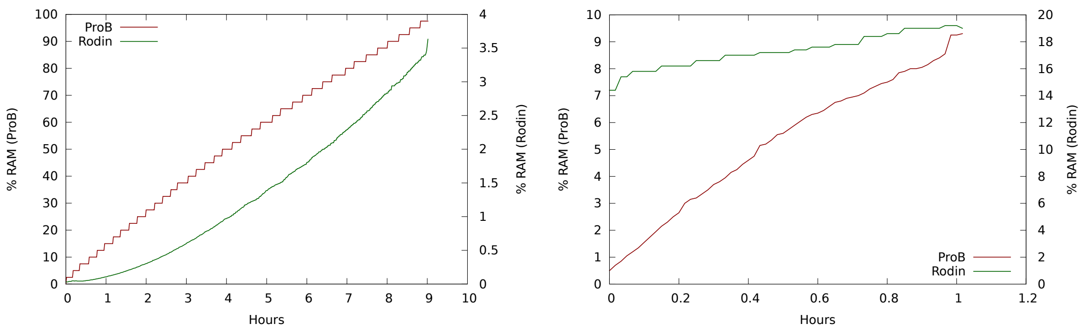


Figure 4.1: MultiSim interfaces and dependencies on other tools



(a) First version of MultiSim (b) Latest version using ProB 2.0 and LTL execution

Figure 4.2: Memory footprint and performance of the Rodin/ProB client during multi-simulation of a smart grid voltage control system [23]. With initial implementation (a) the machine (Intel Core i5@2.6 GHz, 16 GB RAM) runs out of memory at 46% of simulation progress. Latest version (b) runs to completion.

simulated Event-B machine to be replayed and analysed without re-running the resource-intensive simulation. In addition, the results of simulation in the form of output signal values can be plotted in real-time via the Display component, similar to the Scope block in MATLAB Simulink, and recorded to a text file. The ability to record the simulation trace and component outputs has proven to be very useful when analysing unexpected behaviour, such as system deadlocks, and identifying errors in a model. It can also be used to visualise the execution of Event-B machines via the B-Motion Studio plug-in for Rodin [80].

4.4 Implementation

In this section we describe in more detail the architecture and implementation of the tool, starting from the utilised frameworks and application interfaces, followed by a simulation meta-model and a master-slave interface.

The FMI for Co-Simulation standard specifies a set of interface routines, or the Application Programming Interface (API), for the communication between a master and individual simulation tools (slaves) in a co-simulation environment [107]. The MultiSim extension for the Rodin platform is an example of such environment and therefore must implement a master that uses FMI interface to control the simulation. This interface, defined as a collection of C functions, must be further implemented by all simulated Event-B models. As the Rodin platform and the ProB animator is capable of executing Event-B machines are all Java-based tools, we have made a decision to move away from using the C-based FMI interface directly and instead have defined a compatible simulation interface in Java that operates on the abstract level of a generic *Component*. This enables easier integration of the master and Event-B simulation into Rodin. The generic interface also hides component-specific semantics from the master, which makes it extensible for supporting new types of components. Currently supported types include Event-B components, which are implemented by interfacing with ProB, and FMU components that are interfacing with the underlying FMI slaves via the Java FMI wrapper library[3].

We define our simulation interface on an abstract entity called *Component*, which encapsulates the slave-specific data and simulation semantics, hiding it from the master. Hence, a co-simulation composition graph required by the latter comprises a set of components interconnected via inputs and outputs. We combine these entities, their properties and relevant associations into a single data structure called the meta-model. In model-driven engineering (MDA) a meta-model is a model of a modelling language, i.e. a specification of the rules and constructs for creating semantic models [134]. With respect to simulation our meta-model provides the constructs for creating instances of

[3]http://ptolemy.eecs.berkeley.edu/java/jfmi/

the composition graph that contains a collection of specialised components (concrete implementations of the abstract component), each of which may have a set of input and output ports that can be connected to the ports of other components. An instance of the graph serves as a data and semantic model of the simulation, and is used by the master to execute and control the simulation process. The complete class diagram of the meta-model is shown in Figure 4.3. It has been designed using the Eclipse Modelling Framework (EMF), hence class attributes are declared not with the standard Java datatypes, but with equivalent EMF datatypes. The greyed-out classes are the shortcuts to the Event-B EMF meta-model, which is part of the Rodin toolset.

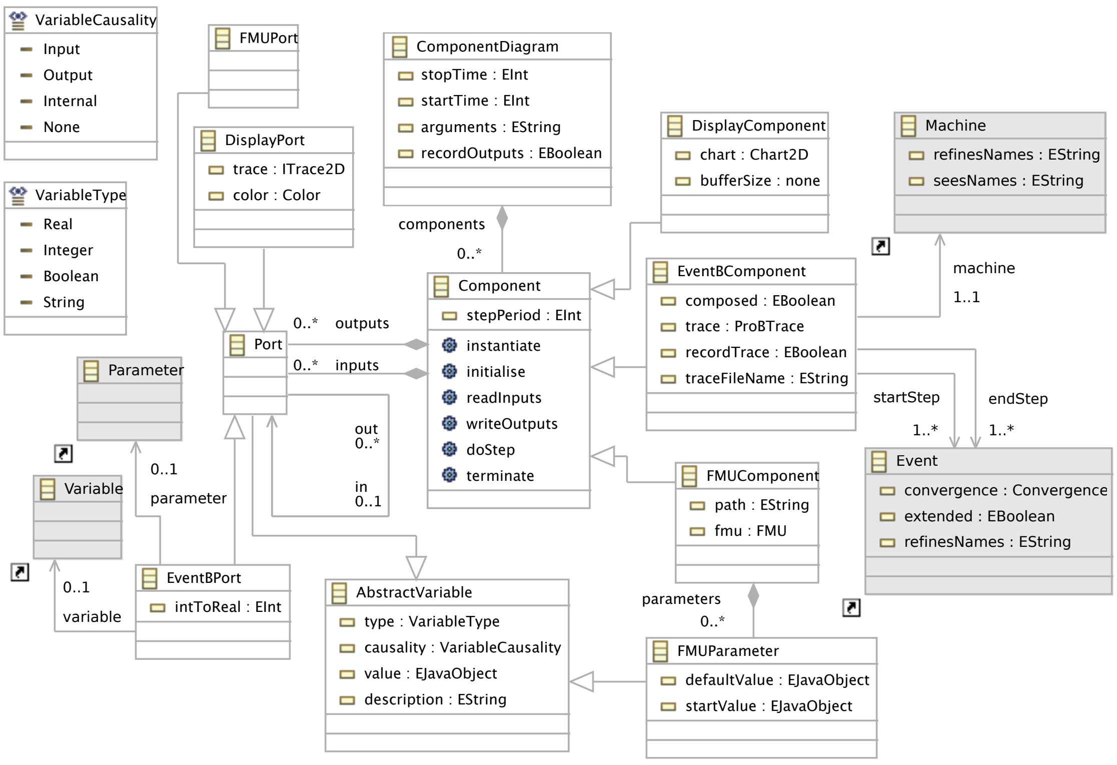


Figure 4.3: Meta-model of the co-simulation framework

The root element of the meta-model is the *ComponentDiagram* class, which models the composition graph of the instances of co-simulated models. The *components* containment association defines a collection of component instances existing on the diagram. The diagram, besides serving as a root container, incorporates a set of attributes essential for the simulation configuration. These include the simulation start time, stop time and step size. Some attributes are added for specific capabilities of the tool, such as the *arguments* attribute for the ProB execution configuration, and the *recordOutputs* flag for the optional recording of output signals to a file.

The *Component* is an abstract class that models a simulation slave. It contains a single attribute *stepPeriod* that allows the component to have a custom simulation step size, different from the diagram (global) step size. The component also defines the master-slave simulation interface, which is specified as a set of operations on a component:

- `instantiate` operation creates an instance of the associated model for the simulation. This may include loading a model file from the filesystem or starting up a dedicated simulator for that model.
- `initialise` operation initialises a component instance prior the simulation. This typically involves setting model parameters to user-defined values as well as setting initial values of the inputs. A call to the *initialise* operation requires simulation start and stop time to be provided as arguments.
- `readInputs` operation reads the values from connected input ports of a component.
- `writeOutputs` operation writes the values to output ports of a component, i.e. makes them available for reading by other (connected) components.
- `doStep` operation is the key operation of the simulation that simulates a component for one step of the time period, provided as an argument. The second argument of the operation is the current simulation time.
- `terminate` operation releases all the resources from memory at the end of simulation and does other post-simulation processing on a component, such as closing an open model file or a loaded library.

The component has an optional set of inputs and outputs, defined by the namesake containment associations to the abstract class *Port*. Ports can be connected depending on their type. This is modelled by associations *in* (incoming connections to an input port) and *out* (outgoing connections from an output port). The main purpose of the port is to store the value of a specific input/output variable of the model instance during the simulation. These values are used by the master to synchronise the signals between components.

The attributes of the port are specified by a generalised abstract class *AbstractVariable*, which includes a data *type* (Real, Integer, Boolean or String), *value*, *causality* (Input or Output for ports and Internal for other variables) and *description* (an optional descriptive comment on the variable).

Three specialisations of the abstract component are implemented in MultiSim: *FMUComponent*, *EventBComponent* and *DisplayComponent*. The FMUComponent class models continuous-time FMI units, or FMUs. In order to load an FMU the component stores its file location in the *path* attribute. The *fmu* attribute holds a reference to the loaded FMU instance. FMU components have a list of optional *parameters*, specified by the *FMIParameter* class – a specialisation of the *AbstractVariable* that stores the values of the associated model parameters. Parameters always have a *defaultValue* (defined internally by the FMU), which can be redefined by the user via *startValue* attribute. Finally, FMU components use ports of the type *FMUPort*, a specialisation of the abstract port.

The *EventBComponent* class models simulated instances of Event-B machines. It contains a set of Event-B specific attributes that define semantics or store vital data of the simulation. The semantics is determined by the reference to a simulated Event-B *machine* and the links to *StartStep* events, which are the events that start the simulation step and optionally read the values from input ports, and *EndStep* events, which denote the end of the step. The *trace* attribute is the key attribute of the simulation – it contains a reference to the current execution trace in ProB, which is used to execute Event-B events and read the values of state variables. The *composed* attribute is a flag to indicate that the associated Event-B machine is a composed machine (a composition of two or more machines). The *recordTrace* flag and the *traceFileName* attribute are used by the tool for recording the trace to a file.

The semantics of Event-B components is additionally determined by their ports. The *EventBPort* is a specialisation of an abstract port that contains a reference to a *parameter* of the specified *StartStep* event or a reference to an internal *variable* of the associated Event-B machine. Depending on the port causality the parameter reference is used in conjunction with the *StartStep* event to read the input value from an input port, whereas the variable reference is used to update the output value of an output port.

Finally, the *DisplayComponent* class models an auxiliary component, which enables visual plotting of the output signals from other components during simulation. Signals are plotted by connecting the associated output ports of components to *DisplayPorts* of the display component. A display port stores the plot*colour* and the *trace* of plotted values, updated throughout the simulation in real time.

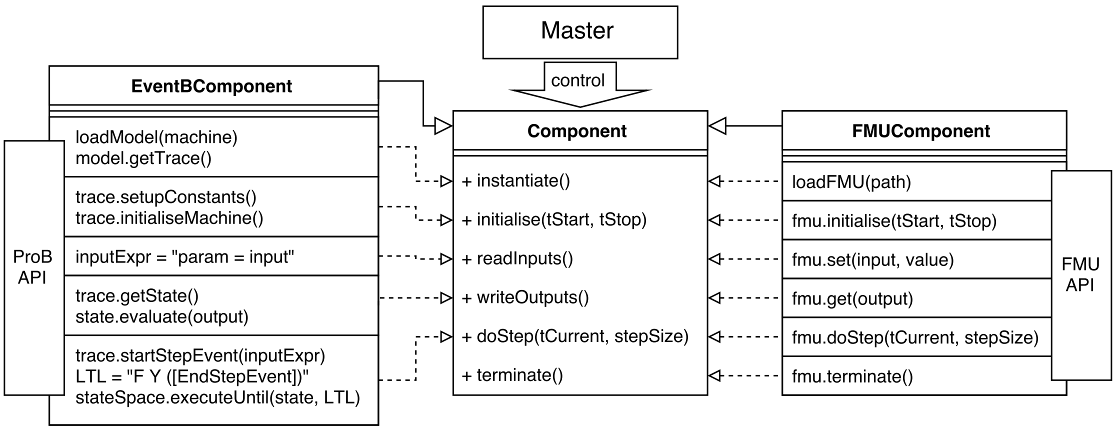


Figure 4.4: Simulation interface implementation by Event-B and FMU components

The developed meta-model has been used to generate the boilerplate code of the simulation data structure via EMF framework. The implementation of Event-B and FMI components maps our simulation interface to manipulations on the underlying model instances, which is illustrated in Figure 4.4 using pseudocode. Since our interface is based on the FMI API, the implementation of FMU components is straightforward – interface routines are mapped directly to the corresponding FMI method calls.

In the case of Event-B components we apply the discrete step semantics, defined in Chapter 3. Implemented methods of the *Component* class execute a trace of events of the associated Event-B machine via ProB. Figure 4.4 shows the key operations from the actual implementation in a pseudocode form:

- `instantiate` method loads an Event-B machine and retrieves its trace.
- `initialise` method executes two operations on the machine. If the machine has a *sees* relationship with an Event-B context, *setup_constants* operation initialises its constants in accordance with the specified axioms. *initialise_macnine* operation initialises machine variables by executing the *INITIALISATION* event.
- `readInputs` method reads the values of input ports of the component and constructs a predicate that binds those values to the *StartStep* event parameters.
- `writeOutputs` method retrieves the current state and evaluates the values of internal variables, associated with the output ports of the component.
- `doStep` method first executes one of the enabled *StartStep* events with an input predicate from `readInputs`. It then constructs an LTL formula of a state where one of the *EndStep* events is enabled and subsequently executed. This formula along with the current state is fed to an *executeUntil* operation in ProB, which, given a state and an LTL formula of the new state, tries to find a trace that leads from former to latter.
- `terminate` method is used by the EventBComponent for peripheral tasks only and does not modify the simulation trace, hence it is not shown on the diagram.

4.5 Operation

The prerequisite for each simulation is a component-connection graph that captures the topology of system components. This graph is used by the simulation master algorithm to orchestrate the simulation and exchange the data between components. MultiSim provides a block-based diagram editor (Figure 4.5), which enables easy composition of components into a graph via input/output port connections. A component can be added to an empty diagram via drag and drop interface from the file system or the Rodin project workspace. The imported component can be either an Event-B machine or an FMU file.

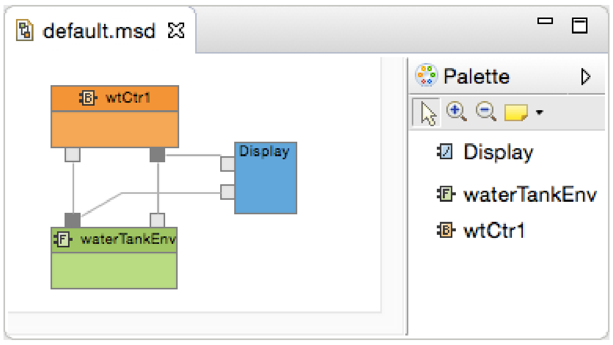
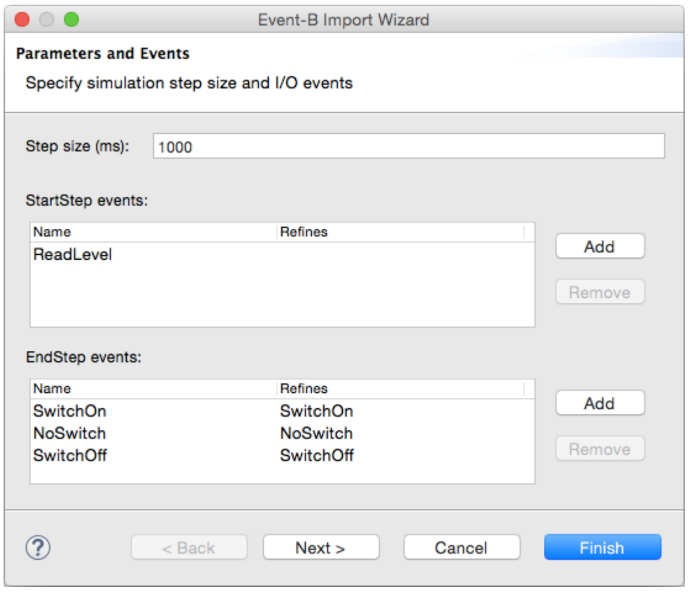


Figure 4.5: MultiSim component diagram editor

When an Event-B machine is imported to a diagram for the first time, a configuration wizard helps the user to assign the simulation semantics to the newly created Event-B component: the user has to specify a step period of the component (data synchronisation period, in milliseconds), at least one or more *StartStep* events and *EndStep* events (Figure 4.6).



Parameters and Events

Specify simulation step size and I/O events

Step size (ms): 1000

StartStep events:

Name	Refines
ReadLevel	

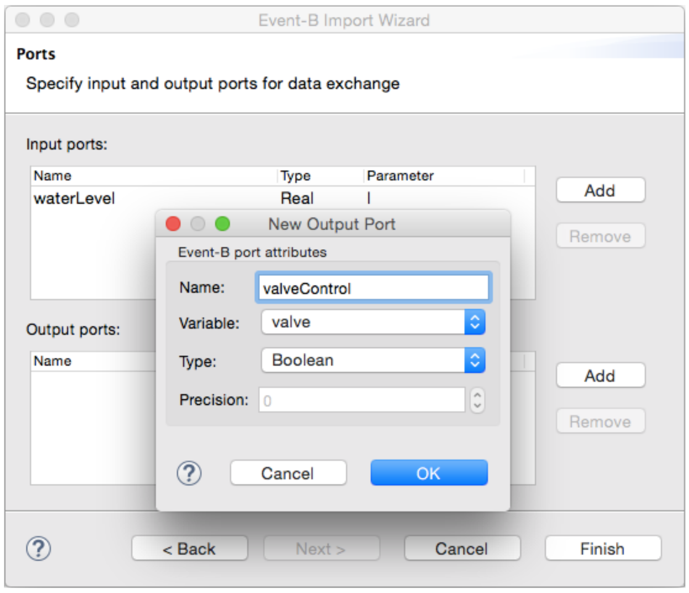
EndStep events:

Name	Refines
SwitchOn	SwitchOn
NoSwitch	NoSwitch
SwitchOff	SwitchOff

Figure 4.6: Configuring Event-B component parameters and events

The user also has to add required input/output ports by mapping parameters of the previously specified *StartStep* events to inputs and the variables of the machine to outputs (Figure 4.7). The definition of each port includes an optional name, a required reference to either a parameter or variable, a signal type (from supported FMI types), and, in case of the Real type, a conversion precision to/from Integer.

When the component is configured and successfully imported to the diagram, it is automatically added to the editor's tool palette, hence enabling multiple instantiation of the same component from the palette. This applies to both Event-B and FMU components. In addition, all semantic information specified for Event-B components during



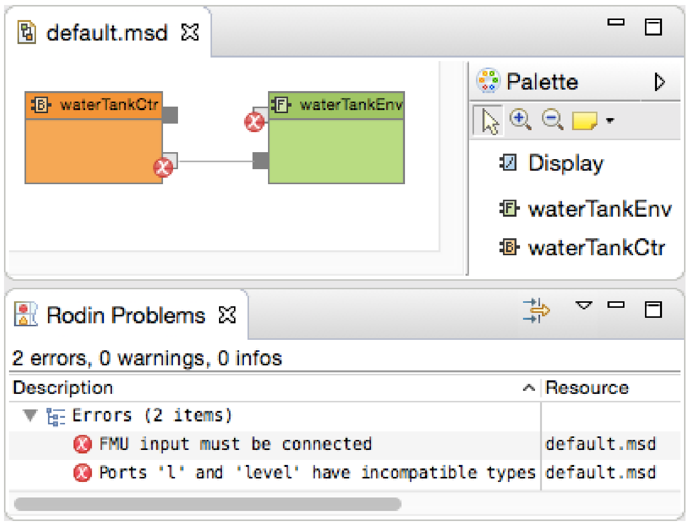
Name	Type	Parameter
waterLevel	Real	l

Name: valveControl
Variable: valve
Type: Boolean
Precision: 0

Figure 4.7: Configuring Event-B component input/output ports

the import process is persisted in the associated Event-B machines as meta-data, which is reused for automatic configuration on subsequent imports.

Prior to simulation the diagram is automatically validated for any errors, such as incompatible types of connected ports, undefined *StartStep*/*EndStep* events of Event-B components, missing references to an FMU file, etc. If a problem is discovered, simulation is interrupted and the user gets notified about errors via error markers on the diagram as well as error messages in the Rodin Problems view (Figure 4.8). Validation problems can be fixed by modifying the component's attributes from the Properties view, which allows the user to add, remove or reconfigure events and ports at any stage of the modelling process. The diagram can be statically checked for validation errors after it has been modified.

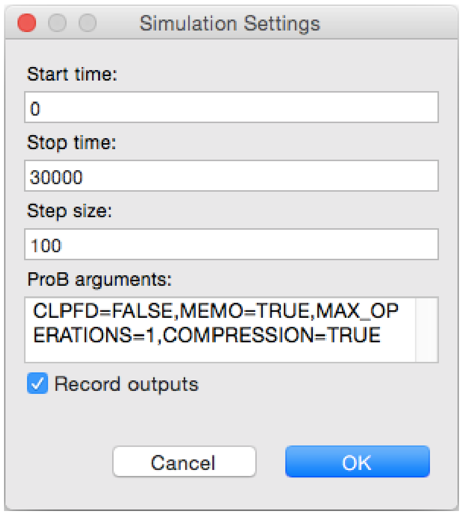


2 errors, 0 warnings, 0 infos

Description	Resource
Errors (2 items)	
FMU input must be connected	default.msd
Ports 'l' and 'level' have incompatible types	default.msd

Figure 4.8: Diagram validation error markers and error messages

After the connection graph is completed and verified, simulation can be started by providing simulation start time, stop time and step size. An optional set of ProB attributes can be added to optimise the simulation of Event-B components, for example, to use memory compression of the state-space. Output value recording can be also turned on (Figure 4.9). Simulation process runs in the background, and the output comes in three forms: an output signal value file in a .csv format (containing time-value rows for each simulation step), a serialised event trace file for individual Event-B components and a real-time plotting during simulation, which requires adding a Display component from the tool palette and connecting it to output signals to be plotted.


Simulation Settings

Start time:
0

Stop time:
30000

Step size:
100

ProB arguments:
CLPFD=FALSE,MEMO=TRUE,MAX_OPERATIONS=1,COMPRESSION=TRUE

✓ Record outputs

Figure 4.9: Simulation start dialog

4.6 Available Documentation

The tool proposal has been presented at the 5th Rodin User and Developer Workshop [129], 2014 Summer Simulation Multi-Conference [126] and Doctoral Symposium at the 4th International Conference on Simulation and Modeling Methodologies, Technologies and Applications [128]. A validation of the tool on a landing gear case study, which is set out in the next chapter, has been presented at the 4th International ABZ 2014 Conference [127]. Both the latest source code and the installation/use instructions of the tool are available online[4].

[4]http://github.com/snursmumrik/multisim

4.7 Summary

The MultiSim tool for Rodin is the key contribution of this work. It addresses the existing limitation of the Rodin toolset – the lack of continuous-time modelling and simulation-based validation of hybrid system models with a physical environment, which is an important requirement for rigorous development of hybrid systems. MultiSim addresses this limitation by providing a mechanism for mapping Event-B models to simulation components that can be composed with physical models. The mapping also caters for refinement, hence the tool supports a systematic modelling and validation workflow, in which a formal model is refined in small incremental steps and validated at each step against the physical model of environment. The validation is supported by the tool via the master algorithm, which simulates the composition of all sub-models. The implemented master is generic and is capable of co-simulating an arbitrary number of components. It is built on the established FMI interface for co-simulation that is an open standard and is supported by many physical modelling tools, including Modelica-based.

In addition to co-simulation the tool provides a graphical environment that allows the models to be imported as component blocks and connected via input and output ports into a component graph. Components can be instantiated and their parameters can be modified for each instance. The component graph is used by the simulation master to orchestrate co-simulation and perform signal exchange between connected ports. The values of signals are visualised during the simulation and the state of each component is saved to an output file.

Our goal was also to reuse as many available technologies and frameworks as possible. This is addressed in MultiSim by adopting the existing FMI libraries and Rodin/Event-B frameworks, as well as the tools like ProB for executing and validating simulations.

During the development of MultiSim we have identified performance and scalability problems. For example, the simulation of a formal model with a large number of events resulted in performance degradation and exponential memory consumption. These issues have been resolved by optimising the master algorithm and by minimising interaction of the tool with the execution engine of Event-B (the ProB kernel).

In the next chapter we validate the tool on the examples of hybrid systems by modelling each system as a combination of the formal model of the control sub-system and the physical model of the environment, and by subsequently verifying and validating the composed model using MultiSim. We then evaluate our approach by comparing it to the traditional simulation and identifying its benefits and limitations.

Chapter 5

Experiments

To validate the functionality of our tool we have conducted a number of experiments that applied MultiSim on the discrete models constructed in Rodin and continuous models created in the Modelica-based environment Dymola, which is a commercial product by Dassault Systèmes that supports the latest revision of the Modelica language and the FMI standard. Some of the case studies involve both co-simulation and formal verification, highlighting the advantages of our integrated approach. The list of case studies was compiled to be representative of the hybrid systems domain and real life problems:

- A controlled water tank – the classical example of a hybrid system that is commonly used in the literature to demonstrate discrete-continuous behaviour.
- Distribution voltage control – a hybrid system similar to the water tank, but from the power systems domain. This case study was selected as a benchmark before the real-scale smart grid case study from the ADVANCE project [23].
- Angle of attack sensor processing – a real life problem that is a good demonstration of the benefits of rigorous analysis techniques.
- Controlled landing gear – a benchmark case study of the joint formal methods conference ABZ 2014 [29].

In order to evaluate the tool we first need to determine the criteria for success. A successful evaluation should demonstrate that:

- The results of the simulation by MultiSim are consistent with the results by other simulation environments (in our case Dymola).
- The approach supported by the tool is beneficial for development: it addresses certain limitations and offers advantages over traditional techniques such as simulation and testing.

- The proposed workflow is feasible: it can be easily adopted by engineers without the fundamental changes to an established engineering process.
- The approach is economically adequate: it can be adopted without significant cost increase.

5.1 Water Tank Control

In the first experiment we evaluated the simulation functionality of the tool on a classical example of a hybrid system – a controlled water tank, briefly described in Chapter 3. In its simplest form the system consists of a water tank that has a constant water outflow and a controlled valve (or a pump in other variations of the system) that supplies water input to the tank and can be actuated by a controller. When the valve is open it delivers water to the tank at some flowrate v_1. Water outflow has a rate v_2. The tank is characterised by a fixed capacity and two water level bounds $0 < L < H$. The goal of the system is to keep the water level within the interval $[L, H]$ by measuring the current water level and controlling the state of the valve. This goal can be formalised as a system invariant:

$$\mathit{level} \in L..H \tag{5.1}$$

As a hybrid system this example has the following components: the environment that is represented by the water tank with an open output port and an input port connected to the valve, the plant comprised of the actuated valve and sensors that monitor the water level, and the controller that controls the state of the valve, which can be either *open* or *closed* (an ideal discrete valve). The controller follows a simple strategy: when the valve is closed and the sensed water level drops below the lower threshold LT, where $LT > L$, the controller opens the valve; when the valve is open and the water level reaches the upper threshold HT, where $HT < H$, the controller closes the valve; in between the thresholds the same mode (open or closed) is maintained by the controller.

The dynamics of the water level in the tank can be specified by a simple differential equation:

$$\mathit{level}'(t) = \begin{cases} v_1 - v_2 & \text{if } \mathit{valve} \text{ is open} \\ -v_2 & \text{if } \mathit{valve} \text{ is closed} \end{cases} \tag{5.2}$$

This equation can be easily translated to Modelica to describe the physical model of the environment. In a more realistic scenario the flowrates v_1 and v_2 are dynamic and dependent on the current water level in the tank as well as thermodynamic properties of the medium and the environment. The advantage of multi-domain physical modelling languages such as Modelica is that they provide facilities to model these aspects of the physical system. Furthermore, it comes with the Modelica Standard Library (MSL) of already developed components for modelling systems of different domains. We have utilised components of the Fluid package (1-dimensional thermo-fluid flow in networks of pipes) of the MSL to model the physical aspect of the water tank system. The resulting model consists of a water *source*, modelled by the boundary source component, a discrete *valve* with linear pressure drop, an open water *tank* with inlet/outlet ports and a second boundary component representing the *sink*. The control signal to the valve is provided by the discrete input port *valveInput*, while the water level reading is connected to the continuous output port *levelOutput*. The model of the sensor is idealised, i.e. we can instantaneously read the current value of the water level at any given time. A similar idealised property is attributable to the valve, which provides a maximum specified flowrate immediately when it is open. The complete model of the plant is shown in Figure 5.1.

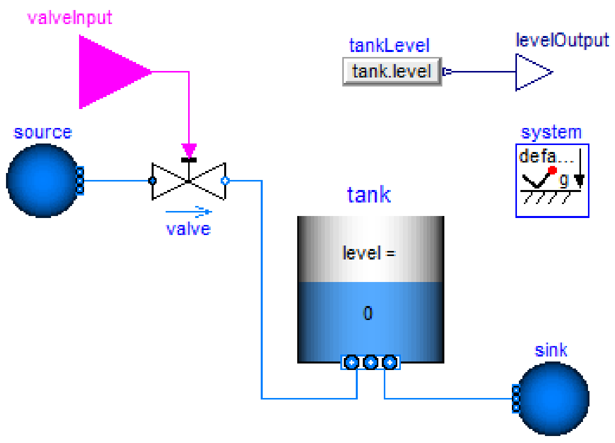


Figure 5.1: Controlled water tank plant subsystem in Modelica

For the simulation scenario the model was initialised via parameters of the water tank and port dimensions, ambient pressure and temperature in the source and sink, and the valve pressure loss. The medium of all Modelica fluid elements was set to the idealised model of the liquid water. The resulting model was exported from Dymola as an FMU for Co-simulation. Note that after the export the mentioned parameters of the model can be modified from MultiSim via the Parameters view.

The discrete Event-B model of controller, specified in Section 3.5, was imported into MultiSim and configured as an Event-B component by augmenting it with simulation semantics, which required: 1) specifying the input reading event *readLevel* (Figure 5.2) as the *StartStep* event and three control order issuing events *closeValve*, *keepValve* and *openValve* as the *EndStep* events, 2) adding a Real input port, associated with an input parameter *l* of the *readLevel* event that reads the water level from the plant, 3) adding a Boolean output port, associated with the machine variable *valve* that models the control order to the valve, and 4) specifying the component's simulation step size, which denotes the sensing/actuation period of the controller, and in this case is equal to 100ms.

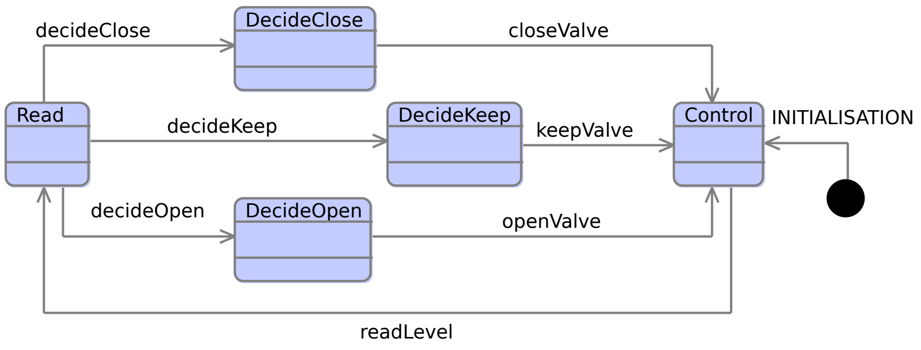


Figure 5.2: State machine of the water tank controller in Event-B

The invariant 5.1 should be preserved by the system since the water level is maintained by the controller roughly within the thresholds LT and HT. However, for the given model it is not possible to prove formally that $level < LT \Rightarrow level \geq L$ (and similarly that $level > HT \Rightarrow level \leq H$) since the modelled water reading from the plant is a physical property and therefore is not constrained by the surrogate properties of the controller. In order to prove these properties and the goal invariant we must introduce additional assumptions on the maximum water level change between two subsequent measurements. At any level reading the following invariant must hold:

$$valve = open \Rightarrow H - level \geq v_1 - v_2 \wedge valve = closed \Rightarrow level - L \geq v_2 \tag{5.3}$$

The control decision should therefore be based on the predicted change for the next measurement, depending on the state of the valve. We leave out the necessary modifications to the formal model and its proof, as this approach has been demonstrated already in multiple works, such as [42] and [140]. Note, however, that the referenced studies modelled a system with constant flowrates.

For the simulation scenario we have initialised the model with the following parameters. The water tank height was set to 3m. The lower bound $L = 0.5m$, and the upper bound $H = 2.5m$. The lower threshold $LT = 1m$, and the upper threshold $HT = 2m$. The initial water level was set to 0 and the valve left open. The composed simulation diagram of the Event-B component, representing the controller, and the FMI component, representing the water tank plant, is shown in Figure 5.3. A Display component has been added to visualise the *valve* and *level* signals during the simulation.

The diagram was simulated for 30s with a simulation step size equal to the step interval of the controller (100ms). The simulation results obtained from MultiSim are shown on a plot in Figure 5.4. From the diagram we can observe non-linear dynamics of the water level and the discrete change of the valve as a result of control orders. For example, the

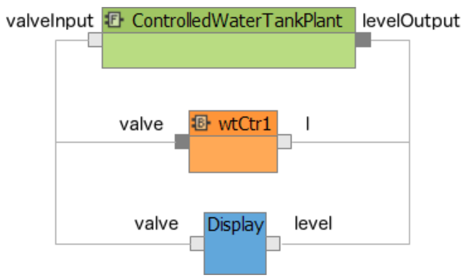


Figure 5.3: Component diagram of the controlled water tank in MultiSim

valve state changes from *open* (1) to *closed* (0) after a control order is issued at $t = 18s$ upon the water level going above the upper threshold of 2m. Similarly, the controller opens the valve again at $t = 26s$ after the water level reaches the lower threshold of 1m.

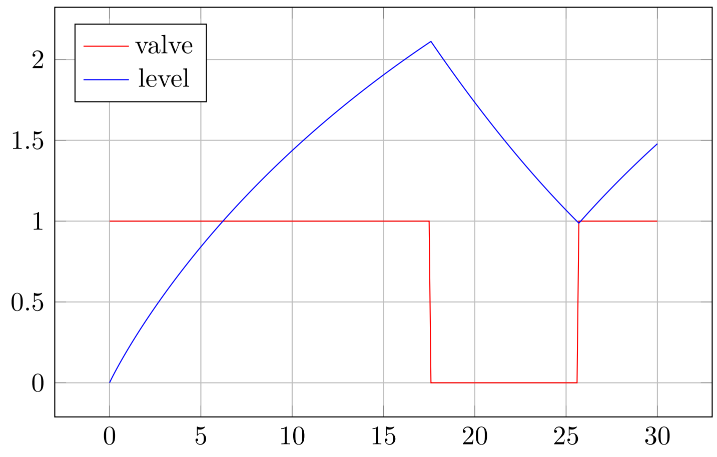


Figure 5.4: Simulation results of the controlled water tank model in MultiSim (simulation time = 30s, step size = 0.1s)

An observable overshoot of the water level above the upper threshold at $t = 16s$ suggests that given a considerable input flowrate, such that $v_1 - v_2 > H - HT$, the goal invariant could have been violated, which directly correlates with inability to prove the invariant formally. The advantage of the formal proof is that a theorem is proven for any instance of the model, whereas simulations require to instantiate a particular scenario. Using MultiSim the two approaches can be combined and the formal proof can be augmented with the simulation-based validation. The discovered flaw in the formal model also shows the importance of the validation of Event-B models against a precise model of environment, and our co-simulation plug-in addresses exactly that. As a future work we are interested in modelling a more complex version of the water tank system, consisting of multiple tanks coupled in a loop.

5.2 Voltage Distribution Control

The next experiment was taken from the Power Systems domain to show how co-simulation can be used in the modelling and verification of Smart Grid systems. A standard electric power system consists of the generating stations (power plants, wind turbines, solar farms), high-voltage transmission lines, distribution networks and residential/commercial consumption loads [144]. The final distribution segment must step down the distribution voltage to a residential voltage that is safe for use by general consumers. The task of stepping the voltage up/down is performed by a transformer with a tap changer that allows to vary the winding ratio between the primary (input) and secondary (output) voltage. The latest generation of the digitally controlled On-Load Tap Changers (OLTC) allows automatic voltage regulation by varying the transformer ratio under load without interruption.

In this experiment we are concerned with modelling the final distribution segment of an electric power system, in which a distribution voltage of 11kV is converted by an OLTC transformer to a consumption voltage of 230V. The system goal is to maintain the reference voltage of 230V within a predefined deadband (safe range) under any load conditions. One of the means of achieving this is by monitoring the voltage under load and controlling the OLTC position. The system (excluding controller) has been modelled in Modelica (Figure 5.5) using the PowerSystems library[1].

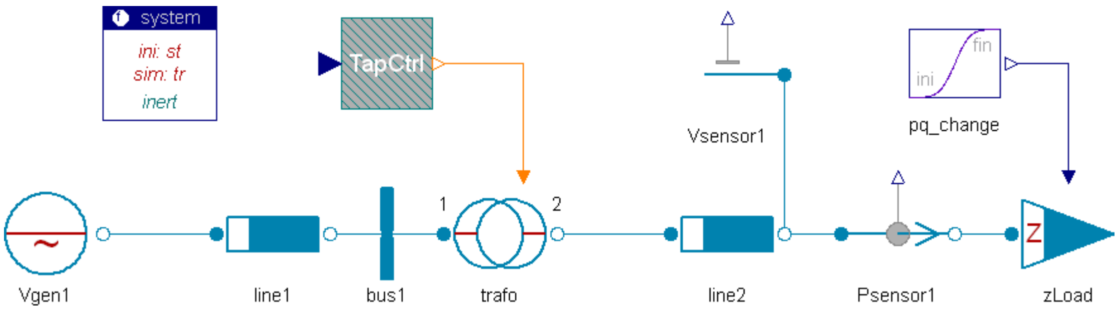


Figure 5.5: Distribution voltage control system in Modelica

The model consists of a constant voltage source *Vgen1* that represents a primary distribution voltage, two power lines split by an OLTC transformer *trafo*, and a load *zLoad* that represents a residential area. For the synthesised simulation scenario the load is set to increase sinusoidally by five times the nominal over a period of 30 seconds, which leads to a corresponding voltage decrease. To regulate the voltage an input control signal can switch the secondary tap of the transformer by providing an index for the tap position. There are 21 positions defined in the transformer, with a 0.2 ratio step between any two consecutive positions. A monitored voltage from the voltage sensor *Vsensor1* is sent back to the controller.

[1] https://github.com/modelica/PowerSystems

The OLTC controller has been modelled in Event-B as a state machine (Figure 5.6) according to [109]. The state machine consists of three states:

- *sIdle*, denoting a normal operation mode where a monitored voltage is within the deadband. The *noChange* event indicates the end of simulation step in this mode.
- *sCount*, denoting a state where the voltage is outside of the deadband, but no tap change action is yet taken. This mode models a detection delay of the OLTC that monitors the voltage for a certain amount of time in hope it goes back to normal (transition *cancelCount*) before taking any action. The delay is modelled in Event-B by a decreasing counter variable *dCounter*. The *delayAction* event is the *EndStep* event in this mode.
- *sAction*, which models a mechanical delay of the tap changer after detection delay expires (transition *startAction*). After another counter variable *mCounter*, involved in this state, reaches zero and depending on the sign of the difference between the reference and monitored voltage a corresponding tap step up/down action is performed (transition *tapUp* or *tapDown*, respectively). The *EndStep* event in this mode is *delayChange*.

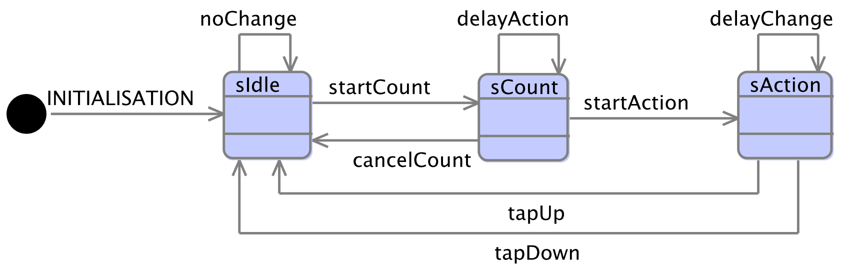


Figure 5.6: Event-B state machine of the OLTC controller

The co-simulation configuration of the OLTC/controller has a detection delay of 10s, a mechanical delay of 1s and a deadband of 2V. The step period of the controller was set to 0.1s and the simulation run for 30s to observe the voltage drop and the reaction of the tap changer, shown in Figure 5.7.

The results show that controller detects a deviation from the deadband ($vNorm < 229$V) at $t = 22$s and switches to the *sCount* state. As voltage continues to stay outside the acceptance range for 10s an action is taken at $t = 32$s and is completed after a mechanical delay of 1s. The tap position changes from 11 (middle position denoting the nominal ratio) to 12, i.e. a tap up is performed to step up the secondary voltage. As a result the voltage jumps to 231.5V at $t = 34$s and becomes outside of the range, but goes back to normal as the load continues to increase.

A more interesting scenario is a number of different factors affecting the voltage (line drop, distribution generation, etc.), as well as a more complex model of the residential load and an intelligent OLTC control algorithm that optimises the number of tap

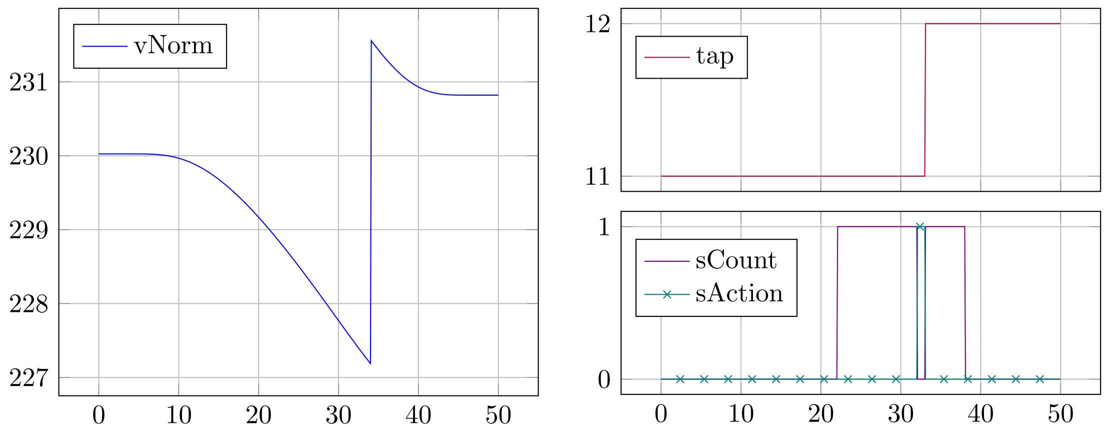


Figure 5.7: Simulation results of the OLTC voltage control (simulation time = 50s, step size = 1s)

changes to minimise the wear of expensive equipment. In fact, MultiSim extension was successfully used in a similar case study done by Critical Software for the ADVANCE project [23]. The case study, provided for the project by Selex ES[2], looked into an automated low voltage control and distribution network of a real residential area, with a dual goal to explore the application of formal methods on the development of smart grid systems and investigate the increased level of automation in a network with distributed micro-generation (photovoltaic cells). The system under study incorporated a voltage optimisation algorithm for the OLTC transformer control, a number of Sensor Interface Units (SIU) that monitor the voltage at various network points and send reports to the control system, and a network communication protocol for transferring the reports. In addition to the voltage regulating function the control algorithm optimises the number of tap changes made in order to maximise the lifetime of the tap changer equipment. A symbolic diagram of the system architecture, borrowed from the ADVANCE deliverable [22], is shown in Figure 5.8.

The control system including the algorithm and communication network was modelled and verified in Event-B, whilst the power network and electrical equipment was designed in Modelica. The developed framework consisting of the Rodin toolset, ProB, MultiSim and visualisation has been used to assess the architecture and protocols of the system, and to identify any counterexamples that violated the key system properties, such as:

- The controller never issues an unsafe command which lowers the voltage when it is already too low.
- The controller never issues an unsafe command which raises the voltage when it is already too high.
- The controller avoids unnecessary tap changes.

