Tampere University

JUSSI-PEKKA PENTTINEN

An Object-Oriented Modelling Framework for Probabilistic Risk and Performance Assessment of Complex Systems

Tampere University Dissertations 283

Tampere University Dissertations 283

JUSSI-PEKKA PENTTINEN

An Object-Oriented Modelling Framework for Probabilistic Risk and Performance Assessment of Complex Systems

ACADEMIC DISSERTATION
To be presented, with the permission of
the Faculty of Engineering and Natural Sciences
of Tampere University,
for public discussion in the auditorium S2
of the Sähkötalo building, Korkeakoulunkatu 3, Tampere,
on 4 September 2020, at 12 o'clock.

ACADEMIC DISSERTATION
Tampere University, Faculty of Engineering and Natural Sciences
Finland

Cover design: Roihu Inc.

ISBN 978-952-03-1634-1 (print)
ISBN 978-952-03-1635-8 (pdf)
ISSN 2489-9860 (print)
ISSN 2490-0028 (pdf)
http://urn.fi/URN:ISBN:978-952-03-1635-8

PunaMusta Oy – Yliopistopaino
Vantaa 2020

PREFACE

This study was carried out at Tampere University of Technology (TUT). My work at Ramentor Oy as a chief architect of ELMAS software has formed the basis of the research. I would like to express my sincere gratitude to TUT graduate school, Kari Koskinen from TUT and Timo Lehtinen from Ramentor Oy for providing me with the possibility of writing this thesis.

I would like to thank Seppo Virtanen and Per-Erik Hagmark for leading me into the area of reliability and risk analysis. I hope that I will be able to adopt at least a bit of the enthusiasm of Seppo Virtanen and the diligent work of Per-Erik Hagmark, which was an admirable combination.

I completed my master's thesis in 2005. Since then my main job has been the research and development of simulation algorithms and analysis tools. I feel privileged for the possibility to apply my interests and studies related to software science and discrete mathematics for answering tangible needs from various industry sectors. I would like to express my gratitude to my co-workers Miikka Tammi, Joel Turpela, Turkka Lehtinen and Ville Vuorela from Ramentor Oy for the smooth and pleasant co-operation. Their feedback during various actual analysis cases has been essential for developing the tools and solutions to the right direction.

In 2015 I started working in a research, which scope was to study the availability design methods that could be applied to CERN particle accelerators. The research was made as a part of the global Future Circular Collider (FCC) study. My responsibility was the creation of an enhanced modelling concept and distributed computing architecture. I would like to thank Johannes Gutleber and Arto Niemi from CERN for various new ideas and thorough validation of my suggestions and solutions during the research. In 2017 this work received the FCC study innovation award, which is given for advancements with high innovation capacity, high socio-economic impact potential or high relevance for the concepts of a frontier particle physics research infrastructure. This thesis improves further the award-winning approach.

Finally, my deepest gratitude to my family, friends and teammates for making this part of my life very enjoyable.

Jussi-Pekka Penttinen
December 3, 2018
Tampere, Finland

ABSTRACT

This thesis introduces a modelling framework, which is developed for risk and performance assessment of large and complex systems with dynamic behaviours. The framework supports the most common reliability and operation modelling techniques, and permits their customisation. This ensures a high degree of freedom for the modeller to describe accurately the system without limitations imposed by an individual technique. The use of an object-oriented paradigm increases flexibility and decreases the semantics gap between the model and real world, which are issues with traditional techniques. The framework is named as Analysis of Things (AoT) to emphasise its universal nature and wide application possibilities. The AoT models are defined by using a Triplets data format, which is platform-independent and tabular. A single table declares the applied modelling techniques, creates the model structure, and assigns the parameter values. The format enables straightforward manual model edit while maintaining direct database compatibility. This thesis also documents a calculation engine, which has been developed for analysis of AoT models. The engine compiles dynamically the most efficient simulation algorithm for each modelling technique. A catalogue of built-in techniques is included in this thesis to demonstrate the application of the framework. The configuration of the simulation algorithm is presented for each technique. The AoT model creation is illustrated by using simple example models. Various techniques can be combined to build a comprehensive risk and performance model that systematically includes all essential details. The advanced features of the AoT framework have wide-ranging applications for analysis of reliability, availability, and operational performance of complex industrial products and processes.

CONTENTS

ABBREVIATIONS

AoT	Analysis of Things
API	Application Programming Interface
BFS	Behavioural Fault Simulation
CDF	Cumulative Distribution Function
CEN	European Committee for Standardization
CERN	European Organisation for Nuclear Research
CM	Corrective Maintenance
CPN	Coloured Petri Net
CSM	Conjoint System Model
CTMC	Continuous-Time Markov Chain
DES	Discrete Event Simulation
DFM	Dynamic Flowgraph Methodology
DFT	Dynamic Fault Tree
DSM	Design Structure Matrix
ELMAS	Event Logic Modelling and Analysis Software
ERP	Enterprise Resource Planning
ESPN	Extended Stochastic Petri Net
ETA	Event Tree Analysis
FCC	Future Circular Collider
FFIP	Functional-Failure Identification and Propagation
FLSA	Failure Logic Synthesis and Analysis

FMEA	Failure Modes and Effects Analysis
FMECA	Failure Modes and Effects and Criticality Analysis
FPTC	Fault Propagation and Transformation Calculus
FPTN	Failure Propagation and Transformation Notation
FTA	Fault Tree Analysis
GSPN	Generalised Stochastic Petri Net
GTS	Guarded Transitions Systems
GUI	Graphical User Interface
HiP-HOPS	Hierarchically Performed Hazard Origin and Propagation Studies
IEC	International Electrotechnical Commission
IoC	Inversion of Control
ISO	International Organization for Standardization
KPI	Key Performance Indicator
LHC	Large Hadron Collider
MBDA	Model-Based Dependability Assessment
MBSA	Model-Based Safety Assessment
MES	Manufacturing Execution Systems
MRP	Markov Renewal Process
MRT	Mean Repair Time
MTTF	Mean operating Time To Failure
MTTR	Mean Time To Restoration
NASA	National Aeronautics and Space Administration
OEE	Overall Equipment Effectiveness
OpenMARS	an Open Modelling approach for Availability and Reliability of Systems
PERT	Project Evaluation and Review Technique
PM	Preventive Maintenance