[2]http://www.selex-es.com

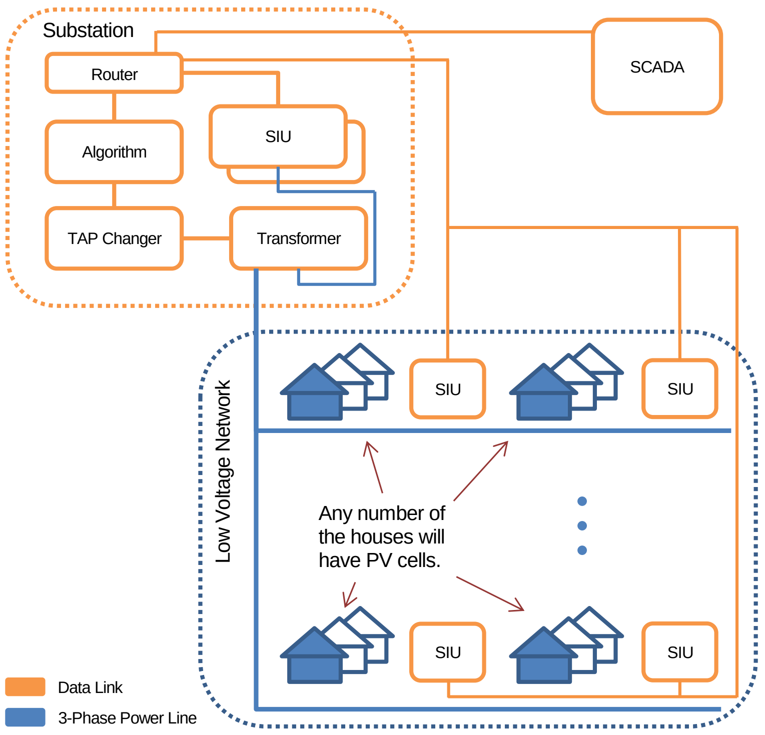


Figure 5.8: Selex ES Smart Grid Network Architecture

In addition, the Event-B model of the control algorithm was verified by applying the System-Theoretic Process Analysis (STPA) approach [95], which involves an exhaustive analysis of all possible actions that a system can emit and potentially cause the violation of safety properties. For each discovered hazard according to STPA a system level constraint is identified that prevent the hazard from occurring. These constraints and violation scenarios are used for identifying test cases. The requirements and constraints of the case study system were formulated in Event-B and used as verification conditions. Formal proof activities on the resulting model helped to identify a number of issues, for example, a significant difference in busbar and target voltages due to the operational characteristics of the tap changer. The algorithm had to be modified accordingly to mitigate the issue, which emphasised the necessity of formal verification.

Co-simulation of the composed system with MultiSim was performed on a fixed network topology model, provided by Selex ES and based on one of their test sites, hence the results of simulation could be compared to the real test measurement data. The continuous model of environment was extended to include a parameterised stochastic model of the network failures and a mutable model of the end-user properties, such as the amount of photovoltaic distribution and the medium voltage input within the network. Simulation with mutation of these properties has helped to explain how the system functions with different degrees of communication problems and to find an acceptable level of packet loss that can still maintain a stable voltage of the network. It has also helped to investigate the response of the system to different levels of power demand and micro-generation.

5.3 Angle of Attack Processing

The next test case is concerned with a common problem of ensuring the fault tolerance in safety-critical systems. Failures in such systems are extremely undesirable as they can lead to dangerous or even fatal incidents like plane crashes. Despite the enormous effort of engineers such systems occasionally do fail due to various internal and external factors. Hence, critical systems often must be fault tolerant[3].

One of the most prevalent techniques for reducing the impact of failures in safety-critical systems is to introduce the redundancy – either static, in which the computation is duplicated and some sort of a voting mechanism is used to determine the result, or dynamic, where the backup resources take over only after a failure of the primary system is detected [73]. Unfortunately, the introduction of fault-tolerance mechanisms inevitably adds complexity to the original design, which can result in a likelihood of failures. In fact, mechanisms of fault tolerance often become the leading cause of failures [123].

In this experiment we explore the application of formal methods and co-simulation in the validation of a representative example of a fault tolerant safety-critical system – an electronic flight control system (EFCS) of the Airbus A330/A340 aircraft. The choice of the object of this case study was driven by the availability of the official documentation and system specification, as well as actual existence of identified design issues that have led to a flight accident on 7 October 2008, when an Airbus A330-303 passenger aircraft on a scheduled Qantas 72 (VH-QPA) flight from Singapore to Perth suffered two sudden uncommanded pitch-down manoeuvres that resulted in serious injuries to passengers and the crew. The following Aviation Occurrence Investigation [40] has revealed that while the aircraft was cruising at 37,000 ft, one of its three air data inertial reference units (ADIRUs) started outputting incorrect values on all flight parameters to other aircraft systems. In response to spikes in the angle of attack (AOA) data the flight control primary computers (FCPCs) commanded the aircraft to pitch down. Despite the fact that the FCPC algorithm utilised for processing AOA data was very effective and included redundancy mechanisms, thorough simulation and analysis revealed that it could not manage a specific scenario of multiple spikes in AOA signal from one ADIRU that were 1.2 seconds apart.

This case is interesting as it demonstrates how a redundancy mechanism itself can be accountable for system failure. We focus on modelling and verifying AOA redundancy logic as well as on reproducing the above failure scenario according to the publicly available VH-QPA recorded flight data. Four scenarios, including the accident, are modelled and simulated in MultiSim by a components diagram in Figure 5.9, which

[3]The term 'fault tolerance' refers to a system's ability to maintain its functionality in the presence of faults. Various techniques for fault-tolerance are employed extensively in the design of hardware and software for safety-critical systems. The fundamental assumption is that faults can never be completely eliminated, but their probability and consequences can be brought to an acceptable level [40].

consists of the Event-B model *mch2* of the AOA monitoring and control logic and the Modelica model *A330AOA.FMI.ScenarioX* of the environment, corresponding to a particular scenario. A *Display* component was added for the output signal visualisation.

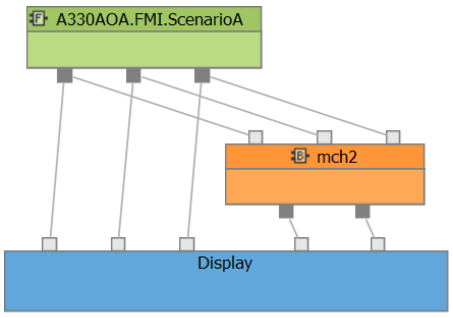


Figure 5.9: MultiSim co-model of the AOA processing test scenario

First, we give a high-level description of the EFCS architecture and the AOA sensor processing logic of the A330 aircraft, which is then modelled and simulated using MultiSim to confirm that its behaviour corresponds to system requirements, followed by synthesising the pitch-down failure scenario. The aim is to demonstrate the applicability of our multi-simulation approach to safety requirements verification and validation of fault-tolerant designs of heterogeneous systems, which consist of highly integrated continuous-time and discrete-time components.

EFCS is a 'fly-by-wire' system that controls the aircraft flight path and attitude by sending movement commands from the flight crew's controls to hydraulic actuators of control surfaces via electrical signals. It consists of several control computers, one of which works as a 'master' and computes control orders according to either normal, alternate or direct control law. In the normal law computers override the flight crew's input to prevent exceeding a safe flight envelope (this includes high AOA and pitch attitude protection). If a failure is detected, the system automatically switches to a different law. The air data inertial reference system (ADIRS) provides vital data about the outside environment and the state of aircraft relative to the outside air and the Earth. For redundancy the ADIRS has three ADIRUs that receive AOA[4] data directly from sensors. EFCS uses this data to control the aircraft's pitch and under normal law, in which EFCS has full control over flight path, protects against a stall by limiting the AOA and sending pitch-down movement commands to aircraft's elevators if a certain angle threshold is exceeded. It is critical that flight data coming from the ADIRUs is accurate.

[4]AOA is the angle between the oncoming air and the wing's reference line. It should not be confused with the pitch angle, which is an angle between the airplane's body and the horizon. A 'stall angle' is a critical AOA angle at which the amount of lift starts to reduce.

The AOA monitoring and computation logic in its simplified form is shown as a flowchart in Figure 5.10 from [40]. Parameters from ADIRUs are monitored as follows:

- FCPC monitors three ADIRUs's output every 40 msec (25 Hz).
- If an ADIRU flags its parameter data as invalid in terms of sign/status matrix (SSM), FCPC ignores it.
- FCPC compares three ADIRUs' parameter values for consistency: if any of the values deviate from the median (of all three) by more than a predefined threshold for over 1 second, the relevant parameter is rejected for the remainder of the flight.

In addition to sensor monitoring the FCPC computes flight control orders, for which it uses valid parameter values from ADIRUs. A common technique is to use the median of all values to ensure that notable deviation in one value would not affect system performance. However, due to the physical position of the sensors (AOA 1 is located on the left side of the fuselage, while AOA 2 and AOA 3 are close together on the right side) the FCPC does not use the median to eliminate the possibility of rejecting a correct value from AOA 1 when both AOA 2 and AOA 3 deviate in a consistent manner, for instance, in an aircraft sideslip situation. To minimise this effect the FCPC uses the average of AOA 1 and AOA 2 and a mechanism to prevent discrepancies in either of the two from influencing the resulting $AOA_{FCPCinput}$ value. The computation logic is as follows:

- If both AOA 1 and AOA 2 are valid, their mean value is used for $AOA_{FCPCinput}$.
- $AOA_{FCPCinput}$ is rate limited to protect from rapid value changes due to short-term anomalies such as turbulence.
- If either AOA 1 or AOA 2 deviate from the median by a 'monitoring threshold', the most recent valid $AOA_{FCPCinput}$ value is memorised and used for 1.2 seconds. During this period the current values of AOA 1 and AOA 2 are not used.
- At the end of a memorisation period the FCPC uses the average of current AOA 1 and AOA 2 values for one sample without applying a rate limiter.
- The monitoring process described earlier happens at the same time as the calculation of $AOA_{FCPCinput}$; therefore, if one of two involved ADRs (air data reference) is rejected as faulty, the average is calculated from AOA 3 and the remaining AOA for all subsequent computations.

Although this AOA processing algorithm is unique to the A330/A340 aircraft, the principle behind it is traditional to aeronautics industry and is known as dissimilarity of architecture in order to avoid a common point of failure. The command and monitoring

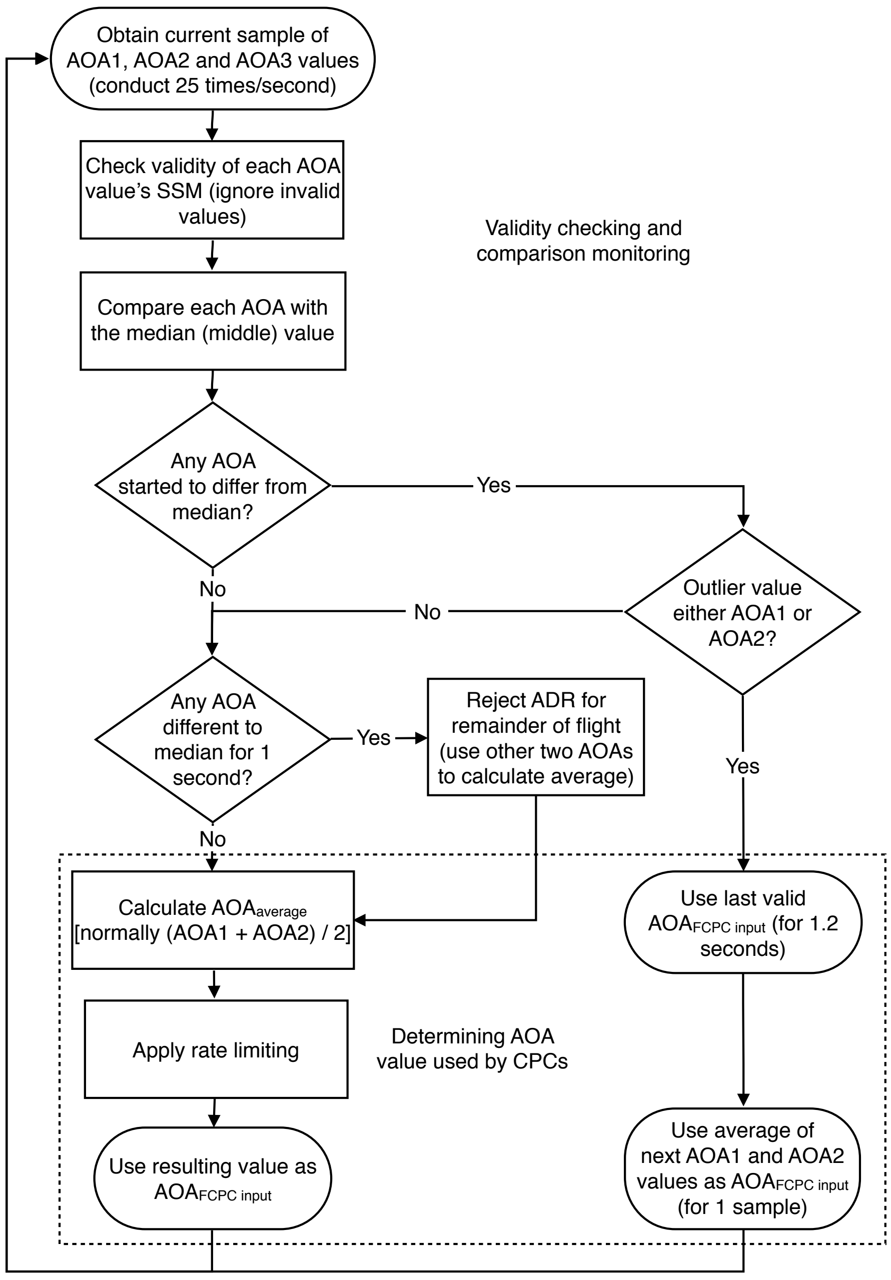

Obtain current sample of AOA1, AOA2 and AOA3 values (conduct 25 times/second)
Check validity of each AOA value's SSM (ignore invalid values)
Compare each AOA with the median (middle) value
Validity checking and comparison monitoring
Any AOA started to differ from median?
Yes
No
Outlier value either AOA1 or AOA2?
No
Yes
Any AOA different to median for 1 second?
Yes
No
Reject ADR for remainder of flight (use other two AOAs to calculate average)
Calculate AOAaverage [normally (AOA1 + AOA2) / 2]
Use last valid AOAFCPC input (for 1.2 seconds)
Apply rate limiting
Determining AOA value used by CPCs
Use resulting value as AOAFCPC input
Use average of next AOA1 and AOA2 values as AOAFCPC input (for 1 sample)

Figure 5.10: Angle of attack processing algorithm

unit architecture is a widely used example of this technique [135], in which all input and output data is permanently monitored while the command unit (COM) sends command data to actuators under control of the monitoring unit (MON). This dual behaviour, present in the A330 design, is of interest in the application of co-simulation between Event-B and continuous-time models. For this example the environment and the command data generation was modelled in Modelica (Figure 5.11) whereas the monitoring unit and control of the output data was modelled in Event-B.

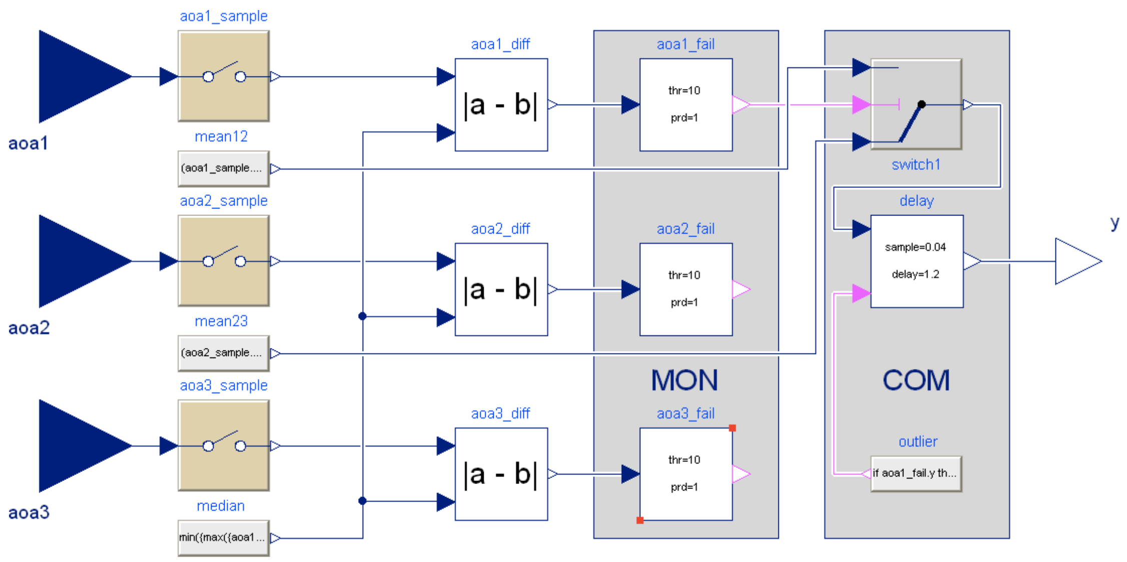


Figure 5.11: Angle of attack sensor processing model in Modelica

The Modelica model consists of three AOA sensor data inputs that are sampled at 25 Hz using Sampler blocks, followed by difference blocks that calculate the deviation of the sampled signal from the median. The logic part of the algorithm is the *MON* and *COM* units that determine sensor failures and the output AOA_{FCPC} value used for control orders, respectively. To simplify the formal model the arithmetic of calculating the median and mean values is shifted outside of the logic blocks. The rate limiter is also omitted (for the following scenarios this omission does not contribute substantial discrepancy in the output). Finally and importantly, AOA 2 and AOA 3 are assumed to be valid and constant during all scenarios; therefore only AOA 1 value fluctuations are considered by the logic.

MON and *COM* modules that are outlined in Figure 5.11 are modelled in Event-B using a state machine plug-in for Rodin [131]. A single control state is composed of two parallel regions (Figure 5.12), one for each mentioned module:

- The top region of the `AOA` state defines the control logic, with a `Normal` state that generates either a nominal (both sensors are valid) or fallback (one sensor is rejected) output, and a `Memorised` state that models the memorisation phase.
- The bottom region specifies the monitoring logic and consists of a `Valid` state (both sensor values are valid), `Invalid` state (sensor value exceeds a threshold) and `Rejected` state (sensor value exceeded a threshold for 1 second, hence it got rejected). Multiple events elaborate a number of transitions with distinct guards and are synchronised across both parallel regions.

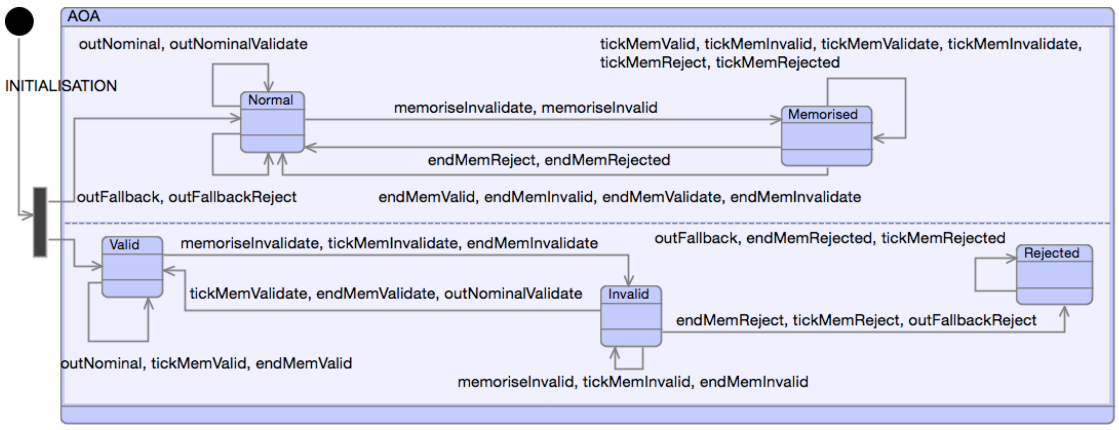

AOA
INITIALISATION
outNominal, outNominalValidate
Normal
memoriseInvalidate, memoriseInvalid
Memorised
tickMemValid, tickMemInvalid, tickMemValidate, tickMemInvalidate, tickMemReject, tickMemRejected
endMemReject, endMemRejected
outFallback, outFallbackReject
endMemValid, endMemInvalid, endMemValidate, endMemInvalidate
Valid
memoriseInvalidate, tickMemInvalidate, endMemInvalidate
outFallback, endMemRejected, tickMemRejected
Rejected
tickMemValidate, endMemValidate, outNominalValidate
Invalid
endMemReject, tickMemReject, outFallbackReject
outNominal, tickMemValid, endMemValid
memoriseInvalid, tickMemInvalid, endMemInvalid

Figure 5.12: State machine of the Event-B AOA controller

Parameter values for the monitoring threshold (`THRESHOLD`), memorisation period of 1.2 seconds (`MEMPERIOD`) and validation period of 1 second (`VALPERIOD`) are specified in Event-B context. Input data from the plant (Modelica) is modelled in Event-B by three variables: `delta1` (deviation of AOA 1 from the median), `mean12` (mean value between AOA 1 and AOA 2, used if both are valid) and `mean23` (mean between AOA 2 and AOA 3, used if AOA 1 is rejected). Inputs are read by a `readInputs` event when controller is in `Input` state. In `Control` state the resulting AOA value is produced and stored in the `aoaFCPC` variable, and the last valid value is stored in `memMean`. The memorisation phase as well as validation phase are modelled using clock variables `memClk` and `valClk`, accordingly, which are set to a period's value on entering the corresponding phase's state and decreased by one of tick events. As you may notice from Figure 5.12 the same events appear in both state machines because they model a composed state space across both parallel regions. The event naming convention here is the name of action/state of the top region + the name of action/state of the bottom region. For example, an event that models a transition from a normal/valid operation to a memorised validation phase when AOA 1 exceeds a threshold looks as follows:

```
event $memoriseInvalidate$
  where $Control = TRUE \wedge Normal = TRUE \wedge Valid = TRUE \wedge delta1 > THRESHOLD$
  then $Control := FALSE \parallel Input := TRUE$
   $Nornal := FALSE \parallel Memorised := TRUE$
   $Valid := FALSE \parallel Invalid := TRUE$
   $memClk := MEMPERIOD - 1 \parallel valClk := VALPERIOD - 1$ // set the clocks; do 1 tick
   $aoaFCPC := memMean$ // use last valid mean for memorisation
  end
```

Tick events count down both clocks to zero or until the input becomes valid, as in the following event that models the transition of the monitor back to the valid state while memorisation is still active:

```
**event** $tickMemValidate$
  **where** $Control = TRUE \wedge Memorised = TRUE \wedge Invalid = TRUE$
    $delta1 \leq THRESHOLD \wedge memClk > 0$
  **then** $Control := FALSE \parallel Input := TRUE$
    $Invalid := FALSE \parallel Valid := TRUE$
    $memClk := memClk - 1$ // decrease memorisation clock
  **end**
```

When one of the clocks reaches zero, the current state is switched. For example, if the memorisation period expires while the validation is still on (AOA 1 has been exceeding the threshold for less than 1 second), COM state is switched back to `Normal` (this case models the 1-sample rule at the end of memorisation from Figure 5.10):

```
**event** $endMemInvalid$
  **where** $Control = TRUE \wedge Memorised = TRUE \wedge Invalid = TRUE$
    $delta1 > THRESHOLD \wedge memClk = 0 \wedge valClk > 0$
  **then** $Control := FALSE \parallel Input := TRUE$
    $Memorised := FALSE \parallel Normal := TRUE$
    $valClk := valClk - 1$ // decrease validation clock
    $aoaFCPC := mean12$ // use current mean without validation
    $memMean := mean12$ // memorise as the last valid output
  **end**
```

Before validating the composed model via multi-simulation, the Event-B model can be validated by ProB and verified using Rodin provers. For the formal analysis we have to add invariants that we want to prove. One of the properties that is apparent from the specification is that if a sensor AOA 1 is rejected, the alternative mean value between AOA 2 and AOA 3 should be used for AOA_{FCPC}, unless the memorisation mode is active. This predicate is expressed as follows:

$$Rejected = TRUE \wedge Input = TRUE \Rightarrow aoaFCPC = mean23 \vee Memorised = TRUE \tag{5.4}$$

and is automatically proven by Rodin provers. The related property that we might want to prove is that if controller is in the memorisation mode, AOA_{FCPC} should still be valid (not exceed a threshold):

$$Memorised = TRUE \Rightarrow aoaFCPC \leq THRESHOLD \tag{5.5}$$

This invariant, however, cannot be proven automatically since it requires the goal $memMean \leq THRESHOLD$ to be proven, which is not possible due to the 1-sample requirement (see the `endMemInvalid` event). Failure of formal proof identifies inconsistency in the requirements. This inconsistency can also be identified via model-checking in ProB: the invariant preservation check discovers a counterexample that invalidates invariant 5.5 on the execution of `invalidateMemorised` when $memMean = 11$.

With multi-simulation we can also determine the actual chain of events that leads to an invalid state. The investigation report [40] gives details of four scenarios that demonstrate how the FCPC algorithm can detect and manage incorrect or inconsistent AOA data. We successfully simulate all four scenarios on a combined model of the plant in Modelica and the algorithm in Event-B. Event-B component has a step period set to 40 ms, equal to the sampling rate of FCPC. The controller takes three inputs from environment: *delta1*, *mean12* and *mean23*, and produces two outputs: *aoaFCPC* and *Rejected* (failure flag of AOA 1).

Scenario A illustrates how a significant step-change in AOA 1 would have no effect on $AOA_{FCPCinput}$, as it would trigger a memorisation period of 1.2 seconds. If the change lasts for more than 1 second, sensor AOA 1 gets rejected and the mean of AOA 2 and AOA 3 is used afterwards (Figure 5.13, A).

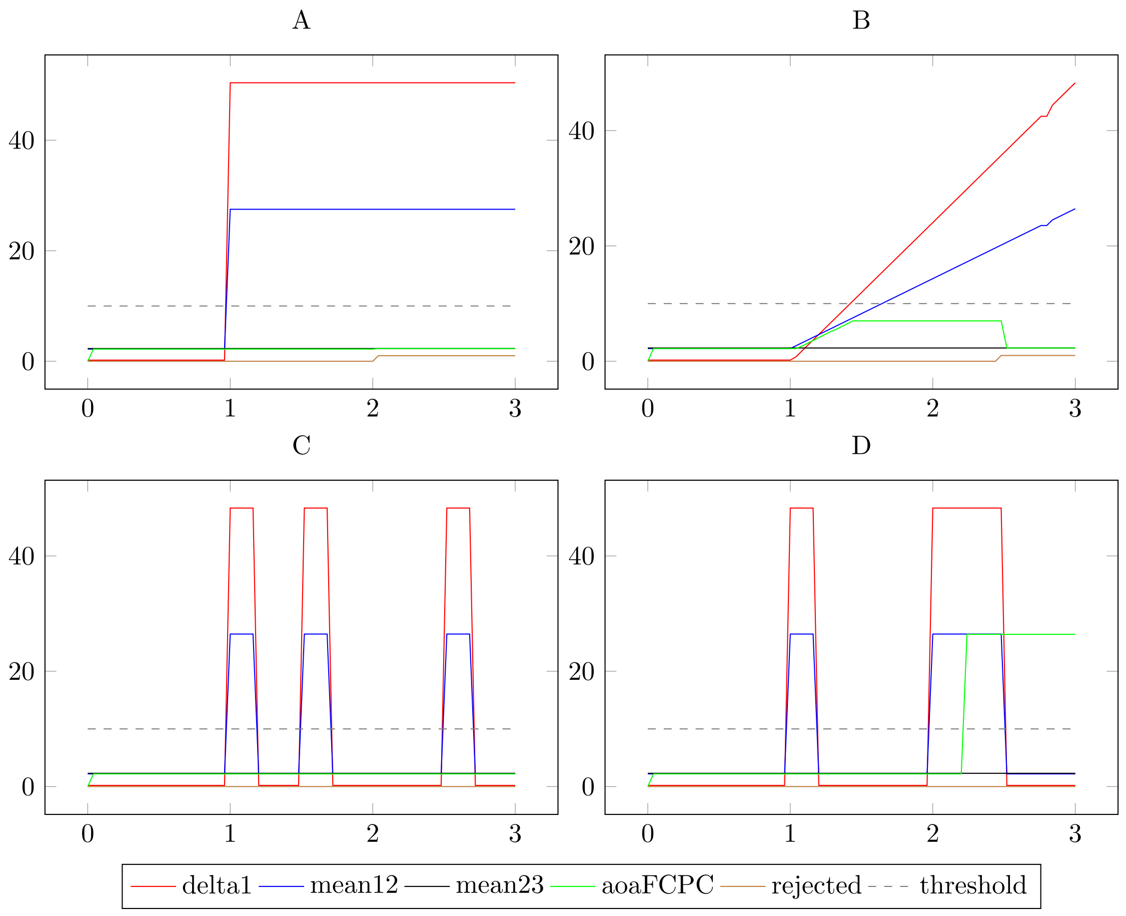


Figure 5.13: Co-simulation of the AOA system in MultiSim (simulation time = 3 s, step size = 40 ms)

Scenario B deals with a runaway (consistently increasing value) of AOA 1 (Figure 5.13, B), which would trigger a 1.2-seconds memorisation period when the value reached the monitoring threshold. After the memorisation period the mean of AOA 2 and AOA 3 is used. Thus, the runaway has only a minor effect on $AOA_{FCPCinput}$ prior the memorisation period. The rate limiter would further mitigate its effect.

Scenario C deals with short-duration spikes. If a short spike is detected in AOA 1, it triggers a memorisation period, at the end of which the current mean is used again for $AOA_{FCPCinput}$. If another spike occurred during the memorisation period, it would not trigger additional memorisation. On the other hand, the occurrence of a spike at the end of the memorisation period would trigger a new memorisation (Figure 5.13, C).

Scenario D was identified by the aircraft manufacturer after the investigation of the accident. Simulations showed that deviation of a single sensor value by two short spikes 1.2 seconds apart could significantly influence the output $AOA_{FCPCinput}$ and potentially contribute to uncontrolled pitch-down manoeuvre. The following scenario is synthesised from the recorder VH-QPA flight data:

- Prior the spikes the value of AOA 1 is 2.1°, whilst the value of both AOA 2 and AOA 3 is 2.3°. $AOA_{FCPCinput}$ is based on the mean between AOA 1 and AOA 2.
- The first spike in AOA 1 has a magnitude of 50.6°, lasting less than a second. It triggers a 1.2-second memorisation period.
- At the end of the memorisation period a second spike of 50.6° int AOA 1 is present (1.2 seconds after the first spike). At this point, according to the algorithm, the values of AOA 1 and AOA 2 are assumed to be valid and a sample of their mean is used to calculate $AOA_{FCPCinput}$, causing a step change of 26° in the latter.
- After the initial sample the logic goes back to normal operating mode, but the value of $AOA_{FCPCinput}$ remains affected by the spike.

The last scenario was identified by the manufacturer during the development of FCPC, but assumed to be highly improbable. The investigation showed that it was one of the main causes of the accident: the incorrect value of $AOA_{FCPCinput}$ was within the range of -10° to +30° and therefore retained the system under normal law, which resulted in two flight envelope mechanisms being activated. The combined effect of both stall prevention and recovery was confirmed to correspond to a recorded elevator deflection. The manufacturer subsequently redesigned the algorithm to prevent this type of faults from occurring again.

Simulation using MultiSim allowed us to execute all four scenarios and validate our formal model of the algorithm agains its specification. Each scenario was synthesised by modelling the sample data from three sensors. In our case the data was available from the investigation report, although theoretically it could be derived from a model-checking counterexample. In the case of the accident scenario the application of MultiSim helped to determine a sequence of events that leads to violation of system safety properties (invariant 5.5). Such an approach is helpful at pinpointing the exact event/requirement, which brings the system to invalid state or deadlock. At the same time, the reverse approach is also beneficial – Rodin tool and ProB model checker can investigate a bounded state space for invariant violations, thus discovering the preconditions and input values, which can then be used for synthesising a simulation scenario that demonstrates system failure.

5.4 Landing Gear System

The MultisSim tool has been also exercised on a landing gear case study proposed as a benchmark for verification tools and techniques at the case study track of the ABZ 2014 conference [28]. This case study presents an aircraft landing system that is responsible for manoeuvring the landing gears and associated doors of an aircraft on pilot request. The system is a representative example of a critical embedded hybrid system in which the actions of the digital controller are dependent on the state of physical devices and their temporal behaviour. The task of the case study was firstly to model the software part that controls extension and retraction sequences of the gears, and secondly to prove safety properties associated with physical behaviour and system state. As we demonstrate below, such a system maps well to the co-modelling and co-simulation framework of the Rodin toolset.

The landing system consists of multiple analog and digital components that are grouped into three parts:

- A mechanical part comprising all mechanical devices and three landing sets (front, left and right) that contain a door, a landing gear and hydraulic cylinders for their movement.
- A digital controller with control software.
- A pilot cockpit interface

The pilot interface is the simplest part of the system, consisting of a two-position input handle and a set of light indicators of the system status. The handle is used by the pilot to issue gear extension and retraction orders to the system. The *up* position of the handle executes gear retraction, whereas the *down* position denotes gear extension. The lights provide the pilot with a visual indicator of the current position of the gears (locked up/down or manoeuvring) and the current health status of the system and its equipment.

The mechanical part consists of three landing sets, each comprising a landing gear up-lock box and a door with latching boxes in the closed position. The manoeuvring of gears and doors is executed by a set of actuating hydraulic cylinders: one cylinder for opening and closing a door and another cylinder for extending and retracting a gear. The hydraulic power is provided to cylinders via a circuit of connected electro-valves. One general electro-valve provides pressure to four manoeuvring electro-valves: for door opening, door closing, gear extension and gear retraction. Each electro-valve is activated by an electrical control order from the digital part. To protect the system from abnormal activation of electro-valves by the digital controller the control order to the general electro-valve additionally goes through an analogue switch (explained below). Finally, a set of discrete sensors indicating each gear's and door's state (locked/unlocked), gear shock absorber state (on ground/in flight), pressure in hydraulic circuit and the analogue switch state (open/closed) feeds the controller with information about the state of equipment. Each sensor is triplicated for redundancy, with a majority-voting mechanism employed to calculate the output value. The architecture of the system is illustrated in Figure 5.14 from [28].

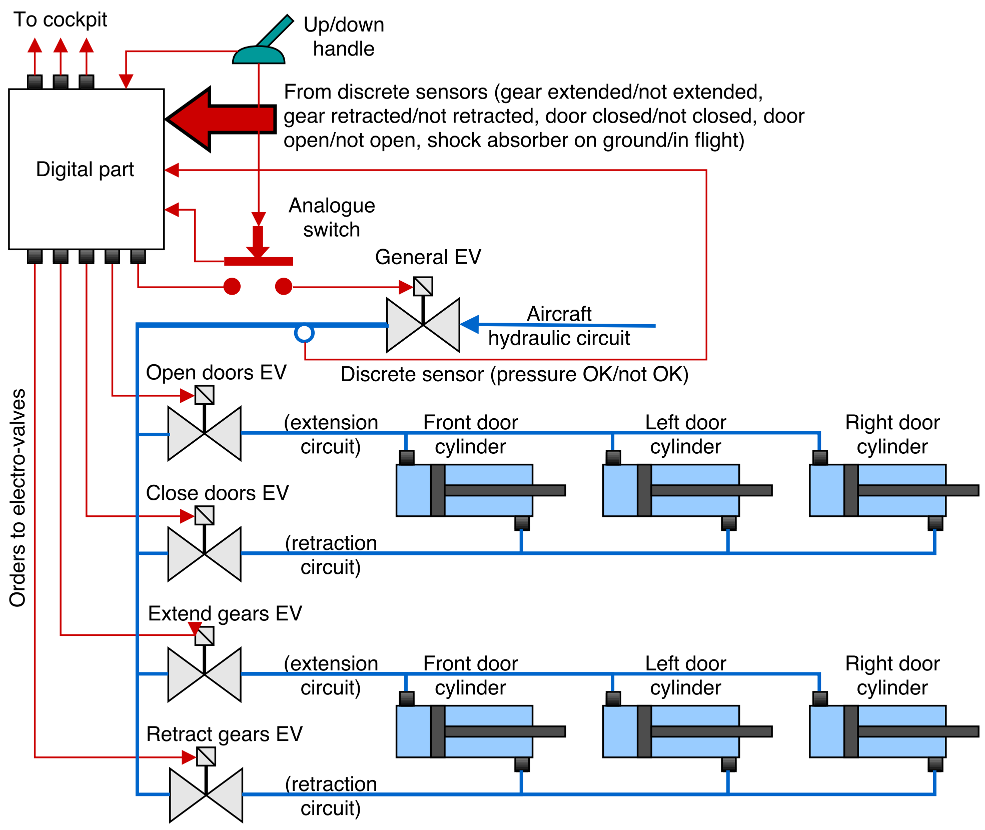


Figure 5.14: Architecture of the landing gear system

As explained earlier, the purpose of the analogue switch is to protect the system from abnormal orders coming from controller. This is achieved by the design of the switch, which passes a digital input only in a closed state. The switch is initially open and it can be closed via an analogue input change. When the switch gets closed, it remains closed for 20 seconds, unless a change is detected again. For inertial reasons the transition from the open to closed state takes 0.8 seconds, and from closed to open – 1.2 seconds. The design of the switch is represented by a hybrid automaton in Figure 5.15. The digital input *in* of the switch is connected to the general electro-valve order of controller, while the analogue input *handle* is connected to the cockpit handle. Whenever a handle movement is detected the analogue switch becomes closed and the general electro-valve can be stimulated by controller ($out := in$) for 20 seconds. The switch is an example of the physical system that can be modelled digitally because it maps well to the discrete semantics of a hybrid state machine. However, since it is designed as analogue according to the specification, we model it as part of the physical plant.

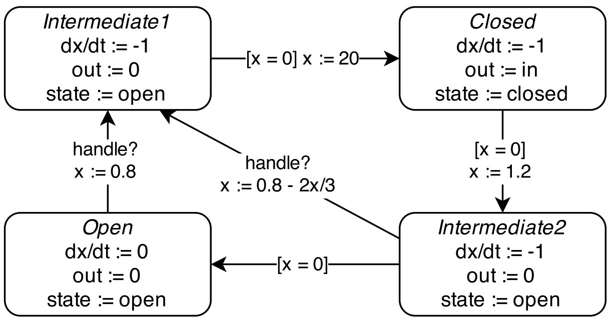


Figure 5.15: Hybrid automaton of the analogue switch

An electro-valve is a hydraulic device with an input pressure port H_{in}, an output pressure port H_{out} and a discrete electrical input E. The behaviour of the electro-valve depends on the electrical order from E: if $E = true$ then $H_{out} = H_{in}$ (maintained as long as E is *true*), otherwise $H_{out} = 0$. Furthermore, due to inertia the output pressure grows continuously from 0 to H_{in} when the valve becomes open. Similarly, pressure declines continuously from H_{in} to 0 when the valve gets closed. In the given model this dynamic behaviour is assumed to be linear, taking 1 second for the opening transition and 3.6 seconds for closing transition. All electro-valves in the system are supposed to demonstrate the same behaviour.

A cylinder is a pure hydraulic device with a moving piston. An opening pressure port on one end of the cylinder and a closing port on the other end provide the received hydraulic pressure into corresponding chamber, thus moving the piston. In addition, cylinders used in the landing system have a latching box mechanism on one or both ends that can lock the cylinder in the fully open or closed position. When a cylinder is locked the pressure does not need to be provided anymore to sustain it in that position. Consequently, to move the piston of a locked cylinder in the opposite direction the applied pressure has to unlock the latching mechanism first. Gear cylinders have latches on both ends (extended and retracted), while door cylinders are only locked in the closed position, hence maintaining the open door position requires a constant pressure in the extension circuit of the cylinder. Sensor data for the cylinder position is determined by the state of corresponding latching boxes, with the exception of the door open position.

For example, a *door_closed* sensor value is *true* if the closing latching box is locked. The behaviour of the cylinders (and attributed latching boxes) is determined by the current pressure in the hydraulic circuit, the position of the piston and their temporal parameters that are given in Table 5.1. The exact values of the given durations can vary up to 20%.

Table 5.1: Duration of cylinder operations

duration (in seconds)	front gear	front door	right gear	right door	left gear	left door
unlock in open position	0.8	-	0.8	-	0.8	-
from open to closed position	1.6	1.2	2	1.6	2	1.6
lock in closed position	0.4	0.3	0.4	0.3	0.4	0.3
unlock in closed position	0.8	0.4	0.8	0.4	0.8	0.4
from closed to open position	1.2	1.2	1.6	1.5	1.6	1.5
unlock in open position	0.4	-	0.4	-	0.4	-

The digital part is composed of two identical computing modules, which execute in parallel the same controller software that reads sensor data, computes control orders for doors and gears, and informs the pilot via cockpit about system status and any detected anomalies. Both modules receive the same input from sensors (each value is triplicated): *handle*, *analogue_switch*, *gear_extended*, *gear_retracted*, *gear_shock_absorber*, *door_closed*, *door_open*, *circuit_pressurised*. From these inputs each module computes a set of control order outputs: *general_EV* (stimulation of the general electro-valve), *close_EV* (door closure EV), *open_EV* (door opening EV), *extend_EV* (gear extension EV) and *retract_EV* (gear retraction EV), and a set of status indicators for the cockpit: *gears_locked_down*, *gears_maneuvering* and *anomaly*. Same outputs from two modules are composed by a logical OR operation.

Digital control is only used in the nominal operation mode, when no anomalies are present. In the event of failure the system is automatically switched to emergency mode and analogue operation. The given specification only covers the nominal mode, though anomaly detection is part of the control software. Hence, the goal of the controller is the operation of the landing system according to pilot orders and the health monitoring of equipment. The first goal is specified by two basic manoeuvring control sequences: gear extension and gear retraction. The extension sequence starts from the retracted gear and closed door position when a pilot moves the handle down, and involves the following control actions:

1. Stimulate the general electro-valve in order to send the pressure to manoeuvring electro-valves.
2. Stimulate the door opening electro-valve.
3. Once all doors are open stimulate the gear extension electro-valve.
4. Once all gears are locked extended stop stimulating the extension electro-valve.

5. Stop stimulating the door opening electro-valve.
6. Stimulate the door closing electro-valve.
7. When all doors are locked closed stop stimulating the closing electro-valve.
8. Stop stimulating the general electro-valve.

Correspondingly, the retraction sequence starts from the extended gear and closed door position when the cockpit handle is pushed up, and involves these steps:

1. Stimulate the general electro-valve.
2. Stimulate the door opening electro-valve.
3. Once all doors are open and if all gear shock absorbers are relaxed stimulate the gear retraction electro-valve, otherwise go to step 5.
4. Once all gears are locked retracted stop stimulating the retraction electro-valve.
5. Stop stimulating the door opening electro-valve.
6. Stimulate the door closing electro-valve.
7. When all doors are locked closed stop stimulating the closing electro-valve.
8. Stop stimulating the general electro-valve.

Both sequences can be interrupted at any step by an opposing order from the pilot (handle can be switched at any time), in which case the reverse sequence should proceed from the point of interruption. In addition, the control software should satisfy the following timing constrains induced by the inertia of the hydraulic pressure:

- Stimulation orders to the general electro-valve and a manoeuvring electro-valve must be separated by at least 200ms.
- Orders to stop stimulation of the general electro-valve and a manoeuvring electro-valve must be separated by at least 1s.
- Two contrary orders (open and close doors, extend and retract gears) must be separated by at least 100ms.

The second goal of the control software – equipment health monitoring and reporting – involves generic sensor monitoring with triple modular redundancy, analogue switch monitoring, pressure sensor monitoring and door/gear motion monitoring. The monitoring requirements are defined in terms of timing constraints on the expected response from sensors. In case a particular constraint is not met, the controller must issue an anomaly signal to the cockpit and switch the system to emergency operation mode. As an example of such a constraint, controller should consider doors as blocked and issue an anomaly if sensor value $door\_closed = false$ is not detected for all three doors within 7 seconds of the stimulation of the opening electro-valve. This experiment only considers generic monitoring and omits the rest of health monitoring constraints. However, we demonstrate at the end how these constraints could easily be modelled in Event-B.

Our modelling the landing gear system starts with a Modelica model of the plant. We simplify the original specification by omitting sensor triplication and combining similar sensor outputs of each landing set (front, left and right) into a single output. Furthermore, the plant incorporates both the mechanical/hydraulic part and the cockpit handle for simplicity of exporting it into Rodin. Hence, our plant model has 7 outputs: *handle*, *switch*, *pressure*, *closed*, *open*, *retracted* and *extended*. The analogue switch automaton from Figure 5.15 is modelled using the freely available Modelica_StateGraph2[5] library for modelling hierarchical state machines. An extra state *Reclosing* is required to delay the transition back to *Closed* state when a handle signal change is detected whilst the switch is opening (Figure 5.16).