PNML	Petri Net Markup Language
PSA	Probabilistic Safety Assessment
R&D	Research and Development
RAM	Reliability, Availability and Maintainability
RAMS	Reliability, Availability, Maintainability and Safety
RBD	Reliability Block Diagram
RM	Risk Management
S^3E	Software-intensive Systems Specification Environment
SAML	Safety Analysis Modelling Language
smartIflow	State Machines for Automation of Reliability-related Tasks using Information FLOWs
SMP	Semi-Markov Process
SPN	Stochastic Petri Net
SRGM	Software Reliability Growth Model
Tekes	Finnish funding agency for technology and innovation
TUT	Tampere University of Technology
UID	Unique Identifier
URL	Uniform Resource Locator
XML	Extensible Markup Language

1 INTRODUCTION

Reliability, availability, and operational performance are integral factors to consider in system design and management. They provide an essential amount of information for risk-informed decision-making. Risk and performance analyses are used, for example, to compare design alternatives, to estimate the return of investment time, and to optimise maintenance of the system. Today's complex systems require sophisticated methods to analyse the effect of failures on overall system performance. This is especially true when considering dynamic interdependencies between failures, production and maintenance. Modern reliability engineering still confronts challenges that relate to the representation of the system and quantification of the model [1]. The traditional methods are not always flexible enough to include all the needed details, which can lead to unrealistic simplifications. For example, Fault Tree Analysis (FTA) [2] is one of the most prominent techniques in risk assessment, but without extensions it lacks the power to express essential dependability patterns [3].

This thesis introduces a modelling framework for probabilistic risk and performance assessment of complex systems with dynamic behaviour. The name of the framework is Analysis of Things (AoT), which emphasises its universal nature and wide application possibilities. AoT supports traditional risk assessment modelling techniques [4], such as FTA, Reliability Block Diagram (RBD) [5], Markov analysis [6], Failure Modes and Effects Analysis (FMEA) [7] and Petri nets [8]. The customisation and the combination of the techniques are enabled. A calculation of Key Performance Indicators (KPIs) [9], such as Overall Equipment Effectiveness (OEE) [10], can be included in the model for assessing the overall effect of stochastic component failures. The AoT framework can be seen as a meta-model, which generalises various modelling techniques, and supports inclusion of any domain-specific features. The possibility to customise the modelling techniques helps avoiding the need of simplified or unnatural solutions. The flexibility and extensibility of AoT ensures the lowest possible semantics gap between the model and real world.

The foundation of the AoT framework is a research that started two decades ago. The reliability-based design methods [11] were first refined to prototype software [12, 13, 14] and then to a commercial Event Logic Modelling and Analysis Software (ELMAS) [15, 16, 17] tool. Several risk assessment cases in various industry sectors have been made with ELMAS [18]. Especially the needs related to the modelling of complex systems and dynamic operation phases required improvements to the traditional techniques [19, 20, 21, 22]. The use of ELMAS in particle accelerator availability modelling [23, 24] motivated us for a research and development of an Open Modelling approach for Availability and Reliability of Systems (OpenMARS) [25, 26]. It enables including customised features with minimal need of programming during modelling. AoT improves OpenMARS further with more efficient data format and more straightforward communication between sub models.

AoT uses the Triplets data format for model definition. The declaration of modelling techniques, the creation of model structures, and the assignment of parameter values can be all done with a single table. Each definition is a triplet that can be stored as a row of a three column table. The format is human-readable, which enables defining models manually with any tool that supports editing tables. In tabular format the model can be stored to a database directly without pre-processing. The Triplets format is platform-independent, open and non-proprietary, which makes it suitable for sharing information between any commercial or public-domain software.

The high flexibility of AoT is achieved by using object-oriented paradigm as a basis of the framework. All models consists of objects, which have attributes. In addition to creation of a model object structure and assignment of parameter values, the model also contains definition of the model object types. Each type is a class, which declares the types of the attributes that are assigned for the object instances. A modelling technique is defined by declaring a catalogue of classes. The framework contains built-in class declarations for traditional techniques, but allows a modeller to improve them to overcome their limitations.

To verify the feasibility of AoT in practice, a calculation engine has been developed for analysis of AoT models. This thesis documents a stochastic Discrete Event Simulation (DES) process for calculation of various explicit and concrete results based on the models. The engine can be configured to analyse any modelling technique that is defined with the AoT framework. The configuration can also be used for including user-defined special features in the simulation algorithm. With the help of dynamic compilation, it is possible to optimise the algorithm based on the content of the simulated model. The adaptive algorithm includes only the essential procedures, which increases the efficiency of the simulating different techniques. The distributed processing architecture of the calculation engine permits parallel Monte-Carlo simulation in a computing cluster, which enables efficient analysis of large models. A future plan is to develop tools with a user-friendly Graphical User Interface (GUI) that are specially designed for the AoT framework. This forms a highly potent environment, which is suitable for comprehensive risk and performance assessment of systems in diverse domains.

To demonstrate the application of the framework, a brief catalogue of built-in techniques is included in this thesis. A class declaration and a configuration of a DES algorithm are presented for each technique. The catalogue can be used as a basis for implementation of an arbitrary new technique. The thesis also presents the creation of simple example models by using the built-in techniques. The examples include interconnecting of various sub models that use different modelling techniques.

1.1 Objectives of the thesis

The main objective of this thesis is to introduce the AoT framework and justify its feasibility for the analysis of today's complex and dynamic systems. Uniquely, the framework enables defining the applied modelling technique before the model creation. With the detailed description of the methodology and the presented versatile examples, this thesis aims to convince the reader that AoT is capable of creating new modelling techniques, and customising and combining of them. This flexibility enables using the framework as a common foundation for a large variety of tools. Each tool can apply a custom-made modelling technique that is optimal for certain special analysis need. When compared to a complex multi-purpose tool, using a highly customised tool can be more efficient and easier to use.

The main difference between AoT and other approaches is the use of a tabular data format for model definition. This thesis documents the Triplets format in such detail that the reader is able to define models manually by editing a table. The novel way of storing object-oriented models into a single table is especially suitable for transferring information between tools and databases. This thesis provides the needed details for implementing Triplets format import and export interfaces for new or existing tools.