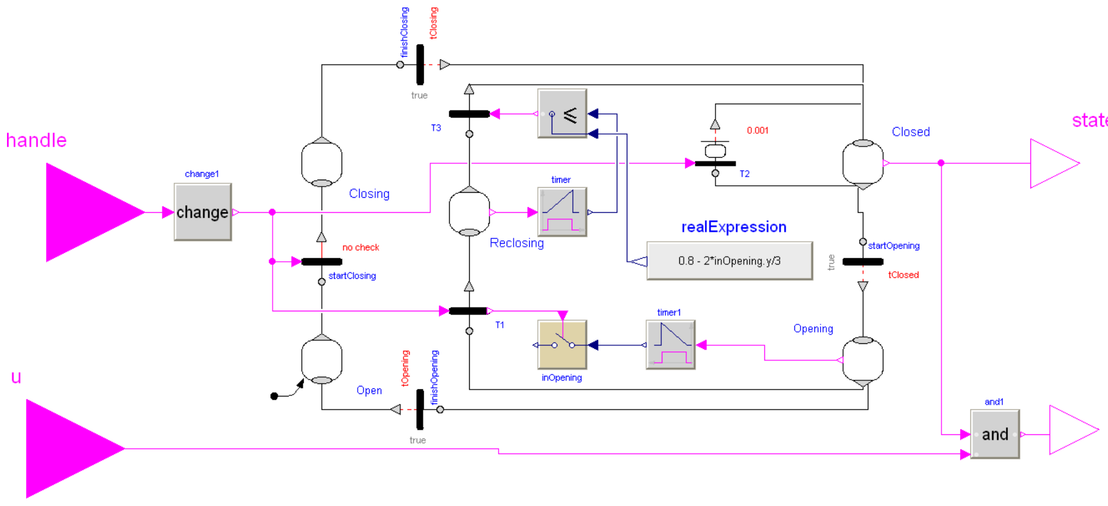


Figure 5.16: Modelica model of the analogue switch

Electro-valves are modelled using a parameterised output ratio R that changes linearly depending on whether the valve is opening or closing. The Modelica model of the valve is given below. We follow the terminology of the original specification, which interprets the open/closed valve as a closed/open circuit. Therefore, an open valve (one that passes the pressure through) is said to be in a closed state.

```
model ElectroValve
  parameter Real closingTime = 1.0 "Closing duration";
  parameter Real openingTime = 3.6 "Opening duration";
protected
```

[5]https://github.com/modelica/Modelica_StateGraph2

```
  parameter Real Rmax = 1.0 "Max opening";
  parameter Real dRcl = Rmax/closingTime "Change rate of R when closing";
  parameter Real dRop = -Rmax/openingTime "Change rate of R when opening";
  Real R(start = 0.0) "Current opening (0-open, 1-closed)";
  discrete Real dR(start = 0.0);
equation
  Hout = Hin*R;
  der(R) = dR;
algorithm
  // closing/opening event
  when E then
    dR := dRcl;
  elsewhen not E then
    dR := dRop;
  end when;
  // limiter of the R value
  when R <= 0 or R >= Rmax then
    dR := 0;
  end when;
end ElectroValve;
```

Hydraulic cylinders are modelled in a similar fashion via a dynamically changing cylinder (piston) position *Pos*, parameterised by the length of cylinder, extension and retraction time. The cylinder contains two optional latching boxes (modelled by *Lock*) at each end, with parameterised locking and unlocking time. The Modelica model of the cylinder with latching boxes is given below:

```
model LatchedCylinder
  Modelica.Blocks.Interfaces.RealInput Hext "Extension pressure input";
  Modelica.Blocks.Interfaces.RealInput Hret "Retraction pressure input";
  Modelica.Blocks.Interfaces.RealOutput Pos(start = 0) "Piston position";
  parameter Real length = 1.0 "Cylinder length";
  parameter Real tExtend = 1.2 "Extension time";
  parameter Real tRetract = 1.6 "Retraction time";
  parameter Boolean extensionLatch = true
    "=true, if cylinder latch in extended position is enabled";
  parameter Boolean retractionLatch = true
    "=true, if cylinder latch in retracted position is enabled";
  parameter Real tLockExtended = 0.4 "Lock time in extended position";
  parameter Real tLockRetracted = 0.4 "Lock time in retracted position";
  parameter Real tUnlockExtended = 0.8 "Unlock time from extended position";
  parameter Real tUnlockRetracted = 0.8 "Unlock time from retracted position";
  parameter Real Hmin = 0.001 "Minimal pressure to move the piston";
  output Boolean lockedRetracted(start = retractionLatch)
    "Status of retraction latch, if enabled";
  output Boolean lockedExtended(start = false)
```

```
    "Status of extension latch, if enabled";
protected
  parameter Real dPosExt = length/tExtend;
  parameter Real dPosRet = -length/tRetract;
public
  Lock extLock(
    initialLock = false,
    lockTime = tLockExtended,
    unlockTime = tUnlockExtended);
  Lock retLock(
    initialLock = true,
    lockTime = tLockRetracted,
    unlockTime = tUnlockRetracted);
... // logical elements for connecting locks
equation
  lockedRetracted = retractionLatch and retLock.locked;
  lockedExtended = extensionLatch and extLock.locked;

  if Hext-Hret >= Hmin and not lockedRetracted then
    der(Pos) = if Pos < length then dPosExt else 0;
  else if Hret-Hext >= Hmin and not lockedExtended then
      der(Pos) = if Pos > 0 then dPosRet else 0;
    else
      der(Pos) = 0;
    end if;
  end if;
... // connect equations for logical elements
end LatchedCylinder;
```

The model of the cockpit handle, which is not shown here due to its simplicity, is a discrete signal source that synthesises a particular scenario of the pilot behaviour with respect to handle movement. The complete model of plant comprising the handle, analogue switch, constant pressure source, general and manoeuvring electro-valves, and six cylinders is available in the Appendix D.

The control software is modelled in Event-B using the State-Machines plug-in for Rodin, presented in [131]. The model is developed in a number of refinement steps, which helps to break down the complexity and verify each iteration with respect to modelled requirements. We also find that the verification process is greatly facilitated by the use of ProB animator tool that helps to execute the model and ensure that it behaves as expected after each refinement step. If modelling with Event-B state machines, one can also use the interactive state machine animator [130], which allows Event-B models specified in terms of state diagrams to be executed by manually picking enabled state transitions and selecting input values; current states and model instances are visually highlighted during the animation, as it is shown in Figure 5.17.

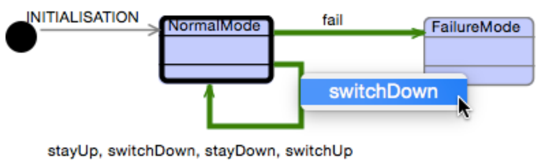


Figure 5.17: Interactive animation of the landing gear control state machine

At the very abstract level the model is only concerned with system operation modes and models the detection of anomalies (states *NormalMode* and *FailureMode* in red in Figure 5.18). It also models the state of the cockpit handle. The boolean input *handle* and a corresponding read/*StartStep* event are introduced at this level, with *handle* = *true* denoting handle down position (gear extension) and *handle* = *false* denoting handle up position (gear retraction). Finally, the abstract level includes a state of the machine to control the simulation. This state can be either *Read* (reading inputs), *Control* (actual control events) or *Failure* (blocking state of the failure mode). Adding simulation semantics to an Event-B machine at this early stage allows us to validate the model of controller using MultiSim even when the specification is very abstract (of course that may require adjusting the co-model of the plant accordingly). Refinement allows us to validate the Event-B model incrementally, so it is useful that co-simulation works at multiple abstraction levels.

The first level of refinement (in green in Figure 5.18) introduces the notion of the general electro-valve control order for gear manoeuvring (state *generalEV*) and two stable states, in which the landing equipment is locked, the gears are either retracted (state *Retracted*) or extended (state *Extended*) and the handle is not moving. Transitions can be taken from a stable state to *generalEV* if a change in the handle position is detected. When in the *generalEV* state, another change of the handle can trigger a cancel action (transitions *cancelExt* and *cancelRet*), modelling the cancellation of initiated extension or retraction sequence (transitions *extend* and *retract*). On successful completion of manoeuvre the control returns from*generalEV* to a stable state (transitions *setExt* and *setRet*).

In the second refinement we add the first timing constraints – the start and stop delay between the simulation of the general electro-valve and manoeuvring electro-valves, which is modelled by state *PendingStart* and *PendingStop* and the abstract state *Maneuvering* (in blue in Figure 5.18). Delays are modelled using clock variables *tStart* and *tStop* and associated constants for 200ms start delay and 1s stop delay in the context. Because the handle can be switched at any time, the stop delay can be intervened to undo the most recent manoeuvre (transitions *undoExt* and *undoRet*).

The third refinement introduces control orders for the door manoeuvring electro-valves *openEV* and *closeEV*, and the *PendingContrDoor* state to model a contrary order delay of 100ms between them (in purple in Figure 5.18). No door sensor inputs are added at this point. Multiple transitions between three new states are the result of possible interrupt by the opposing order from the pilot.

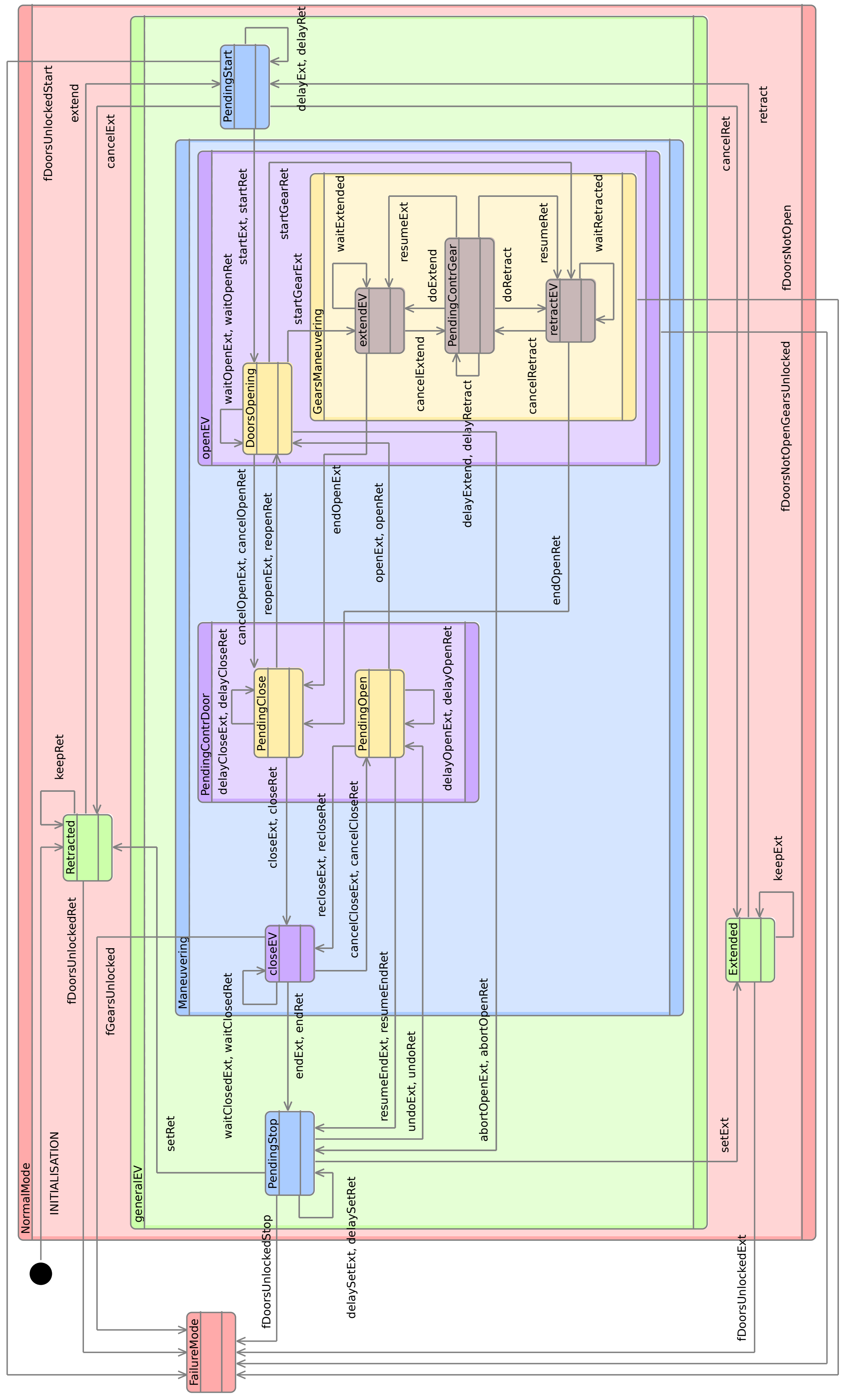


Figure 5.18: Event-B state machine of the landing controller

The fourth refinement adds inputs from the door sensors: *doorsOpen* and *doorsClosed*, which also amends the abstract read event. New states *DoorsOpening* and *GearsManeuvering* (in yellow in Figure 5.18) model the door opening manoeuvre and abstract gear manoeuvre, respectively. Substates , *PendingClose* and *PendingOpen* elaborate the contrary door order delay defending on the current system state and position of the handle. The door closure manoeuvre is modelled by the *closeEV* control state from the previous refinement. Finally, health monitoring of the door sensors is modelled by a set of failure detection events:

- *fDoorsUnlockedRet*, if doors are not locked closed ($doorsClosed = false$) in retracted stable state
- *fDoorsUnlockedExt*, if doors are not locked closed in extended stable state
- *fDoorsUnlockedStart*, if doors are not locked closed before the start of the door opening stimulation
- *fDoorsUnlockedStop*, if doors are not locked closed after the end of the door closure stimulation
- *fDoorsNotOpen*, if doors are not sustained fully open ($doorsOpen = false$) while the gears are manoeuvring

The next refinement level introduces the missing control orders to gear manoeuvring electro-valves: *extendEV* and *retractEV* (in brown in Figure 5.18), and associated gear sensors *gearsExtended* and *gearsRetracted*. A contrary order delay between the gear extension and retraction orders is modelled by a new state *PendingContrGear*. Health monitoring is extended by additional events: *fGearsUnlocked*, if gears are not locked up or down while the doors are closing ($gearsRetracted = false \wedge gearsExtended = false$), and *fDoorsNotOpenGearsUnlocked*, if gears are not locked up or down while the doors are opening ($doorsOpen = false \wedge gearsRetracted = false \wedge gearsExtended = false$). At this point all functional requirements should be fulfilled.

In the last refinement we do not add any new behaviour or states, but introduce the safety invariants to be proved. Although this task could have been started (and ideally should be) at the earlier stages, as soon as necessary information for invariants is available, shifting it to a separate refinement was done intentionally to reduce the number of generated proof obligations in earlier refinements (withal most of the safety properties require gear sensors that were defined last). The first safety property we add states that in the normal operation mode (no detected failures) if the landing gear command handle remains in the DOWN position, then retraction sequence is not observed. This property can be specified as the following invariant, where *Read* is an auxiliary simulation state that is active after each control cycle is completed:

$$Read = TRUE \wedge handle = TRUE \Rightarrow retractEV = FALSE \tag{5.6}$$

A similar property states that in the normal mode if the command handle remains in the UP position, extension sequence is not observed:

$$Read = TRUE \wedge handle = FALSE \Rightarrow extendEV = FALSE \tag{5.7}$$

Another property states that in the normal mode the stimulation of the gears extension or retraction electro-valves can only happen when three doors are locked open. Conversely, the stimulation of the doors opening or closing electro-valves can only happen when three gears are locked down or up. These properties can be split into four separate invariants for each stimulated equipment:

$$Read = TRUE \wedge extendEV = TRUE \Rightarrow doorsOpen = TRUE \tag{5.8a}$$

$$Read = TRUE \wedge retractEV = TRUE \Rightarrow doorsOpen = TRUE \tag{5.8b}$$

$$\begin{gathered} Read = TRUE \wedge closeEV = TRUE \Rightarrow \\ gearsRetracted = TRUE \vee gearsExtended = TRUE \end{gathered} \tag{5.8c}$$

$$\begin{gathered} Read = TRUE \wedge openEV = TRUE \wedge doorsOpen \neq TRUE \Rightarrow \\ gearsRetracted = TRUE \vee gearsExtended = TRUE \end{gathered} \tag{5.8d}$$

The next property is about with contrary orders. It states that in the normal mode door opening and door closure electro-valves are not stimulated simultaneously. Conversely, gear extension and gear retraction electro-valves are not stimulated simultaneously:

$$\neg(openEV = TRUE \wedge closeEV = TRUE) \tag{5.9a}$$

$$\neg(retractEV = TRUE \wedge extendEV = TRUE) \tag{5.9b}$$

The final property we add relates manoeuvring electro-valves to the general electro-valve. It states that in the normal mode it is not possible to stimulate the manoeuvring electro-valves (opening, closure, extension or retraction) without stimulating the general electro-valve:

$$openEV = TRUE \Rightarrow generalEV = TRUE \tag{5.10a}$$

$$closeEV = TRUE \Rightarrow generalEV = TRUE \tag{5.10b}$$

$$extendEV = TRUE \Rightarrow generalEV = TRUE \tag{5.10c}$$

$$retractEV = TRUE \Rightarrow generalEV = TRUE \tag{5.10d}$$

All the specified invariants were successfully proven by the Rodin automatic provers, in part thanks to the properly selected model design. For instance, modelling the contrary manoeuvring orders as orthogonal states ensured that they can never be active at the same time (invariant 5.9). Our modelling process, however, was not completely top-down, and the final design was the result of rethinking a few unsuccessful designs that made the invariants hard to prove. This shows how important it is to make the right design decisions early when modelling a system in Event-B. From this experience our recommendation for developers would be to think ahead about the modelled system from the requirements point of view. A good approach is to start by formulating the requirements and constraints as abstract invariants, e.g. $handle = TRUE \Rightarrow retractEV = FALSE$ for no observed retraction in the handle down position, and then to incorporate those invariants into the model design. The design decisions should take into account how a corresponding invariant would be proven. The closer the design reflects the invariants, the easier the latter are to prove, as in the case of the invariants in Equation 5.9 on contrary control orders that are trivial to prove when the orders are modelled by orthogonal states.

The final safety property of the normal mode, given in the original specification, is a timing constraint: if the landing gear command handle has been pushed DOWN (UP) and stays DOWN (UP), then the gears will be locked down (up) and the doors will be seen closed less than 15 seconds after the handle has been pushed. We did not add this invariant since it required a change in the abstract model – another proof of the importance of thinking ahead about the design derived from requirements. The invariant could be incorporated into the model using a clock counter variable on the handle state. Specifically, in the abstract model a dedicated clock variable *tHandle* could be reset on the handle position change events and incremented by position sustaining events. As those events are refined by all subsequent events in all refinements it would be sufficient to add a guard on the handle clock to events that establish the postcondition of the safety property. The corresponding safety invariant would look as follows:

$$\begin{aligned} Read = TRUE \wedge handle = TRUE \wedge tHandle \geq 150 \Rightarrow \\ gearsExtended = TRUE \wedge doorsClosed = TRUE \end{aligned} \tag{5.11}$$

The remaining safety properties are concerned with the failure mode, which we have omitted in our model. These properties are derived directly from the health monitoring requirements, like the 7-second deadline for receiving sensor value $doorsClosed = false$ after stimulation of the opening electro-valve. Such a property is similar to the timing constraint above, with the only difference being that a failure should be issued and the operation switched to failure mode, hence it can be modelled akin to invariant 5.11 by adding a corresponding time counter (here $tOpenEV$) to the control order:

$$\begin{aligned} openEV = TRUE \wedge tOpenEV > 70 \wedge doorsClosed = TRUE \Rightarrow \\ FailureMode = TRUE \end{aligned} \tag{5.12}$$

For the validation of composed plant and controller models in MultiSim we have synthesised a flight scenario that contains an interrupt in multiple manoeuvring phases. That way we could validate correct handling of interrupting orders from the pilot. The minimal defined time delay of 100ms for the contrary control orders was chosen to be the step period of the controller and the simulation step size. The simulated scenario has the following parameters:

1. At start time = 0s all gears are retracted and doors are closed; the handle is in the UP position ($handle = false$).
2. At time = 1s the handle is pushed DOWN to extend the gears.
3. At time = 11s the handle is pushed UP while the doors are open.
4. At time = 15s the handle is pushed DOWN while the gears are retracting.
5. At time = 24s the handle is pushed UP again while the doors are closing.

The obtained simulation results in terms of the controlled and monitored signal values are plotted in Figure 5.19, showing the behaviour and state of physical components: handle state changes, door and gear cylinder position, and sensor values, – as well as the issued control orders. From the plots we can observe a number of the modelled system properties, for example, a delay of 1 second between the end of stimulation of the door closure electro-valve and the general electro-valve. Other properties, such as the varying lock/unlock duration of hydraulic cylinders, are less apparent, and are shown as a delay between a control order and a state change of the controlled equipment. An interesting dynamics can be seen in the gear position plot at t = 15s, when a handle switch to DOWN position interrupts a gear retraction movement – gears stop in an intermediate position due to pressure loss in the retraction circuit and subsequently resume extension when an opposing pressure is restored to a sufficient level.

The landing system is a good example of the advantages of decomposed development and the use of Rodin for system analysis. Splitting the development into physical and digital components has facilitated the verification process, as the software part could be developed formally in Event-B. Furthermore, the abstraction and refinement capabilities of Rodin have substantially reduced the complexity of the model and allowed us to design it gradually, starting from a very abstract form. For comparison we have developed a model of controller in Modelica using the StateGraph2 library. The result however was complex and difficult to understand due to numerous interleavings between the states that handled possible interrupts. For example, the time delay of 0.1s between two contrary orders results in multiple additional transitions to cope with the delay itself and cancellation/re-activation of the contrary order. Event-B allowed us to introduce such details in small increments, i.e. in separate refinements, and to make sure that the overall invariants of the system hold at every step. In Modelica we would just make the same model more complex and re-run the simulation. There is no clear separation between the complex and the abstract Modelica model (there is no concept of refinement), hence it is more difficult to verify the consistency of the concrete model with its abstraction,

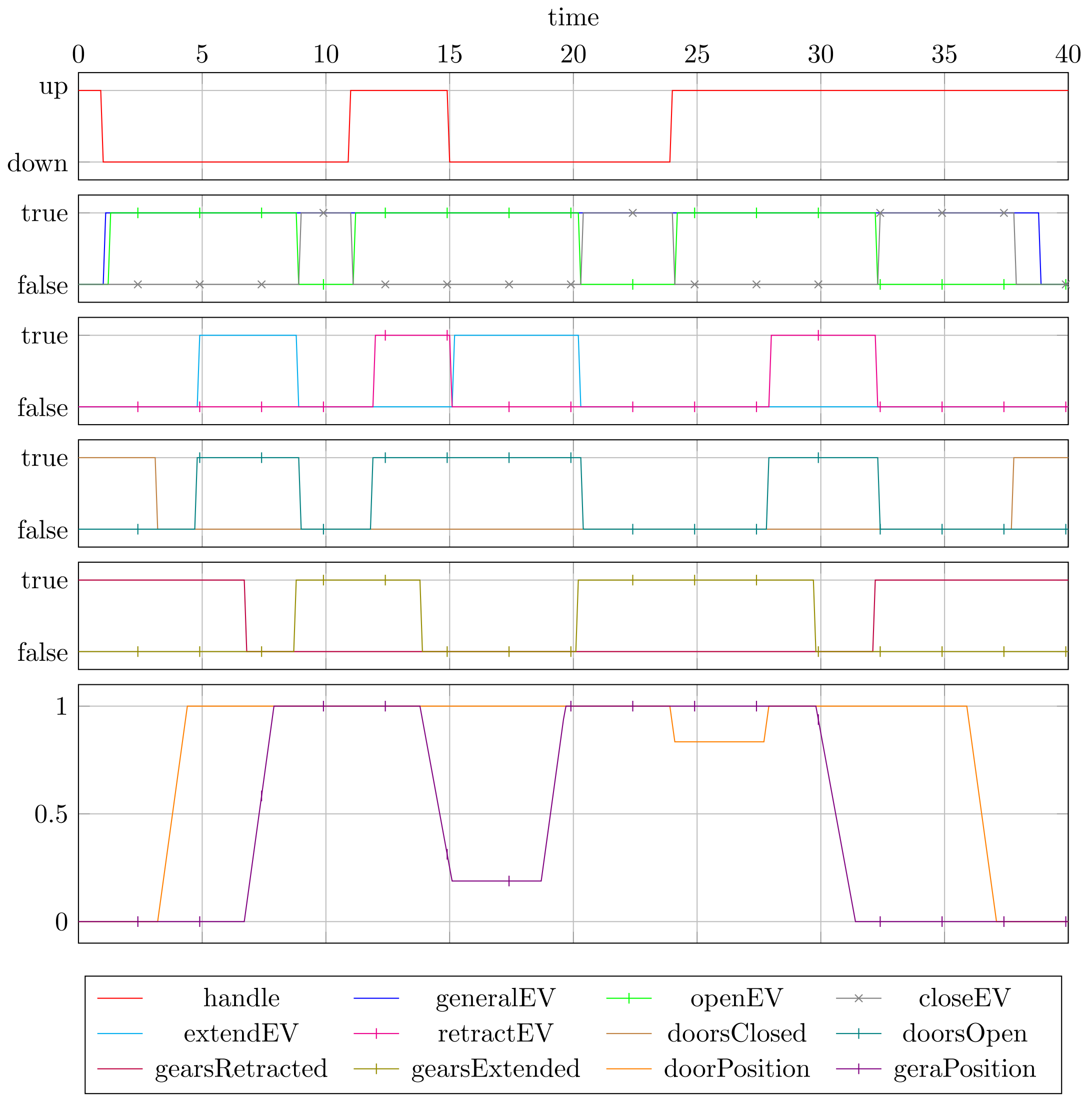


Figure 5.19: Landing system simulation results (simulation time = 40s, step = 0.1s)

as opposed to Event-B, where it is ensured by the gluing invariants. While the problem of complexity can be alleviated in Modelica by splitting the flat structure of the state graph into submodels, the refinement in Rodin is a more elegant approach, which enables not only superposition (horizontal) refinement, but also data (vertical) refinement, in addition to the automated formal proof of system properties. This highlights the role of Event-B as an efficient tool not only for the formal verification, but also for the layered modelling through refinement, which helps to manage the complexity in control system states.

At the same time, the simplicity of assigning simulation semantics to a developed Event-B model of controller enables iterative co-simulation and validation of the formal model and a physical model of the plant. In this respect, the expressiveness of the Modelica language for describing physical dynamics in a natural way and a host of available libraries for multiple domains makes it a compelling technology for modelling the physical aspect of hybrid systems, which cannot be modelled without abstraction on Event-B level. Co-simulation in Rodin coupled with Modelica technology supports validation of formal models against behaviour of the physical components.

5.5 Summary

This chapter illustrated four case studies that we have used to evaluate our approach to hybrid system development. The approach is based on the co-modelling and co-simulation between a formal method (Event-B) and a physical modelling technology (Modelica) that are integrated via the developed MultiSim tool. Our evaluation focused on the following criteria: tool performance, addressed limitations and provided advantages over existing technologies, feasibility and economical value. In all four case studies, as well as in a larger case study on a smart grid system that was implemented for the ADVANCE project (see Section 5.2), the tool has demonstrated good performance. During the evaluation process some memory and performance issues were discovered in the original design of MultiSim that have been successfully resolved by optimising the co-simulation interface and Event-B execution kernel (see Section 4.3.3 for more details).

When compared to the traditional simulation our approach has shown that a combination of the formal analysis and simulation facilitates discovery and understanding of the errors in requirements – according to our workflow the first step of the development is the formal specification of system requirements and constraints in Event-B. For example, formal verification via model-checking and proof allowed us to quickly discover a design flaw in the angle of attack processing logic model before it was simulated (see Section 5.3). An additional benefit of applying the formal method prior to simulation is that formally discovered counterexamples, i.e. invariant violations and system deadlocks, may be used for generating test scenarios that can be subsequently co-simulated in MultiSim against the model of environment. A comparable analysis using the simulation alone may require multiple random sampling executions to detect a failure scenario.

Another advantage of Event-B/Rodin and MultiSim is the step-wise approach to modelling that has shown to be effective when developing a complex system with multiple interrupts between control signals (landing gear case study in Section 5.4). As opposed to a simulation model in Modelica, in which the complexity can be broken down via hierarchy, i.e. by partitioning the model into sub-models, Rodin and MultiSim provide an iterative development approach that relies on the abstraction and systematic refinement of the model in order to gradually increase its complexity. The refinement process in Event-B is verified – each refined model is formally proven to be valid against the requirements and consistent with the abstract model, thus providing a clear link between the abstract model and its refinement. In the landing gear case study the refinement allowed us to model the control system in small steps: each control order was introduced as a new refinement, which substantially reduced complexity of the validated model and simplified the proof, allowing Rodin to discharge all proof obligations automatically.

Based on the experience obtained from the described case studies we think that a combined development using Event-B and physical simulation is a feasible technique that can be adopted in a typical development. Application of the physical simulation is a common practice in system engineering, hence an existing physical model can be imported into Rodin via MultiSim and co-simulated with a formal model. The development of the formal model certainly requires some familiarity with the method and the language. The advantage of Event-B and similar formalisms is a relatively simple logic, supported by a powerful toolset. Furthermore, the key notions of abstraction and refinement make it easier for an engineer to gradually move from the abstract requirements to their formal specification. MultiSim supports this process and enables validation of the formal development against the existing simulation model.

On the other hand, for a formal method user our approach requires a level of experience with the physical modelling and simulation. We alleviate this limitation by adopting a well-established co-simulation standard (FMI), which does not restrict the modeller to a specific simulation technology. All the described case studies have used Modelica, since it is supported by many simulation tools and provides a rich library of modelled physical components. The evaluation showed that the technologies such as Modelica and FMI minimise the level of required expertise to develop a physical simulation model. This allows a formal method user to experiment with simulation and, akin to refinement, iteratively improve the model. Naturally, a domain expert is still required to develop a precise physical model that the concrete formal model must be validated upon.

With respect to the economical sense of applying a formal method and co-simulation instead of relying on a single technology we observed that even for the smallest models a formal specification helps to better understand the requirements, and co-simulation helps to validate those requirements against the model of environment as well as to understand the intricate interactions between components of a hybrid system. With the increased complexity of the system understanding its requirements becomes a challenge. The initial investment that is required for adopting a formal method typically pays off at later stages, when the cost of the design error is significantly higher [66].

Chapter 6

Conclusions

6.1 Goals

Given the increasing engineering focus on the CPS and hybrid systems, and a growing interest in the adoption of formal verification techniques, we believe that a combination of formal methods with the latest simulation technologies is the right step in technology evolution. The existing formal modelling technologies, and Event-B/Rodin in particular, are not fully integrated with the state of the art physical simulation tools. In the context of cyber-physical systems, where both computational and physical aspects of the system are closely coupled, studying them in separation does not provide high assurance, or in some cases gives even false assurance in system correctness, safety and reliability. While the rigorous analysis techniques such as Event-B method are very effective at understanding system requirements and mathematically proving properties of specifications, they are limited in expressiveness to describe complex physical dynamics, so essential for CPS. At the same time, high-end physical modelling and simulation environments do not have formal reasoning, abstraction and refinement capabilities characteristic to formal methods. Both technologies can benefit from each other.

The goal of this work is to provide a development solution targeted at hybrid systems that incorporates formal modelling, verification and physical simulation, in order to ensure safety and reliability of constructed systems. One way to achieve this goal is to implement and offer modellers a single tool that enables both the rigorous, possibly automated analysis of hybrid systems and the simulation-based analysis of interaction with the physical environment. We hope that by integrating these technologies system engineers will benefit in terms of clearer understanding of the intricate interactions between discrete and continuous aspects of a hybrid system, be able to formally verify safety and reliability properties and to analyse system behaviour in a realistic model of environment. Ultimately it should reduce cost and time of the development and improve the quality of implemented software.

The initial idea of an integrated framework was envisaged in two different implementation approaches, both of which have their advantages and application scenarios:

- The first approach is based on the extension of the current Event-B language with continuous-time constructs and meta-data to facilitate the modelling and verification of dynamic systems. Some initial work develops this idea [6, 15]. New constructs can be translated to a simulation language of choice, such as Simulink or Modelica. Additionally, a common abstraction layer, based, for instance, on the hybrid state machines, can be developed on top of both discrete and continuous domains and subsequently mapped to concrete languages via transformation rules. The extensibility of the Rodin platform allows such layers to be implemented on top of the Event-B language, as for instance in a graphical notation of UML-B [137] and Event-B State-machines [131]. Extension of the language is currently possible via the Theory plug-in [44], which is demonstrated on a water tank case study in [43].
- The second approach is based on the co-modelling and co-simulation of discrete and continuous system aspects via integrated tools. The discrete part may be formally specified in Event-B, whereas the continuous part can be either independently developed in a physical modelling environment or generated from augmented Event-B code. The contract on how two models should communicate during simulation process has to be established beforehand. Generally, both solutions can be used in a single process, where the whole system is modelled as discrete in Event-B, augmented and translated to a physical model, with subsequent co-simulation.

This work puts the emphasis on the co-simulation approach as the most feasible integration solution at the moment. Our approach is implemented around a workflow that begins with the requirements specification of a discrete control system using Event-B and proceeds with the step-wise refinement of that specification from the abstract model towards its concrete counterpart. This process is verified by the modeller using the Rodin toolset, which supports formal proof and model-checking. At the same time the continuous environment of the system is modelled using a physical modelling technology, such as Modelica. Our designed co-simulation tool MultiSim couples the formal model of controller with the physical model of environment via the FMI interface for co-simulation. Two models are combined and validated in MultiSim as a single system. This capability enables simulation-based validation of the formal development in Event-B against the wide range of physical models of environment, which was previously not supported by the Rodin toolset. In the same way, it provides the rigorous verification aspect of Event-B to the physical modelling technologies.

6.2 Progress Summary

To accomplish the formulated goal we have completed the following objectives:

- We conducted the analysis of specific features characteristic to CPS and identified key areas for improvement of the current engineering practices and tools. These include the limitations of existing development methods at capturing temporal semantics, high level of design complexity, inadequate models of concurrency and inability to validate entangled interactions between discrete and continuous system aspects. From the current research work in this domain we have distinguished formal methods and simulation technologies as a necessary set of tools for attempting to address some of the stated limitations.

- We reviewed and compared state of the art formal verification methods, simulation technologies and co-simulation frameworks that can be exploited in an integrated development approach. In this work we utilise the Event-B formal language and supporting Rodin platform for their extensibility and step-wise rigorous approach to complex system modelling, and the Functional Mock-Up Interface for its openness and integration with the state of the art physical modelling languages, such as Modelica.

- We defined formal semantics of the discrete-continuous co-simulation step in terms of data exchange and execution progress. The introduced notion of *StartStep* and *EndStep* events for Event-B machines allowed us to define this semantics in the context of the FMI co-simulation and adapt it to a flexible model of Event-B refinement. The latter makes it possible to exercise co-simulation starting from a very abstract model down the refinement chain, thus enabling validation of the formal model along with its construction.

- We specified a composition language of co-simulated models and a simulation data structure using the EMF (Eclipse Modelling Framework) technology. This data structure is implemented as a meta-model that utilises the Event-B EMF framework for Rodin, which enables seamless integration into Rodin infrastructure and easier development of integrated tools and future extensions.

- We specified and implemented a generic simulation orchestration algorithm (master) that can simulate deterministically an arbitrary number of interconnected discrete-event and continuous-time components.

- We implemented an open and extensible MultiSim environment that extends the Rodin toolset with physical co-simulation capability for the simulation-based validation. The environment is designed using the GMF (Graphical Modelling Framework) technology as a graphical composition editor, which allows modellers to import either discrete (Event-B) or continuous (FMI) models, define their interface in terms of the input/output ports and a simulation step size, and execute co-simulation of composed models with a real-time signal visualisation and model-checking via ProB tool [94].

- We validated the developed extension on a number of hybrid system models from several domains, including fluid flows, aircraft equipment control and electric power systems. In each of the case studies the environment was modelled in Modelica language using a commercial simulation tool (Dymola), while the control system was formally specified and verified in Event-B. The composed system was subsequently simulated in MultiSim and compared to the traditional simulation. The experiments have demonstrated that the refinement-based modelling and verification approach supported by the Rodin platform and Event-B works well together with the simulation-based validation. The former yields better understanding of the complex system requirements and associated weaknesses/faults, whereas the latter allows for validation of the formal model on a physical model of environment, which is difficult to verify formally.

6.3 Future Work

As our future work we would like to tackle the existing limitations of the MultiSim tool, mentioned in Chapter 3. Specifically, the designed co-simulation algorithm has a number of limitations due to the complexity of developing a fully-fledged solver for the coupling of continuous components and the limitations of the FMI for Co-simulation standard itself. In particular, we would like to lift the constraint of the MultiSim diagrammatic environment on the direct connectivity between continuous (FMI) components and enable event-driven co-simulation approach that is not restricted to fixed communication points. This would allow for more flexibility when composing a co-simulation model that consists of multiple discrete and continuous components, possibly of different domains and developed with different tools. The event-driven approach should also improve the accuracy of simulation results since it introduces discontinuity into continuous signals only when it is necessary, i.e. at event instants, as opposed to fixed time points.

In order to realise these improvements we need to implement an adaptive step size integration algorithm that is capable of deterministically coupling multiple FMUs. As described in [132, 36] the necessary condition for such an algorithm is to be able to do the multi-step rollback of simulation steps of the co-simulated slaves. A step may get rejected if the step size is too large and causes a discrete event within the step. In such scenarios the master need to roll back the slave's state and states of connected slaves. The FMI 2.0 standard introduces optional function prototypes `fmi2SetFMUstate` and `fmi2GetFMUstate` that do exactly that. Hence, to support the FMI 2.0 standard is one of the key aspect for improving the MultiSim tool. Since the current Java implementation of FMI only supports version 1.0, we are planning to investigate the possibility of developing a Java wrapper library, similar to JFMI [38], on top of the available implementations of the FMI 2.0 in C.

In addition, the support for rollback (backtracking) is required for discrete (Event-B) components, since a rejected step of an FMI component would require each connected and evaluating Event-B component to roll back its state. The backtracking mechanism for Event-B components could be implemented by introducing an internal time in the Event-B slave and by modifying the discrete step semantics to allow for multiple internal steps within a single `doStep(...)` call of the simulation master. The rejection of multiple internal steps within the Event-B slave can be implemented by traversing the event trace of the associated Event-B machine. The trace traversal mechanism is already supported by the execution engine of ProB.

Finally, a discrete component may require to signal the occurrence of an event to the coupled continuous components. While the backtracking and variable step size can facilitate this by rejecting a step and retrying it with a smaller step size to advance the time towards the event instant, current mechanisms provided by FMI do not fully support event handling, as discussed by Broman et al. [37], who outline a set of requirements for developing a hybrid version of the FMI standard that would enable event-driven simulation or fully reactive systems.

Among other possible improvements to our integrated development approach and the MultiSim tool we envision the following ideas:

- *A closer integration of continuous and physical modelling with Event-B*, under which we imply the extension of the existing Event-B language with continuous constructs and capabilities to generate dynamic models, e.g. generating Modelica models from Event-B specifications. Such a capability would enable formal verification of discrete-continuous specifications and a streamlined development process that could capture both formal and simulation-based verification/validation. This task requires substantial research and implementation resources, although the initial proposals are presented in [6], [15] and [43].

- *An integration with other hybrid system modelling and simulation environments*, for example the Ptolemy toolset. This could extend capabilities of the Rodin toolset to heterogeneous simulation of highly concurrent systems. As a possible means of integration we imagine model exchange and co-simulation via dedicated interfaces, such as FMI. The initial experiments in this direction were made by utilising the code generation capability of Ptolemy and a manual translation of the generated models into FMI for Co-Simulation format [48].

- *Broader support of co-simulation interfaces.* Extending supported interfaces with other tools and standards besides the FMI would allow a wider range of modelling and simulation environments to be integrated with Rodin. A possible candidate interface is the HLA architecture, popular in aerospace industry.

We hope that the described ideas will be investigated and potentially developed into improvements of the MultiSim tool and our integrated development approach. Our work has demonstrated that combining a formal method and physical simulation in a single workflow is a practical and useful technique for modelling and validation of multi-domain hybrid systems. One of the difficulties for system engineers who try to adopt a technology or a method in their work process is the tool support. For this work we have used only the open-source community-based technologies, such as Event-B, Modelica and FMI, that are being actively supported by tools and applied on multi-domain projects. In addition, the developed tool is extensible and can be adapted with ease to new requirements. The positive results of the carried out evaluation experiments, the involvement of collaborative parties in the MultiSim development effort and the interest in this technique that has been generated afterwards give us hope that our contribution will be useful for software and system engineers in their challenging work.