The principles of creating a DES algorithm for any modelling technique are described in this thesis. The algorithms are created with the help of dynamic compilation. This differs from the traditional approach of using static simulation algorithms. The flexible procedure of creating simulation algorithms aims to prove that efficient calculation of versatile analysis results is enabled for AoT modelling techniques. The procedure of defining new modelling techniques and simulation algorithms for them forms an environment that lowers the time required for developing new analysis tools for special needs.

Hypothesis: By applying the object-oriented paradigm it is possible to create a single framework that can utilise, combine and customise various risk and performance assessment techniques to answer the challenging analysis needs of today's complex and dynamic systems.

Research questions related to the applying of the object-oriented paradigm to risk and performance assessment:

1. What kind of class declaration is suitable for efficient use of individual modelling techniques and enables a clear way to add connections between them?
2. How the object-oriented models can be defined clearly and efficiently by using a tabular format?
3. What stochastic DES procedure is suitable for generic and versatile analyses of object-oriented models?

The individual objectives of this thesis are summarised as follows:

- To introduce the AoT framework for probabilistic risk and performance assessment of complex systems;
- To document the syntax of the Triplets data format, which is used for efficient model definition;
- To present how the framework covers traditional risk assessment techniques;
- To describe how custom KPIs can be included in models;
- To present how AoT allows combining various modelling techniques and customising them for special needs; and
- To introduce the stochastic DES procedure of the calculation engine, which can be used for producing concrete analysis results from the AoT models.

1.2 Research methods and restrictions

This thesis presents the result of a research, which aim was to develop a new approach for risk and performance assessment of complex systems. The principal idea was to enable utilising entirely the available data in the most efficient manner. The approach should support hybrid modelling by combining qualitative data from experts' knowledge with quantitative data that is collected from various sources. The modelling of basic situations should be as simple as with traditional risk assessment techniques, but without restrictions for advanced modelling of system performance characteristics, dynamic interdependencies or other complex properties. To achieve the needed flexibility, an object-oriented paradigm was selected as a basis.

The result of the research is the AoT framework. It integrates various quantitative risk and performance assessment techniques into a common methodological foundation. In the AoT methodology the qualitative information about cause-consequence relations is collected by defining explicit rules that form a system model. The rules are parametrised with the quantitative information. The definition of probability distributions for the rules enables including stochastic information in the model.

This thesis specifies the methodology of the AoT framework, and presents how it is applied in various modelling situations. The application of the framework is described by using simple and fictional example models. The examples are intended to be as clear as possible for illustrating the features needed for solving the situations. The presented modelling features, such as inclusion of maintenance actions, operation phases, mode-dependent failure rates and user-defined KPIs, have been selected based on concrete needs of various risk assessment cases [18, 19, 20, 21, 22, 23, 24].

The presented examples apply basic features of traditional risk assessment techniques. In addition to the definition of the example model structures, the features of the applied modelling techniques are formally declared. The declaration includes configuring the simulation algorithms for analysis of the models. The framework enables applying similar declaration procedure to formally define an arbitrary modelling technique. This allows including any customised features for advanced or domain-specific needs. The focus in this thesis is on existing and well-known techniques, which aims to verify the applicability of the framework in traditional modelling situations. The common formal approach for declaring various techniques enables combining and extending the presented features.

This thesis does not include a practical analysis case study that could be used for comparing a solution made by applying AoT with solutions from existing techniques. The comparison is made in more general level by classifying different techniques based on their flexibility, modelling power and other disquisition criteria. When considering an individual case, the tool that has been developed for the applied technique affects significantly to the usability and performance of the approach. AoT provides a common formalism for existing and customised techniques, which enables creating tools that are compatible with any of them. Furthermore, the well-defined data format of the framework allows including a support of AoT model creation in existing tools. Selection and implementation of tools to support AoT model creation and analysis, and evaluation of them with respect to practical cases are issues of a follow-up research.

This thesis concentrates on the risk and performance assessment. It does not cover the other issues in the risk management process, such as (i) the data acquisition, (ii) the determination of the goal and the studied KPIs, and (iii) deciding based on the analysis whether the residual risk is tolerable and which actions should be taken.

1.3 Outline and contributions of the thesis

This thesis is divided into 6 chapters, which contents are summarised as follows:

- Chapter 1 is an introduction into the field of risk and performance assessment. The background and motivation for the study are given, followed by the objectives, research methods, restrictions and contributions of the thesis.
- Chapter 2 presents brief reviews of the recent standards that define the risk assessment terminology, the traditional techniques that have been used for the risk assessment of complex systems, and the modern Model-Based Dependability Assessment (MBDA) formalisms.
- Chapter 3 describes the methodology by introducing the AoT framework and documenting the syntax of the Triplets data format. The chapter also describes the stochastic DES process of the calculation engine, which enables analysing AoT models.
- Chapter 4 presents how the framework can be applied with various traditional risk assessment techniques. The declaration of the model element classes and

the configuration of the simulation algorithm are presented for each modelling technique.

- Chapter 5 presents simple example models by using the declared modelling techniques. The examples include special features, such as definition of custom KPIs and combination of different techniques.
- Chapter 6 summarises the thesis. It also contains the discussion about the needs for this type of framework and presents suggestions how AoT could be used in the future.

The introduced AoT framework and its tabular Triplets data format are results of the author's research to improve the applicability of the OpenMARS approach [25, 26]. The most distinctive feature of OpenMARS is the use of tabular model definition format. The Triplets data format improves this unique feature further by reducing the number of required tables from five to one. When compared to markup or programming languages that are used with other similar approaches, the model definition by using tables was found more familiar for basic system engineers.

Large analysis targets create a need for modelling tools that minimise the required manual work. A single table that collects entirely the model data is a significant advantage when developing interfaces for GUIs and databases. AoT is open and non-proprietary permitting any researcher or commercial tool provider to create solutions based on it. The framework can form a common foundation for various analysis tools, which each provide a GUI or an automatised model definition for certain modelling techniques or special analysis needs.

The OpenMARS approach was developed in co-operation between European Organisation for Nuclear Research (CERN), Tampere University of Technology (TUT) and Ramentor Oy. The author's responsibility in the project was the development of the OpenMARS methodology and the implementation of a calculation engine to analyse the models. With minor improvements the same calculation engine is applicable to analysis of AoT models. This thesis publishes first time the principles that make the calculation engine highly configurable for various needs. The flexibility enables using an optimal simulation algorithm for each modelling situation.

This thesis presents configurations for basic risk assessment techniques, which can be used as a basis for configuring calculation of more advanced techniques and their combinations. Arto Niemi together with the author have successfully tested the applicability of the approach for challenging modelling needs from CERN. A particle collider operations model linked fault tree and Markov models, contained failure rates that change based on an active operation mode, and included calculation of luminosity production [23, 24]. The n+1 redundancies with load sharing were modelled in a research of identifying critical systems for a future circular collider [27].

The main contributions of this research can be summarised as follows:

- A novel methodology for probabilistic risk and performance assessment for complex systems is introduced. The AoT framework enables combining various risk assessment techniques, customising them for special needs, and inclusion of any system KPIs in the model.
- Tabular Triplets data format for the definition of AoT models is presented. The platform-independent and non-proprietary format is open for use with any GUI. Manual model edit by using various modelling techniques is enabled with any tool that supports editing tables. The declaration of a new technique and the definition of customised features can be included in the same model table.
- A stochastic DES algorithm of a calculation engine is described. The algorithm is compiled dynamically for each modelling technique. This increases the flexibility when compared to other approaches that use static simulation algorithms. With similar procedure any DES or other algorithm can be formed based on the needs of the analysed model. To enable efficient analysis of large AoT models, the calculation engine supports parallel simulation in a distributed computing cluster.

2 STATE OF THE ART

2.1 Risk and reliability terminology in the latest standards

The field of reliability and risk assessment has decades long history, but the used terminology has not yet been entirely fixed. This section presents the terminology that is used in this thesis. The definitions of the terms are made based on a review of the latest reliability, maintenance and risk assessment standards.

International Organization for Standardization (ISO) defines Risk Management (RM) as coordinated activities to direct and control an organisation with regard to risk [28, Sec. 2.1]. The RM process is included in systems engineering, which is defined by National Aeronautics and Space Administration (NASA) as a methodical, multi-disciplinary approach for the design, realisation, technical management, operations, and retirement of a system [29, p.3]. It focuses on how complex engineering projects should be designed and managed over their life cycles. It is obvious that risk is an integral issue in systems engineering and disciplined RM must be always included.

Risk assessment is defined by ISO as the overall process of risk identification, risk analysis, and risk evaluation [28, Sec. 3.4.1]. This thesis concentrates on quantitative risk assessment, where:

1. the risk identification is made by creating a comprehensive model that unambiguously collects the available information about the uncertainty and other features that affect the risk,
2. the risk analysis is made by using stochastic DES to calculate explicit and concrete results from the model, and
3. the versatile analysis results are evaluated with respect to the the context of the studied case.

The RM can take many forms but always the risk assessment is a crucial part of it. Figure 2.1 illustrates how the three steps of the risk assessment are activities of the RM process, as described by ISO [30, Fig. 3]. A brief description about the content within this thesis is included in each step.

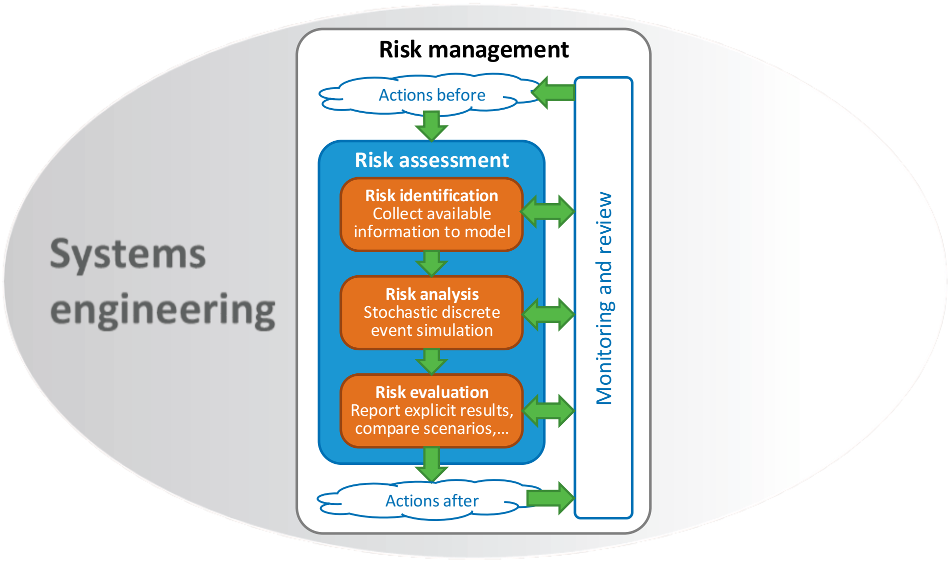

Figure 2.1 The risk assessment process as a part of RM [30, Fig. 3] and systems engineering

Risk is defined by ISO as an effect of uncertainty on objectives. An effect is a positive and/or negative deviation from the expected. Risk can also be expressed in terms of a combination of the consequences of an event and the associated likelihood of occurrence. The uncertainty comes from a lack of full knowledge related to the consequences and/or the likelihood [28, Sec. 1.1]. As an example, the likelihood can be quantified as a probability and the consequences as a cost.

An acronym RAMS is used for terms Reliability, Availability, Maintainability and Safety. These generic and essential risk related system quality attributes can be used for risk management irrespective of the type of item considered. International Electrotechnical Commission (IEC) defines an *item* as an individual part, component, device, subsystem, or system [31, Sec. 192-01-01]. The system is a set of interrelated items that collectively fulfil a requirement [31, Sec. 192-01-03]. A system is considered to have a defined real or abstract boundary. A system structure may be hierarchical, e.g. system, subsystem, component, etc.

RAMS analysis studies the states of items, for example, whether an item is in *up state* or *down state*. European Committee for Standardization (CEN) characterises that in the up state an item can perform as required [32, Sec. 6.4]. In this thesis the basic up state is called *normal state*. Sometimes more than one up states are needed. For example, there can be a up state for each different operation mode or speed.