Appendix A

Models of the Controlled Water Tank System

A.1 Modelica Model of the Plant

```
model ControlledWaterTankPlant
  "Physical aspect of the controlled water tank"
  inner Modelica.Fluid.System system;
  Modelica.Fluid.Vessels.OpenTank tank(
    nPorts=2,
    height=3,
    crossArea=1,
    redeclare package Medium =
      Modelica.Media.Water.ConstantPropertyLiquidWater,
    portsData={Modelica.Fluid.Vessels.BaseClasses.VesselPortsData(
      diameter=0.1,
      height=3),
      Modelica.Fluid.Vessels.BaseClasses.VesselPortsData(
         diameter=0.2)},
    level_start=0);
  Modelica.Fluid.Sources.FixedBoundary source(
    redeclare package Medium =
      Modelica.Media.Water.ConstantPropertyLiquidWater,
    T=system.T_ambient,
    nPorts=1,
    p=2500000);
  Modelica.Fluid.Sources.FixedBoundary sink(
    redeclare package Medium =
      Modelica.Media.Water.ConstantPropertyLiquidWater,
    p=system.p_ambient,
    T=system.T_ambient,
    nPorts=1);
  Modelica.Fluid.Valves.ValveDiscrete valve(
    redeclare package Medium =
      Modelica.Media.Water.ConstantPropertyLiquidWater,
    dp_nominal=100000,
    m_flow_nominal=10);
  Modelica.Blocks.Interfaces.BooleanInput valveInput
    "Control signal of valve state (on/off)";
  Modelica.Blocks.Interfaces.RealOutput levelOutput "Water level";
  Modelica.Blocks.Sources.RealExpression tankLevel(y=tank.level);
equation
  connect(source.ports[1],valve. port_a);
  connect(valve.port_b,tank. ports[1]);
  connect(sink.ports[1],tank. ports[2]);
  connect(valveInput, valve.open);
  connect(tankLevel.y, levelOutput);
end ControlledWaterTankPlant;
```

A.2 Event-B Specification of the Controller

A.2.1 Context ctx

CONTEXT ctx
CONSTANTS

L
H
LT
HT

AXIOMS

axm3 : $L > 0$
axm4 : $H > L$
axm5 : $LT > L$
axm6 : $HT < H$
axm7 : $HT > LT$
axm1 : $LT = 10$
axm2 : $HT = 20$

END

A.2.2 Abstract Machine wtCtr0

MACHINE wtCtr0
SEES ctx
VARIABLES

$valve$

```
INVARIANTS
             inv1 : $valve \in BOOL$
EVENTS
Initialisation
         begin
                 act1 : $valve := TRUE$
         end
Event    $openValve \mathrel{\widehat{=}}$
         any
                 $l$
         where
                 grd1 : $l < LT$
         then
                 act1 : $valve := TRUE$
         end
Event    $keepValve \mathrel{\widehat{=}}$
         any
                 $l$
         where
                 grd1 : $l \geq LT \land l \leq HT$
         then
                 $skip$
         end
Event    $closeValve \mathrel{\widehat{=}}$
         any
                 $l$
         where
                 grd1 : $l > HT$
         then
                 act1 : $valve := FALSE$
         end
END
```

A.2.3 Concrete Machine wtCtr1

```
MACHINE   wtCtr1
REFINES   wtCtr0
SEES   ctx
VARIABLES
           $Read$
           $Control$
           $DecideClose$
           $DecideKeep$
```

DecideOpen

valve

level

INVARIANTS

`typeof_Read` : $Read \in BOOL$

`typeof_Control` : $Control \in BOOL$

`typeof_DecideClose` : $DecideClose \in BOOL$

`typeof_DecideKeep` : $DecideKeep \in BOOL$

`typeof_DecideOpen` : $DecideOpen \in BOOL$

`distinct_states_in_sm` : $partition(\{TRUE\}, \{Read\} \cap \{TRUE\}, \{Control\} \cap \{TRUE\}, \{DecideClose\} \cap \{TRUE\}, \{DecideKeep\} \cap \{TRUE\}, \{DecideOpen\} \cap \{TRUE\})$

`inv1` : $level \in L \,..\, H$
system goal invariant

EVENTS

Initialisation

extended

begin

`act1` : $valve := TRUE$
`init_Control` : $Control := TRUE$
`init_Read` : $Read := FALSE$
`init_DecideClose` : $DecideClose := FALSE$
`init_DecideKeep` : $DecideKeep := FALSE$
`init_DecideOpen` : $DecideOpen := FALSE$
`act2` : $level := 0$

end

Event *openValve* $\widehat{=}$

refines *openValve*

when

`isin_DecideOpen` : $DecideOpen = TRUE$

with

`l` : $l = level$

then

`leave_DecideOpen` : $DecideOpen := FALSE$
`enter_Control` : $Control := TRUE$
`act1` : $valve := TRUE$

end

Event *keepValve* $\widehat{=}$

refines *keepValve*

when

`isin_DecideKeep` : $DecideKeep = TRUE$

with

`l` : $l = level$

then

`leave_DecideKeep` : $DecideKeep := FALSE$

```
        enter_Control : $Control := TRUE$
    end
Event  $closeValve \mathrel{\widehat{=}}$
refines  $closeValve$
    when
        isin_DecideClose : $DecideClose = TRUE$
    with
        l : $l = level$
    then
        leave_DecideClose : $DecideClose := FALSE$
        enter_Control : $Control := TRUE$
        act1 : $valve := FALSE$
    end
Event  $readLevel \mathrel{\widehat{=}}$
    any
        $l$
    where
        grd1 : $l \geq 0$
    then
        act1 : $level := l$
    end
Event  $decideOpen \mathrel{\widehat{=}}$
    when
        isin_Read : $Read = TRUE$
        grd1 : $level < LT$
    then
        leave_Read : $Read := FALSE$
        enter_DecideOpen : $DecideOpen := TRUE$
    end
Event  $decideKeep \mathrel{\widehat{=}}$
    when
        isin_Read : $Read = TRUE$
        grd1 : $level \geq LT \land level \leq HT$
    then
        leave_Read : $Read := FALSE$
        enter_DecideKeep : $DecideKeep := TRUE$
    end
Event  $decideClose \mathrel{\widehat{=}}$
    when
        isin_Read : $Read = TRUE$
        grd1 : $level > HT$
    then
        leave_Read : $Read := FALSE$
        enter_DecideClose : $DecideClose := TRUE$
    end
END
```

Appendix B

Models of the Voltage Control System

B.1 Modelica Model of the Voltage Distribution Network

```
model OLTCDistrPlant
  inner ElectricPower.System system(ref="inert");
  ElectricPower.AC3ph_abc.Sources.Voltage Vgen1(V_nom=11e3, v0=1.006);
  ElectricPower.AC3ph_abc.Lines.RXline line1(
    steadyIni_en=false,
    par(
      V_nom=11e3,
      r=0.02e-3,
      x=0.2e-3),
    len=10000);
  ElectricPower.AC3ph_abc.Loads.Zload zLoad(
    pType=2,
    V_nom=230,
    S_nom=500e6);
  ElectricPower.Blocks.Signals.Transient[2] pq_change(
    s_ini={1,0.1},
    s_fin={10,0.1},
    t_change=25,
    each t_duration=50);
  ElectricPower.AC3ph_abc.Transformers.TrafoIdeal trafo(
    dynTC_2=true, par(V_nom={48,1},
    v_tc2={0.8,0.82,0.84,0.86,0.88,0.9,0.92,0.94,0.96,0.98,1.0,
          1.02,1.04,1.06,1.08,1.1,1.12,1.14,1.16,1.18,1.2}));
  ElectricPower.AC3ph_abc.Lines.RXline line2(
    steadyIni_en=false,
    len=1000,
    par(
      V_nom=230,
      r=0.02e-3,
      x=0.2e-3));
  ElectricPower.AC3ph_abc.Sensors.Psensor Psensor1;
  ElectricPower.AC3ph_abc.Sensors.VnormSensor Vsensor1;
  ElectricPower.AC3ph_abc.Nodes.BusBar bus1;
  Modelica.Blocks.Interfaces.RealOutput vNorm;
  Modelica.Blocks.Interfaces.IntegerInput tapIndex;
  Modelica.Blocks.Interfaces.RealOutput pActive;
equation
  connect(zLoad.p, pq_change.y);
  connect(trafo.term_2, line2.term_p);
  connect(Vgen1.ACterm, line1.term_p);
  connect(line2.term_n, Psensor1.term_p);
  connect(zLoad.term, Psensor1.term_n);
  connect(line1.term_n, bus1.term);
  connect(trafo.term_1, bus1.term);
  connect(line2.term_n, Vsensor1.term);
  connect(Vsensor1.v_norm,vNorm);
  connect(tapIndex, trafo.tap_2);
  connect(Psensor1.p[1],pActive);
end OLTCDistrPlant;
```

B.2 Event-B Specification of the Tap Controller

B.2.1 Context controlCtx

CONTEXT controlCtx
CONSTANTS

$DB2$	voltage deadband / 2
$VREF$	reference voltage
$TAPN$	total number of taps
TD	detection delay
TM	mechanical delay
$VINIT$	initialisation value of the voltage
$TAPINIT$	initialisation value of the tap position

AXIOMS

axm1 : $DB2 > 0$

axm2 : $VREF > 0$

axm3 : $TAPN > 0$

axm4 : $TD \in \mathbb{N}$

axm5 : $TM \in \mathbb{N}$

axm6 : $VINIT = VREF$

axm7 : $TAPINIT > 0$

axm8 : $DB2 = 10$

axm9 : $VREF = 2300$

axm10 : $TAPN = 21$

axm11 : $TD = 10$

axm12 : $TM = 1$

axm13 : $TAPINIT = 11$

END

B.2.2 Machine tapController

MACHINE tapController
SEES controlCtx
VARIABLES

$sIdle$

$sCount$

$sAction$

$sRead$

$sDecide$

$voltage$

tap

$dTimer$

$mTimer$

$changeDirection$

INVARIANTS

typeof_sIdle : $sIdle \in BOOL$

typeof_sCount : $sCount \in BOOL$

typeof_sAction : $sAction \in BOOL$

distinct_states_in_smControl_1 : $partition(\{TRUE\}, \{sIdle\} \cap \{TRUE\}, \{sCount\} \cap \{TRUE\}, \{sAction\} \cap \{TRUE\})$

_txLQwXxIEeOmNfMgYpZZsw : $(sCount = TRUE) \Rightarrow (dTimer \geq 0)$

_txLQwnxIEeOmNfMgYpZZsw : $(sCount = TRUE) \Rightarrow (dTimer \leq 10)$

typeof_sRead : $sRead \in BOOL$

typeof_sDecide : $sDecide \in BOOL$

distinct_states_in_smFMU_1 : $partition(\{TRUE\}, \{sRead\} \cap \{TRUE\}, \{sDecide\} \cap \{TRUE\})$

inv1 : $voltage \in \mathbb{N}$

inv2 : $tap > 0$

inv3 : $tap \leq TAPN$

inv4 : $dTimer \in \mathbb{N}$

inv5 : $mTimer \in \mathbb{N}$

inv6 : $changeDirection \in BOOL$

```
EVENTS
Initialisation
      begin
              act1 : $voltage := VINIT$
              act2 : $tap := TAPINIT$
              init_sRead : $sRead := TRUE$
              init_sDecide : $sDecide := FALSE$
              act4 : $mTimer := 0$
              init_sIdle : $sIdle := TRUE$
              act3 : $dTimer := 0$
              init_sAction : $sAction := FALSE$
              init_sCount : $sCount := FALSE$
              act5 : $changeDirection := FALSE$
      end
Event   $readInputs \mathrel{\widehat{=}}$
      any
             $v$
      where
              isin_sRead : $sRead = TRUE$
              grd1 : $v \in \mathbb{N}$
      then
              leave_sRead : $sRead := FALSE$
              enter_sDecide : $sDecide := TRUE$
              act1 : $voltage := v$
      end
Event   $noChange \mathrel{\widehat{=}}$
      when
              grd1 : $VREF - voltage \leq DB2$
              isin_sDecide : $sDecide = TRUE$
              grd2 : $voltage - VREF \leq DB2$
              isin_sIdle : $sIdle = TRUE$
      then
              enter_sRead : $sRead := TRUE$
              leave_sDecide : $sDecide := FALSE$
      end
Event   $delayAction \mathrel{\widehat{=}}$
      when
              grd2 : $dTimer > 0$
              isin_sDecide : $sDecide = TRUE$
              isin_sCount : $sCount = TRUE$
              grd1 : $VREF - voltage > DB2 \lor voltage - VREF > DB2$
      then
              enter_sRead : $sRead := TRUE$
              leave_sDecide : $sDecide := FALSE$
              act1 : $dTimer := dTimer - 1$
```

```
    end
Event  delayChange ≙
    when
        isin_sDecide : $sDecide = TRUE$
        grd1 : $mTimer > 0$
        isin_sAction : $sAction = TRUE$
    then
        leave_sDecide : $sDecide := FALSE$
        act1 : $mTimer := mTimer - 1$
        enter_sRead : $sRead := TRUE$
    end
Event  startCount ≙
    when
        grd2 : $VREF - voltage > DB2 \Rightarrow tap < TAPN$
        grd3 : $voltage - VREF > DB2 \Rightarrow tap > 1$
        grd1 : $VREF - voltage > DB2 \lor voltage - VREF > DB2$
        isin_sIdle : $sIdle = TRUE$
    then
        act1 : $dTimer := TD$
        leave_sIdle : $sIdle := FALSE$
        enter_sCount : $sCount := TRUE$
    end
Event  cancelCount ≙
    when
        grd2 : $voltage - VREF \leq DB2$
        grd1 : $VREF - voltage \leq DB2$
        isin_sCount : $sCount = TRUE$
    then
        leave_sCount : $sCount := FALSE$
        enter_sIdle : $sIdle := TRUE$
    end
Event  startAction ≙
    when
        grd1 : $dTimer = 0$
        isin_sCount : $sCount = TRUE$
    then
        enter_sAction : $sAction := TRUE$
        act1 : $mTimer := TM$
        act2 : $changeDirection := bool(VREF - voltage > DB2)$
        leave_sCount : $sCount := FALSE$
    end
Event  tapUp ≙
    when
        grd2 : $changeDirection = TRUE$
        isin_sAction : $sAction = TRUE$
```

```
            grd1 : $mTimer = 0$
    then
            enter_sIdle : $sIdle := TRUE$
            act1 : $tap := tap + 1$
            leave_sAction : $sAction := FALSE$
    end
Event   $tapDown \mathrel{\widehat{=}}$
    when
            isin_sAction : $sAction = TRUE$
            grd1 : $mTimer = 0$
            grd2 : $changeDirection = FALSE$
    then
            act1 : $tap := tap - 1$
            leave_sAction : $sAction := FALSE$
            enter_sIdle : $sIdle := TRUE$
    end
END
```

Appendix C

Models of the Angle of Attack Processing

C.1 Modelica Models of Test Scenarios and the FCPC Environment

C.1.1 Scenario A

```
model ScenarioA
  FCPC fCPC;
  Modelica.Blocks.Sources.Constant aoa2(k=2.3);
  Modelica.Blocks.Sources.Constant aoa3(k=2.3);
  Modelica.Blocks.Sources.Step aoa1(
    startTime=1,
    height=50.6,
    offset=2.1);
  Modelica.Blocks.Interfaces.RealOutput mean23
    "Mean value between AOA 2 and AOA 3";
  Modelica.Blocks.Interfaces.RealOutput delta1
    "Difference between AOA 1 and the median";
  Modelica.Blocks.Interfaces.RealOutput mean12
    "Mean value between AOA 1 and AOA 2";
equation
  connect(aoa1.y, fCPC.aoa1);
  connect(aoa2.y, fCPC.aoa2);
  connect(aoa3.y, fCPC.aoa3);
  connect(fCPC.m23, mean23);
  connect(fCPC.d1, delta1);
  connect(mean12, fCPC.m12);
end ScenarioA;
```

C.1.2 Scenario B

```
model ScenarioB
  FCPC fCPC;
  Modelica.Blocks.Sources.Constant aoa2(k=2.3);
  Modelica.Blocks.Sources.Constant aoa3(k=2.3);
  Modelica.Blocks.Interfaces.RealOutput mean23
    "Mean value between AOA 2 and AOA 3";
  Modelica.Blocks.Interfaces.RealOutput delta1
    "Difference between AOA 1 and the median";
  Modelica.Blocks.Interfaces.RealOutput mean12
    "Mean value between AOA 1 and AOA 2";
  Modelica.Blocks.Sources.Ramp aoa4(
    startTime=1,
    duration=2,
    height=48.5,
    offset=2.1);
equation
  connect(aoa2.y, fCPC.aoa2);
  connect(aoa3.y, fCPC.aoa3);
  connect(fCPC.m23, mean23);
  connect(fCPC.d1, delta1);
  connect(mean12, fCPC.m12);
  connect(fCPC.aoa1, aoa4.y);
end ScenarioB;
```

C.1.3 Scenario C

```
model ScenarioC
  FCPC fCPC;
  Modelica.Blocks.Sources.Constant aoa2(k=2.3);
  Modelica.Blocks.Sources.Constant aoa3(k=2.3);
  Modelica.Blocks.Interfaces.RealOutput mean23
    "Mean value between AOA 2 and AOA 3";
  Modelica.Blocks.Interfaces.RealOutput delta1
    "Difference between AOA 1 and the median";
  Modelica.Blocks.Interfaces.RealOutput mean12
    "Mean value between AOA 1 and AOA 2";
  Modelica.Blocks.Sources.Pulse aoa1(
    offset=2.1,
    startTime=1,
    amplitude=48.5,
    width=40,
    period=0.5,
    nperiod=2);
  Modelica.Blocks.Sources.Pulse aoa4(
    amplitude=48.5,
    width=40,
    period=0.5,
    nperiod=1,
    startTime=2.5,
    offset=0);
  Modelica.Blocks.Math.Add add;
equation
  connect(aoa2.y, fCPC.aoa2);
  connect(aoa3.y, fCPC.aoa3);
  connect(fCPC.m23, mean23);
  connect(fCPC.d1, delta1);
  connect(mean12, fCPC.m12);
  connect(aoa4.y, add.u1);
  connect(aoa1.y, add.u2);
  connect(add.y, fCPC.aoa1);
end ScenarioC;
```

C.1.4 Scenario D

```
model ScenarioD
  FCPC fCPC;
  Modelica.Blocks.Sources.Constant aoa2(k=2.3);
  Modelica.Blocks.Sources.Constant aoa3(k=2.3);
  Modelica.Blocks.Interfaces.RealOutput mean23
    "Mean value between AOA 2 and AOA 3";
  Modelica.Blocks.Interfaces.RealOutput delta1
    "Difference between AOA 1 and the median";
  Modelica.Blocks.Interfaces.RealOutput mean12
    "Mean value between AOA 1 and AOA 2";
  Modelica.Blocks.Math.Add add;
  Modelica.Blocks.Sources.Pulse aoa5(
    offset=2.1,
    startTime=1,
    amplitude=48.5,
    width=40,
    period=0.5,
    nperiod=1);
  Modelica.Blocks.Sources.Pulse aoa6(
    amplitude=48.5,
    nperiod=1,
    period=1,
    startTime=2);
equation
  connect(aoa2.y, fCPC.aoa2);
  connect(aoa3.y, fCPC.aoa3);
  connect(fCPC.m23, mean23);
  connect(fCPC.d1, delta1);
  connect(mean12, fCPC.m12);
  connect(add.y, fCPC.aoa1);
  connect(add.u1, aoa6.y);
  connect(add.u2, aoa5.y);
end ScenarioD;
```

C.1.5 FCPC

```
model FCPC
  parameter Real threshold=3;
  Modelica.Blocks.Interfaces.RealInput aoa1;
  Modelica.Blocks.Interfaces.RealInput aoa2;
  Modelica.Blocks.Interfaces.RealInput aoa3;
  Modelica.Blocks.Discrete.Sampler aoa1_sample
    (samplePeriod(displayUnit="s")=0.04);
  Modelica.Blocks.Discrete.Sampler aoa3_sample
    (samplePeriod(displayUnit="s")=0.04);
  Modelica.Blocks.Discrete.Sampler aoa2_sample
    (samplePeriod(displayUnit="s")=0.04);
  Diff aoa1_diff;
  Diff aoa2_diff;
  Diff aoa3_diff;
  Modelica.Blocks.Interfaces.RealOutput m23;
  Modelica.Blocks.Sources.RealExpression mean12
    (y=(aoa1_sample.y + aoa2_sample.y)/2);
  Modelica.Blocks.Sources.RealExpression median
    (y=min({max({aoa1_sample.y,aoa2_sample.y}),
            max({aoa2_sample.y,aoa3_sample.y}),
            max({aoa1_sample.y,aoa3_sample.y})}));
  Modelica.Blocks.Sources.RealExpression mean23
    (y=(aoa2_sample.y + aoa3_sample.y)/2);
  Modelica.Blocks.Interfaces.RealOutput d1;
  Modelica.Blocks.Interfaces.RealOutput m12;
equation
  connect(aoa1_sample.u, aoa1);
  connect(aoa2, aoa2_sample.u);
  connect(aoa3, aoa3_sample.u);
  connect(aoa1_sample.y, aoa1_diff.u1);
  connect(aoa3_sample.y, aoa3_diff.u1);
  connect(aoa2_sample.y, aoa2_diff.u1);
  connect(median.y, aoa1_diff.u2);
  connect(median.y, aoa2_diff.u2);
  connect(median.y, aoa3_diff.u2);
  connect(aoa1_diff.y, d1);
  connect(mean23.y, m23);
  connect(mean12.y, m12);
end FCPC;
```

C.1.6 Auxiliary Block Diff

```
block A330AOA.Diff "Absolute difference between two Real values"
  extends Modelica.Blocks.Interfaces.SI2SO;
equation
  y = abs(u1 - u2);
end Diff;
```

C.2 Event-B Specification of the COM/MON Unit

C.2.1 Context ctx

CONTEXT ctx
CONSTANTS

THRESHOLD

MEMPERIOD

VALPERIOD

AXIOMS

axm1 : $THRESHOLD \in \mathbb{N}_1$

axm2 : $MEMPERIOD \in \mathbb{N}_1$

axm3 : $VALPERIOD \in \mathbb{N}_1$

axm4 : $MEMPERIOD > VALPERIOD$

axm5 : $THRESHOLD = 100$

axm6 : $MEMPERIOD = 30$

axm7 : $VALPERIOD = 25$

END

C.2.2 Machine mch2

MACHINE mch2
SEES ctx
VARIABLES

Control

Input

Rejected

Invalid

Valid

$Memorised$

$Normal$

AOA

$mean12$

$mean23$

$delta1$

$aoaFCPC$

$memClk$

$memMean$

$valClk$

INVARIANTS

`typeof_Control` : $Control \in BOOL$

`typeof_Input` : $Input \in BOOL$

`distinct_states_in_sm2` : $partition(\{TRUE\}, \{Input\} \cap \{TRUE\}, \{Control\} \cap \{TRUE\})$

`typeof_Rejected` : $Rejected \in BOOL$

`typeof_Invalid` : $Invalid \in BOOL$

`typeof_Valid` : $Valid \in BOOL$

`typeof_Memorised` : $Memorised \in BOOL$

`typeof_Normal` : $Normal \in BOOL$

`typeof_AOA` : $AOA \in BOOL$

`distinct_states_in_MON` : $(AOA = TRUE) \Rightarrow partition(\{TRUE\}, \{Valid\} \cap \{TRUE\}, \{Invalid\} \cap \{TRUE\}, \{Rejected\} \cap \{TRUE\})$

`distinct_states_in_COM` : $(AOA = TRUE) \Rightarrow partition(\{TRUE\}, \{Normal\} \cap \{TRUE\}, \{Memorised\} \cap \{TRUE\})$

`Rejected_substateof_AOA` : $(Rejected = TRUE) \Rightarrow (AOA = TRUE)$

`Invalid_substateof_AOA` : $(Invalid = TRUE) \Rightarrow (AOA = TRUE)$

`Valid_substateof_AOA` : $(Valid = TRUE) \Rightarrow (AOA = TRUE)$

`Memorised_substateof_AOA` : $(Memorised = TRUE) \Rightarrow (AOA = TRUE)$

`Normal_substateof_AOA` : $(Normal = TRUE) \Rightarrow (AOA = TRUE)$

`inv1` : $mean12 \in \mathbb{Z}$

`inv2` : $mean23 \in \mathbb{Z}$

`inv3` : $delta1 \in \mathbb{N}$

`inv5` : $aoaFCPC \in \mathbb{Z}$

`inv6` : $memClk \in \mathbb{N}$

`inv7` : $memMean \in \mathbb{Z}$

`inv8` : $valClk \in \mathbb{N}$

`inv9` : $Rejected = TRUE \land Input = TRUE \Rightarrow aoaFCPC = mean23 \lor Memorised = TRUE$

if AOA is rejected, a fallback mean value shall
be used for output, unless in memorisation mode

`inv11` : $Memorised = TRUE \Rightarrow aoaFCPC \leq THRESHOLD$

EVENTS

Initialisation

  begin

    init_Input : $Input := TRUE$
    init_Control : $Control := FALSE$
    init_Valid : $Valid := TRUE$
    init_AOA : $AOA := TRUE$
    init_Normal : $Normal := TRUE$
    init_Rejected : $Rejected := FALSE$
    init_Invalid : $Invalid := FALSE$
    init_Memorised : $Memorised := FALSE$
    act1 : $mean12 := 0$
    act2 : $mean23 := 0$
    act3 : $delta1 := 0$
    act5 : $aoaFCPC := 0$
    act6 : $memClk := 0$
    act7 : $memMean := 0$
    act8 : $valClk := 0$

  end

Event $readInputs \mathrel{\widehat{=}}$

  any

    $d1$
    $m12$
    $m23$

  where

    isin_Input : $Input = TRUE$
    grd1 : $d1 \in \mathbb{N}$
    grd3 : $m12 \in \mathbb{Z}$
    grd4 : $m23 \in \mathbb{Z}$

  then

    leave_Input : $Input := FALSE$
    enter_Control : $Control := TRUE$
    act1 : $delta1 := d1$
    act3 : $mean12 := m12$
    act4 : $mean23 := m23$

  end

Event $outNominal \mathrel{\widehat{=}}$

  when

    isin_Control : $Control = TRUE$
    isin_Valid : $Valid = TRUE$
    MON_guards1 : $delta1 \leq THRESHOLD$
    isin_Normal : $Normal = TRUE$

  then

    leave_Control : $Control := FALSE$
    enter_Input : $Input := TRUE$
    COM_actions2 : $aoaFCPC := mean12$
    COM_actions1 : $memMean := mean12$

```
    end
Event  outNominalValidate ≙
    when
        isin_Control : Control = TRUE
        isin_Invalid : Invalid = TRUE
        MON_guards5 : delta1 ≤ THRESHOLD
        isin_Normal : Normal = TRUE
    then
        leave_Control : Control := FALSE
        enter_Input : Input := TRUE
        leave_Invalid : Invalid := FALSE
        enter_Valid : Valid := TRUE
        COM_actions2 : aoaFCPC := mean12
        COM_actions1 : memMean := mean12
    end
Event  outFallback ≙
    when
        isin_Control : Control = TRUE
        isin_Rejected : Rejected = TRUE
        isin_Normal : Normal = TRUE
    then
        leave_Control : Control := FALSE
        enter_Input : Input := TRUE
        COM_actions4 : aoaFCPC := mean23
        COM_actions3 : memMean := mean23
    end
Event  outFallbackReject ≙
    when
        isin_Control : Control = TRUE
        isin_Invalid : Invalid = TRUE
        MON_guards7 : valClk = 0
        MON_guards6 : delta1 > THRESHOLD
        isin_Normal : Normal = TRUE
    then
        leave_Control : Control := FALSE
        enter_Input : Input := TRUE
        leave_Invalid : Invalid := FALSE
        enter_Rejected : Rejected := TRUE
        COM_actions4 : aoaFCPC := mean23
        COM_actions3 : memMean := mean23
    end
Event  memoriseInvalidate ≙
    when
        isin_Control : Control = TRUE
        isin_Valid : Valid = TRUE
        isin_Normal : Normal = TRUE
        COM_guards1 : $delta1 > THRESHOLD$
    then
        leave_Control : $Control := FALSE$
        enter_Input : $Input := TRUE$
        leave_Valid : $Valid := FALSE$
        leave_Normal : $Normal := FALSE$
        enter_Invalid : $Invalid := TRUE$
        MON_actions1 : $valClk := VALPERIOD - 1$
        enter_Memorised : $Memorised := TRUE$
        COM_setOutput : $aoaFCPC := memMean$
        COM_setClock : $memClk := MEMPERIOD - 1$
    end
Event memoriseInvalid $\widehat{=}$
    when
        isin_Control : $Control = TRUE$
        isin_Invalid : $Invalid = TRUE$
        MON_guards3 : $valClk > 0$
        isin_Normal : $Normal = TRUE$
        COM_guards1 : $delta1 > THRESHOLD$
    then
        leave_Control : $Control := FALSE$
        enter_Input : $Input := TRUE$
        leave_Normal : $Normal := FALSE$
        MON_tick : $valClk := valClk - 1$
        enter_Memorised : $Memorised := TRUE$
        COM_setOutput : $aoaFCPC := memMean$
        COM_setClock : $memClk := MEMPERIOD - 1$
    end
Event tickMemValidate $\widehat{=}$
    when
        isin_Control : $Control = TRUE$
        isin_Invalid : $Invalid = TRUE$
        MON_guards5 : $delta1 \leq THRESHOLD$
        isin_Memorised : $Memorised = TRUE$
        COM_guards2 : $memClk > 0$
    then
        leave_Control : $Control := FALSE$
        enter_Input : $Input := TRUE$
        leave_Invalid : $Invalid := FALSE$
        enter_Valid : $Valid := TRUE$
        COM_tick : $memClk := memClk - 1$
    end
Event tickMemValid $\widehat{=}$
    when
        isin_Control : $Control = TRUE$
        isin_Valid : $Valid = TRUE$
        MON_guards1 : $delta1 \leq THRESHOLD$
        isin_Memorised : $Memorised = TRUE$
        COM_guards2 : $memClk > 0$
    then
        leave_Control : $Control := FALSE$
        enter_Input : $Input := TRUE$
        COM_tick : $memClk := memClk - 1$
    end
Event tickMemInvalidate $\widehat{=}$
    when
        isin_Control : $Control = TRUE$
        isin_Valid : $Valid = TRUE$
        MON_guards4 : $delta1 > THRESHOLD$
        isin_Memorised : $Memorised = TRUE$
        COM_guards2 : $memClk > 0$
    then
        leave_Control : $Control := FALSE$
        enter_Input : $Input := TRUE$
        leave_Valid : $Valid := FALSE$
        enter_Invalid : $Invalid := TRUE$
        MON_actions1 : $valClk := VALPERIOD - 1$
        COM_tick : $memClk := memClk - 1$
    end
Event tickMemInvalid $\widehat{=}$
    when
        isin_Control : $Control = TRUE$
        isin_Invalid : $Invalid = TRUE$
        MON_guards3 : $valClk > 0$
        MON_guards2 : $delta1 > THRESHOLD$
        isin_Memorised : $Memorised = TRUE$
        COM_guards2 : $memClk > 0$
    then
        leave_Control : $Control := FALSE$
        enter_Input : $Input := TRUE$
        MON_tick : $valClk := valClk - 1$
        COM_tick : $memClk := memClk - 1$
    end
Event tickMemReject $\widehat{=}$
    when
        isin_Control : $Control = TRUE$
        isin_Invalid : $Invalid = TRUE$
        MON_guards7 : $valClk = 0$
        MON_guards6 : $delta1 > THRESHOLD$
        isin_Memorised : $Memorised = TRUE$
```

```
            COM_guards2 : $memClk > 0$
      then
            leave_Control : $Control := FALSE$
            enter_Input : $Input := TRUE$
            leave_Invalid : $Invalid := FALSE$
            enter_Rejected : $Rejected := TRUE$
            COM_tick : $memClk := memClk - 1$
      end
Event   tickMemRejected ≙
      when
            isin_Control : $Control = TRUE$
            isin_Rejected : $Rejected = TRUE$
            isin_Memorised : $Memorised = TRUE$
            COM_guards2 : $memClk > 0$
      then
            leave_Control : $Control := FALSE$
            enter_Input : $Input := TRUE$
            MON_actions3 : $aoaFCPC := mean23$
            MON_actions2 : $memMean := mean23$
            COM_tick : $memClk := memClk - 1$
      end
Event   endMemValidate ≙
      when
            isin_Control : $Control = TRUE$
            isin_Invalid : $Invalid = TRUE$
            MON_guards5 : $delta1 \leq THRESHOLD$
            isin_Memorised : $Memorised = TRUE$
            COM_guards3 : $memClk = 0$
      then
            leave_Control : $Control := FALSE$
            enter_Input : $Input := TRUE$
            leave_Memorised : $Memorised := FALSE$
            leave_Invalid : $Invalid := FALSE$
            enter_Valid : $Valid := TRUE$
            enter_Normal : $Normal := TRUE$
            COM_actions8 : $aoaFCPC := mean12$
            COM_actions7 : $memMean := mean12$
      end
Event   endMemValid ≙
      when
            isin_Control : $Control = TRUE$
            isin_Valid : $Valid = TRUE$
            MON_guards1 : $delta1 \leq THRESHOLD$
            isin_Memorised : $Memorised = TRUE$
            COM_guards3 : $memClk = 0$
```

then

`leave_Control` : $Control := FALSE$
`enter_Input` : $Input := TRUE$
`leave_Memorised` : $Memorised := FALSE$
`enter_Normal` : $Normal := TRUE$
`COM_actions8` : $aoaFCPC := mean12$
`COM_actions7` : $memMean := mean12$

end

Event *endMemInvalidate* $\widehat{=}$

when

`isin_Control` : $Control = TRUE$
`isin_Valid` : $Valid = TRUE$
`MON_guards4` : $delta1 > THRESHOLD$
`isin_Memorised` : $Memorised = TRUE$
`COM_guards3` : $memClk = 0$

then

`leave_Control` : $Control := FALSE$
`enter_Input` : $Input := TRUE$
`leave_Memorised` : $Memorised := FALSE$
`leave_Valid` : $Valid := FALSE$
`enter_Invalid` : $Invalid := TRUE$
`MON_actions1` : $valClk := VALPERIOD - 1$
`enter_Normal` : $Normal := TRUE$
`COM_actions8` : $aoaFCPC := mean12$
`COM_actions7` : $memMean := mean12$

end

Event *endMemInvalid* $\widehat{=}$

when

`isin_Control` : $Control = TRUE$
`isin_Invalid` : $Invalid = TRUE$
`MON_guards3` : $valClk > 0$
`MON_guards2` : $delta1 > THRESHOLD$
`isin_Memorised` : $Memorised = TRUE$
`COM_guards3` : $memClk = 0$

then

`leave_Control` : $Control := FALSE$
`enter_Input` : $Input := TRUE$
`leave_Memorised` : $Memorised := FALSE$
`MON_tick` : $valClk := valClk - 1$
`enter_Normal` : $Normal := TRUE$
`COM_actions8` : $aoaFCPC := mean12$
`COM_actions7` : $memMean := mean12$

end

Event *endMemReject* $\widehat{=}$

when

`isin_Control` : $Control = TRUE$

```
        isin_Invalid : $Invalid = TRUE$
        MON_guards7 : $valClk = 0$
        MON_guards6 : $delta1 > THRESHOLD$
        isin_Memorised : $Memorised = TRUE$
    then
        leave_Control : $Control := FALSE$
        enter_Input : $Input := TRUE$
        leave_Memorised : $Memorised := FALSE$
        leave_Invalid : $Invalid := FALSE$
        enter_Rejected : $Rejected := TRUE$
        enter_Normal : $Normal := TRUE$
        act1 : $memMean := mean23$
        act2 : $aoaFCPC := mean23$
    end
Event endMemRejected $\widehat{=}$
    when
        isin_Control : $Control = TRUE$
        isin_Rejected : $Rejected = TRUE$
        isin_Memorised : $Memorised = TRUE$
    then
        leave_Control : $Control := FALSE$
        enter_Input : $Input := TRUE$
        leave_Memorised : $Memorised := FALSE$
        enter_Normal : $Normal := TRUE$
        COM_actions6 : $aoaFCPC := mean23$
        COM_actions5 : $memMean := mean23$
    end
END
```

Appendix D

Models of the Landing Gear System

D.1 Modelica Models of the Plant, Cockpit and Landing Equipment

D.1.1 Plant

```
model Plant
  ElectroValve OpenDoorsEV;
  ElectroValve CloseDoorsEV;
  LatchedCylinder FrontDoor
    (extendedLatch=false,
    Hmin=HydraulicPressure.k,
    tRetract=1.2,
    tRetractedLock=0.3,
    tRetractedUnlock=0.4);
  LatchedCylinder LeftDoor
    (extendedLatch=false,
    Hmin=HydraulicPressure.k,
    tExtend=1.5,
    tRetractedLock=0.3,
    tRetractedUnlock=0.4);
  LatchedCylinder RightDoor
    (extendedLatch=false,
    Hmin=HydraulicPressure.k,
    tRetractedLock=0.3,
    tRetractedUnlock=0.4,
    tExtend=1.5);
  ElectroValve ExtendGearsEV;
  ElectroValve RetractGearsEV;
```

```
LatchedCylinder FrontGear(Hmin=HydraulicPressure.k);
LatchedCylinder LeftGear(Hmin=HydraulicPressure.k,
  tExtend=1.6,
  tRetract=2.0);
LatchedCylinder RightGear(Hmin=HydraulicPressure.k,
  tExtend=1.6,
  tRetract=2.0);
AnalogicalSwitch analogicalSwitch;
Modelica.Blocks.Sources.Constant HydraulicPressure(k=5);
ElectroValve GeneralEV;
Modelica.Blocks.Sources.BooleanExpression booleanExpression
  (y = GeneralEV.Hout >= HydraulicPressure.k);
Modelica.Blocks.Sources.BooleanExpression booleanExpression1
  (y = FrontDoor.retractedLocked);
Modelica.Blocks.Sources.BooleanExpression booleanExpression2
  (y = LeftDoor.retractedLocked);
Modelica.Blocks.Sources.BooleanExpression booleanExpression3
  (y = RightDoor.retractedLocked);
Modelica.Blocks.Sources.BooleanExpression booleanExpression4
  (y = FrontDoor.Pos >= FrontDoor.length
      and FrontDoor.Hext >= HydraulicPressure.k);
Modelica.Blocks.Sources.BooleanExpression booleanExpression5
  (y = RightDoor.Pos >= RightDoor.length
      and RightDoor.Hext >= HydraulicPressure.k);
Modelica.Blocks.Sources.BooleanExpression booleanExpression6
  (y = LeftDoor.Pos >= LeftDoor.length
      and LeftDoor.Hext >= HydraulicPressure.k);
Modelica.Blocks.Sources.BooleanExpression booleanExpression7
  (y = FrontGear.retractedLocked);
Modelica.Blocks.Sources.BooleanExpression booleanExpression8
  (y = RightGear.retractedLocked);
Modelica.Blocks.Sources.BooleanExpression booleanExpression9
  (y = LeftGear.retractedLocked);
Modelica.Blocks.Sources.BooleanExpression booleanExpression10
  (y = FrontGear.extendedLocked);
Modelica.Blocks.Sources.BooleanExpression booleanExpression11
  (y = RightGear.extendedLocked);
Modelica.Blocks.Sources.BooleanExpression booleanExpression12
  (y = LeftGear.extendedLocked);
Modelica.Blocks.Interfaces.BooleanOutput pressure;
Modelica.Blocks.Interfaces.BooleanInput generalEV;
Modelica.Blocks.Interfaces.BooleanInput openEV;
Modelica.Blocks.Interfaces.BooleanInput closeEV;
Modelica.Blocks.Interfaces.BooleanInput extendEV;
Modelica.Blocks.Interfaces.BooleanInput retractEV;
Modelica.Blocks.Interfaces.BooleanOutput retracted;
Modelica.Blocks.Interfaces.BooleanOutput open;
```

```
  Modelica.Blocks.Interfaces.BooleanOutput closed;
  Modelica.Blocks.Interfaces.BooleanOutput extended;
  Modelica.Blocks.Interfaces.BooleanOutput switch;
  And3 and3_1;
  And3 and3_2;
  And3 and3_3;
  And3 and3_4;
  Cockpit cockpit(switchTable={1,11,15,24});
  Modelica.Blocks.Interfaces.BooleanOutput handle;
  Modelica.Blocks.Sources.RealExpression realExpression(y=FrontDoor.Pos);
  Modelica.Blocks.Sources.RealExpression realExpression1(y=FrontGear.Pos);
  Modelica.Blocks.Interfaces.RealOutput doorPosition;
  Modelica.Blocks.Interfaces.RealOutput gearPosition;
equation
  connect(HydraulicPressure.y, GeneralEV.Hin);
  connect(analogicalSwitch.y, GeneralEV.E);
  connect(GeneralEV.Hout, OpenDoorsEV.Hin);
  connect(GeneralEV.Hout, ExtendGearsEV.Hin);
  connect(GeneralEV.Hout, CloseDoorsEV.Hin);
  connect(GeneralEV.Hout, RetractGearsEV.Hin);
  connect(OpenDoorsEV.Hout, FrontDoor.Hext);
  connect(LeftDoor.Hext, OpenDoorsEV.Hout);
  connect(RightDoor.Hext, OpenDoorsEV.Hout);
  connect(FrontDoor.Hret, CloseDoorsEV.Hout);
  connect(LeftDoor.Hret, CloseDoorsEV.Hout);
  connect(RightDoor.Hret, CloseDoorsEV.Hout);
  connect(ExtendGearsEV.Hout, FrontGear.Hext);
  connect(LeftGear.Hext, FrontGear.Hext);
  connect(RightGear.Hext, FrontGear.Hext);
  connect(FrontGear.Hret, RetractGearsEV.Hout);
  connect(LeftGear.Hret, RetractGearsEV.Hout);
  connect(RightGear.Hret, RetractGearsEV.Hout);
  connect(generalEV, analogicalSwitch.u);
  connect(openEV, OpenDoorsEV.E);
  connect(closeEV, CloseDoorsEV.E);
  connect(extendEV, ExtendGearsEV.E);
  connect(retractEV, RetractGearsEV.E);
  connect(booleanExpression.y, pressure);
  connect(analogicalSwitch.state, switch);
  connect(booleanExpression1.y, and3_1.u1);
  connect(booleanExpression3.y, and3_1.u2);
  connect(booleanExpression2.y, and3_1.u3);
  connect(and3_1.y, closed);
  connect(and3_2.y, open);
  connect(booleanExpression6.y, and3_2.u3);
  connect(booleanExpression4.y, and3_2.u1);
  connect(booleanExpression5.y, and3_2.u2);
  connect(booleanExpression9.y, and3_4.u3);
  connect(booleanExpression7.y, and3_4.u1);
  connect(booleanExpression8.y, and3_4.u2);
  connect(and3_4.y, retracted);
  connect(extended, and3_3.y);
  connect(booleanExpression10.y, and3_3.u1);
  connect(booleanExpression12.y, and3_3.u3);
  connect(booleanExpression11.y, and3_3.u2);
  connect(cockpit.handle, analogicalSwitch.handle);
  connect(cockpit.handle, handle);
  connect(realExpression1.y, gearPosition);
  connect(doorPosition, realExpression.y);
end Plant;
```

D.1.2 Cockpit

```
model LandingGear.Cockpit
  parameter Real switchTable[:] = {0,1} "Table of handle switch events";
  Modelica.Blocks.Interfaces.BooleanOutput handle
    "Handle position (true=down/extend)";
  Modelica.Blocks.Sources.BooleanTable booleanTable(table=switchTable);
equation
  connect(handle, booleanTable.y);
end Cockpit;
```

D.1.3 Electro-Valve

```
model LandingGear.ElectroValve
  "Abstract model of the hydraulic electro-valve with delays based on
    the varying valve opening"
  Modelica.Blocks.Interfaces.BooleanInput E(start = false)
    "Input electrical order";
  Modelica.Blocks.Interfaces.RealInput Hin "Hydraulic input pressure";
  Modelica.Blocks.Interfaces.RealOutput Hout(start = 0.0)
    "Hydraulic output pressure";
  parameter Real closingTime = 1.0 "Closing duration";
  parameter Real openingTime = 3.6 "Opening duration";
protected
  parameter Real Rmax = 1.0;
  parameter Real dRcl = Rmax/closingTime
    "Rate of change of the ratio R when closing";
  parameter Real dRop = -Rmax/openingTime
    "Rate of change of the ratio R when opening";
  Real R(start = 0.0)
    "Closed circuit ratio (0.0 - completely open, 1.0 - fully closed)";
  discrete Real dR(start = 0.0);
equation
  Hout = Hin*R;
  der(R) = dR;
algorithm
  // closing/opening event
  when E then
    dR := dRcl;
     elsewhen
          not E then
    dR := dRop;
  end when;
  // limiter of the R value
  when R <= 0 or R >= Rmax then
    dR := 0;
  end when;
end ElectroValve;
```

D.1.4 Latched Cylinder

```
model LandingGear.LatchedCylinder
  "Hydraulic pressure cylinder with two latches"
  parameter Real length = 1.0 "Cylinder length";
  parameter Real initPos = 0
    "Initial position of the cylinde (from 0 to length)";
  parameter Boolean initRetLatch = true "Initial state of retracted latch";
  parameter Boolean initExtLatch = false "Initial state of extended latch";
  Modelica.Blocks.Interfaces.RealInput Hext
    "Hydraulic pressure input to extend the cylinder";
  Modelica.Blocks.Interfaces.RealInput Hret
    "Hydraulic pressure input to retract the cylinder";
  Modelica.Blocks.Interfaces.RealOutput Pos(start = initPos)
    "Cylinder position";
  parameter Real tExtend = 1.2 "Extension duration";
  parameter Real tRetract = 1.6 "Retraction duration";
  parameter Boolean extendedLatch = true
    "=true, if cylinder latch in extended position is enabled";
  parameter Real tExtendedLock = 0.4 "Lock delay in extended position";
  parameter Real tExtendedUnlock = 0.8 "Unlock delay from extended position";
  parameter Boolean retractedLatch = true
    "=true, if cylinder latch in retracted position is enabled";
  parameter Real tRetractedLock = 0.4 "Lock delay in retracted position";
  parameter Real tRetractedUnlock = 0.8 "Unlock delay from retracted position";
  parameter Real Hmin = 0.001 "Minimal pressure to move the cylinder";
  output Boolean retractedLocked(start = initRetLatch)
    "Status of retracted latch, if enabled";
  output Boolean extendedLocked(start = initExtLatch)
    "Status of extended latch, if enabled";
protected
  parameter Real dPosExt = length/tExtend;
  parameter Real dPosRet = -length/tRetract;
public
  Lock retLock(
    lockTime=tRetractedLock,
    unlockTime=tRetractedUnlock,
    initialLock=initRetLatch);
  Modelica.Blocks.Logical.GreaterEqual greaterEqual;
  Modelica.Blocks.Sources.RealExpression realExpression(y=Hmin);
  Lock extLock(
    initialLock=initExtLatch,
    lockTime=tExtendedLock,
    unlockTime=tExtendedUnlock);
  Modelica.Blocks.Logical.GreaterEqual greaterEqual1;
  Modelica.Blocks.Logical.And and1;
  Modelica.Blocks.Sources.BooleanExpression booleanExpression(y=Pos<=0);
  Modelica.Blocks.Logical.And and2;
  Modelica.Blocks.Logical.And and3;
  Modelica.Blocks.Logical.And and4;
  Modelica.Blocks.Sources.BooleanExpression booleanExpression1(
      y=Pos>=length);
  Modelica.Blocks.Math.Add add(k1=-1, k2=+1);
  Modelica.Blocks.Math.Add add1(k1=-1, k2=+1);
equation
  retractedLocked = retractedLatch and retLock.locked;
  extendedLocked = extendedLatch and extLock.locked;

  if Hext-Hret >= Hmin and not retractedLocked then
    der(Pos) = if Pos < length then dPosExt else 0;
  else
    if Hret-Hext >= Hmin and not extendedLocked then
      der(Pos) = if Pos > 0 then dPosRet else 0;
    else
      der(Pos) = 0;
    end if;
  end if;

  connect(realExpression.y, greaterEqual.u2);
  connect(greaterEqual.y, and1.u1);
  connect(and1.y, retLock.uUnlock);
  connect(and2.y, retLock.uLock);
  connect(realExpression.y, greaterEqual1.u2);
  connect(booleanExpression.y, and2.u2);
  connect(booleanExpression1.y, and4.u1);
  connect(and4.y, extLock.uLock);
  connect(and3.y, extLock.uUnlock);
  connect(greaterEqual1.y, and2.u1);
  connect(greaterEqual1.y, and3.u2);
  connect(greaterEqual.y, and4.u2);
  connect(retLock.locked, and1.u2);
  connect(extLock.locked, and3.u1);
  connect(greaterEqual.u1, add.y);
  connect(Hext, add.u2);
  connect(Hret, add.u1);
  connect(Hret, add1.u2);
  connect(Hext, add1.u1);
  connect(add1.y, greaterEqual1.u1);
end LatchedCylinder;
```