A basic down state is called *fault state*. It is started by a *failure*, which is defined as a loss of the ability of an item to perform a required function [32, Sec. 5.1]. Corrective Maintenance (CM) is made during the fault state. The event that re-establishes the up state is called *restoration* [32, Sec. 8.9]. Preventive Maintenance (PM) is made during a *service state*, which is an example of other down state. Like with the fault state, the start and the end of the PM task define the duration of the service state.

An example of an item's state changes is shown in Figure 2.2. The names of the illustrated states and events are based on a CEN standard [32]. It is also possible that some PM tasks do not necessarily interrupt an item to perform as required.

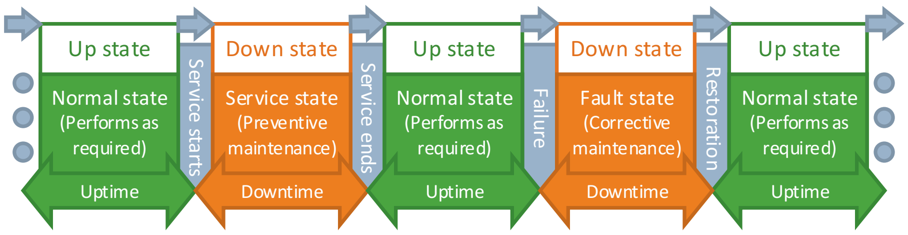
Up state
Normal state (Performs as required)
Uptime
Service starts
Down state
Service state (Preventive maintenance)
Downtime
Service ends
Up state
Normal state (Performs as required)
Uptime
Failure
Down state
Fault state (Corrective maintenance)
Downtime
Restoration
Up state
Normal state (Performs as required)
Uptime

Figure 2.2 An example of state changes of an item

IEC defines *reliability* as the ability to perform as required, without failure, for a given time interval, under given conditions [31, Sec. 192-01-24]. Reliability describes how long the item stays in up state. Reliability may be quantified as a probability or performance indicators by using appropriate measures, and is then referred to as reliability performance. When quantified, it is assumed that the item is in a state to perform as required at the beginning of the time interval.

Reliability measure is the probability of performing as required for certain time interval, under given conditions [31, Sec. 192-05-05]. *Unreliability* is the probability of the complement situation, which is preferred in this thesis because it refers more clearly to the quantified reliability performance. In addition, because of E-notation it is more convenient to have a value near zero when highly reliable systems are studied. For example, it is simpler to display clearly an unreliability 3.7E-8 when compared to a corresponding reliability 0.999999963.

There exists various functions and measures to quantify the reliability performance. In this thesis detailed quantification is made by using the Cumulative Distribution Function (CDF), which defines the unreliability for different time intervals. In addition to CDF, for example, Mean operating Time To Failure (MTTF) [31, Sec. 192-05-11], or mean failure rate [31, Sec. 192-05-07] are ways to quantify the reliability performance.

Availability is the ability to be in a state to perform as and when required, under given conditions, assuming that the necessary external resources are provided [32, Sec. 4.7]. Required external resources, other than maintenance resources, do not affect the availability of the item although the item may not be available from the user's viewpoint. It should be defined for each case which required resources are considered external and which are internal. This thesis uses a general definition: An item is available when it can perform as required. Tangible definitions should be made when the framework is applied in practice.

Like reliability, availability may be quantified using appropriate measures or indicators, and is then referred to as availability performance. As with reliability, complementary probability called *unavailability* is preferred for the quantified availability performance. *Instantaneous unavailability* is the probability that an item is not in a state to perform as required at a given instant [31, Sec. 192-08-04]. In addition, *mean unavailability* [31, Sec. 192-08-06] over the given time period is used to quantify the availability performance.

Maintainability is the ability of an item under given conditions of use, to be retained in, or restored to, a state in which it can perform a required function, when maintenance is performed under given conditions and using stated procedures and resources [32, Sec. 4.5]. Maintainability describes the ability to perform the CM and PM tasks, which affects the durations of the fault and service states.

In addition to the active repair time of the CM, the duration of the fault state can contain various delays. Similarly, the service state can contain delays in addition to the active PM task time. For example, logistic or technical delays, or the time needed for finding of the failure can affect the maintainability. *Maintenance supportability* is the ability of an organisation to have the correct maintenance support at the necessary place to perform the required CM or PM activity.

Like reliability and availability, maintainability may be quantified using appropriate measures or indicators and is then referred to as maintainability performance. The total CM and PM time can be quantified with a CDF. If needed, the functions can be defined separately for active maintenance time, and for each type of delay that is included. Mean Repair Time (MRT) [32, Sec. 11.4] and Mean Time To Restoration (MTTR) [32, Sec. 11.5] are examples of other ways to quantify the CM.

In this thesis, the maintenance supportability is handled like any other delay that could affect the durations of the fault and service states. The length of the delay can be quantified as detailed as necessary. Separate analysis should be made later about the principles or origins of a delay if it is found significant from the risk point of view.

IEC defines *dependability* as the ability to perform as and when required. It is used as a collective term for item's time-related quality characteristics, such as availability, reliability, maintainability and maintenance supportability [31, Sec. 192-01-22]. Figure 2.3 illustrates the dependencies of the dependability characteristics, as described by IEC [31, Sec. 192-01-23]. The up time [31, Sec. 192-02-02] is characterised by reliability, and the down time [31, Sec. 192-02-21] by maintainability and maintenance supportability.

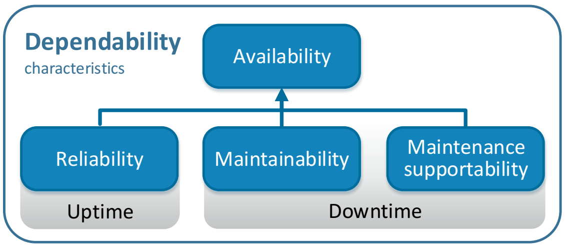

Figure 2.3 Dependability characteristics described by IEC [31, Sec. 192-01-23]

The definition of dependability mentions that the characteristics are time-related. This time does not necessarily refer to calendar time. The studies can also be made related to the operation time of the system if the operation is not made continuously. Additionally, number of cycles, number of kilometres, etc. can be considered instead of time unit. If other than calendar time is used, the durations of the CM and PM tasks needs to be defined carefully. Some kind of usage profile should be used to describe the relation between units used with operation and non-operation times.

The dependability requires avoiding the situations when the items are not able to perform as required. These situations can be *risk sources*, which alone or in combination has the intrinsic potential to give rise to risk [28, Sec. 3.5.1.2]. The consequences of these dependability related risk sources are, for example, break and downtime costs of the down state, CM material and resource costs of the fault state, and PM costs of the service state.

In addition to dependability that includes Reliability, Availability and Maintainability (RAM), also *Safety* is included in Reliability, Availability, Maintainability and Safety (RAMS). Safety is related to special risk sources called hazards. *Hazard* [28, Sec. 3.5.1.4] is a source of potential *harm* [33, Sec. 3.1], which is injury or damage to the health of people, or damage to property or the environment. Harms are consequences of safety related risk sources. The basis for the both dependability and safety related risk sources comes from the states of the items.

The risks of a system can be divided to availability and safety risks. The combination of likelihoods and consequences of dependability related risk sources form *availability risks* of the system. Similarly, the combination of likelihoods and consequences of hazards form *safety risks*. The terms of the risks and RAMS, as described in standards [28, 31, 32, 33], are illustrated in Figure 2.4.

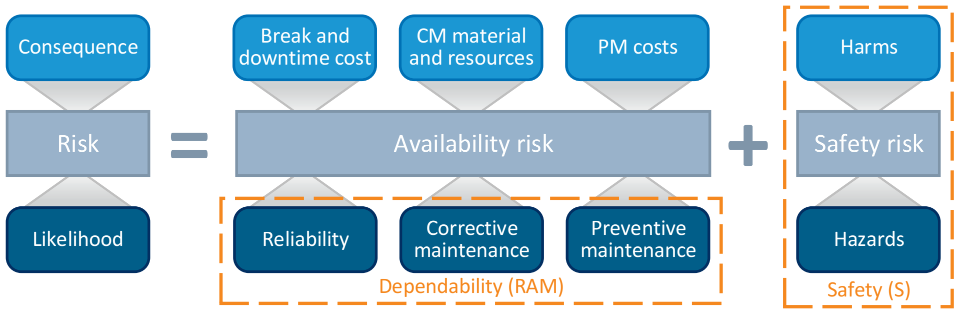

Figure 2.4 The terms of risk and RAMS, as described in standards [28, 31, 32, 33]

The acceptable risk depends on the considered situation. Usually the assessment of risk is made for more than one scenarios, which allows prioritising different design alternatives. As an example, the reduced risk can be used to justify the cost of an investment.

2.2 Traditional techniques for systems' risk assessment

This section briefly describes traditional techniques that are commonly used for quantitative risk assessment. Estimating the probabilities of events or failure rates of components is required in order to obtain quantitative results. The estimation requires experts' judgements or history data. If this information is not available, the same methods and formalisms are usually suitable for qualitative analyses.

The traditional techniques are not always flexible enough for obtaining quantitative results in complex situations. For example, a simple technique can be limited to analyse only non-repairable systems. Extensions and new techniques can avoid the limitations by increasing the modelling power. The possibility to reduce the manual modelling work has also been a motivation for improvements. Various modelling targets with different needs have lead to a large variety of different modelling techniques for systems' risk assessment.

Fault Tree Analysis (FTA) [2] is one of the most prominent techniques for quantitative risk assessment. A fault tree is an organised graphical representation of the factors contributing to the occurrence of a defined outcome, referred to as the *top event*. Deductive (top-down) approach aims finding the combinations of conditions that can cause the defined top event. Figure 2.5 illustrates a fault tree model structure, where the top event is a system failure. Component faults are *primary events*, which are at the bottom of the fault tree. *Intermediate events* are between the top event and the primary events. A *gate* defines the causal relation between an output event and the contributing input events. For example, a gate can have a logic rule OR, AND, Vote or XOR. Traditionally FTA uses special symbols for different type of elements, but they are not strictly followed in this thesis.

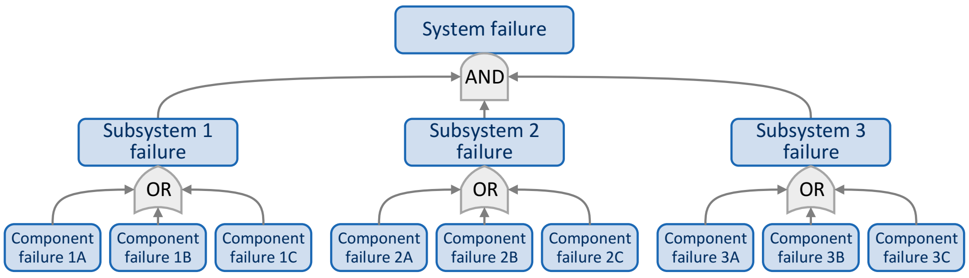

Figure 2.5 A fault tree model structure

Similar analyses can be made also with Reliability Block Diagram (RBD) [5], which is made up of *blocks* that are connected in series or parallel configuration. Any block failure along a series path causes the entire path to fail. Parallel paths are redundant, which means that all blocks must fail. RBD models are formed by combining series and parallel configurations. Figure 2.6 illustrates a RBD model structure, where the system consist of three redundant subsystems, which each have three components. RBD can include also k-out-of-n configurations, where at least k out of n items must operate for the redundant system to operate.

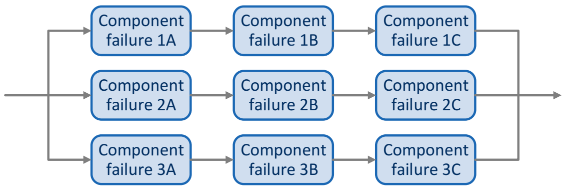

Figure 2.6 A reliability block diagram model structure

A fault tree can be converted to a success tree by applying the two rules of de Morgans's theorem[1]. A success tree can be then converted to a RBD by replacing AND gates with series paths, and OR gates with parallel paths. Similar conversion can also be made from a RBD to a fault tree. Possibility to convert the models to both directions implies that the techniques have the same modelling power.