D.1.5 Analogical Switch

```
model LandingGear.AnalogicalSwitch
  "Handle-controlled four-state analogical switch that passes input
    signal only in the closed state"
  Modelica_StateGraph2.Step Open(initialStep=true, nOut=1,
    nIn=1);
  Modelica_StateGraph2.Transition startClosing(use_conditionPort=true,
      loopCheck=false);
  Modelica_StateGraph2.Step Closing(
    use_activePort=true,
    nIn=1,
    nOut=1);
  Modelica_StateGraph2.Transition finishClosing(use_conditionPort=true);
  Modelica.Blocks.Interfaces.BooleanInput handle;
  Modelica_StateGraph2.Blocks.MathBoolean.OnDelay onDelay(
    delayTime=tClose);
  parameter Real tClose = 0.8 "Closing delay";
  parameter Real tOpen = 1.2 "Opening delay";
  parameter Real tClosed = 20.0 "Duration in the closed state";
  Modelica_StateGraph2.Step Closed(
    nOut=2,
    initialStep=false,
    nIn=3,
    use_activePort=true);
  Modelica_StateGraph2.Transition startOpening(use_conditionPort=true);
  Modelica_StateGraph2.Step Opening(
    nOut=2,
    initialStep=false,
    nIn=1,
    use_activePort=true);
  Modelica_StateGraph2.Blocks.MathBoolean.OnDelay onDelay1(
    delayTime=tClosed);
  Modelica.Blocks.Interfaces.BooleanOutput state "=true, if closed";
  Modelica_StateGraph2.Transition finishOpening(
    use_conditionPort=true);
  Modelica_StateGraph2.Blocks.MathBoolean.OnDelay onDelay2(
    delayTime=tOpen);
  Modelica.Blocks.Logical.Or or1;
  Modelica.Blocks.Interfaces.BooleanInput u;
  Modelica.Blocks.Interfaces.BooleanOutput y;
  Modelica.Blocks.Logical.And and1;
  Modelica_StateGraph2.Transition T1(use_conditionPort=true,
      delayedTransition=false);
  Modelica_StateGraph2.Step Closed2(
    nIn=1,
    nOut=1,
    use_activePort=true);
  Modelica_StateGraph2.LoopBreakingTransition T2(use_conditionPort=false);
  Modelica.Blocks.Logical.Timer timer;
  Modelica.Blocks.Logical.GreaterEqual greater;
  Modelica.Blocks.Sources.RealExpression realExpression(
      y=0.8 - 2*tClosing.y/3);
  Modelica.Blocks.Logical.Timer timer1;
  Modelica.Blocks.Discrete.TriggeredSampler tClosing;
  Modelica_StateGraph2.Transition T3(use_conditionPort=true,
      use_firePort=true);
  Modelica_StateGraph2.Step Closing2(
    use_activePort=true,
    nIn=1,
    nOut=1);
  Modelica_StateGraph2.LoopBreakingTransition T5(use_conditionPort=true);
  Modelica.Blocks.Logical.Change change1;
equation
  connect(Open.outPort[1], startClosing.inPort);
  connect(startClosing.outPort, Closing.inPort[1]);
  connect(Closing.outPort[1], finishClosing.inPort);
  connect(Closing.activePort, onDelay.u);
  connect(onDelay.y, finishClosing.conditionPort);
  connect(finishClosing.outPort, Closed.inPort[1]);
  connect(Closed.outPort[1], startOpening.inPort);
  connect(Closed.activePort, onDelay1.u);
  connect(onDelay1.y, startOpening.conditionPort);
  connect(startOpening.outPort, Opening.inPort[1]);
  connect(Opening.outPort[1], finishOpening.inPort);
  connect(Opening.activePort, onDelay2.u);
  connect(onDelay2.y, finishOpening.conditionPort);
  connect(finishOpening.outPort, Open.inPort[1]);
  connect(or1.y, state);
  connect(y, and1.y);
  connect(T1.outPort, Closed2.inPort[1]);
  connect(Closed2.outPort[1], T2.inPort);
  connect(T2.outPort, Closed.inPort[2]);
  connect(T1.inPort, Closed.outPort[2]);
  connect(u, and1.u2);
  connect(or1.y, and1.u1);
  connect(Closed2.activePort, or1.u1);
  connect(Closed.activePort, or1.u2);
  connect(Opening.activePort, timer1.u);
  connect(timer.y, greater.u1);
  connect(T3.inPort, Opening.outPort[2]);
  connect(T3.outPort, Closing2.inPort[1]);
  connect(Closing2.outPort[1], T5.inPort);
  connect(T5.outPort, Closed.inPort[3]);
  connect(T3.firePort, tClosing.trigger);
  connect(timer1.y, tClosing.u);
  connect(realExpression.y, greater.u2);
  connect(T5.conditionPort, greater.y);
  connect(Closing2.activePort, timer.u);
  connect(startClosing.conditionPort, change1.y);
  connect(T1.conditionPort, change1.y);
  connect(T3.conditionPort, change1.y);
  connect(handle, change1.u);
end AnalogicalSwitch;
```

D.1.6 Latching Lock

```
model LandingGear.Lock
  "Analogical two-state lock with a lock/unlock delay"
  parameter Boolean initialLock = true "Initially locked";
  Modelica_StateGraph2.Step Unlocked(
    nOut=1,
    nIn=1,
    use_activePort=false,
    initialStep=not initialLock);
  Modelica_StateGraph2.Transition lock(use_conditionPort=true,
      delayedTransition=true,
    waitTime=lockTime);
  Modelica_StateGraph2.Step Locked(
    nIn=1,
    nOut=1,
    use_activePort=true,
    initialStep=initialLock);
  Modelica_StateGraph2.Transition unlock(
    use_conditionPort=true,
    delayedTransition=true,
    waitTime=unlockTime);
  Modelica.Blocks.Interfaces.BooleanInput uLock;
  Modelica.Blocks.Interfaces.BooleanOutput locked;
  parameter Real lockTime = 0.4 "Locking duration";
  parameter Real unlockTime = 0.8 "Unlocking duration";
  Modelica.Blocks.Interfaces.BooleanInput uUnlock;
equation
  connect(Unlocked.outPort[1], lock.inPort);
  connect(lock.outPort, Locked.inPort[1]);
  connect(Locked.outPort[1], unlock.inPort);
  connect(unlock.outPort, Unlocked.inPort[1]);
  connect(uLock, lock.conditionPort);
  connect(Locked.activePort, locked);
  connect(uUnlock, unlock.conditionPort);
end Lock;
```

D.1.7 Auxiliary Model And3

```
model LandingGear.And3 "Logical 'and': y = u1 and u2 and u3"
  Modelica.Blocks.Interfaces.BooleanInput u1;
  Modelica.Blocks.Interfaces.BooleanInput u2;
  Modelica.Blocks.Interfaces.BooleanInput u3;
  Modelica.Blocks.Interfaces.BooleanOutput y;
equation
  y = u1 and u2 and u3;
end And3;
```

D.2 Event-B Specifications of the Controller

D.2.1 Abstract Context ctx1

CONTEXT ctx1
CONSTANTS

$START\_INTERVAL$

$STOP\_INTERVAL$

AXIOMS

axm1 : $START\_INTERVAL = 2$

axm2 : $STOP\_INTERVAL = 10$

END

D.2.2 Concrete Context ctx2

CONTEXT ctx2
EXTENDS ctx1
CONSTANTS

$CONTR\_INTERVAL$

AXIOMS

`axm1` : $CONTR\_INTERVAL = 1$

END

D.2.3 Abstract Machine mch1

MACHINE mch1
VARIABLES

$HandleDown$

$HandleUp$

$FailureMode$

$NormalMode$

$Block$

$Control$

$Read$

$handle$

INVARIANTS

`typeof_HandleDown` : $HandleDown \in BOOL$

`typeof_HandleUp` : $HandleUp \in BOOL$

`distinct_states_in_smHandle` : $partition(\{TRUE\}, \{HandleUp\} \cap \{TRUE\}, \{HandleDown\} \cap \{TRUE\})$

`typeof_FailureMode` : $FailureMode \in BOOL$

`typeof_NormalMode` : $NormalMode \in BOOL$

`distinct_states_in_smControl` : $partition(\{TRUE\}, \{NormalMode\} \cap \{TRUE\}, \{FailureMode\} \cap \{TRUE\})$

`typeof_Block` : $Block \in BOOL$

`typeof_Control` : $Control \in BOOL$

`typeof_Read` : $Read \in BOOL$

`distinct_states_in_smFMU` : $partition(\{TRUE\}, \{Read\} \cap \{TRUE\}, \{Control\} \cap \{TRUE\}, \{Block\} \cap \{TRUE\})$

`inv1` : $handle \in BOOL$

EVENTS

Initialisation

begin

`init_HandleUp` : $HandleUp := TRUE$

`init_HandleDown` : $HandleDown := FALSE$

`init_NormalMode` : $NormalMode := TRUE$

```
        init_FailureMode : FailureMode := FALSE
        init_Read : Read := TRUE
        init_Block : Block := FALSE
        init_Control : Control := FALSE
        act1 : handle := FALSE
    end
Event stayUp ≙
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards1 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
    end
Event switchDown ≙
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards2 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
    then
        leave_HandleUp : HandleUp := FALSE
        enter_HandleDown : HandleDown := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
    end
Event stayDown ≙
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards3 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
    end
Event switchUp ≙
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards4 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
    then
        leave_HandleDown : HandleDown := FALSE
```

`enter_HandleUp` : $HandleUp := TRUE$
`leave_Control` : $Control := FALSE$
`enter_Read` : $Read := TRUE$

end

Event *fail* $\widehat{=}$

when

`isin_NormalMode` : $NormalMode = TRUE$
`isin_Control` : $Control = TRUE$

then

`leave_NormalMode` : $NormalMode := FALSE$
`enter_FailureMode` : $FailureMode := TRUE$
`leave_Control` : $Control := FALSE$
`enter_Block` : $Block := TRUE$

end

Event *readInput* $\widehat{=}$

any

h handle position (TRUE = down, FALSE = up)

where

`isin_Read` : $Read = TRUE$
`h_type` : $h \in BOOL$

then

`leave_Read` : $Read := FALSE$
`enter_Control` : $Control := TRUE$
`smFMU_actions1` : $handle := h$

end

END

D.2.4 First Refinement mch2

MACHINE mch2
REFINES mch1
VARIABLES

FailureMode
generalEV
Extended
Retracted
NormalMode
Block
Control
Read
HandleDown
HandleUp
handle

INVARIANTS

typeof_generalEV : $generalEV \in BOOL$

typeof_Extended : $Extended \in BOOL$

typeof_Retracted : $Retracted \in BOOL$

distinct_states_in_smNormalMode : $(NormalMode = TRUE) \Rightarrow partition(\{TRUE\}, \{Retracted\} \cap \{TRUE\}, \{Extended\} \cap \{TRUE\}, \{generalEV\} \cap \{TRUE\})$

generalEV_substateof_NormalMode : $(generalEV = TRUE) \Rightarrow (NormalMode = TRUE)$

Extended_substateof_NormalMode : $(Extended = TRUE) \Rightarrow (NormalMode = TRUE)$

Retracted_substateof_NormalMode : $(Retracted = TRUE) \Rightarrow (NormalMode = TRUE)$

EVENTS

Initialisation

extended

begin

init_HandleUp : $HandleUp := TRUE$

init_HandleDown : $HandleDown := FALSE$

init_NormalMode : $NormalMode := TRUE$

init_FailureMode : $FailureMode := FALSE$

init_Read : $Read := TRUE$

init_Block : $Block := FALSE$

init_Control : $Control := FALSE$

act1 : $handle := FALSE$

init_Retracted : $Retracted := TRUE$

init_generalEV : $generalEV := FALSE$

init_Extended : $Extended := FALSE$

end

Event *stayUp* $\widehat{=}$

extends *stayUp*

when

isin_HandleUp : $HandleUp = TRUE$

smHandle_guards1 : $handle = FALSE$

isin_NormalMode : $NormalMode = TRUE$

isin_Control : $Control = TRUE$

isin_generalEV : $generalEV = TRUE$

then

leave_Control : $Control := FALSE$

enter_Read : $Read := TRUE$

end

Event *stayDown* $\widehat{=}$

extends *stayDown*

when

isin_HandleDown : $HandleDown = TRUE$

smHandle_guards3 : $handle = TRUE$

isin_NormalMode : $NormalMode = TRUE$

```
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event   $switchUp$ $\widehat{=}$
extends $switchUp$
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards4 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
      then
            leave_HandleDown : $HandleDown := FALSE$
            enter_HandleUp : $HandleUp := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event   $switchDown$ $\widehat{=}$
extends $switchDown$
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards2 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
      then
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event   $fail$ $\widehat{=}$
extends $fail$
      when
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
      then
            leave_NormalMode : $NormalMode := FALSE$
            enter_FailureMode : $FailureMode := TRUE$
            leave_Control : $Control := FALSE$
            enter_Block : $Block := TRUE$
            leave_generalEV : $generalEV := FALSE$
            leave_Extended : $Extended := FALSE$
```

```
        leave_Retracted : $Retracted := FALSE$
    end
Event  readInput $\widehat{=}$
extends  readInput
    any
        $h$  handle position (TRUE = down, FALSE = up)
    where
        isin_Read : $Read = TRUE$
        h_type : $h \in BOOL$
    then
        leave_Read : $Read := FALSE$
        enter_Control : $Control := TRUE$
        smFMU_actions1 : $handle := h$
    end
Event  extend $\widehat{=}$
extends  switchDown
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards2 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_Retracted : $Retracted = TRUE$
    then
        leave_HandleUp : $HandleUp := FALSE$
        enter_HandleDown : $HandleDown := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_Retracted : $Retracted := FALSE$
        enter_generalEV : $generalEV := TRUE$
    end
Event  retract $\widehat{=}$
extends  switchUp
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards4 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_Extended : $Extended = TRUE$
    then
        leave_HandleDown : $HandleDown := FALSE$
        enter_HandleUp : $HandleUp := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_Extended : $Extended := FALSE$
        enter_generalEV : $generalEV := TRUE$
    end
```

```
Event  keepRet ≙
extends  stayUp
     when
          isin_HandleUp : HandleUp = TRUE
          smHandle_guards1 : handle = FALSE
          isin_NormalMode : NormalMode = TRUE
          isin_Control : Control = TRUE
          isin_Retracted : Retracted = TRUE
     then
          leave_Control : Control := FALSE
          enter_Read : Read := TRUE
     end
Event  keepExt ≙
extends  stayDown
     when
          isin_HandleDown : HandleDown = TRUE
          smHandle_guards3 : handle = TRUE
          isin_NormalMode : NormalMode = TRUE
          isin_Control : Control = TRUE
          isin_Extended : Extended = TRUE
     then
          leave_Control : Control := FALSE
          enter_Read : Read := TRUE
     end
Event  setRet ≙
extends  stayUp
     when
          isin_HandleUp : HandleUp = TRUE
          smHandle_guards1 : handle = FALSE
          isin_NormalMode : NormalMode = TRUE
          isin_Control : Control = TRUE
          isin_generalEV : generalEV = TRUE
     then
          leave_Control : Control := FALSE
          enter_Read : Read := TRUE
          leave_generalEV : generalEV := FALSE
          enter_Retracted : Retracted := TRUE
     end
Event  cancelExt ≙
extends  switchUp
     when
          isin_HandleDown : HandleDown = TRUE
          smHandle_guards4 : handle = FALSE
          isin_NormalMode : NormalMode = TRUE
          isin_Control : Control = TRUE
          isin_generalEV : generalEV = TRUE
```

```
    then
        leave_HandleDown : HandleDown := FALSE
        enter_HandleUp : HandleUp := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_generalEV : generalEV := FALSE
        enter_Retracted : Retracted := TRUE
    end
Event  setExt ≙
extends  stayDown
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards3 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_generalEV : generalEV := FALSE
        enter_Extended : Extended := TRUE
    end
Event  cancelRet ≙
extends  switchDown
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards2 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
    then
        leave_HandleUp : HandleUp := FALSE
        enter_HandleDown : HandleDown := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_generalEV : generalEV := FALSE
        enter_Extended : Extended := TRUE
    end
END
```

D.2.5 Second Refinement mch3

```
MACHINE  mch3
REFINES  mch2
SEES  ctx1
VARIABLES
```

Block

Control

Read

FailureMode

Maneuvering

PendingStart

PendingStop

generalEV

Extended

Retracted

NormalMode

HandleDown

HandleUp

handle

tStart

tStop

INVARIANTS

`typeof_Maneuvering` : $Maneuvering \in BOOL$

`typeof_PendingStart` : $PendingStart \in BOOL$

`typeof_PendingStop` : $PendingStop \in BOOL$

`distinct_states_in_smGeneralEV` : $(generalEV = TRUE) \Rightarrow$
$partition(\{TRUE\}, \{PendingStop\} \cap \{TRUE\},$
$\{PendingStart\} \cap \{TRUE\}, \{Maneuvering\} \cap \{TRUE\})$

`Maneuvering_substateof_generalEV` : $(Maneuvering = TRUE) \Rightarrow$
$(generalEV = TRUE)$

`PendingStart_substateof_generalEV` : $(PendingStart = TRUE) \Rightarrow$
$(generalEV = TRUE)$

`PendingStop_substateof_generalEV` : $(PendingStop = TRUE) \Rightarrow$
$(generalEV = TRUE)$

`inv1` : $tStart \in \mathbb{N}$

`inv2` : $tStop \in \mathbb{N}$

EVENTS

Initialisation

extended

begin

`init_HandleUp` : $HandleUp := TRUE$
`init_HandleDown` : $HandleDown := FALSE$
`init_NormalMode` : $NormalMode := TRUE$
`init_FailureMode` : $FailureMode := FALSE$
`init_Read` : $Read := TRUE$
`init_Block` : $Block := FALSE$
`init_Control` : $Control := FALSE$
`act1` : $handle := FALSE$
`init_Retracted` : $Retracted := TRUE$

```
            init_generalEV : generalEV := FALSE
            init_Extended : Extended := FALSE
            init_Maneuvering : Maneuvering := FALSE
            init_PendingStart : PendingStart := FALSE
            init_PendingStop : PendingStop := FALSE
            act2 : tStart := 0
            act3 : tStop := 0
      end
Event  stayUp ≙
extends stayUp
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
      end
Event  stayDown ≙
extends stayDown
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
      end
Event  switchUp ≙
extends switchUp
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
      then
            leave_HandleDown : HandleDown := FALSE
            enter_HandleUp : HandleUp := TRUE
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event   switchDown $\widehat{=}$
extends switchDown
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards2 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
      then
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event   fail $\widehat{=}$
extends fail
      when
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
      then
            leave_NormalMode : $NormalMode := FALSE$
            enter_FailureMode : $FailureMode := TRUE$
            leave_Control : $Control := FALSE$
            enter_Block : $Block := TRUE$
            leave_generalEV : $generalEV := FALSE$
            leave_Extended : $Extended := FALSE$
            leave_Retracted : $Retracted := FALSE$
            leave_Maneuvering : $Maneuvering := FALSE$
            leave_PendingStart : $PendingStart := FALSE$
            leave_PendingStop : $PendingStop := FALSE$
      end
Event   readInput $\widehat{=}$
extends readInput
      any
           $h$   handle position (TRUE = down, FALSE = up)
      where
            isin_Read : $Read = TRUE$
            h_type : $h \in BOOL$
      then
            leave_Read : $Read := FALSE$
            enter_Control : $Control := TRUE$
            smFMU_actions1 : $handle := h$
```

```
    end
Event  extend ≙
extends  extend
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards2 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_Retracted : Retracted = TRUE
    then
        leave_HandleUp : HandleUp := FALSE
        enter_HandleDown : HandleDown := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_Retracted : Retracted := FALSE
        enter_generalEV : generalEV := TRUE
        enter_PendingStart : PendingStart := TRUE
        smNormalMode_actions1 : tStart := 1
    end
Event  retract ≙
extends  retract
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards4 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_Extended : Extended = TRUE
    then
        leave_HandleDown : HandleDown := FALSE
        enter_HandleUp : HandleUp := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_Extended : Extended := FALSE
        enter_generalEV : generalEV := TRUE
        enter_PendingStart : PendingStart := TRUE
        smNormalMode_actions2 : tStart := 1
    end
Event  keepRet ≙
extends  keepRet
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards1 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_Retracted : Retracted = TRUE
    then
        leave_Control : Control := FALSE
        enter_Read : $Read := TRUE$
  end
Event  keepExt $\widehat{=}$
extends  keepExt
  when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards3 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_Extended : $Extended = TRUE$
  then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
  end
Event  setRet $\widehat{=}$
extends  setRet
  when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards1 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_PendingStop : $PendingStop = TRUE$
        smNormalMode_guards1 : $tStop \geq STOP\_INTERVAL$
  then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_generalEV : $generalEV := FALSE$
        enter_Retracted : $Retracted := TRUE$
        leave_PendingStop : $PendingStop := FALSE$
  end
Event  cancelExt $\widehat{=}$
extends  cancelExt
  when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards4 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_PendingStart : $PendingStart = TRUE$
  then
        leave_HandleDown : $HandleDown := FALSE$
        enter_HandleUp : $HandleUp := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_generalEV : $generalEV := FALSE$
```

```
            enter_Retracted : $Retracted := TRUE$
            leave_PendingStart : $PendingStart := FALSE$
      end
Event   $setExt \mathrel{\widehat{=}}$
extends $setExt$
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStop : $PendingStop = TRUE$
            smNormalMode_guards2 : $tStop \geq STOP\_INTERVAL$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_generalEV : $generalEV := FALSE$
            enter_Extended : $Extended := TRUE$
            leave_PendingStop : $PendingStop := FALSE$
      end
Event   $cancelRet \mathrel{\widehat{=}}$
extends $cancelRet$
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards2 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
      then
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_generalEV : $generalEV := FALSE$
            enter_Extended : $Extended := TRUE$
            leave_PendingStart : $PendingStart := FALSE$
      end
Event   $startExt \mathrel{\widehat{=}}$
extends $stayDown$
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            _guards3 : $tStart \geq START\_INTERVAL$
     then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingStart : $PendingStart := FALSE$
            enter_Maneuvering : $Maneuvering := TRUE$
     end
Event  $startRet \ \widehat{=}$
extends  $stayUp$
     when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            _guards3 : $tStart \geq START\_INTERVAL$
     then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingStart : $PendingStart := FALSE$
            enter_Maneuvering : $Maneuvering := TRUE$
     end
Event  $endExt \ \widehat{=}$
extends  $stayDown$
     when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            _guards2 : $tStop < STOP\_INTERVAL$
     then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_Maneuvering : $Maneuvering := FALSE$
            enter_PendingStop : $PendingStop := TRUE$
            _actions3 : $tStop := 1$
     end
Event  $endRet \ \widehat{=}$
extends  $stayUp$
     when
            isin_HandleUp : $HandleUp = TRUE$
```

```
        smHandle_guards1 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        _guards2 : tStop < STOP_INTERVAL
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_Maneuvering : Maneuvering := FALSE
        enter_PendingStop : PendingStop := TRUE
        _actions3 : tStop := 1
    end
Event abortExt ≙
extends switchUp
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards4 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        _guards2 : tStop < STOP_INTERVAL
    then
        leave_HandleDown : HandleDown := FALSE
        enter_HandleUp : HandleUp := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_Maneuvering : Maneuvering := FALSE
        enter_PendingStop : PendingStop := TRUE
        _actions3 : tStop := 1
    end
Event abortRet ≙
extends switchDown
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards2 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        _guards2 : tStop < STOP_INTERVAL
    then
        leave_HandleUp : HandleUp := FALSE
        enter_HandleDown : HandleDown := TRUE
        leave_Control : Control := FALSE
```

```
            enter_Read : $Read := TRUE$
            leave_Maneuvering : $Maneuvering := FALSE$
            enter_PendingStop : $PendingStop := TRUE$
            _actions3 : $tStop := 1$
      end
Event  undoExt ≙
extends  switchUp
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards4 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStop : $PendingStop = TRUE$
      then
            leave_HandleDown : $HandleDown := FALSE$
            enter_HandleUp : $HandleUp := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingStop : $PendingStop := FALSE$
            enter_Maneuvering : $Maneuvering := TRUE$
      end
Event  undoRet ≙
extends  switchDown
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards2 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStop : $PendingStop = TRUE$
      then
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingStop : $PendingStop := FALSE$
            enter_Maneuvering : $Maneuvering := TRUE$
      end
Event  delayExt ≙
extends  stayDown
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            _guards1 : $tStart < START\_INTERVAL$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            _actions1 : $tStart := tStart + 1$
      end
Event  delayRet $\hat{=}$
extends stayUp
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            _guards1 : $tStart < START\_INTERVAL$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            _actions1 : $tStart := tStart + 1$
      end
Event  delaySetExt $\hat{=}$
extends stayDown
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStop : $PendingStop = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            _actions2 : $tStop := tStop + 1$
      end
Event  delaySetRet $\hat{=}$
extends stayUp
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStop : $PendingStop = TRUE$
```

then

`leave_Control` : $Control := FALSE$
`enter_Read` : $Read := TRUE$
`_actions2` : $tStop := tStop + 1$

end

END

D.2.6 Third Refinement mch4

MACHINE mch4
REFINES mch3
SEES ctx2
VARIABLES

$FailureMode$
$PendingContrDoor$
$openEV$
$closeEV$
$Maneuvering$
$PendingStart$
$PendingStop$
$generalEV$
$Extended$
$Retracted$
$NormalMode$
$Block$
$Control$
$Read$
$HandleDown$
$HandleUp$
$handle$
$tStart$
$tStop$
$tContrDoor$

INVARIANTS

`typeof_PendingContrDoor` : $PendingContrDoor \in BOOL$
`typeof_openEV` : $openEV \in BOOL$
`typeof_closeEV` : $closeEV \in BOOL$
`distinct_states_in_smManeuvering` : $(Maneuvering = TRUE) \Rightarrow partition(\{TRUE\}, \{closeEV\} \cap \{TRUE\}, \{openEV\} \cap \{TRUE\}, \{PendingContrDoor\} \cap \{TRUE\})$
`PendingContrDoor_substateof_Maneuvering` : $(PendingContrDoor = TRUE) \Rightarrow (Maneuvering = TRUE)$

openEV_substateof_Maneuvering : $(openEV = TRUE) \Rightarrow (Maneuvering = TRUE)$
closeEV_substateof_Maneuvering : $(closeEV = TRUE) \Rightarrow (Maneuvering = TRUE)$
inv1 : $tContrDoor \in \mathbb{N}$

EVENTS

Initialisation

extended

begin

init_HandleUp : $HandleUp := TRUE$
init_HandleDown : $HandleDown := FALSE$
init_NormalMode : $NormalMode := TRUE$
init_FailureMode : $FailureMode := FALSE$
init_Read : $Read := TRUE$
init_Block : $Block := FALSE$
init_Control : $Control := FALSE$
act1 : $handle := FALSE$
init_Retracted : $Retracted := TRUE$
init_generalEV : $generalEV := FALSE$
init_Extended : $Extended := FALSE$
init_Maneuvering : $Maneuvering := FALSE$
init_PendingStart : $PendingStart := FALSE$
init_PendingStop : $PendingStop := FALSE$
act2 : $tStart := 0$
act3 : $tStop := 0$
init_PendingContrDoor : $PendingContrDoor := FALSE$
init_openEV : $openEV := FALSE$
init_closeEV : $closeEV := FALSE$
act4 : $tContrDoor := 0$

end

Event *stayUp* $\widehat{=}$
extends *stayUp*

when

isin_HandleUp : $HandleUp = TRUE$
smHandle_guards1 : $handle = FALSE$
isin_NormalMode : $NormalMode = TRUE$
isin_Control : $Control = TRUE$
isin_generalEV : $generalEV = TRUE$
isin_Maneuvering : $Maneuvering = TRUE$
isin_openEV : $openEV = TRUE$

then

leave_Control : $Control := FALSE$
enter_Read : $Read := TRUE$

end

Event *stayDown* $\widehat{=}$
extends *stayDown*

```
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards3 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
    end
Event  switchUp $\widehat{=}$
extends switchUp
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards4 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
    then
        leave_HandleDown : $HandleDown := FALSE$
        enter_HandleUp : $HandleUp := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
    end
Event  switchDown $\widehat{=}$
extends switchDown
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards2 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
    then
        leave_HandleUp : $HandleUp := FALSE$
        enter_HandleDown : $HandleDown := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
    end
Event  fail $\widehat{=}$
extends fail
```

```
    when
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
    then
        leave_NormalMode : NormalMode := FALSE
        enter_FailureMode : FailureMode := TRUE
        leave_Control : Control := FALSE
        enter_Block : Block := TRUE
        leave_generalEV : generalEV := FALSE
        leave_Extended : Extended := FALSE
        leave_Retracted : Retracted := FALSE
        leave_Maneuvering : Maneuvering := FALSE
        leave_PendingStart : PendingStart := FALSE
        leave_PendingStop : PendingStop := FALSE
        leave_PendingContrDoor : PendingContrDoor := FALSE
        leave_openEV : openEV := FALSE
        leave_closeEV : closeEV := FALSE
    end
Event  readInput ≙
extends readInput
    any
        h   handle position (TRUE = down, FALSE = up)
    where
        isin_Read : Read = TRUE
        h_type : h ∈ BOOL
    then
        leave_Read : Read := FALSE
        enter_Control : Control := TRUE
        smFMU_actions1 : handle := h
    end
Event  extend ≙
extends extend
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards2 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_Retracted : Retracted = TRUE
    then
        leave_HandleUp : HandleUp := FALSE
        enter_HandleDown : HandleDown := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_Retracted : Retracted := FALSE
        enter_generalEV : generalEV := TRUE
        enter_PendingStart : PendingStart := TRUE
        smNormalMode_actions1 : tStart := 1
    end
Event  retract ≙
extends  retract
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards4 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_Extended : Extended = TRUE
    then
        leave_HandleDown : HandleDown := FALSE
        enter_HandleUp : HandleUp := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_Extended : Extended := FALSE
        enter_generalEV : generalEV := TRUE
        enter_PendingStart : PendingStart := TRUE
        smNormalMode_actions2 : tStart := 1
    end
Event  keepRet ≙
extends  keepRet
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards1 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_Retracted : Retracted = TRUE
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
    end
Event  keepExt ≙
extends  keepExt
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards3 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_Extended : Extended = TRUE
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
    end
Event  setRet ≙
extends  setRet
```

when

    isin_HandleUp : $HandleUp = TRUE$
    smHandle_guards1 : $handle = FALSE$
    isin_NormalMode : $NormalMode = TRUE$
    isin_Control : $Control = TRUE$
    isin_generalEV : $generalEV = TRUE$
    isin_PendingStop : $PendingStop = TRUE$
    smNormalMode_guards1 : $tStop \geq STOP\_INTERVAL$

then

    leave_Control : $Control := FALSE$
    enter_Read : $Read := TRUE$
    leave_generalEV : $generalEV := FALSE$
    enter_Retracted : $Retracted := TRUE$
    leave_PendingStop : $PendingStop := FALSE$

end

Event *cancelExt* $\widehat{=}$
extends *cancelExt*

when

    isin_HandleDown : $HandleDown = TRUE$
    smHandle_guards4 : $handle = FALSE$
    isin_NormalMode : $NormalMode = TRUE$
    isin_Control : $Control = TRUE$
    isin_generalEV : $generalEV = TRUE$
    isin_PendingStart : $PendingStart = TRUE$

then

    leave_HandleDown : $HandleDown := FALSE$
    enter_HandleUp : $HandleUp := TRUE$
    leave_Control : $Control := FALSE$
    enter_Read : $Read := TRUE$
    leave_generalEV : $generalEV := FALSE$
    enter_Retracted : $Retracted := TRUE$
    leave_PendingStart : $PendingStart := FALSE$

end

Event *setExt* $\widehat{=}$
extends *setExt*

when

    isin_HandleDown : $HandleDown = TRUE$
    smHandle_guards3 : $handle = TRUE$
    isin_NormalMode : $NormalMode = TRUE$
    isin_Control : $Control = TRUE$
    isin_generalEV : $generalEV = TRUE$
    isin_PendingStop : $PendingStop = TRUE$
    smNormalMode_guards2 : $tStop \geq STOP\_INTERVAL$

then

    leave_Control : $Control := FALSE$
    enter_Read : $Read := TRUE$

```
            leave_generalEV : $generalEV := FALSE$
            enter_Extended : $Extended := TRUE$
            leave_PendingStop : $PendingStop := FALSE$
      end
Event  cancelRet ≙
extends  cancelRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards2 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
      then
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_generalEV : $generalEV := FALSE$
            enter_Extended : $Extended := TRUE$
            leave_PendingStart : $PendingStart := FALSE$
      end
Event  startExt ≙
extends  startExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            _guards3 : $tStart \geq START\_INTERVAL$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingStart : $PendingStart := FALSE$
            enter_Maneuvering : $Maneuvering := TRUE$
            enter_openEV : $openEV := TRUE$
      end
Event  startRet ≙
extends  startRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            _guards3 : $tStart \geq START\_INTERVAL$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingStart : $PendingStart := FALSE$
            enter_Maneuvering : $Maneuvering := TRUE$
            enter_openEV : $openEV := TRUE$
      end
Event  endExt $\widehat{=}$
extends  endExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            _guards2 : $tStop < STOP\_INTERVAL$
            isin_closeEV : $closeEV = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_Maneuvering : $Maneuvering := FALSE$
            enter_PendingStop : $PendingStop := TRUE$
            _actions3 : $tStop := 1$
            leave_closeEV : $closeEV := FALSE$
      end
Event  endRet $\widehat{=}$
extends  endRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            _guards2 : $tStop < STOP\_INTERVAL$
            isin_closeEV : $closeEV = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_Maneuvering : $Maneuvering := FALSE$
            enter_PendingStop : $PendingStop := TRUE$
            _actions3 : $tStop := 1$
```

```
        leave_closeEV : closeEV := FALSE
    end
Event   undoExt ≙
extends undoExt
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards4 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_PendingStop : PendingStop = TRUE
    then
        leave_HandleDown : HandleDown := FALSE
        enter_HandleUp : HandleUp := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_PendingStop : PendingStop := FALSE
        enter_Maneuvering : Maneuvering := TRUE
        enter_PendingContrDoor : PendingContrDoor := TRUE
        smGeneralEV_actions1 : tContrDoor := tStop
    end
Event   undoRet ≙
extends undoRet
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards2 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_PendingStop : PendingStop = TRUE
    then
        leave_HandleUp : HandleUp := FALSE
        enter_HandleDown : HandleDown := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_PendingStop : PendingStop := FALSE
        enter_Maneuvering : Maneuvering := TRUE
        enter_PendingContrDoor : PendingContrDoor := TRUE
        smGeneralEV_actions1 : tContrDoor := tStop
    end
Event   delayExt ≙
extends delayExt
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards3 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
```

```
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStart : PendingStart = TRUE
            _guards1 : tStart < START_INTERVAL
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            _actions1 : tStart := tStart + 1
      end
Event   delayRet ≙
extends delayRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStart : PendingStart = TRUE
            _guards1 : tStart < START_INTERVAL
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            _actions1 : tStart := tStart + 1
      end
Event   delaySetExt ≙
extends delaySetExt
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStop : PendingStop = TRUE
            smGeneralEV_guards1 : tStop < STOP_INTERVAL
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            _actions2 : tStop := tStop + 1
      end
Event   delaySetRet ≙
extends delaySetRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStop : PendingStop = TRUE
            smGeneralEV_guards1 : tStop < STOP_INTERVAL
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            _actions2 : tStop := tStop + 1
      end
Event   resumeEndExt ≙
extends  abortRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            _guards2 : tStop < STOP_INTERVAL
            isin_PendingContrDoor : PendingContrDoor = TRUE
      then
            leave_HandleUp : HandleUp := FALSE
            enter_HandleDown : HandleDown := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_Maneuvering : Maneuvering := FALSE
            enter_PendingStop : PendingStop := TRUE
            _actions3 : tStop := 1
            leave_PendingContrDoor : PendingContrDoor := FALSE
      end
Event   resumeEndRet ≙
extends  abortExt
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            _guards2 : tStop < STOP_INTERVAL
            isin_PendingContrDoor : PendingContrDoor = TRUE
      then
            leave_HandleDown : HandleDown := FALSE
            enter_HandleUp : HandleUp := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_Maneuvering : Maneuvering := FALSE
            enter_PendingStop : $PendingStop := TRUE$
            _actions3 : $tStop := 1$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
     end
Event  abortOpenExt $\widehat{=}$
extends  abortExt
     when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards4 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            _guards2 : $tStop < STOP\_INTERVAL$
            isin_openEV : $openEV = TRUE$
     then
            leave_HandleDown : $HandleDown := FALSE$
            enter_HandleUp : $HandleUp := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_Maneuvering : $Maneuvering := FALSE$
            enter_PendingStop : $PendingStop := TRUE$
            _actions3 : $tStop := 1$
            leave_openEV : $openEV := FALSE$
     end
Event  abortOpenRet $\widehat{=}$
extends  abortRet
     when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards2 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            _guards2 : $tStop < STOP\_INTERVAL$
            isin_openEV : $openEV = TRUE$
     then
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_Maneuvering : $Maneuvering := FALSE$
            enter_PendingStop : $PendingStop := TRUE$
            _actions3 : $tStop := 1$
            leave_openEV : $openEV := FALSE$
     end
```

Event *closeExt* $\widehat{=}$
extends *stayDown*

when

```
isin_HandleDown : HandleDown = TRUE
smHandle_guards3 : handle = TRUE
isin_NormalMode : NormalMode = TRUE
isin_Control : Control = TRUE
isin_generalEV : generalEV = TRUE
isin_Maneuvering : Maneuvering = TRUE
isin_PendingContrDoor : PendingContrDoor = TRUE
smManeuvering_guards3 : tContrDoor ≥ CONTR_INTERVAL
```

then

```
leave_Control : Control := FALSE
enter_Read : Read := TRUE
leave_PendingContrDoor : PendingContrDoor := FALSE
enter_closeEV : closeEV := TRUE
```

end

Event *closeRet* $\widehat{=}$
extends *stayUp*

when

```
isin_HandleUp : HandleUp = TRUE
smHandle_guards1 : handle = FALSE
isin_NormalMode : NormalMode = TRUE
isin_Control : Control = TRUE
isin_generalEV : generalEV = TRUE
isin_Maneuvering : Maneuvering = TRUE
isin_PendingContrDoor : PendingContrDoor = TRUE
smManeuvering_guards3 : tContrDoor ≥ CONTR_INTERVAL
```

then

```
leave_Control : Control := FALSE
enter_Read : Read := TRUE
leave_PendingContrDoor : PendingContrDoor := FALSE
enter_closeEV : closeEV := TRUE
```

end

Event *cancelCloseExt* $\widehat{=}$
extends *switchUp*

when

```
isin_HandleDown : HandleDown = TRUE
smHandle_guards4 : handle = FALSE
isin_NormalMode : NormalMode = TRUE
isin_Control : Control = TRUE
isin_generalEV : generalEV = TRUE
isin_Maneuvering : Maneuvering = TRUE
isin_closeEV : closeEV = TRUE
```

then

```
leave_HandleDown : HandleDown := FALSE
```

```
            enter_HandleUp : HandleUp := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_closeEV : closeEV := FALSE
            enter_PendingContrDoor : PendingContrDoor := TRUE
            smManeuvering_actions3 : tContrDoor := 1
      end
Event   cancelCloseRet ≙
extends switchDown
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_closeEV : closeEV = TRUE
      then
            leave_HandleUp : HandleUp := FALSE
            enter_HandleDown : HandleDown := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_closeEV : closeEV := FALSE
            enter_PendingContrDoor : PendingContrDoor := TRUE
            smManeuvering_actions3 : tContrDoor := 1
      end
Event   recloseExt ≙
extends switchDown
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_PendingContrDoor : PendingContrDoor = TRUE
      then
            leave_HandleUp : HandleUp := FALSE
            enter_HandleDown : HandleDown := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingContrDoor : PendingContrDoor := FALSE
            enter_closeEV : closeEV := TRUE
      end
Event   recloseRet ≙
extends switchUp
```

```
     when
          isin_HandleDown : $HandleDown = TRUE$
          smHandle_guards4 : $handle = FALSE$
          isin_NormalMode : $NormalMode = TRUE$
          isin_Control : $Control = TRUE$
          isin_generalEV : $generalEV = TRUE$
          isin_Maneuvering : $Maneuvering = TRUE$
          isin_PendingContrDoor : $PendingContrDoor = TRUE$
     then
          leave_HandleDown : $HandleDown := FALSE$
          enter_HandleUp : $HandleUp := TRUE$
          leave_Control : $Control := FALSE$
          enter_Read : $Read := TRUE$
          leave_PendingContrDoor : $PendingContrDoor := FALSE$
          enter_closeEV : $closeEV := TRUE$
     end
Event  delayContrDoorExt $\widehat{=}$
extends  stayDown
     when
          isin_HandleDown : $HandleDown = TRUE$
          smHandle_guards3 : $handle = TRUE$
          isin_NormalMode : $NormalMode = TRUE$
          isin_Control : $Control = TRUE$
          isin_generalEV : $generalEV = TRUE$
          isin_Maneuvering : $Maneuvering = TRUE$
          isin_PendingContrDoor : $PendingContrDoor = TRUE$
          smManeuvering_guards2 : $tContrDoor < CONTR\_INTERVAL$
     then
          leave_Control : $Control := FALSE$
          enter_Read : $Read := TRUE$
          smManeuvering_actions2 : $tContrDoor := tContrDoor + 1$
     end
Event  delayContrDoorRet $\widehat{=}$
extends  stayUp
     when
          isin_HandleUp : $HandleUp = TRUE$
          smHandle_guards1 : $handle = FALSE$
          isin_NormalMode : $NormalMode = TRUE$
          isin_Control : $Control = TRUE$
          isin_generalEV : $generalEV = TRUE$
          isin_Maneuvering : $Maneuvering = TRUE$
          isin_PendingContrDoor : $PendingContrDoor = TRUE$
          smManeuvering_guards2 : $tContrDoor < CONTR\_INTERVAL$
     then
          leave_Control : $Control := FALSE$
          enter_Read : $Read := TRUE$
```

```
        smManeuvering_actions2 : tContrDoor := tContrDoor + 1
    end
Event  endOpenExt ≙
extends stayDown
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards3 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_openEV : openEV = TRUE
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_openEV : openEV := FALSE
        enter_PendingContrDoor : PendingContrDoor := TRUE
        smManeuvering_actions1 : tContrDoor := 1
    end
Event  endOpenRet ≙
extends stayUp
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards1 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_openEV : openEV = TRUE
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_openEV : openEV := FALSE
        enter_PendingContrDoor : PendingContrDoor := TRUE
        smManeuvering_actions1 : tContrDoor := 1
    end
Event  cancelOpenExt ≙
extends switchUp
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards4 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_openEV : $openEV = TRUE$
    then
        leave_HandleDown : $HandleDown := FALSE$
        enter_HandleUp : $HandleUp := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_openEV : $openEV := FALSE$
        enter_PendingContrDoor : $PendingContrDoor := TRUE$
        smManeuvering_actions1 : $tContrDoor := 1$
    end
Event cancelOpenRet $\widehat{=}$
extends switchDown
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards2 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
    then
        leave_HandleUp : $HandleUp := FALSE$
        enter_HandleDown : $HandleDown := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_openEV : $openEV := FALSE$
        enter_PendingContrDoor : $PendingContrDoor := TRUE$
        smManeuvering_actions1 : $tContrDoor := 1$
    end
Event reopenExt $\widehat{=}$
extends switchDown
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards2 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_PendingContrDoor : $PendingContrDoor = TRUE$
    then
        leave_HandleUp : $HandleUp := FALSE$
        enter_HandleDown : $HandleDown := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_PendingContrDoor : $PendingContrDoor := FALSE$
        enter_openEV : $openEV := TRUE$
    end
Event   reopenRet ≙
extends  switchUp
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards4 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_PendingContrDoor : PendingContrDoor = TRUE
    then
        leave_HandleDown : HandleDown := FALSE
        enter_HandleUp : HandleUp := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_PendingContrDoor : PendingContrDoor := FALSE
        enter_openEV : openEV := TRUE
    end
Event   openExt ≙
extends  stayDown
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards3 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_PendingContrDoor : PendingContrDoor = TRUE
        smManeuvering_guards1 : tContrDoor ≥ CONTR_INTERVAL
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_PendingContrDoor : PendingContrDoor := FALSE
        enter_openEV : openEV := TRUE
    end
Event   openRet ≙
extends  stayUp
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards1 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_PendingContrDoor : PendingContrDoor = TRUE
```

```
            smManeuvering_guards1 : $tContrDoor \geq CONTR\_INTERVAL$
    then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            enter_openEV : $openEV := TRUE$
    end
Event  waitClosedExt ≙
extends  stayDown
    when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_closeEV : $closeEV = TRUE$
    then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
    end
Event  waitClosedRet ≙
extends  stayUp
    when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_closeEV : $closeEV = TRUE$
    then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
    end
END
```

D.2.7 Fourth Refinement mch5

```
MACHINE  mch5
REFINES  mch4
SEES  ctx2
VARIABLES
        Block
        Control
```

Read

FailureMode

PendingClose

PendingOpen

PendingContrDoor

GearsManeuvering

DoorsOpening

openEV

closeEV

Maneuvering

PendingStart

PendingStop

generalEV

Extended

Retracted

NormalMode

HandleDown

HandleUp

handle

tStart

tStop

tContrDoor

doorsOpen

doorsClosed

INVARIANTS

`typeof_PendingClose` : $PendingClose \in BOOL$

`typeof_PendingOpen` : $PendingOpen \in BOOL$

`typeof_GearsManeuvering` : $GearsManeuvering \in BOOL$

`typeof_DoorsOpening` : $DoorsOpening \in BOOL$

`distinct_states_in_smPendingContrDoor` : $(PendingContrDoor = TRUE) \Rightarrow partition(\{TRUE\}, \{PendingOpen\} \cap \{TRUE\}, \{PendingClose\} \cap \{TRUE\})$

`distinct_states_in_smOpenEV` : $(openEV = TRUE) \Rightarrow partition(\{TRUE\}, \{DoorsOpening\} \cap \{TRUE\}, \{GearsManeuvering\} \cap \{TRUE\})$

`PendingClose_substateof_PendingContrDoor` : $(PendingClose = TRUE) \Rightarrow (PendingContrDoor = TRUE)$

`PendingOpen_substateof_PendingContrDoor` : $(PendingOpen = TRUE) \Rightarrow (PendingContrDoor = TRUE)$

`GearsManeuvering_substateof_openEV` : $(GearsManeuvering = TRUE) \Rightarrow (openEV = TRUE)$

`DoorsOpening_substateof_openEV` : $(DoorsOpening = TRUE) \Rightarrow (openEV = TRUE)$

`inv1` : $doorsOpen \in BOOL$

`inv2` : $doorsClosed \in BOOL$

EVENTS

Initialisation

extended

begin

`init_HandleUp` : $HandleUp := TRUE$
`init_HandleDown` : $HandleDown := FALSE$
`init_NormalMode` : $NormalMode := TRUE$
`init_FailureMode` : $FailureMode := FALSE$
`init_Read` : $Read := TRUE$
`init_Block` : $Block := FALSE$
`init_Control` : $Control := FALSE$
`act1` : $handle := FALSE$
`init_Retracted` : $Retracted := TRUE$
`init_generalEV` : $generalEV := FALSE$
`init_Extended` : $Extended := FALSE$
`init_Maneuvering` : $Maneuvering := FALSE$
`init_PendingStart` : $PendingStart := FALSE$
`init_PendingStop` : $PendingStop := FALSE$
`act2` : $tStart := 0$
`act3` : $tStop := 0$
`init_PendingContrDoor` : $PendingContrDoor := FALSE$
`init_openEV` : $openEV := FALSE$
`init_closeEV` : $closeEV := FALSE$
`act4` : $tContrDoor := 0$
`init_PendingClose` : $PendingClose := FALSE$
`init_PendingOpen` : $PendingOpen := FALSE$
`init_GearsManeuvering` : $GearsManeuvering := FALSE$
`init_DoorsOpening` : $DoorsOpening := FALSE$
`act5` : $doorsOpen := FALSE$
`act6` : $doorsClosed := TRUE$

end

Event *stayUp* $\widehat{=}$
extends *stayUp*

when

`isin_HandleUp` : $HandleUp = TRUE$
`smHandle_guards1` : $handle = FALSE$
`isin_NormalMode` : $NormalMode = TRUE$
`isin_Control` : $Control = TRUE$
`isin_generalEV` : $generalEV = TRUE$
`isin_Maneuvering` : $Maneuvering = TRUE$
`isin_openEV` : $openEV = TRUE$
`isin_GearsManeuvering` : $GearsManeuvering = TRUE$
`smControl_guards1` : $doorsOpen = TRUE$

then

`leave_Control` : $Control := FALSE$
`enter_Read` : $Read := TRUE$

end

Event *stayDown* $\widehat{=}$

```
extends  stayDown
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_openEV : openEV = TRUE
            isin_GearsManeuvering : GearsManeuvering = TRUE
            smControl_guards1 : doorsOpen = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
      end
Event  switchUp ≙
extends  switchUp
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_openEV : openEV = TRUE
            isin_GearsManeuvering : GearsManeuvering = TRUE
            smControl_guards1 : doorsOpen = TRUE
      then
            leave_HandleDown : HandleDown := FALSE
            enter_HandleUp : HandleUp := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
      end
Event  switchDown ≙
extends  switchDown
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_openEV : openEV = TRUE
            isin_GearsManeuvering : GearsManeuvering = TRUE
            smControl_guards1 : doorsOpen = TRUE
      then
```

```
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event  fail ≙
extends  fail
      when
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
      then
            leave_NormalMode : $NormalMode := FALSE$
            enter_FailureMode : $FailureMode := TRUE$
            leave_Control : $Control := FALSE$
            enter_Block : $Block := TRUE$
            leave_generalEV : $generalEV := FALSE$
            leave_Extended : $Extended := FALSE$
            leave_Retracted : $Retracted := FALSE$
            leave_Maneuvering : $Maneuvering := FALSE$
            leave_PendingStart : $PendingStart := FALSE$
            leave_PendingStop : $PendingStop := FALSE$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            leave_openEV : $openEV := FALSE$
            leave_closeEV : $closeEV := FALSE$
            leave_PendingClose : $PendingClose := FALSE$
            leave_PendingOpen : $PendingOpen := FALSE$
            leave_GearsManeuvering : $GearsManeuvering := FALSE$
            leave_DoorsOpening : $DoorsOpening := FALSE$
      end
Event  readInput ≙
extends  readInput
      any
           $h$   handle position (TRUE = down, FALSE = up)
           $dc$  doors closed
           $do$  doors open
      where
            isin_Read : $Read = TRUE$
            h_type : $h \in BOOL$
            dc_type : $dc \in BOOL$
            do_type : $do \in BOOL$
      then
            leave_Read : $Read := FALSE$
            enter_Control : $Control := TRUE$
            smFMU_actions1 : $handle := h$
            smFMU_actions3 : $doorsClosed := dc$
            smFMU_actions2 : $doorsOpen := do$
```

```
    end
Event    extend ≙
extends  extend
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards2 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_Retracted : Retracted = TRUE
        smNormalMode_guards8 : doorsClosed = TRUE
    then
        leave_HandleUp : HandleUp := FALSE
        enter_HandleDown : HandleDown := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_Retracted : Retracted := FALSE
        enter_generalEV : generalEV := TRUE
        enter_PendingStart : PendingStart := TRUE
        smNormalMode_actions1 : tStart := 1
    end
Event    retract ≙
extends  retract
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards4 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_Extended : Extended = TRUE
        smNormalMode_guards7 : doorsClosed = TRUE
    then
        leave_HandleDown : HandleDown := FALSE
        enter_HandleUp : HandleUp := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_Extended : Extended := FALSE
        enter_generalEV : generalEV := TRUE
        enter_PendingStart : PendingStart := TRUE
        smNormalMode_actions2 : tStart := 1
    end
Event    keepRet ≙
extends  keepRet
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards1 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
```

```
            isin_Retracted : $Retracted = TRUE$
            smNormalMode_guards4 : $doorsClosed = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event   keepExt $\widehat{=}$
extends keepExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_Extended : $Extended = TRUE$
            smNormalMode_guards5 : $doorsClosed = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event   setRet $\widehat{=}$
extends setRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStop : $PendingStop = TRUE$
            smNormalMode_guards1 : $tStop \geq STOP\_INTERVAL$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_generalEV : $generalEV := FALSE$
            enter_Retracted : $Retracted := TRUE$
            leave_PendingStop : $PendingStop := FALSE$
      end
Event   cancelExt $\widehat{=}$
extends cancelExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards4 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            smGeneralEV_guards6 : $doorsClosed = TRUE$
```

```
    then
        leave_HandleDown : HandleDown := FALSE
        enter_HandleUp : HandleUp := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_generalEV : generalEV := FALSE
        enter_Retracted : Retracted := TRUE
        leave_PendingStart : PendingStart := FALSE
    end
Event  setExt ≙
extends setExt
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards3 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_PendingStop : PendingStop = TRUE
        smNormalMode_guards2 : tStop ≥ STOP_INTERVAL
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_generalEV : generalEV := FALSE
        enter_Extended : Extended := TRUE
        leave_PendingStop : PendingStop := FALSE
    end
Event  cancelRet ≙
extends cancelRet
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards2 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_PendingStart : PendingStart = TRUE
        smGeneralEV_guards4 : doorsClosed = TRUE
    then
        leave_HandleUp : HandleUp := FALSE
        enter_HandleDown : HandleDown := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_generalEV : generalEV := FALSE
        enter_Extended : Extended := TRUE
        leave_PendingStart : PendingStart := FALSE
    end
Event  startExt ≙
```

```
extends startExt
     when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStart : PendingStart = TRUE
            _guards3 : tStart ≥ START_INTERVAL
     then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingStart : PendingStart := FALSE
            enter_Maneuvering : Maneuvering := TRUE
            enter_openEV : openEV := TRUE
            enter_DoorsOpening : DoorsOpening := TRUE
     end
Event  startRet ≙
extends startRet
     when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStart : PendingStart = TRUE
            _guards3 : tStart ≥ START_INTERVAL
     then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingStart : PendingStart := FALSE
            enter_Maneuvering : Maneuvering := TRUE
            enter_openEV : openEV := TRUE
            enter_DoorsOpening : DoorsOpening := TRUE
     end
Event  endExt ≙
extends endExt
     when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            _guards2 : tStop < STOP_INTERVAL
            isin_closeEV : closeEV = TRUE
            smGeneralEV_guards2 : doorsClosed = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_Maneuvering : Maneuvering := FALSE
            enter_PendingStop : PendingStop := TRUE
            _actions3 : tStop := 1
            leave_closeEV : closeEV := FALSE
      end
Event   endRet ≙
extends  endRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            _guards2 : tStop < STOP_INTERVAL
            isin_closeEV : closeEV = TRUE
            smGeneralEV_guards2 : doorsClosed = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_Maneuvering : Maneuvering := FALSE
            enter_PendingStop : PendingStop := TRUE
            _actions3 : tStop := 1
            leave_closeEV : closeEV := FALSE
      end
Event   undoExt ≙
extends  undoExt
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStop : PendingStop = TRUE
      then
            leave_HandleDown : HandleDown := FALSE
            enter_HandleUp : HandleUp := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingStop : PendingStop := FALSE
            enter_Maneuvering : Maneuvering := TRUE
            enter_PendingContrDoor : PendingContrDoor := TRUE
            smGeneralEV_actions1 : tContrDoor := tStop
            enter_PendingOpen : PendingOpen := TRUE
      end
Event   undoRet ≙
extends  undoRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStop : PendingStop = TRUE
      then
            leave_HandleUp : HandleUp := FALSE
            enter_HandleDown : HandleDown := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingStop : PendingStop := FALSE
            enter_Maneuvering : Maneuvering := TRUE
            enter_PendingContrDoor : PendingContrDoor := TRUE
            smGeneralEV_actions1 : tContrDoor := tStop
            enter_PendingOpen : PendingOpen := TRUE
      end
Event   delayExt ≙
extends  delayExt
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStart : PendingStart = TRUE
            _guards1 : tStart < START_INTERVAL
            smGeneralEV_guards3 : doorsClosed = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            _actions1 : tStart := tStart + 1
      end
Event   delayRet ≙
extends  delayRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            _guards1 : $tStart < START\_INTERVAL$
            smGeneralEV_guards3 : $doorsClosed = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            _actions1 : $tStart := tStart + 1$
      end
Event  delaySetExt $\widehat{=}$
extends  delaySetExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStop : $PendingStop = TRUE$
            smGeneralEV_guards1 : $tStop < STOP\_INTERVAL$
            smGeneralEV_guards5 : $doorsClosed = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            _actions2 : $tStop := tStop + 1$
      end
Event  delaySetRet $\widehat{=}$
extends  delaySetRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStop : $PendingStop = TRUE$
            smGeneralEV_guards1 : $tStop < STOP\_INTERVAL$
            smGeneralEV_guards5 : $doorsClosed = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            _actions2 : $tStop := tStop + 1$
      end
Event  resumeEndExt $\widehat{=}$
extends  resumeEndExt
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards2 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            _guards2 : $tStop < STOP\_INTERVAL$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            isin_PendingOpen : $PendingOpen = TRUE$
            smManeuvering_guards5 : $doorsClosed = TRUE$
      then
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_Maneuvering : $Maneuvering := FALSE$
            enter_PendingStop : $PendingStop := TRUE$
            _actions3 : $tStop := 1$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            leave_PendingOpen : $PendingOpen := FALSE$
      end
Event   $resumeEndRet \widehat{=}$
extends $resumeEndRet$
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards4 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            _guards2 : $tStop < STOP\_INTERVAL$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            isin_PendingOpen : $PendingOpen = TRUE$
            smManeuvering_guards5 : $doorsClosed = TRUE$
      then
            leave_HandleDown : $HandleDown := FALSE$
            enter_HandleUp : $HandleUp := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_Maneuvering : $Maneuvering := FALSE$
            enter_PendingStop : $PendingStop := TRUE$
            _actions3 : $tStop := 1$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            leave_PendingOpen : $PendingOpen := FALSE$
      end
Event   $abortOpenExt \widehat{=}$
extends $abortOpenExt$
```

when

```
    isin_HandleDown : $HandleDown = TRUE$
    smHandle_guards4 : $handle = FALSE$
    isin_NormalMode : $NormalMode = TRUE$
    isin_Control : $Control = TRUE$
    isin_generalEV : $generalEV = TRUE$
    isin_Maneuvering : $Maneuvering = TRUE$
    _guards2 : $tStop < STOP\_INTERVAL$
    isin_openEV : $openEV = TRUE$
    isin_DoorsOpening : $DoorsOpening = TRUE$
    smManeuvering_guards6 : $doorsClosed = TRUE$
```

then

```
    leave_HandleDown : $HandleDown := FALSE$
    enter_HandleUp : $HandleUp := TRUE$
    leave_Control : $Control := FALSE$
    enter_Read : $Read := TRUE$
    leave_Maneuvering : $Maneuvering := FALSE$
    enter_PendingStop : $PendingStop := TRUE$
    _actions3 : $tStop := 1$
    leave_openEV : $openEV := FALSE$
    leave_DoorsOpening : $DoorsOpening := FALSE$
```

end

Event *abortOpenRet* $\widehat{=}$
extends *abortOpenRet*

when

```
    isin_HandleUp : $HandleUp = TRUE$
    smHandle_guards2 : $handle = TRUE$
    isin_NormalMode : $NormalMode = TRUE$
    isin_Control : $Control = TRUE$
    isin_generalEV : $generalEV = TRUE$
    isin_Maneuvering : $Maneuvering = TRUE$
    _guards2 : $tStop < STOP\_INTERVAL$
    isin_openEV : $openEV = TRUE$
    isin_DoorsOpening : $DoorsOpening = TRUE$
    smManeuvering_guards6 : $doorsClosed = TRUE$
```

then

```
    leave_HandleUp : $HandleUp := FALSE$
    enter_HandleDown : $HandleDown := TRUE$
    leave_Control : $Control := FALSE$
    enter_Read : $Read := TRUE$
    leave_Maneuvering : $Maneuvering := FALSE$
    enter_PendingStop : $PendingStop := TRUE$
    _actions3 : $tStop := 1$
    leave_openEV : $openEV := FALSE$
    leave_DoorsOpening : $DoorsOpening := FALSE$
```

end

Event *closeExt* $\widehat{=}$
extends *closeExt*

when

```
isin_HandleDown : $HandleDown = TRUE$
smHandle_guards3 : $handle = TRUE$
isin_NormalMode : $NormalMode = TRUE$
isin_Control : $Control = TRUE$
isin_generalEV : $generalEV = TRUE$
isin_Maneuvering : $Maneuvering = TRUE$
isin_PendingContrDoor : $PendingContrDoor = TRUE$
smManeuvering_guards3 : $tContrDoor \geq CONTR\_INTERVAL$
isin_PendingClose : $PendingClose = TRUE$
```

then

```
leave_Control : $Control := FALSE$
enter_Read : $Read := TRUE$
leave_PendingContrDoor : $PendingContrDoor := FALSE$
enter_closeEV : $closeEV := TRUE$
leave_PendingClose : $PendingClose := FALSE$
```

end

Event *closeRet* $\widehat{=}$
extends *closeRet*

when

```
isin_HandleUp : $HandleUp = TRUE$
smHandle_guards1 : $handle = FALSE$
isin_NormalMode : $NormalMode = TRUE$
isin_Control : $Control = TRUE$
isin_generalEV : $generalEV = TRUE$
isin_Maneuvering : $Maneuvering = TRUE$
isin_PendingContrDoor : $PendingContrDoor = TRUE$
smManeuvering_guards3 : $tContrDoor \geq CONTR\_INTERVAL$
isin_PendingClose : $PendingClose = TRUE$
```

then

```
leave_Control : $Control := FALSE$
enter_Read : $Read := TRUE$
leave_PendingContrDoor : $PendingContrDoor := FALSE$
enter_closeEV : $closeEV := TRUE$
leave_PendingClose : $PendingClose := FALSE$
```

end

Event *cancelCloseExt* $\widehat{=}$
extends *cancelCloseExt*

when

```
isin_HandleDown : $HandleDown = TRUE$
smHandle_guards4 : $handle = FALSE$
isin_NormalMode : $NormalMode = TRUE$
isin_Control : $Control = TRUE$
isin_generalEV : $generalEV = TRUE$
```

```
        isin_Maneuvering : Maneuvering = TRUE
        isin_closeEV : closeEV = TRUE
    then
        leave_HandleDown : HandleDown := FALSE
        enter_HandleUp : HandleUp := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_closeEV : closeEV := FALSE
        enter_PendingContrDoor : PendingContrDoor := TRUE
        smManeuvering_actions3 : tContrDoor := 1
        enter_PendingOpen : PendingOpen := TRUE
    end
Event  cancelCloseRet ≙
extends  cancelCloseRet
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards2 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_closeEV : closeEV = TRUE
    then
        leave_HandleUp : HandleUp := FALSE
        enter_HandleDown : HandleDown := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_closeEV : closeEV := FALSE
        enter_PendingContrDoor : PendingContrDoor := TRUE
        smManeuvering_actions3 : tContrDoor := 1
        enter_PendingOpen : PendingOpen := TRUE
    end
Event  recloseExt ≙
extends  recloseExt
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards2 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_PendingContrDoor : PendingContrDoor = TRUE
        isin_PendingOpen : PendingOpen = TRUE
    then
        leave_HandleUp : HandleUp := FALSE
        enter_HandleDown : HandleDown := TRUE
```

```
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingContrDoor : PendingContrDoor := FALSE
            enter_closeEV : closeEV := TRUE
            leave_PendingOpen : PendingOpen := FALSE
     end
Event   recloseRet ≙
extends  recloseRet
     when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_PendingContrDoor : PendingContrDoor = TRUE
            isin_PendingOpen : PendingOpen = TRUE
     then
            leave_HandleDown : HandleDown := FALSE
            enter_HandleUp : HandleUp := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingContrDoor : PendingContrDoor := FALSE
            enter_closeEV : closeEV := TRUE
            leave_PendingOpen : PendingOpen := FALSE
     end
Event   endOpenExt ≙
extends  endOpenExt
     when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_openEV : openEV = TRUE
            isin_GearsManeuvering : GearsManeuvering = TRUE
            _guards5 : doorsOpen = TRUE
     then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_openEV : openEV := FALSE
            enter_PendingContrDoor : PendingContrDoor := TRUE
            smManeuvering_actions1 : tContrDoor := 1
            leave_GearsManeuvering : GearsManeuvering := FALSE
            enter_PendingClose : PendingClose := TRUE
```

```
    end
Event  endOpenRet ≙
extends  endOpenRet
    when
          isin_HandleUp : HandleUp = TRUE
          smHandle_guards1 : handle = FALSE
          isin_NormalMode : NormalMode = TRUE
          isin_Control : Control = TRUE
          isin_generalEV : generalEV = TRUE
          isin_Maneuvering : Maneuvering = TRUE
          isin_openEV : openEV = TRUE
          isin_GearsManeuvering : GearsManeuvering = TRUE
          _guards5 : doorsOpen = TRUE
    then
          leave_Control : Control := FALSE
          enter_Read : Read := TRUE
          leave_openEV : openEV := FALSE
          enter_PendingContrDoor : PendingContrDoor := TRUE
          smManeuvering_actions1 : tContrDoor := 1
          leave_GearsManeuvering : GearsManeuvering := FALSE
          enter_PendingClose : PendingClose := TRUE
    end
Event  cancelOpenExt ≙
extends  cancelOpenExt
    when
          isin_HandleDown : HandleDown = TRUE
          smHandle_guards4 : handle = FALSE
          isin_NormalMode : NormalMode = TRUE
          isin_Control : Control = TRUE
          isin_generalEV : generalEV = TRUE
          isin_Maneuvering : Maneuvering = TRUE
          isin_openEV : openEV = TRUE
          isin_DoorsOpening : DoorsOpening = TRUE
          smManeuvering_guards7 : doorsClosed = FALSE
    then
          leave_HandleDown : HandleDown := FALSE
          enter_HandleUp : HandleUp := TRUE
          leave_Control : Control := FALSE
          enter_Read : Read := TRUE
          leave_openEV : openEV := FALSE
          enter_PendingContrDoor : PendingContrDoor := TRUE
          smManeuvering_actions1 : tContrDoor := 1
          leave_DoorsOpening : DoorsOpening := FALSE
          enter_PendingClose : PendingClose := TRUE
    end
Event  cancelOpenRet ≙
```

```
extends  cancelOpenRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_openEV : openEV = TRUE
            isin_DoorsOpening : DoorsOpening = TRUE
            smManeuvering_guards7 : doorsClosed = FALSE
      then
            leave_HandleUp : HandleUp := FALSE
            enter_HandleDown : HandleDown := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_openEV : openEV := FALSE
            enter_PendingContrDoor : PendingContrDoor := TRUE
            smManeuvering_actions1 : tContrDoor := 1
            leave_DoorsOpening : DoorsOpening := FALSE
            enter_PendingClose : PendingClose := TRUE
      end
Event  reopenExt ≙
extends  reopenExt
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_PendingContrDoor : PendingContrDoor = TRUE
            isin_PendingClose : PendingClose = TRUE
      then
            leave_HandleUp : HandleUp := FALSE
            enter_HandleDown : HandleDown := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingContrDoor : PendingContrDoor := FALSE
            enter_openEV : openEV := TRUE
            leave_PendingClose : PendingClose := FALSE
            enter_DoorsOpening : DoorsOpening := TRUE
      end
Event  reopenRet ≙
extends  reopenRet
      when
```

```
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards4 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            isin_PendingClose : $PendingClose = TRUE$
      then
            leave_HandleDown : $HandleDown := FALSE$
            enter_HandleUp : $HandleUp := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            enter_openEV : $openEV := TRUE$
            leave_PendingClose : $PendingClose := FALSE$
            enter_DoorsOpening : $DoorsOpening := TRUE$
      end
Event  openExt $\widehat{=}$
extends  openExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            smManeuvering_guards1 : $tContrDoor \geq CONTR\_INTERVAL$
            isin_PendingOpen : $PendingOpen = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            enter_openEV : $openEV := TRUE$
            leave_PendingOpen : $PendingOpen := FALSE$
            enter_DoorsOpening : $DoorsOpening := TRUE$
      end
Event  openRet $\widehat{=}$
extends  openRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
```

```
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_PendingContrDoor : $PendingContrDoor = TRUE$
        smManeuvering_guards1 : $tContrDoor \geq CONTR\_INTERVAL$
        isin_PendingOpen : $PendingOpen = TRUE$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_PendingContrDoor : $PendingContrDoor := FALSE$
        enter_openEV : $openEV := TRUE$
        leave_PendingOpen : $PendingOpen := FALSE$
        enter_DoorsOpening : $DoorsOpening := TRUE$
    end
Event  $waitClosedExt \widehat{=}$
extends $waitClosedExt$
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards3 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_closeEV : $closeEV = TRUE$
        smManeuvering_guards4 : $doorsClosed = FALSE$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
    end
Event  $waitClosedRet \widehat{=}$
extends $waitClosedRet$
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards1 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_closeEV : $closeEV = TRUE$
        smManeuvering_guards4 : $doorsClosed = FALSE$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
    end
Event  $waitOpenExt \widehat{=}$
extends $stayDown$
    when
        isin_HandleDown : $HandleDown = TRUE$
```

```
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_openEV : $openEV = TRUE$
            isin_DoorsOpening : $DoorsOpening = TRUE$
            _guards2 : $doorsOpen = FALSE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event  waitOpenRet $\widehat{=}$
extends stayUp
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_openEV : $openEV = TRUE$
            isin_DoorsOpening : $DoorsOpening = TRUE$
            _guards2 : $doorsOpen = FALSE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event  startGearExt $\widehat{=}$
extends stayDown
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_openEV : $openEV = TRUE$
            isin_DoorsOpening : $DoorsOpening = TRUE$
            _guards4 : $doorsOpen = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_DoorsOpening : $DoorsOpening := FALSE$
            enter_GearsManeuvering : $GearsManeuvering := TRUE$
      end
```

```
Event  startGearRet ≙
extends stayUp
     when
           isin_HandleUp : HandleUp = TRUE
           smHandle_guards1 : handle = FALSE
           isin_NormalMode : NormalMode = TRUE
           isin_Control : Control = TRUE
           isin_generalEV : generalEV = TRUE
           isin_Maneuvering : Maneuvering = TRUE
           isin_openEV : openEV = TRUE
           isin_DoorsOpening : DoorsOpening = TRUE
           _guards4 : doorsOpen = TRUE
     then
           leave_Control : Control := FALSE
           enter_Read : Read := TRUE
           leave_DoorsOpening : DoorsOpening := FALSE
           enter_GearsManeuvering : GearsManeuvering := TRUE
     end
Event  fDoorsNotOpen ≙
refines fail
     when
           isin_Control : Control = TRUE
           isin_GearsManeuvering : GearsManeuvering = TRUE
           smOpenEV_guards1 : doorsOpen = FALSE
     then
           leave_Control : Control := FALSE
           enter_Block : Block := TRUE
           leave_NormalMode : NormalMode := FALSE
           leave_generalEV : generalEV := FALSE
           leave_Maneuvering : Maneuvering := FALSE
           leave_openEV : openEV := FALSE
           leave_GearsManeuvering : GearsManeuvering := FALSE
           enter_FailureMode : FailureMode := TRUE
     end
Event  fDoorsUnlockedRet ≙
refines fail
     when
           isin_Control : Control = TRUE
           isin_Retracted : Retracted = TRUE
           smNormalMode_guards3 : doorsClosed = FALSE
     then
           leave_Control : Control := FALSE
           enter_Block : Block := TRUE
           leave_NormalMode : NormalMode := FALSE
           leave_Retracted : Retracted := FALSE
           enter_FailureMode : FailureMode := TRUE
```

```
        end
Event   fDoorsUnlockedExt ≙
refines fail
        when
            isin_Control : Control = TRUE
            isin_Extended : Extended = TRUE
            smNormalMode_guards6 : doorsClosed = FALSE
        then
            leave_Control : Control := FALSE
            enter_Block : Block := TRUE
            leave_NormalMode : NormalMode := FALSE
            leave_Extended : Extended := FALSE
            enter_FailureMode : FailureMode := TRUE
        end
Event   fDoorsUnlockedStart ≙
refines fail
        when
            isin_Control : Control = TRUE
            isin_PendingStart : PendingStart = TRUE
            smGeneralEV_guards7 : doorsClosed = FALSE
        then
            leave_Control : Control := FALSE
            enter_Block : Block := TRUE
            leave_NormalMode : NormalMode := FALSE
            leave_generalEV : generalEV := FALSE
            leave_PendingStart : PendingStart := FALSE
            enter_FailureMode : FailureMode := TRUE
        end
Event   fDoorsUnlockedStop ≙
refines fail
        when
            isin_Control : Control = TRUE
            isin_PendingStop : PendingStop = TRUE
            smGeneralEV_guards8 : doorsClosed = FALSE
        then
            leave_Control : Control := FALSE
            enter_Block : Block := TRUE
            leave_NormalMode : NormalMode := FALSE
            leave_generalEV : generalEV := FALSE
            leave_PendingStop : PendingStop := FALSE
            enter_FailureMode : FailureMode := TRUE
        end
Event   delayOpenExt ≙
extends delayContrDoorExt
        when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_PendingContrDoor : PendingContrDoor = TRUE
            smManeuvering_guards2 : tContrDoor < CONTR_INTERVAL
            isin_PendingOpen : PendingOpen = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            smManeuvering_actions2 : tContrDoor := tContrDoor + 1
      end
Event   delayOpenRet ≙
extends  delayContrDoorRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_PendingContrDoor : PendingContrDoor = TRUE
            smManeuvering_guards2 : tContrDoor < CONTR_INTERVAL
            isin_PendingOpen : PendingOpen = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            smManeuvering_actions2 : tContrDoor := tContrDoor + 1
      end
Event   delayCloseExt ≙
extends  delayContrDoorExt
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_PendingContrDoor : PendingContrDoor = TRUE
            smManeuvering_guards2 : tContrDoor < CONTR_INTERVAL
            isin_PendingClose : PendingClose = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            smManeuvering_actions2 : tContrDoor := tContrDoor + 1
```

```
    end
Event  delayCloseRet ≙
extends  delayContrDoorRet
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards1 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_PendingContrDoor : PendingContrDoor = TRUE
        smManeuvering_guards2 : tContrDoor < CONTR_INTERVAL
        isin_PendingClose : PendingClose = TRUE
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        smManeuvering_actions2 : tContrDoor := tContrDoor + 1
    end
END
```

D.2.8 Fifth Refinement mch6

```
MACHINE  mch6
REFINES  mch5
SEES  ctx2
VARIABLES
        Block
        Control
        Read
        FailureMode
        PendingClose
        PendingOpen
        PendingContrDoor
        retractEV
        PendingContrGear
        extendEV
        GearsManeuvering
        DoorsOpening
        openEV
        closeEV
        Maneuvering
        PendingStart
        PendingStop
```

generalEV

Extended

Retracted

NormalMode

HandleDown

HandleUp

handle

tStart

tStop

tContrDoor

doorsOpen

doorsClosed

tContrGear

gearsExtended

gearsRetracted

INVARIANTS

`typeof_retractEV` : $retractEV \in BOOL$

`typeof_PendingContrGear` : $PendingContrGear \in BOOL$

`typeof_extendEV` : $extendEV \in BOOL$

`distinct_states_in_smGearManeuvering` : $(GearsManeuvering = TRUE) \Rightarrow partition(\{TRUE\}, \{extendEV\} \cap \{TRUE\}, \{PendingContrGear\} \cap \{TRUE\}, \{retractEV\} \cap \{TRUE\})$

`retractEV_substateof_GearsManeuvering` : $(retractEV = TRUE) \Rightarrow (GearsManeuvering = TRUE)$

`PendingContrGear_substateof_GearsManeuvering` : $(PendingContrGear = TRUE) \Rightarrow (GearsManeuvering = TRUE)$

`extendEV_substateof_GearsManeuvering` : $(extendEV = TRUE) \Rightarrow (GearsManeuvering = TRUE)$

`inv1` : $tContrGear \in \mathbb{N}$

`inv2` : $gearsExtended \in BOOL$

`inv3` : $gearsRetracted \in BOOL$

EVENTS

Initialisation

extended

begin

`init_HandleUp` : $HandleUp := TRUE$
`init_HandleDown` : $HandleDown := FALSE$
`init_NormalMode` : $NormalMode := TRUE$
`init_FailureMode` : $FailureMode := FALSE$
`init_Read` : $Read := TRUE$
`init_Block` : $Block := FALSE$
`init_Control` : $Control := FALSE$
`act1` : $handle := FALSE$
`init_Retracted` : $Retracted := TRUE$

```
        init_generalEV : $generalEV := FALSE$
        init_Extended : $Extended := FALSE$
        init_Maneuvering : $Maneuvering := FALSE$
        init_PendingStart : $PendingStart := FALSE$
        init_PendingStop : $PendingStop := FALSE$
        act2 : $tStart := 0$
        act3 : $tStop := 0$
        init_PendingContrDoor : $PendingContrDoor := FALSE$
        init_openEV : $openEV := FALSE$
        init_closeEV : $closeEV := FALSE$
        act4 : $tContrDoor := 0$
        init_PendingClose : $PendingClose := FALSE$
        init_PendingOpen : $PendingOpen := FALSE$
        init_GearsManeuvering : $GearsManeuvering := FALSE$
        init_DoorsOpening : $DoorsOpening := FALSE$
        act5 : $doorsOpen := FALSE$
        act6 : $doorsClosed := TRUE$
        init_retractEV : $retractEV := FALSE$
        init_PendingContrGear : $PendingContrGear := FALSE$
        init_extendEV : $extendEV := FALSE$
        act7 : $tContrGear := 0$
        act8 : $gearsExtended := FALSE$
        act9 : $gearsRetracted := TRUE$
    end
Event readInput $\widehat{=}$
extends readInput
    any
        $h$  handle position (TRUE = down, FALSE = up)
        $dc$  doors closed
        $do$  doors open
        $gr$  gears retracted
        $ge$  gears extended
    where
        isin_Read : $Read = TRUE$
        h_type : $h \in BOOL$
        dc_type : $dc \in BOOL$
        do_type : $do \in BOOL$
        gr_type : $gr \in BOOL$
        ge_type : $ge \in BOOL$
    then
        leave_Read : $Read := FALSE$
        enter_Control : $Control := TRUE$
        smFMU_actions1 : $handle := h$
        smFMU_actions3 : $doorsClosed := dc$
        smFMU_actions2 : $doorsOpen := do$
        smFMU_actions5 : $gearsRetracted := gr$
```

```
            smFMU_actions4 : $gearsExtended := ge$
      end
Event   $extend \;\widehat{=}$
extends $extend$
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards2 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_Retracted : $Retracted = TRUE$
            smNormalMode_guards8 : $doorsClosed = TRUE$
      then
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_Retracted : $Retracted := FALSE$
            enter_generalEV : $generalEV := TRUE$
            enter_PendingStart : $PendingStart := TRUE$
            smNormalMode_actions1 : $tStart := 1$
      end
Event   $retract \;\widehat{=}$
extends $retract$
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards4 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_Extended : $Extended = TRUE$
            smNormalMode_guards7 : $doorsClosed = TRUE$
      then
            leave_HandleDown : $HandleDown := FALSE$
            enter_HandleUp : $HandleUp := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_Extended : $Extended := FALSE$
            enter_generalEV : $generalEV := TRUE$
            enter_PendingStart : $PendingStart := TRUE$
            smNormalMode_actions2 : $tStart := 1$
      end
Event   $keepRet \;\widehat{=}$
extends $keepRet$
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
```

```
        isin_Control : Control = TRUE
        isin_Retracted : Retracted = TRUE
        smNormalMode_guards4 : doorsClosed = TRUE
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
    end
Event   keepExt ≙
extends keepExt
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards3 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_Extended : Extended = TRUE
        smNormalMode_guards5 : doorsClosed = TRUE
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
    end
Event   setRet ≙
extends setRet
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards1 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_PendingStop : PendingStop = TRUE
        smNormalMode_guards1 : tStop ≥ STOP_INTERVAL
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_generalEV : generalEV := FALSE
        enter_Retracted : Retracted := TRUE
        leave_PendingStop : PendingStop := FALSE
    end
Event   cancelExt ≙
extends cancelExt
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards4 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_PendingStart : PendingStart = TRUE
        smGeneralEV_guards6 : $doorsClosed = TRUE$
    then
        leave_HandleDown : $HandleDown := FALSE$
        enter_HandleUp : $HandleUp := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_generalEV : $generalEV := FALSE$
        enter_Retracted : $Retracted := TRUE$
        leave_PendingStart : $PendingStart := FALSE$
    end
Event  setExt $\widehat{=}$
extends  setExt
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards3 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_PendingStop : $PendingStop = TRUE$
        smNormalMode_guards2 : $tStop \geq STOP\_INTERVAL$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_generalEV : $generalEV := FALSE$
        enter_Extended : $Extended := TRUE$
        leave_PendingStop : $PendingStop := FALSE$
    end
Event  cancelRet $\widehat{=}$
extends  cancelRet
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards2 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_PendingStart : $PendingStart = TRUE$
        smGeneralEV_guards4 : $doorsClosed = TRUE$
    then
        leave_HandleUp : $HandleUp := FALSE$
        enter_HandleDown : $HandleDown := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_generalEV : $generalEV := FALSE$
        enter_Extended : $Extended := TRUE$
        leave_PendingStart : $PendingStart := FALSE$
    end
```

Event *startExt* $\widehat{=}$
extends *startExt*

  when

    `isin_HandleDown` : $HandleDown = TRUE$
    `smHandle_guards3` : $handle = TRUE$
    `isin_NormalMode` : $NormalMode = TRUE$
    `isin_Control` : $Control = TRUE$
    `isin_generalEV` : $generalEV = TRUE$
    `isin_PendingStart` : $PendingStart = TRUE$
    `_guards3` : $tStart \geq START\_INTERVAL$
    `smGeneralEV_guards9` : $gearsRetracted = TRUE \lor$
   $gearsExtended = TRUE$
  then

    `leave_Control` : $Control := FALSE$
    `enter_Read` : $Read := TRUE$
    `leave_PendingStart` : $PendingStart := FALSE$
    `enter_Maneuvering` : $Maneuvering := TRUE$
    `enter_openEV` : $openEV := TRUE$
    `enter_DoorsOpening` : $DoorsOpening := TRUE$
  end

Event *startRet* $\widehat{=}$
extends *startRet*

  when

    `isin_HandleUp` : $HandleUp = TRUE$
    `smHandle_guards1` : $handle = FALSE$
    `isin_NormalMode` : $NormalMode = TRUE$
    `isin_Control` : $Control = TRUE$
    `isin_generalEV` : $generalEV = TRUE$
    `isin_PendingStart` : $PendingStart = TRUE$
    `_guards3` : $tStart \geq START\_INTERVAL$
    `smGeneralEV_guards9` : $gearsRetracted = TRUE \lor$
   $gearsExtended = TRUE$
  then

    `leave_Control` : $Control := FALSE$
    `enter_Read` : $Read := TRUE$
    `leave_PendingStart` : $PendingStart := FALSE$
    `enter_Maneuvering` : $Maneuvering := TRUE$
    `enter_openEV` : $openEV := TRUE$
    `enter_DoorsOpening` : $DoorsOpening := TRUE$
  end

Event *endExt* $\widehat{=}$
extends *endExt*

  when

    `isin_HandleDown` : $HandleDown = TRUE$
    `smHandle_guards3` : $handle = TRUE$
    `isin_NormalMode` : $NormalMode = TRUE$

```
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            _guards2 : tStop < STOP_INTERVAL
            isin_closeEV : closeEV = TRUE
            smGeneralEV_guards2 : doorsClosed = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_Maneuvering : Maneuvering := FALSE
            enter_PendingStop : PendingStop := TRUE
            _actions3 : tStop := 1
            leave_closeEV : closeEV := FALSE
      end
Event  endRet ≙
extends  endRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            _guards2 : tStop < STOP_INTERVAL
            isin_closeEV : closeEV = TRUE
            smGeneralEV_guards2 : doorsClosed = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_Maneuvering : Maneuvering := FALSE
            enter_PendingStop : PendingStop := TRUE
            _actions3 : tStop := 1
            leave_closeEV : closeEV := FALSE
      end
Event  undoExt ≙
extends  undoExt
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStop : PendingStop = TRUE
      then
            leave_HandleDown : HandleDown := FALSE
            enter_HandleUp : HandleUp := TRUE
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingStop : $PendingStop := FALSE$
            enter_Maneuvering : $Maneuvering := TRUE$
            enter_PendingContrDoor : $PendingContrDoor := TRUE$
            smGeneralEV_actions1 : $tContrDoor := tStop$
            enter_PendingOpen : $PendingOpen := TRUE$
      end
Event   undoRet ≙
extends  undoRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards2 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStop : $PendingStop = TRUE$
      then
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingStop : $PendingStop := FALSE$
            enter_Maneuvering : $Maneuvering := TRUE$
            enter_PendingContrDoor : $PendingContrDoor := TRUE$
            smGeneralEV_actions1 : $tContrDoor := tStop$
            enter_PendingOpen : $PendingOpen := TRUE$
      end
Event   delayExt ≙
extends  delayExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            _guards1 : $tStart < START\_INTERVAL$
            smGeneralEV_guards3 : $doorsClosed = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            _actions1 : $tStart := tStart + 1$
      end
Event   delayRet ≙
extends  delayRet
```

```
    when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStart : PendingStart = TRUE
            _guards1 : tStart < START_INTERVAL
            smGeneralEV_guards3 : doorsClosed = TRUE
    then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            _actions1 : tStart := tStart + 1
    end
Event  delaySetExt ≙
extends  delaySetExt
    when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStop : PendingStop = TRUE
            smGeneralEV_guards1 : tStop < STOP_INTERVAL
            smGeneralEV_guards5 : doorsClosed = TRUE
    then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            _actions2 : tStop := tStop + 1
    end
Event  delaySetRet ≙
extends  delaySetRet
    when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStop : PendingStop = TRUE
            smGeneralEV_guards1 : tStop < STOP_INTERVAL
            smGeneralEV_guards5 : doorsClosed = TRUE
    then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            _actions2 : tStop := tStop + 1
    end
```

```
Event   resumeEndExt ≙
extends  resumeEndExt
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            _guards2 : tStop < STOP_INTERVAL
            isin_PendingContrDoor : PendingContrDoor = TRUE
            isin_PendingOpen : PendingOpen = TRUE
            smManeuvering_guards5 : doorsClosed = TRUE
      then
            leave_HandleUp : HandleUp := FALSE
            enter_HandleDown : HandleDown := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_Maneuvering : Maneuvering := FALSE
            enter_PendingStop : PendingStop := TRUE
            _actions3 : tStop := 1
            leave_PendingContrDoor : PendingContrDoor := FALSE
            leave_PendingOpen : PendingOpen := FALSE
      end
Event   resumeEndRet ≙
extends  resumeEndRet
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            _guards2 : tStop < STOP_INTERVAL
            isin_PendingContrDoor : PendingContrDoor = TRUE
            isin_PendingOpen : PendingOpen = TRUE
            smManeuvering_guards5 : doorsClosed = TRUE
      then
            leave_HandleDown : HandleDown := FALSE
            enter_HandleUp : HandleUp := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_Maneuvering : Maneuvering := FALSE
            enter_PendingStop : PendingStop := TRUE
            _actions3 : tStop := 1
            leave_PendingContrDoor : PendingContrDoor := FALSE
```

```
        leave_PendingOpen : $PendingOpen := FALSE$
    end
Event  $abortOpenExt \mathrel{\widehat{=}}$
extends  $abortOpenExt$
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards4 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        _guards2 : $tStop < STOP\_INTERVAL$
        isin_openEV : $openEV = TRUE$
        isin_DoorsOpening : $DoorsOpening = TRUE$
        smManeuvering_guards6 : $doorsClosed = TRUE$
    then
        leave_HandleDown : $HandleDown := FALSE$
        enter_HandleUp : $HandleUp := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_Maneuvering : $Maneuvering := FALSE$
        enter_PendingStop : $PendingStop := TRUE$
        _actions3 : $tStop := 1$
        leave_openEV : $openEV := FALSE$
        leave_DoorsOpening : $DoorsOpening := FALSE$
    end
Event  $abortOpenRet \mathrel{\widehat{=}}$
extends  $abortOpenRet$
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards2 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        _guards2 : $tStop < STOP\_INTERVAL$
        isin_openEV : $openEV = TRUE$
        isin_DoorsOpening : $DoorsOpening = TRUE$
        smManeuvering_guards6 : $doorsClosed = TRUE$
    then
        leave_HandleUp : $HandleUp := FALSE$
        enter_HandleDown : $HandleDown := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_Maneuvering : $Maneuvering := FALSE$
        enter_PendingStop : $PendingStop := TRUE$
```

```
            _actions3 : $tStop := 1$
            leave_openEV : $openEV := FALSE$
            leave_DoorsOpening : $DoorsOpening := FALSE$
      end
Event   closeExt ≘
extends closeExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            smManeuvering_guards3 : $tContrDoor \geq CONTR\_INTERVAL$
            isin_PendingClose : $PendingClose = TRUE$
            smManeuvering_guards13 : $gearsRetracted = TRUE \vee$
           $gearsExtended = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            enter_closeEV : $closeEV := TRUE$
            leave_PendingClose : $PendingClose := FALSE$
      end
Event   closeRet ≘
extends closeRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            smManeuvering_guards3 : $tContrDoor \geq CONTR\_INTERVAL$
            isin_PendingClose : $PendingClose = TRUE$
            smManeuvering_guards13 : $gearsRetracted = TRUE \vee$
           $gearsExtended = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            enter_closeEV : $closeEV := TRUE$
            leave_PendingClose : $PendingClose := FALSE$
      end
```

```
Event  cancelCloseExt ≙
extends  cancelCloseExt
     when
          isin_HandleDown : HandleDown = TRUE
          smHandle_guards4 : handle = FALSE
          isin_NormalMode : NormalMode = TRUE
          isin_Control : Control = TRUE
          isin_generalEV : generalEV = TRUE
          isin_Maneuvering : Maneuvering = TRUE
          isin_closeEV : closeEV = TRUE
     then
          leave_HandleDown : HandleDown := FALSE
          enter_HandleUp : HandleUp := TRUE
          leave_Control : Control := FALSE
          enter_Read : Read := TRUE
          leave_closeEV : closeEV := FALSE
          enter_PendingContrDoor : PendingContrDoor := TRUE
          smManeuvering_actions3 : tContrDoor := 1
          enter_PendingOpen : PendingOpen := TRUE
     end
Event  cancelCloseRet ≙
extends  cancelCloseRet
     when
          isin_HandleUp : HandleUp = TRUE
          smHandle_guards2 : handle = TRUE
          isin_NormalMode : NormalMode = TRUE
          isin_Control : Control = TRUE
          isin_generalEV : generalEV = TRUE
          isin_Maneuvering : Maneuvering = TRUE
          isin_closeEV : closeEV = TRUE
     then
          leave_HandleUp : HandleUp := FALSE
          enter_HandleDown : HandleDown := TRUE
          leave_Control : Control := FALSE
          enter_Read : Read := TRUE
          leave_closeEV : closeEV := FALSE
          enter_PendingContrDoor : PendingContrDoor := TRUE
          smManeuvering_actions3 : tContrDoor := 1
          enter_PendingOpen : PendingOpen := TRUE
     end
Event  recloseExt ≙
extends  recloseExt
     when
          isin_HandleUp : HandleUp = TRUE
          smHandle_guards2 : handle = TRUE
          isin_NormalMode : NormalMode = TRUE
```