The modelling power of the FTA formalism can be enhanced by allowing repeated events. For example, the component C of the model that was illustrated in Figure 2.5 can be shared between sub systems 1 and 2. The failure of the component causes both sub systems to fail. If repeated events are not allowed, FTA is not capable for modelling the dependability of such system. Figure 2.7 illustrates a fault tree model structure, where the failure of the shared component C occurs in two locations. An analogous RBD with repeated blocks is shown in Figure 2.8. With repeats allowed, FTA and RBD have the same modelling power [34].

Both FTA and RBD are combinatorial modelling techniques, which map the operational dependency of a system on its components. Even with repeated events, they are not suitable for modelling of certain kind of dependencies, such as shared repair facilities, which are enabled with state-space modelling techniques [35].

[1]`not(A or B) = not(A) and not(B)`
`not(A and B) = not(A) or not(B)`

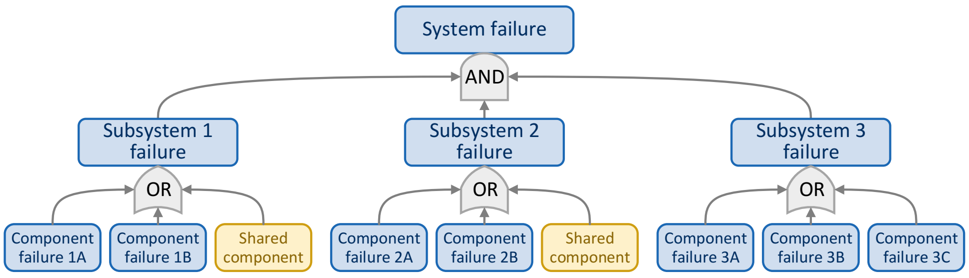

Figure 2.7 A fault tree model structure with a repeated event

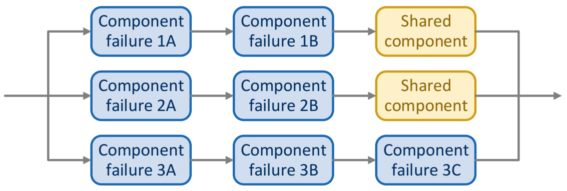

Figure 2.8 A reliability block diagram model structure with a repeated block

Markov analysis [6] is a simple state-space modelling technique. It can be used if the future state of a system depends only upon its present state. This rule is called a Markov property. For example, in a system with states functioning, degraded, and failed, the Markov property is valid if the next state depends only upon the current state, and not from the states before it. A Markov model consist of *states* and *transitions*, which define the state changes. Table 2.1 gives an example of a transition matrix, which represents a system by defining the probabilities of its state changes [4, Tab. B.2].

Table 2.1 A transition matrix that represents state changes of the example system [4, Tab. B.2]

		Current state		
		FUNCTIONING	DEGRADED	FAILED
Next state	FUNCTIONING	0.95	0.3	0.2
	DEGRADED	0.04	0.65	0.6
	FAILED	0.01	0.05	0.2

In addition to a transition matrix, a Markov diagram can be used to represent a system. Figure 2.9 illustrates the states and transitions that was defined in Table 2.1 [4, Fig. B.9]. The arrows to a state to itself are not shown. The probability of such arrow can be calculated based on other arrows, because the sum of probabilities of all transitions of a state must be always 1. This thesis does not strictly follow the symbols that are commonly used in Markov diagrams.

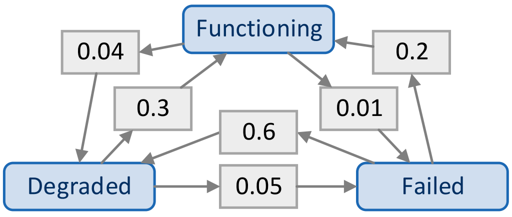

Figure 2.9 A Markov diagram that represents state changes of the example system [4, Fig. B.9]

A Markov chain is the stochastic model that is defined by a transition matrix or a Markov diagram. If exponentially distributed delays are defined for transitions, the model is called Continuous-Time Markov Chain (CTMC) [36]. Also the name Markov process is used. CTMC uses rates of state changes in the transition matrix to define the model.

Semi-Markov Process (SMP) generalises CTMC by removing the restriction that all transition durations must be exponentially distributed. Unlike CTMCs, SMPs have the Markov property only at certain time points. For example, failures and restorations of a system can be such time points, and the jumps between them are defined by using arbitrary distribution functions. The duration of each jump must depend only on the two states between which the move is being made. Markov Renewal Process (MRP) is equivalent to SMP, but their definition is different [37].

Another state-space modelling technique is Petri net [8], which is a graphical tool with an exact mathematical definition. Unlike Markov models, Petri nets can have multiple active elements at the same time. The model consist of *places* and *transitions*, which are connected by directed *arcs*. The state of the model at certain point in time is defined by dynamic *tokens*, which move between places when the transitions are fired. Petri nets have various application areas, but the formalism can also be considered as a traditional risk assessment technique.

Figure 2.10 illustrates Petri nets that define OR and AND logic rules for two input places A and B. Arcs connect the inputs through transitions to an output place. A transition is enabled, when each of its input places contains at least one token. When a discrete time step is taken, each enabled transition fire by removing a token from each input place and placing a token in an output place. If there are more than one outputs, they each will have a token. The tokens define the state of the model. Traditionally Petri nets use special symbols for different type of elements, but they are not strictly followed in this thesis.