isin_Control : $Control = TRUE$
isin_generalEV : $generalEV = TRUE$
isin_Maneuvering : $Maneuvering = TRUE$
isin_PendingContrDoor : $PendingContrDoor = TRUE$
isin_PendingOpen : $PendingOpen = TRUE$
smManeuvering_guards14 : $gearsRetracted = TRUE \lor$
$gearsExtended = TRUE$

then

leave_HandleUp : $HandleUp := FALSE$
enter_HandleDown : $HandleDown := TRUE$
leave_Control : $Control := FALSE$
enter_Read : $Read := TRUE$
leave_PendingContrDoor : $PendingContrDoor := FALSE$
enter_closeEV : $closeEV := TRUE$
leave_PendingOpen : $PendingOpen := FALSE$

end

Event *recloseRet* $\widehat{=}$
extends *recloseRet*

when

isin_HandleDown : $HandleDown = TRUE$
smHandle_guards4 : $handle = FALSE$
isin_NormalMode : $NormalMode = TRUE$
isin_Control : $Control = TRUE$
isin_generalEV : $generalEV = TRUE$
isin_Maneuvering : $Maneuvering = TRUE$
isin_PendingContrDoor : $PendingContrDoor = TRUE$
isin_PendingOpen : $PendingOpen = TRUE$
smManeuvering_guards14 : $gearsRetracted = TRUE \lor$
$gearsExtended = TRUE$

then

leave_HandleDown : $HandleDown := FALSE$
enter_HandleUp : $HandleUp := TRUE$
leave_Control : $Control := FALSE$
enter_Read : $Read := TRUE$
leave_PendingContrDoor : $PendingContrDoor := FALSE$
enter_closeEV : $closeEV := TRUE$
leave_PendingOpen : $PendingOpen := FALSE$

end

Event *endOpenExt* $\widehat{=}$
extends *endOpenExt*

when

isin_HandleDown : $HandleDown = TRUE$
smHandle_guards3 : $handle = TRUE$
isin_NormalMode : $NormalMode = TRUE$
isin_Control : $Control = TRUE$
isin_generalEV : $generalEV = TRUE$

```
            isin_Maneuvering : Maneuvering = TRUE
            isin_openEV : openEV = TRUE
            isin_GearsManeuvering : GearsManeuvering = TRUE
            _guards5 : doorsOpen = TRUE
            isin_extendEV : extendEV = TRUE
            smOpenEV_guards4 : gearsExtended = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_openEV : openEV := FALSE
            enter_PendingContrDoor : PendingContrDoor := TRUE
            smManeuvering_actions1 : tContrDoor := 1
            leave_GearsManeuvering : GearsManeuvering := FALSE
            enter_PendingClose : PendingClose := TRUE
            leave_extendEV : extendEV := FALSE
      end
Event   endOpenRet ≙
extends endOpenRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_openEV : openEV = TRUE
            isin_GearsManeuvering : GearsManeuvering = TRUE
            _guards5 : doorsOpen = TRUE
            isin_retractEV : retractEV = TRUE
            _guards10 : gearsRetracted = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_openEV : openEV := FALSE
            enter_PendingContrDoor : PendingContrDoor := TRUE
            smManeuvering_actions1 : tContrDoor := 1
            leave_GearsManeuvering : GearsManeuvering := FALSE
            enter_PendingClose : PendingClose := TRUE
            leave_retractEV : retractEV := FALSE
      end
Event   cancelOpenExt ≙
extends cancelOpenExt
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
```

```
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_DoorsOpening : $DoorsOpening = TRUE$
        smManeuvering_guards7 : $doorsClosed = FALSE$
    then
        leave_HandleDown : $HandleDown := FALSE$
        enter_HandleUp : $HandleUp := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_openEV : $openEV := FALSE$
        enter_PendingContrDoor : $PendingContrDoor := TRUE$
        smManeuvering_actions1 : $tContrDoor := 1$
        leave_DoorsOpening : $DoorsOpening := FALSE$
        enter_PendingClose : $PendingClose := TRUE$
    end
Event cancelOpenRet ≙
extends cancelOpenRet
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards2 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_DoorsOpening : $DoorsOpening = TRUE$
        smManeuvering_guards7 : $doorsClosed = FALSE$
    then
        leave_HandleUp : $HandleUp := FALSE$
        enter_HandleDown : $HandleDown := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_openEV : $openEV := FALSE$
        enter_PendingContrDoor : $PendingContrDoor := TRUE$
        smManeuvering_actions1 : $tContrDoor := 1$
        leave_DoorsOpening : $DoorsOpening := FALSE$
        enter_PendingClose : $PendingClose := TRUE$
    end
Event reopenExt ≙
extends reopenExt
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards2 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_PendingContrDoor : PendingContrDoor = TRUE
        isin_PendingClose : PendingClose = TRUE
        smManeuvering_guards9 : gearsRetracted = TRUE ∨
       gearsExtended = TRUE
    then
        leave_HandleUp : HandleUp := FALSE
        enter_HandleDown : HandleDown := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_PendingContrDoor : PendingContrDoor := FALSE
        enter_openEV : openEV := TRUE
        leave_PendingClose : PendingClose := FALSE
        enter_DoorsOpening : DoorsOpening := TRUE
    end
Event  reopenRet ≙
extends reopenRet
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards4 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_PendingContrDoor : PendingContrDoor = TRUE
        isin_PendingClose : PendingClose = TRUE
        smManeuvering_guards9 : gearsRetracted = TRUE ∨
       gearsExtended = TRUE
    then
        leave_HandleDown : HandleDown := FALSE
        enter_HandleUp : HandleUp := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_PendingContrDoor : PendingContrDoor := FALSE
        enter_openEV : openEV := TRUE
        leave_PendingClose : PendingClose := FALSE
        enter_DoorsOpening : DoorsOpening := TRUE
    end
Event  openExt ≙
extends openExt
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards3 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
```

```
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            smManeuvering_guards1 : $tContrDoor \geq CONTR\_INTERVAL$
            isin_PendingOpen : $PendingOpen = TRUE$
            smManeuvering_guards10 : $gearsRetracted = TRUE \lor$
           $gearsExtended = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            enter_openEV : $openEV := TRUE$
            leave_PendingOpen : $PendingOpen := FALSE$
            enter_DoorsOpening : $DoorsOpening := TRUE$
      end
Event  openRet ≙
extends  openRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            smManeuvering_guards1 : $tContrDoor \geq CONTR\_INTERVAL$
            isin_PendingOpen : $PendingOpen = TRUE$
            smManeuvering_guards10 : $gearsRetracted = TRUE \lor$
           $gearsExtended = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            enter_openEV : $openEV := TRUE$
            leave_PendingOpen : $PendingOpen := FALSE$
            enter_DoorsOpening : $DoorsOpening := TRUE$
      end
Event  waitClosedExt ≙
extends  waitClosedExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_closeEV : $closeEV = TRUE$
            smManeuvering_guards4 : $doorsClosed = FALSE$
            smManeuvering_guards12 : $gearsRetracted = TRUE \vee$
           $gearsExtended = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event   waitClosedRet $\widehat{=}$
extends  waitClosedRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_closeEV : $closeEV = TRUE$
            smManeuvering_guards4 : $doorsClosed = FALSE$
            smManeuvering_guards12 : $gearsRetracted = TRUE \vee$
           $gearsExtended = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event   waitOpenExt $\widehat{=}$
extends  waitOpenExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_openEV : $openEV = TRUE$
            isin_DoorsOpening : $DoorsOpening = TRUE$
            _guards2 : $doorsOpen = FALSE$
            smOpenEV_guards2 : $gearsRetracted = TRUE \vee$
           $gearsExtended = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
      end
Event   waitOpenRet $\widehat{=}$
extends  waitOpenRet
```

```
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards1 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_DoorsOpening : $DoorsOpening = TRUE$
        _guards2 : $doorsOpen = FALSE$
        smOpenEV_guards2 : $gearsRetracted = TRUE \lor$
       $gearsExtended = TRUE$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
    end
Event  startGearExt $\widehat{=}$
extends  startGearExt
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards3 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_DoorsOpening : $DoorsOpening = TRUE$
        _guards4 : $doorsOpen = TRUE$
        smOpenEV_guards3 : $gearsRetracted = TRUE \lor$
       $gearsExtended = TRUE$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_DoorsOpening : $DoorsOpening := FALSE$
        enter_GearsManeuvering : $GearsManeuvering := TRUE$
        enter_extendEV : $extendEV := TRUE$
    end
Event  startGearRet $\widehat{=}$
extends  startGearRet
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards1 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
```

```
            isin_openEV : $openEV = TRUE$
            isin_DoorsOpening : $DoorsOpening = TRUE$
            _guards4 : $doorsOpen = TRUE$
            smOpenEV_guards5 : $gearsRetracted = TRUE \lor$
           $gearsExtended = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_DoorsOpening : $DoorsOpening := FALSE$
            enter_GearsManeuvering : $GearsManeuvering := TRUE$
            enter_retractEV : $retractEV := TRUE$
      end
Event   fDoorsNotOpen $\widehat{=}$
extends  fDoorsNotOpen
      when
            isin_Control : $Control = TRUE$
            isin_GearsManeuvering : $GearsManeuvering = TRUE$
            smOpenEV_guards1 : $doorsOpen = FALSE$
      then
            leave_Control : $Control := FALSE$
            enter_Block : $Block := TRUE$
            leave_NormalMode : $NormalMode := FALSE$
            leave_generalEV : $generalEV := FALSE$
            leave_Maneuvering : $Maneuvering := FALSE$
            leave_openEV : $openEV := FALSE$
            leave_GearsManeuvering : $GearsManeuvering := FALSE$
            enter_FailureMode : $FailureMode := TRUE$
            leave_retractEV : $retractEV := FALSE$
            leave_PendingContrGear : $PendingContrGear := FALSE$
            leave_extendEV : $extendEV := FALSE$
      end
Event   fDoorsUnlockedRet $\widehat{=}$
extends  fDoorsUnlockedRet
      when
            isin_Control : $Control = TRUE$
            isin_Retracted : $Retracted = TRUE$
            smNormalMode_guards3 : $doorsClosed = FALSE$
      then
            leave_Control : $Control := FALSE$
            enter_Block : $Block := TRUE$
            leave_NormalMode : $NormalMode := FALSE$
            leave_Retracted : $Retracted := FALSE$
            enter_FailureMode : $FailureMode := TRUE$
      end
Event   fDoorsUnlockedExt $\widehat{=}$
extends  fDoorsUnlockedExt
```

```
    when
          isin_Control : Control = TRUE
          isin_Extended : Extended = TRUE
          smNormalMode_guards6 : doorsClosed = FALSE
    then
          leave_Control : Control := FALSE
          enter_Block : Block := TRUE
          leave_NormalMode : NormalMode := FALSE
          leave_Extended : Extended := FALSE
          enter_FailureMode : FailureMode := TRUE
    end
Event   fDoorsUnlockedStart ≙
extends fDoorsUnlockedStart
    when
          isin_Control : Control = TRUE
          isin_PendingStart : PendingStart = TRUE
          smGeneralEV_guards7 : doorsClosed = FALSE
    then
          leave_Control : Control := FALSE
          enter_Block : Block := TRUE
          leave_NormalMode : NormalMode := FALSE
          leave_generalEV : generalEV := FALSE
          leave_PendingStart : PendingStart := FALSE
          enter_FailureMode : FailureMode := TRUE
    end
Event   fDoorsUnlockedStop ≙
extends fDoorsUnlockedStop
    when
          isin_Control : Control = TRUE
          isin_PendingStop : PendingStop = TRUE
          smGeneralEV_guards8 : doorsClosed = FALSE
    then
          leave_Control : Control := FALSE
          enter_Block : Block := TRUE
          leave_NormalMode : NormalMode := FALSE
          leave_generalEV : generalEV := FALSE
          leave_PendingStop : PendingStop := FALSE
          enter_FailureMode : FailureMode := TRUE
    end
Event   delayOpenExt ≙
extends delayOpenExt
    when
          isin_HandleDown : HandleDown = TRUE
          smHandle_guards3 : handle = TRUE
          isin_NormalMode : NormalMode = TRUE
          isin_Control : Control = TRUE
```

```
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            smManeuvering_guards2 : $tContrDoor < CONTR\_INTERVAL$
            isin_PendingOpen : $PendingOpen = TRUE$
     then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            smManeuvering_actions2 : $tContrDoor := tContrDoor + 1$
     end
Event   delayOpenRet ≙
extends delayOpenRet
     when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            smManeuvering_guards2 : $tContrDoor < CONTR\_INTERVAL$
            isin_PendingOpen : $PendingOpen = TRUE$
     then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            smManeuvering_actions2 : $tContrDoor := tContrDoor + 1$
     end
Event   delayCloseExt ≙
extends delayCloseExt
     when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            smManeuvering_guards2 : $tContrDoor < CONTR\_INTERVAL$
            isin_PendingClose : $PendingClose = TRUE$
     then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            smManeuvering_actions2 : $tContrDoor := tContrDoor + 1$
     end
Event   delayCloseRet ≙
extends delayCloseRet
```

```
    when
          isin_HandleUp : HandleUp = TRUE
          smHandle_guards1 : handle = FALSE
          isin_NormalMode : NormalMode = TRUE
          isin_Control : Control = TRUE
          isin_generalEV : generalEV = TRUE
          isin_Maneuvering : Maneuvering = TRUE
          isin_PendingContrDoor : PendingContrDoor = TRUE
          smManeuvering_guards2 : tContrDoor < CONTR_INTERVAL
          isin_PendingClose : PendingClose = TRUE
    then
          leave_Control : Control := FALSE
          enter_Read : Read := TRUE
          smManeuvering_actions2 : tContrDoor := tContrDoor + 1
    end
Event   waitExtended ≙
extends  stayDown
    when
          isin_HandleDown : HandleDown = TRUE
          smHandle_guards3 : handle = TRUE
          isin_NormalMode : NormalMode = TRUE
          isin_Control : Control = TRUE
          isin_generalEV : generalEV = TRUE
          isin_Maneuvering : Maneuvering = TRUE
          isin_openEV : openEV = TRUE
          isin_GearsManeuvering : GearsManeuvering = TRUE
          smControl_guards1 : doorsOpen = TRUE
          isin_extendEV : extendEV = TRUE
          _guards6 : gearsExtended = FALSE
    then
          leave_Control : Control := FALSE
          enter_Read : Read := TRUE
    end
Event   waitRetracted ≙
extends  stayUp
    when
          isin_HandleUp : HandleUp = TRUE
          smHandle_guards1 : handle = FALSE
          isin_NormalMode : NormalMode = TRUE
          isin_Control : Control = TRUE
          isin_generalEV : generalEV = TRUE
          isin_Maneuvering : Maneuvering = TRUE
          isin_openEV : openEV = TRUE
          isin_GearsManeuvering : GearsManeuvering = TRUE
          smControl_guards1 : doorsOpen = TRUE
          isin_retractEV : retractEV = TRUE
```

```
        _guards7 : $gearsRetracted = FALSE$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
    end
Event  $delayExtend \widehat{=}$
extends $stayDown$
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards3 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_GearsManeuvering : $GearsManeuvering = TRUE$
        smControl_guards1 : $doorsOpen = TRUE$
        isin_PendingContrGear : $PendingContrGear = TRUE$
        smControl_guards2 : $tContrGear < CONTR\_INTERVAL$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        smControl_actions1 : $tContrGear := tContrGear + 1$
    end
Event  $delayRetract \widehat{=}$
extends $stayDown$
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards3 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_GearsManeuvering : $GearsManeuvering = TRUE$
        smControl_guards1 : $doorsOpen = TRUE$
        isin_PendingContrGear : $PendingContrGear = TRUE$
        smControl_guards2 : $tContrGear < CONTR\_INTERVAL$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        smControl_actions1 : $tContrGear := tContrGear + 1$
    end
Event  $cancelExtend \widehat{=}$
extends $switchUp$
    when
```

```
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_openEV : openEV = TRUE
            isin_GearsManeuvering : GearsManeuvering = TRUE
            smControl_guards1 : doorsOpen = TRUE
            isin_extendEV : extendEV = TRUE
      then
            leave_HandleDown : HandleDown := FALSE
            enter_HandleUp : HandleUp := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_extendEV : extendEV := FALSE
            enter_PendingContrGear : PendingContrGear := TRUE
            _actions6 : tContrGear := 1
      end
Event  cancelRetract ≙
extends  switchDown
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_openEV : openEV = TRUE
            isin_GearsManeuvering : GearsManeuvering = TRUE
            smControl_guards1 : doorsOpen = TRUE
            isin_retractEV : retractEV = TRUE
      then
            leave_HandleUp : HandleUp := FALSE
            enter_HandleDown : HandleDown := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_retractEV : retractEV := FALSE
            enter_PendingContrGear : PendingContrGear := TRUE
            _actions5 : tContrGear := 1
      end
Event  doExtend ≙
extends  stayDown
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_openEV : $openEV = TRUE$
            isin_GearsManeuvering : $GearsManeuvering = TRUE$
            smControl_guards1 : $doorsOpen = TRUE$
            isin_PendingContrGear : $PendingContrGear = TRUE$
            _guards8 : $tContrGear \geq CONTR\_INTERVAL$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrGear : $PendingContrGear := FALSE$
            enter_extendEV : $extendEV := TRUE$
      end
Event   doRetract $\widehat{=}$
extends  stayUp
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_openEV : $openEV = TRUE$
            isin_GearsManeuvering : $GearsManeuvering = TRUE$
            smControl_guards1 : $doorsOpen = TRUE$
            isin_PendingContrGear : $PendingContrGear = TRUE$
            _guards9 : $tContrGear \geq CONTR\_INTERVAL$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrGear : $PendingContrGear := FALSE$
            enter_retractEV : $retractEV := TRUE$
      end
Event   resumeExt $\widehat{=}$
extends  switchDown
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards2 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_openEV : $openEV = TRUE$
            isin_GearsManeuvering : $GearsManeuvering = TRUE$
            smControl_guards1 : $doorsOpen = TRUE$
            isin_PendingContrGear : PendingContrGear = TRUE
      then
            leave_HandleUp : HandleUp := FALSE
            enter_HandleDown : HandleDown := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingContrGear : PendingContrGear := FALSE
            enter_extendEV : extendEV := TRUE
      end
Event  resumeRet ≙
extends  switchUp
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_openEV : openEV = TRUE
            isin_GearsManeuvering : GearsManeuvering = TRUE
            smControl_guards1 : doorsOpen = TRUE
            isin_PendingContrGear : PendingContrGear = TRUE
      then
            leave_HandleDown : HandleDown := FALSE
            enter_HandleUp : HandleUp := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingContrGear : PendingContrGear := FALSE
            enter_retractEV : retractEV := TRUE
      end
Event  fDoorsNotOpenGearsUnlocked ≙
refines  fail
      when
            isin_Control : Control = TRUE
            isin_openEV : openEV = TRUE
            smManeuvering_guards8 : doorsOpen = FALSE ∧
           gearsRetracted = FALSE ∧ gearsExtended = FALSE
      then
            leave_Control : Control := FALSE
            enter_Block : Block := TRUE
            leave_NormalMode : NormalMode := FALSE
            leave_generalEV : generalEV := FALSE
            leave_Maneuvering : Maneuvering := FALSE
            leave_retractEV : retractEV := FALSE
            leave_PendingContrGear : PendingContrGear := FALSE
            leave_extendEV : extendEV := FALSE
            leave_GearsManeuvering : $GearsManeuvering := FALSE$
            leave_DoorsOpening : $DoorsOpening := FALSE$
            leave_openEV : $openEV := FALSE$
            enter_FailureMode : $FailureMode := TRUE$
      end
Event  $fGearsUnlocked \;\widehat{=}$
refines  $fail$
      when
            isin_Control : $Control = TRUE$
            isin_closeEV : $closeEV = TRUE$
            smManeuvering_guards11 : $gearsRetracted = FALSE \wedge$
           $gearsExtended = FALSE$
      then
            leave_Control : $Control := FALSE$
            enter_Block : $Block := TRUE$
            leave_NormalMode : $NormalMode := FALSE$
            leave_generalEV : $generalEV := FALSE$
            leave_Maneuvering : $Maneuvering := FALSE$
            leave_closeEV : $closeEV := FALSE$
            enter_FailureMode : $FailureMode := TRUE$
      end
END
```

D.2.9 Sixth Refinement mch7

```
MACHINE  mch7
REFINES  mch6
SEES  ctx2
VARIABLES
        $Read$
        $Read\_sm1\_S1$
        $Read\_sm1\_S2$
        $Control$
        $Block$
        $FailureMode$
        $PendingClose$
        $PendingOpen$
        $PendingContrDoor$
        $retractEV$
        $PendingContrGear$
        $extendEV$
        $GearsManeuvering$
        $DoorsOpening$
        $openEV$
```

$closeEV$

$Maneuvering$

$PendingStart$

$PendingStop$

$generalEV$

$Extended$

$Retracted$

$NormalMode$

$HandleDown$

$HandleUp$

$handle$

$tStart$

$tStop$

$tContrDoor$

$doorsOpen$

$doorsClosed$

$tContrGear$

$gearsExtended$

$gearsRetracted$

INVARIANTS

`typeof_Read_sm1_S1` : $Read\_sm1\_S1 \in BOOL$

`typeof_Read_sm1_S2` : $Read\_sm1\_S2 \in BOOL$

`distinct_states_in_Read_sm1` : $(Read = TRUE) \Rightarrow$
$partition(\{TRUE\}, \{Read\_sm1\_S1\} \cap \{TRUE\}, \{Read\_sm1\_S2\} \cap \{TRUE\})$

`Read_sm1_S1_substateof_Read` : $(Read\_sm1\_S1 = TRUE) \Rightarrow (Read = TRUE)$

`Read_sm1_S2_substateof_Read` : $(Read\_sm1\_S2 = TRUE) \Rightarrow (Read = TRUE)$

`R21` : $Read = TRUE \wedge handle = TRUE \Rightarrow retractEV = FALSE$
if the landing gear command handle remains in the DOWN position,
then retraction sequence is not observed

`R22` : $Read = TRUE \wedge handle = FALSE \Rightarrow extendEV = FALSE$
if the landing gear command handle remains in the UP position,
then outgoing sequence is not observed

`R311` : $Read = TRUE \wedge extendEV = TRUE \Rightarrow doorsOpen = TRUE$
the stimulation of the gears outgoing electro-valves can only happen
when the three doors are locked open

`R312` : $Read = TRUE \wedge retractEV = TRUE \Rightarrow doorsOpen = TRUE$
the stimulation of the retraction electro-valves can only happen when
the three doors are locked open

`R321` : $Read = TRUE \wedge closeEV = TRUE \Rightarrow$
$gearsRetracted = TRUE \vee gearsExtended = TRUE$
the stimulation of the doors closure electro-valves can only happen
when the three gears are locked down or up

`R322` : $Read = TRUE \wedge openEV = TRUE \wedge doorsOpen \neq TRUE \Rightarrow$
$gearsRetracted = TRUE \vee gearsExtended = TRUE$
the stimulation of the doors opening electro-valves can only happen
when the three gears are locked down or up

R41 : $\neg\,(openEV = TRUE \wedge closeEV = TRUE)$
opening and closure doors electro-valves are not
stimulated simultaneously
R42 : $\neg\,(retractEV = TRUE \wedge extendEV = TRUE)$
outgoing and retraction gears electro-valves are not
stimulated simultaneously
R511 : $openEV = TRUE \Rightarrow generalEV = TRUE$
it is not possible to stimulate the opening electro-valve without
stimulating the general electro-valve
R512 : $closeEV = TRUE \Rightarrow generalEV = TRUE$
it is not possible to stimulate the closure electro-valve without
stimulating the general electro-valve
R521 : $extendEV = TRUE \Rightarrow generalEV = TRUE$
it is not possible to stimulate the outgoing electro-valve without
stimulating the general electro-valve
R522 : $retractEV = TRUE \Rightarrow generalEV = TRUE$
it is not possible to stimulate the retraction electro-valve without
stimulating the general electro-valve

EVENTS

Initialisation

extended

begin

init_HandleUp : $HandleUp := TRUE$
init_HandleDown : $HandleDown := FALSE$
init_NormalMode : $NormalMode := TRUE$
init_FailureMode : $FailureMode := FALSE$
init_Read : $Read := TRUE$
init_Block : $Block := FALSE$
init_Control : $Control := FALSE$
act1 : $handle := FALSE$
init_Retracted : $Retracted := TRUE$
init_generalEV : $generalEV := FALSE$
init_Extended : $Extended := FALSE$
init_Maneuvering : $Maneuvering := FALSE$
init_PendingStart : $PendingStart := FALSE$
init_PendingStop : $PendingStop := FALSE$
act2 : $tStart := 0$
act3 : $tStop := 0$
init_PendingContrDoor : $PendingContrDoor := FALSE$
init_openEV : $openEV := FALSE$
init_closeEV : $closeEV := FALSE$
act4 : $tContrDoor := 0$
init_PendingClose : $PendingClose := FALSE$
init_PendingOpen : $PendingOpen := FALSE$
init_GearsManeuvering : $GearsManeuvering := FALSE$
init_DoorsOpening : $DoorsOpening := FALSE$
act5 : $doorsOpen := FALSE$

```
            act6 : doorsClosed := TRUE
            init_retractEV : retractEV := FALSE
            init_PendingContrGear : PendingContrGear := FALSE
            init_extendEV : extendEV := FALSE
            act7 : tContrGear := 0
            act8 : gearsExtended := FALSE
            act9 : gearsRetracted := TRUE
            init_Read_sm1_S1 : Read_sm1_S1 := TRUE
            init_Read_sm1_S2 : Read_sm1_S2 := FALSE
      end
Event  readInput ≙
extends  readInput
      any
           h   handle position (TRUE = down, FALSE = up)
           dc   doors closed
           do   doors open
           gr   gears retracted
           ge   gears extended
      where
            isin_Read : Read = TRUE
            h_type : h ∈ BOOL
            dc_type : dc ∈ BOOL
            do_type : do ∈ BOOL
            gr_type : gr ∈ BOOL
            ge_type : ge ∈ BOOL
            isin_Read_sm1_S1 : Read_sm1_S1 = TRUE
      then
            leave_Read : Read := FALSE
            enter_Control : Control := TRUE
            smFMU_actions1 : handle := h
            smFMU_actions3 : doorsClosed := dc
            smFMU_actions2 : doorsOpen := do
            smFMU_actions5 : gearsRetracted := gr
            smFMU_actions4 : gearsExtended := ge
            leave_Read_sm1_S1 : Read_sm1_S1 := FALSE
      end
Event  extend ≙
extends  extend
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_Retracted : Retracted = TRUE
            smNormalMode_guards8 : doorsClosed = TRUE
      then
```

```
            leave_HandleUp : HandleUp := FALSE
            enter_HandleDown : HandleDown := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_Retracted : Retracted := FALSE
            enter_generalEV : generalEV := TRUE
            enter_PendingStart : PendingStart := TRUE
            smNormalMode_actions1 : tStart := 1
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
      end
Event   retract ≙
extends retract
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_Extended : Extended = TRUE
            smNormalMode_guards7 : doorsClosed = TRUE
      then
            leave_HandleDown : HandleDown := FALSE
            enter_HandleUp : HandleUp := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_Extended : Extended := FALSE
            enter_generalEV : generalEV := TRUE
            enter_PendingStart : PendingStart := TRUE
            smNormalMode_actions2 : tStart := 1
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
      end
Event   keepRet ≙
extends keepRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_Retracted : Retracted = TRUE
            smNormalMode_guards4 : doorsClosed = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
      end
Event   keepExt ≙
extends keepExt
```

```
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_Extended : $Extended = TRUE$
            smNormalMode_guards5 : $doorsClosed = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event   setRet $\widehat{=}$
extends setRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStop : $PendingStop = TRUE$
            smNormalMode_guards1 : $tStop \geq STOP\_INTERVAL$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_generalEV : $generalEV := FALSE$
            enter_Retracted : $Retracted := TRUE$
            leave_PendingStop : $PendingStop := FALSE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event   cancelExt $\widehat{=}$
extends cancelExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards4 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            smGeneralEV_guards6 : $doorsClosed = TRUE$
      then
            leave_HandleDown : $HandleDown := FALSE$
            enter_HandleUp : $HandleUp := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_generalEV : $generalEV := FALSE$
            enter_Retracted : $Retracted := TRUE$
            leave_PendingStart : $PendingStart := FALSE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event  setExt ≙
extends setExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStop : $PendingStop = TRUE$
            smNormalMode_guards2 : $tStop \geq STOP\_INTERVAL$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_generalEV : $generalEV := FALSE$
            enter_Extended : $Extended := TRUE$
            leave_PendingStop : $PendingStop := FALSE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event  cancelRet ≙
extends cancelRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards2 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            smGeneralEV_guards4 : $doorsClosed = TRUE$
      then
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_generalEV : $generalEV := FALSE$
            enter_Extended : $Extended := TRUE$
            leave_PendingStart : $PendingStart := FALSE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event  startExt ≙
extends startExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            _guards3 : $tStart \geq START\_INTERVAL$
            smGeneralEV_guards9 : $gearsRetracted = TRUE \lor$
                   $gearsExtended = TRUE$
     then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingStart : $PendingStart := FALSE$
            enter_Maneuvering : $Maneuvering := TRUE$
            enter_openEV : $openEV := TRUE$
            enter_DoorsOpening : $DoorsOpening := TRUE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
     end
Event  startRet $\widehat{=}$
extends  startRet
     when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStart : $PendingStart = TRUE$
            _guards3 : $tStart \geq START\_INTERVAL$
            smGeneralEV_guards9 : $gearsRetracted = TRUE \lor$
                   $gearsExtended = TRUE$
     then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingStart : $PendingStart := FALSE$
            enter_Maneuvering : $Maneuvering := TRUE$
            enter_openEV : $openEV := TRUE$
            enter_DoorsOpening : $DoorsOpening := TRUE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
     end
Event  endExt $\widehat{=}$
extends  endExt
     when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            _guards2 : $tStop < STOP\_INTERVAL$
            isin_closeEV : $closeEV = TRUE$
            smGeneralEV_guards2 : $doorsClosed = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_Maneuvering : $Maneuvering := FALSE$
            enter_PendingStop : $PendingStop := TRUE$
            _actions3 : $tStop := 1$
            leave_closeEV : $closeEV := FALSE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event   endRet ≙
extends endRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            _guards2 : $tStop < STOP\_INTERVAL$
            isin_closeEV : $closeEV = TRUE$
            smGeneralEV_guards2 : $doorsClosed = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_Maneuvering : $Maneuvering := FALSE$
            enter_PendingStop : $PendingStop := TRUE$
            _actions3 : $tStop := 1$
            leave_closeEV : $closeEV := FALSE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event   undoExt ≙
extends undoExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards4 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_PendingStop : $PendingStop = TRUE$
      then
            leave_HandleDown : $HandleDown := FALSE$
            enter_HandleUp : $HandleUp := TRUE$
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingStop : PendingStop := FALSE
            enter_Maneuvering : Maneuvering := TRUE
            enter_PendingContrDoor : PendingContrDoor := TRUE
            smGeneralEV_actions1 : tContrDoor := tStop
            enter_PendingOpen : PendingOpen := TRUE
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
      end
Event   undoRet ≙
extends  undoRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStop : PendingStop = TRUE
      then
            leave_HandleUp : HandleUp := FALSE
            enter_HandleDown : HandleDown := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingStop : PendingStop := FALSE
            enter_Maneuvering : Maneuvering := TRUE
            enter_PendingContrDoor : PendingContrDoor := TRUE
            smGeneralEV_actions1 : tContrDoor := tStop
            enter_PendingOpen : PendingOpen := TRUE
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
      end
Event   delayExt ≙
extends  delayExt
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_PendingStart : PendingStart = TRUE
            _guards1 : tStart < START_INTERVAL
            smGeneralEV_guards3 : doorsClosed = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            _actions1 : tStart := tStart + 1
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
```

end

Event *delayRet* $\widehat{=}$
extends *delayRet*

when

- `isin_HandleUp` : $HandleUp = TRUE$
- `smHandle_guards1` : $handle = FALSE$
- `isin_NormalMode` : $NormalMode = TRUE$
- `isin_Control` : $Control = TRUE$
- `isin_generalEV` : $generalEV = TRUE$
- `isin_PendingStart` : $PendingStart = TRUE$
- `_guards1` : $tStart < START\_INTERVAL$
- `smGeneralEV_guards3` : $doorsClosed = TRUE$

then

- `leave_Control` : $Control := FALSE$
- `enter_Read` : $Read := TRUE$
- `_actions1` : $tStart := tStart + 1$
- `enter_Read_sm1_S2` : $Read\_sm1\_S2 := TRUE$

end

Event *delaySetExt* $\widehat{=}$
extends *delaySetExt*

when

- `isin_HandleDown` : $HandleDown = TRUE$
- `smHandle_guards3` : $handle = TRUE$
- `isin_NormalMode` : $NormalMode = TRUE$
- `isin_Control` : $Control = TRUE$
- `isin_generalEV` : $generalEV = TRUE$
- `isin_PendingStop` : $PendingStop = TRUE$
- `smGeneralEV_guards1` : $tStop < STOP\_INTERVAL$
- `smGeneralEV_guards5` : $doorsClosed = TRUE$

then

- `leave_Control` : $Control := FALSE$
- `enter_Read` : $Read := TRUE$
- `_actions2` : $tStop := tStop + 1$
- `enter_Read_sm1_S2` : $Read\_sm1\_S2 := TRUE$

end

Event *delaySetRet* $\widehat{=}$
extends *delaySetRet*

when

- `isin_HandleUp` : $HandleUp = TRUE$
- `smHandle_guards1` : $handle = FALSE$
- `isin_NormalMode` : $NormalMode = TRUE$
- `isin_Control` : $Control = TRUE$
- `isin_generalEV` : $generalEV = TRUE$
- `isin_PendingStop` : $PendingStop = TRUE$
- `smGeneralEV_guards1` : $tStop < STOP\_INTERVAL$
- `smGeneralEV_guards5` : $doorsClosed = TRUE$

then

```
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        _actions2 : $tStop := tStop + 1$
        enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
```

end

Event *resumeEndExt* $\widehat{=}$
extends *resumeEndExt*

when

```
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards2 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        _guards2 : $tStop < STOP\_INTERVAL$
        isin_PendingContrDoor : $PendingContrDoor = TRUE$
        isin_PendingOpen : $PendingOpen = TRUE$
        smManeuvering_guards5 : $doorsClosed = TRUE$
```

then

```
        leave_HandleUp : $HandleUp := FALSE$
        enter_HandleDown : $HandleDown := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_Maneuvering : $Maneuvering := FALSE$
        enter_PendingStop : $PendingStop := TRUE$
        _actions3 : $tStop := 1$
        leave_PendingContrDoor : $PendingContrDoor := FALSE$
        leave_PendingOpen : $PendingOpen := FALSE$
        enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
```

end

Event *resumeEndRet* $\widehat{=}$
extends *resumeEndRet*

when

```
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards4 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        _guards2 : $tStop < STOP\_INTERVAL$
        isin_PendingContrDoor : $PendingContrDoor = TRUE$
        isin_PendingOpen : $PendingOpen = TRUE$
        smManeuvering_guards5 : $doorsClosed = TRUE$
```

then

```
        leave_HandleDown : $HandleDown := FALSE$
```

```
            enter_HandleUp : HandleUp := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_Maneuvering : Maneuvering := FALSE
            enter_PendingStop : PendingStop := TRUE
            _actions3 : tStop := 1
            leave_PendingContrDoor : PendingContrDoor := FALSE
            leave_PendingOpen : PendingOpen := FALSE
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
      end
Event   abortOpenExt ≙
extends abortOpenExt
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            _guards2 : tStop < STOP_INTERVAL
            isin_openEV : openEV = TRUE
            isin_DoorsOpening : DoorsOpening = TRUE
            smManeuvering_guards6 : doorsClosed = TRUE
      then
            leave_HandleDown : HandleDown := FALSE
            enter_HandleUp : HandleUp := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_Maneuvering : Maneuvering := FALSE
            enter_PendingStop : PendingStop := TRUE
            _actions3 : tStop := 1
            leave_openEV : openEV := FALSE
            leave_DoorsOpening : DoorsOpening := FALSE
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
      end
Event   abortOpenRet ≙
extends abortOpenRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            _guards2 : tStop < STOP_INTERVAL
            isin_openEV : openEV = TRUE
```

```
        isin_DoorsOpening : $DoorsOpening = TRUE$
        smManeuvering_guards6 : $doorsClosed = TRUE$
    then
        leave_HandleUp : $HandleUp := FALSE$
        enter_HandleDown : $HandleDown := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_Maneuvering : $Maneuvering := FALSE$
        enter_PendingStop : $PendingStop := TRUE$
        _actions3 : $tStop := 1$
        leave_openEV : $openEV := FALSE$
        leave_DoorsOpening : $DoorsOpening := FALSE$
        enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
    end
Event  closeExt ≙
extends  closeExt
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards3 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_PendingContrDoor : $PendingContrDoor = TRUE$
        smManeuvering_guards3 : $tContrDoor \geq CONTR\_INTERVAL$
        isin_PendingClose : $PendingClose = TRUE$
        smManeuvering_guards13 : $gearsRetracted = TRUE \vee$
              $gearsExtended = TRUE$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_PendingContrDoor : $PendingContrDoor := FALSE$
        enter_closeEV : $closeEV := TRUE$
        leave_PendingClose : $PendingClose := FALSE$
        enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
    end
Event  closeRet ≙
extends  closeRet
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards1 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_PendingContrDoor : $PendingContrDoor = TRUE$
```

```
            smManeuvering_guards3 : tContrDoor ≥ CONTR_INTERVAL
            isin_PendingClose : PendingClose = TRUE
            smManeuvering_guards13 : gearsRetracted = TRUE ∨
                    gearsExtended = TRUE
      then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_PendingContrDoor : PendingContrDoor := FALSE
            enter_closeEV : closeEV := TRUE
            leave_PendingClose : PendingClose := FALSE
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
      end
Event   cancelCloseExt ≙
extends  cancelCloseExt
      when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards4 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_closeEV : closeEV = TRUE
      then
            leave_HandleDown : HandleDown := FALSE
            enter_HandleUp : HandleUp := TRUE
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            leave_closeEV : closeEV := FALSE
            enter_PendingContrDoor : PendingContrDoor := TRUE
            smManeuvering_actions3 : tContrDoor := 1
            enter_PendingOpen : PendingOpen := TRUE
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
      end
Event   cancelCloseRet ≙
extends  cancelCloseRet
      when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards2 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_closeEV : closeEV = TRUE
      then
            leave_HandleUp : HandleUp := FALSE
            enter_HandleDown : HandleDown := TRUE
            leave_Control : Control := FALSE
```

```
        enter_Read : $Read := TRUE$
        leave_closeEV : $closeEV := FALSE$
        enter_PendingContrDoor : $PendingContrDoor := TRUE$
        smManeuvering_actions3 : $tContrDoor := 1$
        enter_PendingOpen : $PendingOpen := TRUE$
        enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
    end
Event  recloseExt ≙
extends recloseExt
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards2 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_PendingContrDoor : $PendingContrDoor = TRUE$
        isin_PendingOpen : $PendingOpen = TRUE$
        smManeuvering_guards14 : $gearsRetracted = TRUE \lor$
                $gearsExtended = TRUE$
    then
        leave_HandleUp : $HandleUp := FALSE$
        enter_HandleDown : $HandleDown := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_PendingContrDoor : $PendingContrDoor := FALSE$
        enter_closeEV : $closeEV := TRUE$
        leave_PendingOpen : $PendingOpen := FALSE$
        enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
    end
Event  recloseRet ≙
extends recloseRet
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards4 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_PendingContrDoor : $PendingContrDoor = TRUE$
        isin_PendingOpen : $PendingOpen = TRUE$
        smManeuvering_guards14 : $gearsRetracted = TRUE \lor$
                $gearsExtended = TRUE$
    then
        leave_HandleDown : $HandleDown := FALSE$
        enter_HandleUp : $HandleUp := TRUE$
```

```
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            enter_closeEV : $closeEV := TRUE$
            leave_PendingOpen : $PendingOpen := FALSE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event  endOpenExt ≙
extends  endOpenExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_openEV : $openEV = TRUE$
            isin_GearsManeuvering : $GearsManeuvering = TRUE$
            _guards5 : $doorsOpen = TRUE$
            isin_extendEV : $extendEV = TRUE$
            smOpenEV_guards4 : $gearsExtended = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_openEV : $openEV := FALSE$
            enter_PendingContrDoor : $PendingContrDoor := TRUE$
            smManeuvering_actions1 : $tContrDoor := 1$
            leave_GearsManeuvering : $GearsManeuvering := FALSE$
            enter_PendingClose : $PendingClose := TRUE$
            leave_extendEV : $extendEV := FALSE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event  endOpenRet ≙
extends  endOpenRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_openEV : $openEV = TRUE$
            isin_GearsManeuvering : $GearsManeuvering = TRUE$
            _guards5 : $doorsOpen = TRUE$
            isin_retractEV : $retractEV = TRUE$
            _guards10 : $gearsRetracted = TRUE$
```

```
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_openEV : openEV := FALSE
        enter_PendingContrDoor : PendingContrDoor := TRUE
        smManeuvering_actions1 : tContrDoor := 1
        leave_GearsManeuvering : GearsManeuvering := FALSE
        enter_PendingClose : PendingClose := TRUE
        leave_retractEV : retractEV := FALSE
        enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
    end
Event  cancelOpenExt ≙
extends  cancelOpenExt
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards4 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_openEV : openEV = TRUE
        isin_DoorsOpening : DoorsOpening = TRUE
        smManeuvering_guards7 : doorsClosed = FALSE
    then
        leave_HandleDown : HandleDown := FALSE
        enter_HandleUp : HandleUp := TRUE
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        leave_openEV : openEV := FALSE
        enter_PendingContrDoor : PendingContrDoor := TRUE
        smManeuvering_actions1 : tContrDoor := 1
        leave_DoorsOpening : DoorsOpening := FALSE
        enter_PendingClose : PendingClose := TRUE
        enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
    end
Event  cancelOpenRet ≙
extends  cancelOpenRet
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards2 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_openEV : openEV = TRUE
        isin_DoorsOpening : DoorsOpening = TRUE
```

```
            smManeuvering_guards7 : $doorsClosed = FALSE$
      then
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_openEV : $openEV := FALSE$
            enter_PendingContrDoor : $PendingContrDoor := TRUE$
            smManeuvering_actions1 : $tContrDoor := 1$
            leave_DoorsOpening : $DoorsOpening := FALSE$
            enter_PendingClose : $PendingClose := TRUE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event   reopenExt ≙
extends reopenExt
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards2 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            isin_PendingClose : $PendingClose = TRUE$
            smManeuvering_guards9 : $gearsRetracted = TRUE \lor$
                  $gearsExtended = TRUE$
      then
            leave_HandleUp : $HandleUp := FALSE$
            enter_HandleDown : $HandleDown := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            enter_openEV : $openEV := TRUE$
            leave_PendingClose : $PendingClose := FALSE$
            enter_DoorsOpening : $DoorsOpening := TRUE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event   reopenRet ≙
extends reopenRet
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards4 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            isin_PendingClose : $PendingClose = TRUE$
            smManeuvering_guards9 : $gearsRetracted = TRUE \lor$
                  $gearsExtended = TRUE$
      then
            leave_HandleDown : $HandleDown := FALSE$
            enter_HandleUp : $HandleUp := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            enter_openEV : $openEV := TRUE$
            leave_PendingClose : $PendingClose := FALSE$
            enter_DoorsOpening : $DoorsOpening := TRUE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event   openExt $\widehat{=}$
extends  openExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            smManeuvering_guards1 : $tContrDoor \geq CONTR\_INTERVAL$
            isin_PendingOpen : $PendingOpen = TRUE$
            smManeuvering_guards10 : $gearsRetracted = TRUE \lor$
                  $gearsExtended = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrDoor : $PendingContrDoor := FALSE$
            enter_openEV : $openEV := TRUE$
            leave_PendingOpen : $PendingOpen := FALSE$
            enter_DoorsOpening : $DoorsOpening := TRUE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event   openRet $\widehat{=}$
extends  openRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
```

`isin_Maneuvering` : $Maneuvering = TRUE$
`isin_PendingContrDoor` : $PendingContrDoor = TRUE$
`smManeuvering_guards1` : $tContrDoor \geq CONTR\_INTERVAL$
`isin_PendingOpen` : $PendingOpen = TRUE$
`smManeuvering_guards10` : $gearsRetracted = TRUE \lor$
$gearsExtended = TRUE$

then

`leave_Control` : $Control := FALSE$
`enter_Read` : $Read := TRUE$
`leave_PendingContrDoor` : $PendingContrDoor := FALSE$
`enter_openEV` : $openEV := TRUE$
`leave_PendingOpen` : $PendingOpen := FALSE$
`enter_DoorsOpening` : $DoorsOpening := TRUE$
`enter_Read_sm1_S2` : $Read\_sm1\_S2 := TRUE$

end

Event *waitClosedExt* $\widehat{=}$
extends *waitClosedExt*

when

`isin_HandleDown` : $HandleDown = TRUE$
`smHandle_guards3` : $handle = TRUE$
`isin_NormalMode` : $NormalMode = TRUE$
`isin_Control` : $Control = TRUE$
`isin_generalEV` : $generalEV = TRUE$
`isin_Maneuvering` : $Maneuvering = TRUE$
`isin_closeEV` : $closeEV = TRUE$
`smManeuvering_guards4` : $doorsClosed = FALSE$
`smManeuvering_guards12` : $gearsRetracted = TRUE \lor$
$gearsExtended = TRUE$

then

`leave_Control` : $Control := FALSE$
`enter_Read` : $Read := TRUE$
`enter_Read_sm1_S2` : $Read\_sm1\_S2 := TRUE$

end

Event *waitClosedRet* $\widehat{=}$
extends *waitClosedRet*

when

`isin_HandleUp` : $HandleUp = TRUE$
`smHandle_guards1` : $handle = FALSE$
`isin_NormalMode` : $NormalMode = TRUE$
`isin_Control` : $Control = TRUE$
`isin_generalEV` : $generalEV = TRUE$
`isin_Maneuvering` : $Maneuvering = TRUE$
`isin_closeEV` : $closeEV = TRUE$
`smManeuvering_guards4` : $doorsClosed = FALSE$
`smManeuvering_guards12` : $gearsRetracted = TRUE \lor$
$gearsExtended = TRUE$

then

  `leave_Control` : $Control := FALSE$
  `enter_Read` : $Read := TRUE$
  `enter_Read_sm1_S2` : $Read\_sm1\_S2 := TRUE$

end

Event *waitOpenExt* $\widehat{=}$
extends *waitOpenExt*

when

  `isin_HandleDown` : $HandleDown = TRUE$
  `smHandle_guards3` : $handle = TRUE$
  `isin_NormalMode` : $NormalMode = TRUE$
  `isin_Control` : $Control = TRUE$
  `isin_generalEV` : $generalEV = TRUE$
  `isin_Maneuvering` : $Maneuvering = TRUE$
  `isin_openEV` : $openEV = TRUE$
  `isin_DoorsOpening` : $DoorsOpening = TRUE$
  `_guards2` : $doorsOpen = FALSE$
  `smOpenEV_guards2` : $gearsRetracted = TRUE \lor gearsExtended = TRUE$

then

  `leave_Control` : $Control := FALSE$
  `enter_Read` : $Read := TRUE$
  `enter_Read_sm1_S2` : $Read\_sm1\_S2 := TRUE$

end

Event *waitOpenRet* $\widehat{=}$
extends *waitOpenRet*

when

  `isin_HandleUp` : $HandleUp = TRUE$
  `smHandle_guards1` : $handle = FALSE$
  `isin_NormalMode` : $NormalMode = TRUE$
  `isin_Control` : $Control = TRUE$
  `isin_generalEV` : $generalEV = TRUE$
  `isin_Maneuvering` : $Maneuvering = TRUE$
  `isin_openEV` : $openEV = TRUE$
  `isin_DoorsOpening` : $DoorsOpening = TRUE$
  `_guards2` : $doorsOpen = FALSE$
  `smOpenEV_guards2` : $gearsRetracted = TRUE \lor gearsExtended = TRUE$

then

  `leave_Control` : $Control := FALSE$
  `enter_Read` : $Read := TRUE$
  `enter_Read_sm1_S2` : $Read\_sm1\_S2 := TRUE$

end

Event *startGearExt* $\widehat{=}$
extends *startGearExt*

when

`isin_HandleDown` : $HandleDown = TRUE$
`smHandle_guards3` : $handle = TRUE$
`isin_NormalMode` : $NormalMode = TRUE$
`isin_Control` : $Control = TRUE$
`isin_generalEV` : $generalEV = TRUE$
`isin_Maneuvering` : $Maneuvering = TRUE$
`isin_openEV` : $openEV = TRUE$
`isin_DoorsOpening` : $DoorsOpening = TRUE$
`_guards4` : $doorsOpen = TRUE$
`smOpenEV_guards3` : $gearsRetracted = TRUE \lor$
$gearsExtended = TRUE$

then

`leave_Control` : $Control := FALSE$
`enter_Read` : $Read := TRUE$
`leave_DoorsOpening` : $DoorsOpening := FALSE$
`enter_GearsManeuvering` : $GearsManeuvering := TRUE$
`enter_extendEV` : $extendEV := TRUE$
`enter_Read_sm1_S2` : $Read\_sm1\_S2 := TRUE$

end

Event *startGearRet* $\widehat{=}$
extends *startGearRet*

when

`isin_HandleUp` : $HandleUp = TRUE$
`smHandle_guards1` : $handle = FALSE$
`isin_NormalMode` : $NormalMode = TRUE$
`isin_Control` : $Control = TRUE$
`isin_generalEV` : $generalEV = TRUE$
`isin_Maneuvering` : $Maneuvering = TRUE$
`isin_openEV` : $openEV = TRUE$
`isin_DoorsOpening` : $DoorsOpening = TRUE$
`_guards4` : $doorsOpen = TRUE$
`smOpenEV_guards5` : $gearsRetracted = TRUE \lor$
$gearsExtended = TRUE$

then

`leave_Control` : $Control := FALSE$
`enter_Read` : $Read := TRUE$
`leave_DoorsOpening` : $DoorsOpening := FALSE$
`enter_GearsManeuvering` : $GearsManeuvering := TRUE$
`enter_retractEV` : $retractEV := TRUE$
`enter_Read_sm1_S2` : $Read\_sm1\_S2 := TRUE$

end

Event *fDoorsNotOpen* $\widehat{=}$
extends *fDoorsNotOpen*

when

`isin_Control` : $Control = TRUE$
`isin_GearsManeuvering` : $GearsManeuvering = TRUE$

```
        smOpenEV_guards1 : doorsOpen = FALSE
    then
        leave_Control : Control := FALSE
        enter_Block : Block := TRUE
        leave_NormalMode : NormalMode := FALSE
        leave_generalEV : generalEV := FALSE
        leave_Maneuvering : Maneuvering := FALSE
        leave_openEV : openEV := FALSE
        leave_GearsManeuvering : GearsManeuvering := FALSE
        enter_FailureMode : FailureMode := TRUE
        leave_retractEV : retractEV := FALSE
        leave_PendingContrGear : PendingContrGear := FALSE
        leave_extendEV : extendEV := FALSE
    end
Event  fDoorsUnlockedRet ≙
extends fDoorsUnlockedRet
    when
        isin_Control : Control = TRUE
        isin_Retracted : Retracted = TRUE
        smNormalMode_guards3 : doorsClosed = FALSE
    then
        leave_Control : Control := FALSE
        enter_Block : Block := TRUE
        leave_NormalMode : NormalMode := FALSE
        leave_Retracted : Retracted := FALSE
        enter_FailureMode : FailureMode := TRUE
    end
Event  fDoorsUnlockedExt ≙
extends fDoorsUnlockedExt
    when
        isin_Control : Control = TRUE
        isin_Extended : Extended = TRUE
        smNormalMode_guards6 : doorsClosed = FALSE
    then
        leave_Control : Control := FALSE
        enter_Block : Block := TRUE
        leave_NormalMode : NormalMode := FALSE
        leave_Extended : Extended := FALSE
        enter_FailureMode : FailureMode := TRUE
    end
Event  fDoorsUnlockedStart ≙
extends fDoorsUnlockedStart
    when
        isin_Control : Control = TRUE
        isin_PendingStart : PendingStart = TRUE
        smGeneralEV_guards7 : doorsClosed = FALSE
    then
        leave_Control : Control := FALSE
        enter_Block : Block := TRUE
        leave_NormalMode : NormalMode := FALSE
        leave_generalEV : generalEV := FALSE
        leave_PendingStart : PendingStart := FALSE
        enter_FailureMode : FailureMode := TRUE
    end
Event  fDoorsUnlockedStop ≙
extends  fDoorsUnlockedStop
    when
        isin_Control : Control = TRUE
        isin_PendingStop : PendingStop = TRUE
        smGeneralEV_guards8 : doorsClosed = FALSE
    then
        leave_Control : Control := FALSE
        enter_Block : Block := TRUE
        leave_NormalMode : NormalMode := FALSE
        leave_generalEV : generalEV := FALSE
        leave_PendingStop : PendingStop := FALSE
        enter_FailureMode : FailureMode := TRUE
    end
Event  delayOpenExt ≙
extends  delayOpenExt
    when
        isin_HandleDown : HandleDown = TRUE
        smHandle_guards3 : handle = TRUE
        isin_NormalMode : NormalMode = TRUE
        isin_Control : Control = TRUE
        isin_generalEV : generalEV = TRUE
        isin_Maneuvering : Maneuvering = TRUE
        isin_PendingContrDoor : PendingContrDoor = TRUE
        smManeuvering_guards2 : tContrDoor < CONTR_INTERVAL
        isin_PendingOpen : PendingOpen = TRUE
    then
        leave_Control : Control := FALSE
        enter_Read : Read := TRUE
        smManeuvering_actions2 : tContrDoor := tContrDoor + 1
        enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
    end
Event  delayOpenRet ≙
extends  delayOpenRet
    when
        isin_HandleUp : HandleUp = TRUE
        smHandle_guards1 : handle = FALSE
        isin_NormalMode : NormalMode = TRUE
```

```
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            smManeuvering_guards2 : $tContrDoor < CONTR\_INTERVAL$
            isin_PendingOpen : $PendingOpen = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            smManeuvering_actions2 : $tContrDoor := tContrDoor + 1$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event   delayCloseExt $\widehat{=}$
extends  delayCloseExt
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards3 : $handle = TRUE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            smManeuvering_guards2 : $tContrDoor < CONTR\_INTERVAL$
            isin_PendingClose : $PendingClose = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            smManeuvering_actions2 : $tContrDoor := tContrDoor + 1$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event   delayCloseRet $\widehat{=}$
extends  delayCloseRet
      when
            isin_HandleUp : $HandleUp = TRUE$
            smHandle_guards1 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_PendingContrDoor : $PendingContrDoor = TRUE$
            smManeuvering_guards2 : $tContrDoor < CONTR\_INTERVAL$
            isin_PendingClose : $PendingClose = TRUE$
      then
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            smManeuvering_actions2 : $tContrDoor := tContrDoor + 1$
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
    end
Event  waitExtended ≙
extends  waitExtended
    when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_openEV : openEV = TRUE
            isin_GearsManeuvering : GearsManeuvering = TRUE
            smControl_guards1 : doorsOpen = TRUE
            isin_extendEV : extendEV = TRUE
            _guards6 : gearsExtended = FALSE
    then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
    end
Event  waitRetracted ≙
extends  waitRetracted
    when
            isin_HandleUp : HandleUp = TRUE
            smHandle_guards1 : handle = FALSE
            isin_NormalMode : NormalMode = TRUE
            isin_Control : Control = TRUE
            isin_generalEV : generalEV = TRUE
            isin_Maneuvering : Maneuvering = TRUE
            isin_openEV : openEV = TRUE
            isin_GearsManeuvering : GearsManeuvering = TRUE
            smControl_guards1 : doorsOpen = TRUE
            isin_retractEV : retractEV = TRUE
            _guards7 : gearsRetracted = FALSE
    then
            leave_Control : Control := FALSE
            enter_Read : Read := TRUE
            enter_Read_sm1_S2 : Read_sm1_S2 := TRUE
    end
Event  delayExtend ≙
extends  delayExtend
    when
            isin_HandleDown : HandleDown = TRUE
            smHandle_guards3 : handle = TRUE
            isin_NormalMode : NormalMode = TRUE
```

```
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_GearsManeuvering : $GearsManeuvering = TRUE$
        smControl_guards1 : $doorsOpen = TRUE$
        isin_PendingContrGear : $PendingContrGear = TRUE$
        smControl_guards2 : $tContrGear < CONTR\_INTERVAL$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        smControl_actions1 : $tContrGear := tContrGear + 1$
        enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
    end
Event   delayRetract ≙
extends  delayRetract
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards3 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_GearsManeuvering : $GearsManeuvering = TRUE$
        smControl_guards1 : $doorsOpen = TRUE$
        isin_PendingContrGear : $PendingContrGear = TRUE$
        smControl_guards2 : $tContrGear < CONTR\_INTERVAL$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        smControl_actions1 : $tContrGear := tContrGear + 1$
        enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
    end
Event   cancelExtend ≙
extends  cancelExtend
    when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards4 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_GearsManeuvering : $GearsManeuvering = TRUE$
        smControl_guards1 : $doorsOpen = TRUE$
        isin_extendEV : $extendEV = TRUE$
then
        leave_HandleDown : $HandleDown := FALSE$
        enter_HandleUp : $HandleUp := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_extendEV : $extendEV := FALSE$
        enter_PendingContrGear : $PendingContrGear := TRUE$
        _actions6 : $tContrGear := 1$
        enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
end
```

Event *cancelRetract* $\widehat{=}$
extends *cancelRetract*

```
when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards2 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_GearsManeuvering : $GearsManeuvering = TRUE$
        smControl_guards1 : $doorsOpen = TRUE$
        isin_retractEV : $retractEV = TRUE$
then
        leave_HandleUp : $HandleUp := FALSE$
        enter_HandleDown : $HandleDown := TRUE$
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_retractEV : $retractEV := FALSE$
        enter_PendingContrGear : $PendingContrGear := TRUE$
        _actions5 : $tContrGear := 1$
        enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
end
```

Event *doExtend* $\widehat{=}$
extends *doExtend*

```
when
        isin_HandleDown : $HandleDown = TRUE$
        smHandle_guards3 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_GearsManeuvering : $GearsManeuvering = TRUE$
        smControl_guards1 : $doorsOpen = TRUE$
        isin_PendingContrGear : $PendingContrGear = TRUE$
        _guards8 : $tContrGear \geq CONTR\_INTERVAL$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_PendingContrGear : $PendingContrGear := FALSE$
        enter_extendEV : $extendEV := TRUE$
        enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
    end
Event  doRetract $\widehat{=}$
extends  doRetract
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards1 : $handle = FALSE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_GearsManeuvering : $GearsManeuvering = TRUE$
        smControl_guards1 : $doorsOpen = TRUE$
        isin_PendingContrGear : $PendingContrGear = TRUE$
        _guards9 : $tContrGear \geq CONTR\_INTERVAL$
    then
        leave_Control : $Control := FALSE$
        enter_Read : $Read := TRUE$
        leave_PendingContrGear : $PendingContrGear := FALSE$
        enter_retractEV : $retractEV := TRUE$
        enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
    end
Event  resumeExt $\widehat{=}$
extends  resumeExt
    when
        isin_HandleUp : $HandleUp = TRUE$
        smHandle_guards2 : $handle = TRUE$
        isin_NormalMode : $NormalMode = TRUE$
        isin_Control : $Control = TRUE$
        isin_generalEV : $generalEV = TRUE$
        isin_Maneuvering : $Maneuvering = TRUE$
        isin_openEV : $openEV = TRUE$
        isin_GearsManeuvering : $GearsManeuvering = TRUE$
        smControl_guards1 : $doorsOpen = TRUE$
        isin_PendingContrGear : $PendingContrGear = TRUE$
    then
        leave_HandleUp : $HandleUp := FALSE$
        enter_HandleDown : $HandleDown := TRUE$
        leave_Control : $Control := FALSE$
```

```
            enter_Read : $Read := TRUE$
            leave_PendingContrGear : $PendingContrGear := FALSE$
            enter_extendEV : $extendEV := TRUE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event  resumeRet ≙
extends  resumeRet
      when
            isin_HandleDown : $HandleDown = TRUE$
            smHandle_guards4 : $handle = FALSE$
            isin_NormalMode : $NormalMode = TRUE$
            isin_Control : $Control = TRUE$
            isin_generalEV : $generalEV = TRUE$
            isin_Maneuvering : $Maneuvering = TRUE$
            isin_openEV : $openEV = TRUE$
            isin_GearsManeuvering : $GearsManeuvering = TRUE$
            smControl_guards1 : $doorsOpen = TRUE$
            isin_PendingContrGear : $PendingContrGear = TRUE$
      then
            leave_HandleDown : $HandleDown := FALSE$
            enter_HandleUp : $HandleUp := TRUE$
            leave_Control : $Control := FALSE$
            enter_Read : $Read := TRUE$
            leave_PendingContrGear : $PendingContrGear := FALSE$
            enter_retractEV : $retractEV := TRUE$
            enter_Read_sm1_S2 : $Read\_sm1\_S2 := TRUE$
      end
Event  fDoorsNotOpenGearsUnlocked ≙
extends  fDoorsNotOpenGearsUnlocked
      when
            isin_Control : $Control = TRUE$
            isin_openEV : $openEV = TRUE$
            smManeuvering_guards8 : $doorsOpen = FALSE \land$
                  $gearsRetracted = FALSE \land gearsExtended = FALSE$
      then
            leave_Control : $Control := FALSE$
            enter_Block : $Block := TRUE$
            leave_NormalMode : $NormalMode := FALSE$
            leave_generalEV : $generalEV := FALSE$
            leave_Maneuvering : $Maneuvering := FALSE$
            leave_retractEV : $retractEV := FALSE$
            leave_PendingContrGear : $PendingContrGear := FALSE$
            leave_extendEV : $extendEV := FALSE$
            leave_GearsManeuvering : $GearsManeuvering := FALSE$
            leave_DoorsOpening : $DoorsOpening := FALSE$
            leave_openEV : $openEV := FALSE$
```

```
        enter_FailureMode : $FailureMode := TRUE$
    end
Event  $fGearsUnlocked \widehat{=}$
extends $fGearsUnlocked$
    when
        isin_Control : $Control = TRUE$
        isin_closeEV : $closeEV = TRUE$
        smManeuvering_guards11 : $gearsRetracted = FALSE \land$
                $gearsExtended = FALSE$
    then
        leave_Control : $Control := FALSE$
        enter_Block : $Block := TRUE$
        leave_NormalMode : $NormalMode := FALSE$
        leave_generalEV : $generalEV := FALSE$
        leave_Maneuvering : $Maneuvering := FALSE$
        leave_closeEV : $closeEV := FALSE$
        enter_FailureMode : $FailureMode := TRUE$
    end
Event  $wait \widehat{=}$
    when
        isin_Read_sm1_S2 : $Read\_sm1\_S2 = TRUE$
    then
        leave_Read_sm1_S2 : $Read\_sm1\_S2 := FALSE$
        enter_Read_sm1_S1 : $Read\_sm1\_S1 := TRUE$
    end
END
```

References

[1] Abadi, M. and Cardelli, L. (1998). *A theory of objects*. Springer.

[2] Abrial, J. (2010). *Modeling in Event-B: system and software engineering*. Cambridge University Press.

[3] Abrial, J., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., and Voisin, L. (2010). Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(6):447–466.

[4] Abrial, J., Lee, M., Neilson, D., Scharbach, P., and Sørensen, I. (1991). The B-method. In *VDM'91 Formal Software Development Methods*, pages 398–405. Springer.

[5] Abrial, J.-R. (2005). *The B-Book: Assigning programs to meanings*. Cambridge University Press.

[6] Abrial, J.-R., Su, W., and Zhu, H. (2012). Formalizing hybrid systems with Event-B. In *Abstract State Machines, Alloy, B, VDM, and Z*, pages 178–193. Springer.

[7] Åkesson, J., Årzén, K.-E., Gäfvert, M., Bergdahl, T., and Tummescheit, H. (2010). Modeling and optimization with Optimica and JModelica. org — Languages and tools for solving large-scale dynamic optimization problems. *Computers & Chemical Engineering*, 34(11):1737–1749.

[8] Alur, R. and Dill, D. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.

[9] Association, M. et al. (2000). Modelica–a unified object-oriented language for physical systems modeling: Tutorial version 1.4. *[Online] http://www.modelica.org/publications*.

[10] Association, M. et al. (2004). A unified object-oriented language for physical systems modeling: Language specification version 2.1. *Modelica Association*.

[11] Association, M. et al. (2005). Modelica–a unified object-oriented language for physical systems modeling. *Language Specification, Version*, 2.

[12] Back, R. and Sere, K. (1989). Stepwise refinement of action systems. In *Mathematics of Program Construction*, pages 115–138. Springer.

[13] Bagnato, A., Brosse, E., Quadri, I., and Sadovykh, A. (2015). Into-cps: An integrated "tool chain" for comprehensive: model-based design of cyber-physical systems. In *ICSSEA 2015 Proceedings*.

[14] Balasubramanian, K., Gokhale, A., Karsai, G., Sztipanovits, J., and Neema, S. (2006). Developing applications using model-driven design environments. *Computer*, 39(2):33–40.

[15] Banach, R. and Butler, M. (2014). A hybrid Event-B study of lane centering. In *Complex Systems Design & Management*, pages 97–111. Springer.

[16] Bastian, J., Clauß, C., Wolf, S., and Schneider, P. (2011). Master for co-simulation using FMI. In *8th International Modelica Conference. Dresden*.

[17] Bellini, P., Mattolini, R., and Nesi, P. (2000). Temporal logics for real-time system specification. *ACM Computing Surveys (CSUR)*, 32(1):12–42.

[18] Bendisposto, J., Birkhoff, M., Clark, J., Dobrikov, I., Fontaine, M., Fritz, F., R. Goebbels, D. Hansen, P. K., Koerner, P., Krings, S., Ladenberger, L., Luo, L., Leuschel, M., Plagge, D., , and Spermann, C. (2016). ProB 2.0 source code. http://github.com/bendisposto/prob2.

[19] Bengtsson, J., Larsen, K., Larsson, F., Pettersson, P., and Yi, W. (1996). Uppaal—a tool suite for automatic verification of real-time systems. *Hybrid Systems III*, pages 232–243.

[20] Bert, D., Boulmé, S., Potet, M.-L., Requet, A., and Voisin, L. (2003). Adaptable translator of b specifications to embedded c programs. In *FME 2003: Formal Methods*, pages 94–113. Springer.

[21] Bicarregui, J. C., Fitzgerald, J. S., Lindsay, P. A., Moore, R., and Ritchie, B. (1994). *Proof in VDM: a practitioner's guide*. Springer-Verlag New York, Inc.

[22] Bicknell, B. and Kanso, K. (2014). D2.4 – Full application in the smart energy domain. Technical report, ADVANCE.

[23] Bicknell, B., Kanso, K., Reis, J., Rampton, N., and McLeod, D. (2014). A rigorous approach for smart grid systems engineering using co-simulation. In *Proceedings of the 11th International Multidisciplinary Modelling and Simulation Multiconference*, Bordeaux, France.

[24] Blochwitz, T., Otter, M., Akesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D., et al. (2012). Functional Mockup Interface 2.0: The standard for tool independent exchange of simulation models. In *9th International Modelica Conference, Munich*.

[25] Blochwitz, T., Otter, M., Arnold, M., Bausch, C., Clauß, C., Elmqvist, H., Junghanns, A., Mauss, J., Monteiro, M., Neidhold, T., et al. (2011). The Functional Mockup Interface for tool independent exchange of simulation models. In *Modelica'2011 Conference*, pages 20–22.

[26] Bollella, G. and Gosling, J. (2000). The real-time specification for Java. *Computer*, 33(6):47–54.

[27] Bonakdarpour, B. (2008). Challenges in transformation of existing real-time embedded systems to cyber-physical systems. *ACM SIGBED Review*, 5(1):11.

[28] Boniol, F. and Wiels, V. (2013). Landing gear system. http://www.irit.fr/ABZ2014/landing_system.pdf.

[29] Boniol, F. and Wiels, V. (2014). The landing gear system case study. In *ABZ 2014: The Landing Gear Case Study*, pages 1–18. Springer.

[30] Borshchev, A. and Filippov, A. (2004). Anylogic—multi-paradigm simulation for business, engineering and research. In *The 6th IIE annual simulation solutions conference*, volume 150, page 45.

[31] Borshchev, A., Karpov, Y., and Kharitonov, V. (2002). Distributed simulation of hybrid systems with AnyLogic and HLA. *Future Generation Computer Systems*, 18(6):829–839.

[32] Borshchev, A., Kolesov, Y., and Senichenkov, Y. (2000). Java engine for UML based hybrid state machines. In *Simulation Conference Proceedings, 2000. Winter*, volume 2, pages 1888–1894. IEEE.

[33] Breunese, A. and Broenink, J. (1997). Modeling mechatronic systems using the SIDOPS+ language. *Simulation Series*, 29:301–306.

[34] Broenink, J. F. (1997). Modelling, simulation and analysis with 20-sim. *Journal A*, 38(3):22–25.

[35] Broenink, J. F. (1999). 20-sim software for hierarchical bond-graph/block-diagram models. *Simulation Practice and Theory*, 7(5):481–492.

[36] Broman, D., Brooks, C., Greenberg, L., Lee, E. A., Masin, M., Tripakis, S., and Wetter, M. (2013). Determinate composition of FMUs for co-simulation. In *Embedded Software (EMSOFT), 2013 Proceedings of the International Conference on*, pages 1–12. IEEE.

[37] Broman, D., Greenberg, L., Lee, E. A., Masin, M., Tripakis, S., and Wetter, M. (2015). Requirements for hybrid cosimulation standards. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 179–188. ACM.

[38] Brooks, C., Lee, E., Wetter, M., Nouidui, T., Broman, D., and Tripakis, S. (2012). Jfmi-a java wrapper for the functional mock-up interface. http://ptolemy.eecs.berkeley.edu/java/jfmi/.

[39] Brooks, C., Lee, E. A., and Tripakis, S. (2010). Exploring models of computation with Ptolemy II. In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 331–332. IEEE.

[40] Bureau, A. T. S. (2011). In-flight upset 154 km west of Learmonth, WA, 7 October 2008, VH-QPA, Airbus A330-303. ATSB Transport Safety Report, Aviation Occurence Investigation AO-2008-070 Final, http://www.atsb.gov.au/media/3532398/ao2008070.pdf, ATSB.

[41] Burns, A. and Wellings, A. (2007). *Concurrent and real-time programming in Ada 2005*. Cambridge Univ Pr.

[42] Butler, M. (2009). Using Event-B refinement to verify a control strategy. Working Paper. ECS, University of Southampton.

[43] Butler, M., Abrial, J.-R., and Banach, R. (2015). Modelling and refining hybrid systems in Event-B and Rodin. In *From Action System to Distributed Systems: The Refinement Approach*. Taylor & Francis.

[44] Butler, M. and Maamria, I. (2010). Mathematical extension in Event-B through the Rodin Theory component. Technical report, Technical Report. Deploy Project.

[45] Cansell, D. and Méry, D. (2012). Foundations of the B method. *Computing and informatics*, 22(3-4):221–256.

[46] Cansell, D., Méry, D., and Rehm, J. (2006). Time constraint patterns for Event-B development. *B 2007: Formal Specification and Development in B*, pages 140–154.

[47] Carloni, L., Passerone, R., and Pinto, A. (2006). *Languages and tools for hybrid systems design*, volume 1. now Publishers Inc.

[48] Chaudemar, J.-C., Savicks, V., and Butler, M. (2014). Co-simulation of Event-B and Ptolemy II Models via FMI. In *Embedded Real-time software and systems (ERTSS 2014)*.

[49] Dahmann, J. S. (1997). High Level Architecture for simulation. In *Distributed Interactive Simulation and Real Time Applications, 1997., First International Workshop on*, pages 9–14. IEEE.

[50] Dahmann, J. S., Fujimoto, R. M., and Weatherly, R. M. (1997). The Department of Defense High Level Architecture. In *Proceedings of the 29th conference on Winter simulation*, pages 142–149. IEEE Computer Society.

[51] Dijkstra, E. W., Dijkstra, E. W., Dijkstra, E. W., and Dijkstra, E. W. (1976). *A discipline of programming*, volume 1. prentice-hall Englewood Cliffs.

[52] Elliott, C., Vijayakumar, V., Zink, W., and Hansen, R. (2007). National instruments labview: a programming environment for laboratory automation and measurement. *Journal of the Association for Laboratory Automation*, 12(1):17–24.

[53] Fathabadi, A. S., Butler, M., and Rezazadeh, A. (2015). Language and tool support for event refinement structures in Event-B. *Formal Aspects of Computing*, 27(3):499–523.

[54] Fidge, C. (1991). Specification and verification of real-time behaviour using Z and RTL. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 393–409. Springer.

[55] Fitzgerald, J., Larsen, P. G., and Verhoef, M. (2014). *Collaborative Design for Embedded Systems*. Springer.

[56] Fitzgerald, J. S. and Jones, C. B. (2008). The connection between two ways of reasoning about partial functions. *Information Processing Letters*, 107(3):128–132.

[57] Fitzgerald, J. S., Larsen, P. G., and Verhoef, M. (2008). Vienna Development Method. *Wiley Encyclopedia of Computer Science and Engineering*.

[58] Fritzson, P. (2015). *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Wiley.

[59] Fritzson, P. and Bunus, P. (2002). Modelica – a general object-oriented language for continuous and discrete-event system modeling and simulation. In *Simulation Symposium, 2002. Proceedings. 35th Annual*, pages 365–380. IEEE.

[60] Giese, H. and Burmester, S. (2003). Real-time statechart semantics. Technical report, Hasso Plattner Institute, University of Paderborn.

[61] Halbwachs, N. (1998). Synchronous programming of reactive systems. In *Computer Aided Verification*, pages 1–16. Springer.

[62] Hall, A. (1990). Seven myths of formal methods. *Software, IEEE*, 7(5):11–19.

[63] Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

[64] Hayhurst, K. J., Veerhusen, D. S., Chilenski, J. J., and Rierson, L. K. (2001). A practical tutorial on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, National Aeronautics and Space Administration, Langley Research Center.

[65] Henzinger, T. (1996). The theory of hybrid automata. In *Logic in Computer Science, 1996. LICS'96. Proceedings., Eleventh Annual IEEE Symposium on*, pages 278–292. IEEE.

[66] Hinchey, M. and Monin, J. (2012). *Understanding Formal Methods*. Springer London.

[67] Hřebíček, J., Řezáč, M., et al. (2008). Modelling with maple and maplesim. In *22nd European Conference on Modelling and Simulation ECMS 2008 Proceedings*, pages 60–66.

[68] IEEE (2010a). Standard for Modeling and Simulation High Level Architecture (HLA) – Federate Interface Specification. *IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000)*, pages 1–378.

[69] IEEE (2010b). Standard for Modeling and Simulation High Level Architecture (HLA) – Framework and Rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pages 1–38.

[70] IEEE (2010c). Standard for Modeling and Simulation High Level Architecture (HLA) – Object Model Template (OMT) Specification. *IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000)*, pages 1–110.

[71] Jaspan, C., Keeling, M., Maccherone, L., Zenarosa, G. L., and Shaw, M. (2009). Software mythbusters explore formal methods. *IEEE Software*, 26(6):60.

[72] Jazequel, J.-M. and Meyer, B. (1997). Design by contract: The lessons of Ariane. *Computer*, 30(1):129–130.

[73] Johnson, C. (2009). The dangers of interaction with modular and self-healing avionics applications: redundancy considered harmful. In *27th International Conference on Systems Safety*, Huntsville, Alabama, USA.

[74] Karnopp, D., Rosenberg, R. C., Karnopp, D. C., and Karnopp, D. C. (1968). *Analysis and simulation of multiport systems: The bond graph approach to physical system dynamics*, volume 1. MIT press Cambridge, MA.

[75] Kernighan, B. W., Ritchie, D. M., and Ejeklint, P. (1988). *The C programming language*, volume 2. prentice-Hall Englewood Cliffs.

[76] Kesten, Y. and Pnueli, A. (1991). Timed and hybrid statecharts and their textual representation. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 591–620. Springer.

[77] Kneuper, R. (1997). Limits of formal methods. *Formal Aspects of Computing*, 9(4):379–394.

[78] Krogh, B., Lee, E., Lee, I., Mok, A., Rajkumar, R., Sha, L., Vincentelli, A., Shin, K., Stankovic, J., Sztipanovits, J., et al. (2008). Cyber-physical systems, executive summary. *CPS Steering Group, Washington DC, March.*

[79] Kübler, R. and Schiehlen, W. (2000). Two methods of simulator coupling. *Mathematical and Computer Modelling of Dynamical Systems*, 6(2):93–113.

[80] Ladenberger, L., Bendisposto, J., and Leuschel, M. (2009). Visualising Event-B models with B-Motion Studio. In *Formal Methods for Industrial Critical Systems*, pages 202–204. Springer.

[81] Lamport, L. (1994). The temporal logic of actions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3):872–923.

[82] Lamport, L. (2000). A summary of TLA+. Currently available from `http://research.microsoft.com/users/lamport/tla/tla.html`.

[83] Langstraat, P. (1993). *A guide to VHDL.* Springer Science & Business Media.

[84] Larsen, P. G., Gamble, C., Pierce, K., Ribeiro, A., and Lausdahl, K. (2014). Support for co-modelling and co-simulation: The crescendo tool. In *Collaborative Design for Embedded Systems*, pages 97–114. Springer.

[85] Lausdahl, K. G., Ribeiro, A., Visser, P., and Groen, F. (2011). D3.2b – co-simulation. Technical report, The DESTECS Project (INFSOICT-248134).

[86] Lavagno, L., Martin, G., and Selic, B. (2003). *UML for real: design of embedded real-time systems.* Springer.

[87] Lavazza, L., Quaroni, G., and Venturelli, M. (2001). Combining UML and formal notations for modelling real-time systems. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 196–206. ACM.

[88] Lee, E. (2006). Cyber-physical systems – are computing foundations adequate? In *Position Paper for NSF Workshop On Cyber-Physical Systems: Research Motivation, Techniques and Roadmap*, volume 1, pages 1–9. Citeseer.

[89] Lee, E. et al. (2006). The problem with threads. *Computer*, 39(5):33–42.

[90] Lee, E. and Seshia, S. (2011). *Introduction to Embedded Systems-A Cyber-Physical Systems Approach.* Lee & Seshia.

[91] Lee, E. A. (2008). Cyber physical systems: Design challenges. In *International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC).* Invited Paper.

[92] Lee, E. A. (2009). Computing needs time. *Communications of the ACM*, 52(5):70–79.

[93] Lee, E. A. (2011). Heterogeneous actor modeling. In *Proceedings of the ninth ACM international conference on Embedded software*, EMSOFT '11, pages 3–12, New York, NY, USA. ACM.

[94] Leuschel, M. and Butler, M. (2008). ProB: an automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2):185–203.

[95] Leveson, N. (2011). *Engineering a safer world: Systems thinking applied to safety.* MIT Press.

[96] Liu, H., Liu, X., and Lee, E. (2001). Modeling distributed hybrid systems in Ptolemy II. In *American Control Conference, 2001. Proceedings of the 2001*, volume 6, pages 4984–4985. IEEE.

[97] Mahony, B. and Dong, J. S. (2000). Timed communicating Object Z. *IEEE Transactions on Software Engineering*, 26(2):150–177.

[98] Mallet, F. and André, C. (2009). On the semantics of uml/marte clock constraints. In *Object/Component/Service-Oriented Real-Time Distributed Computing, 2009. ISORC'09. IEEE International Symposium on*, pages 305–312. IEEE.

[99] Manna, Z. and Pnueli, A. (1992). The temporal logic of reactive and concurrent systems: Specifications. the temporal logic of reactive and concurrent systems.

[100] Martin, C., Urquia, A., Sanchez, J., Dormido, S., Esquembre, F., Guzman, J., and Berenguel, M. (2004). Interactive simulation of object-oriented hybrid models, by combined use of Ejs, Matlab/Simulink and Modelica/Dymola. In *Proc. 18th European Simulation Multiconference*, pages 210–215.

[101] Marwedel, P. (2010). Embedded and cyber-physical systems in a nutshell. *DAC. COM Knowledge Center Article*, 20(10).

[102] MathWorks (2012). Simulink getting started guide. http://www.mathworks.co.uk/help/pdf_doc/simulink/sl_gs.pdf.

[103] MathWorks, T. (2005). Inc., Stateflow and Stateflow Coder User's Guide, Version 6. www.mathworks.com/help/pdf_doc/stateflow/sf_ug.pdf.

[104] McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195.

[105] Messner, B. and Tilbury, D. (2009). Control tutorials for MATLAB and Simulink. http://www.engin.umich.edu/class/ctms/.

[106] Meyer, B. (1992). Applying 'design by contract'. *Computer*, 25(10):40–51.

[107] MODELISAR (2010). Functional Mock-up Interface for Co-Simulation, Version 1.0. https://svn.fmi-standard.org/fmi/tags/v1.0/FMI_for_CoSimulation_v1.0.pdf.

[108] Möller, B. (2012). HLA Evolved Starter Kit Tutorial. http://www.pitch.se/hlatutorial.

[109] Moradzadeh, M. and Boel, R. (2010). A hybrid framework for coordinated voltage control of power systems. In *IPEC, 2010 Conference Proceedings*, pages 304–309. IEEE.

[110] Noll, C., Blochwitz, T., Neidhold, T., and Kehrer, C. (2011). Implementation of Modelisar Functional Mock-up Interfaces in SimulationX. In *8th International Modelica Conference. Dresden*.

[111] OMG (2011). UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1, formal/2011-06-02. http://www.omg.org/spec/MARTE/1.1/PDF.

[112] Oppenheim, A., Willsky, A., and Nawab, S. (1997). *Signals and Systems*. Prentice-Hall signal processing series. Prentice Hall.

[113] Otter, M. (2010). Functional mockup interface (fmi) for model exchange. *Modelica Newsletter*, 1.

[114] Otter, M., Årzén, K., and Dressler, I. (2005). StateGraph – a Modelica library for hierarchical state machines. In *Paper presented at the 4th International Modelica Conference*. Citeseer.

[115] Otter, M., Elmqvist, H., and Mattsson, S. (1999). Hybrid modeling in Modelica based on the synchronous data flow principle. In *Computer Aided Control System Design, 1999. Proceedings of the 1999 IEEE International Symposium on*, pages 151–157. IEEE.

[116] Petridis, K. and Clauß, C. (2015). Test of basic co-simulation algorithms using fmi. In *Proceedings of the 11th International Modelica Conference, Versailles, France, September 21-23, 2015*, Linköping Electronic Conference Proceedings. Linköping University Electronic Press, Linköpings universitet.

[117] Pnueli, A. (1977). The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57. IEEE.

[118] Pohlmann, U., Schäfer, W., Reddehase, H., Röckemann, J., and Wagner, R. (2012). Generating Functional Mockup Units from software specifications. In *Proceedings of the 9th International MODELICA Conference*, pages 765–774.

[119] QTronic (2012). FMU SDK (FMU Software Development Kit). http://www.qtronic.de/de/fmusdk.html.

[120] Rajkumar, R., Lee, I., Sha, L., and Stankovic, J. (2010). Cyber-physical systems: the next computing revolution. In *Proceedings of the 47th Design Automation Conference*, pages 731–736. ACM.

[121] Robinson, K. (2012). System Modelling & Design Using Event-B. Technical report, School of Computer Science and Engineering, The University of New South Wales.

[122] Romanowicz, E. (2008). B method - an overview through example. McMaster University, Hamilton, Ontario, Canada.

[123] Rushby, J. (2009). Safety, Fault-tolerance, Verification, and Certification for Embedded Systems. http://chess.eecs.berkeley.edu/eecs149/lectures/Rushby-SFVC.pdf.

[124] Sahbani, A. and Pascal, J. (2000). Simulation of hybrid systems using stateflow. In *Proceedings of the 14th European Simulation Multiconference on Simulation and Modelling: Enablers for a Better Quality of Life*, pages 271–275.

[125] Sarshogh, M. and Butler, M. (2011). Specification and refinement of discrete timing properties in Event-B. *Automated Verification of Critical Systems 2011*, 46.

[126] Savicks, V., Butler, M., and Colley, J. (2014a). Co-simulating Event-B and continuous models via FMI. In *Proceedings of the 2014 Summer Simulation Multiconference*, page 37. Society for Computer Simulation International.

[127] Savicks, V., Butler, M., and Colley, J. (2014b). Co-simulation environment for Rodin: Landing gear case study. In *ABZ 2014: The Landing Gear Case Study*, pages 148–153. Springer.

[128] Savicks, V., Butler, M., and Colley, J. (2014c). Integrating formal verification and simulation of hybrid systems – Rodin multi-simulation plug-in. In *Proceedings of DCSIMULTECH 2014/SIMULTECH*. SCITEPRESS Digital Library.

[129] Savicks, V., Butler, M., Colley, J., and Bendisposto, J. (2014d). Rodin multi-simulation plug-in. *5th Rodin User and Developer Workshop*.

[130] Savicks, V., Snook, C., and Butler, M. (2009). Animation of UML-B State-machines. In *Rodin workshop, Dusseldorf*. University of Southampton.

[131] Savicks, V., Snook, C., and Butler, M. (2011). Event-B wiki: Event-B Statemachines. http://wiki.event-b.org/index.php/Event-B_Statemachines.

[132] Schierz, T., Arnold, M., and Clauß, C. (2012). Co-simulation with communication step size control in an FMI compatible master algorithm. In *9th International Modelica Conference. Munich*.

[133] Schneider, S. (2001). *The B-method: An Introduction*. Cornerstones of computing. Palgrave.

[134] Seidewitz, E. (2003). What models mean. *Software, IEEE*, 20(5):26–32.

[135] Sghairi, M., Aubert, J.-J., Brot, P., de Bonneval, A., Crouzet, Y., and Laarouchi, Y. (2009). Distributed and reconfigurable architecture for flight control system. In *Digital Avionics Systems Conference, 2009. DASC'09. IEEE/AIAA 28th*, pages 6–B. IEEE.

[136] Sistla, A. P. (1994). Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–511.

[137] Snook, C. and Butler, M. (2008). Uml-b and event-b: an integration of languages and tools. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 336–341. ACTA Press.

[138] Spivey, J. M. and Abrial, J. (1992). *The Z notation*. Prentice Hall Hemel Hempstead.

[139] Struth, G. (2005). Software verification and testing: Course material. Technical report, Department of Computer Science, The University of Sheffield.

[140] Su, W., Abrial, J.-R., and Zhu, H. (2012). Complementary methodologies for developing hybrid systems with Event-B. In *Formal Methods and Software Engineering*, pages 230–248. Springer.

[141] Tiwari, A. (2002). Formal semantics and analysis methods for Simulink Stateflow models. *Unpublished report, SRI International*.

[142] Tudoret, S., Nadjm-Tehrani, S., Benveniste, A., and Strömberg, J. (2000). Co-simulation of hybrid systems: Signal-Simulink. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 623–639. Springer.

[143] Verhoef, M. H. G. (2009). *Modeling and validating distributed embedded real-time control systems*. [Sl: sn].

[144] Weedy, B. M., Cory, B. J., Jenkins, N., Ekanayake, J., and Strbac, G. (2012). *Electric power systems*. John Wiley & Sons.

[145] Wen, R. B. Z. H. S. and Xiaofeng, W. (2012). Adding continuous behaviour to Event-B. In *Proc. ABZ 2012: 3rd International Conference on Abstract State Machines Alloy B and Z*, LNCS. Spinger-Verlag.

[146] Widl, E., Muller, W., Elsheikh, A., Hortenhuber, M., and Palensky, P. (2013). The FMI++ library: A high-level utility package for FMI for model exchange. In *Modeling and Simulation of Cyber-Physical Energy Systems (MSCPES), 2013 Workshop on*, pages 1–6. IEEE.

[147] Wolfram (2015). SystemModeler. http://www.wolfram.com/system-modeler/.

[148] Woodcock, J. and Davies, J. (1996). *Using Z: specification, refinement, and proof.* Prentice-Hall, Inc.

[149] Yoon, Y. and Jang, J. (2012). SimulationX, multi-domain simulation and modeling tool for the design, analysis, and optimization of complex systems. *Journal of Drive and Control*, 9(1):56–69.