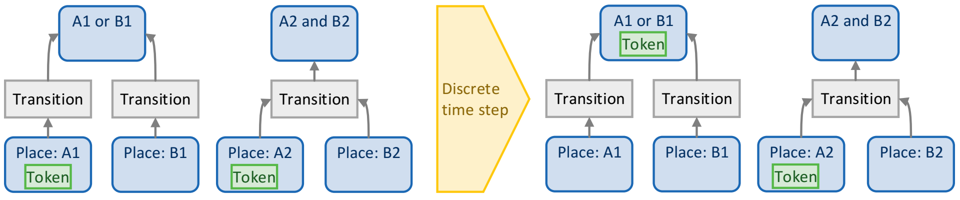

Figure 2.10 Petri nets for OR and AND logic rules

The discrete time Petri nets can be used, for example, to investigate the deadlocks of a computer system. Stochastic Petri Net (SPN) includes exponentially distributed firing times for transitions. The extension increases its modelling power to the same as CTMC [38]. Also various other extensions exist. Generalised Stochastic Petri Net (GSPN) [39] contains both immediate and exponentially distributed transitions. GSPN does not have more modelling power than SPN, but the presence of immediate transitions allows a higher parametrisation degree [40]. Coloured Petri Net (CPN) [41] belongs to high-level Petri nets [42], which are characterised by the combination of Petri Nets and programming languages. The extension focuses on the practical use of the formalism instead of increasing its modelling power. Extended Stochastic Petri Net (ESPN) uses arbitrary distribution functions for transitions, which increases the modelling power [43].

Markov analysis and Petri nets are more powerful than FTA and RBD, but for certain needs the use of state-space modelling techniques can be unnecessarily complex. The most suitable technique depends on the studied case. Combination of different techniques allows using the most practical technique for each part of the model. For example, the blocks of a RBD can be modules that are modelled by using a fault tree or a Petri net. The sub models define first the failure rates of the modules, which are then used to evaluate the RBD. RBD driven Petri net [44] and a Conjoint System Model (CSM) [45] are examples of hybrid techniques that combine Petri nets and RBD.

In a basic situation the gates in FTA can model only static relations. By adding *dynamic gates* FTA can be extended to handle dynamic relations. Dynamic gates, such as Priority AND, sequential gate and spare, model situations where the order of the events matters [2, Sec. 5.4.3]. Their evaluation can be made with the Markov analysis. Once evaluated, the gate and its inputs may be replaced by a single primary event of a fault tree. Various extensions exist for FTA also to handle e.g. (i) uncertain probabilities, (ii) dependency between events, (iii) temporal requirements, (iv) repairable components, (v) inclusion of more than two states for components, and (vi) different distribution types for failure rate [3].

Several software tools have been created to enable efficient use of different techniques. For example, CPN Tools [46] is for editing and simulation of CPN models, GRIF BStoK [47] is for RBD driven Petri nets, and REALIST [48] for CSM. Various software tools also exist for different FTA extensions [49]. For example, ELMAS [15] is a tool for an advanced FTA modelling technique [19], which formed the basis of the AoT framework. The advanced FTA includes various extensions in the traditional FTA, and permits including user-defined procedure codes to support modelling of domain-specific features. This enables, for example, (i) combining FTA and FMEA analysis [20], (ii) multi-state modelling of partial process flows, (iii) including dynamic rules for backup power supply [21], and (iv) defining exclusive stochastic consequences [22].

In addition to the previously described techniques FTA, RBD and Markov analysis, the standard IEC/ISO 31010:2009 [4] refers to various other risk assessment techniques that have also reached a satisfactory level of professional consensus. For example, a widely used qualitative technique Failure Modes and Effects Analysis (FMEA) [7] and its extension Failure Modes and Effects and Criticality Analysis (FMECA) are mentioned. FMECA can be quantitative if suitable failure rate data and consequences are used for describing the occurrence and effect of failures. PSK 6800 [50] is an example of similar qualitative technique for classification of criticality. The inductive FMEA can be used together with the deductive FTA for having a more comprehensive approach.

Another inductive technique is Event Tree Analysis (ETA) [4, Sec. B.15], which displays potential scenarios following certain initiating event. Event trees represent graphically sequences of events, which are not possible to represent when using fault trees. Cause-consequence analysis [4, Sec. B.16] is a combination of FTA and ETA. The technique analyses a critical event, which is equivalent to the top event of a fault tree and the initiating event of an event tree.

In addition, recent research has proposed various new techniques that satisfy certain special needs of quantitative risk assessment. For example, Functional-Failure Identification and Propagation (FFIP) [51] is applied at early conceptual design phase before a top event of FTA and the mechanisms leading to it are known. Dynamic Flowgraph Methodology (DFM) [52] produces a self-contained system model from which many fault trees can be derived via algorithmic procedures. Software Reliability Growth Model (SRGM) [53] enables estimating the decrease of software failures during the testing phase. Translation of FTA and RBD models to a Bayesian network [54] allows the inclusion of new features, such as probabilistic gates, multi-state variables, uncertainty on model parameters, and dependence between components, in the model.

2.3 Model-based dependability assessment

Over the past twenty years, researchers have made continuous efforts to simplify the analysis process by automatically synthesising dependability related data from system models [49]. This has led to the emergence of the field of Model-Based Dependability Assessment (MBDA), which is also referred to as Model-Based Safety Assessment (MBSA) [55]. While certain techniques focus on making the analysis process more manageable, other MBDA techniques have been developed to address the limitations of traditional approaches [56]. The field of MBDA encompasses a large variety of techniques, such as FPTN [57], FPTC [58], HiP-HOPS [59], SAML [60], smartIflow [61], AltaRica [62], and Figaro [63]. This section presents a brief review of different MBDA techniques and a comparison of their features.

Failure Propagation and Transformation Notation (FPTN) [57] is among the pioneering MBDA methods. It was designed to address limitations and improve the integration of FTA and FMEA. FPTN proposes a systematic model of failure propagation, which resembles data-flow based methods. The model consists of boxes with input and output failure modes, and a set of predicates describing the relationship between them. Fault trees can be created for each output failure mode based on the information of the boxes.

Fault Propagation and Transformation Calculus (FPTC) [58] attempts to overcome deficiencies in FPTN by using an actual architectural model of the system. The FPTN diagram includes only the parts that are currently known to cause failures. In FPTC the changes to the architectural model do not require a new failure model to be built, which improves the synchronisation of the model and reality. FPTN was designed to support both inductive and deductive analysis, but FPTC is primarily inductive in nature.

Another technique for the synthesis of fault trees and FMEAs is Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) [59]. Unlike FPTC, HiP-HOPS is a deductive technique. Thus, it is less prone to combinatorial explosion [64]. A model starts at functional level and proceeds all the way down to the low levels of the hardware and software implementation. HiP-HOPS has interfaces to a number of different modelling tools, including Matlab Simulink, Eclipse-based UML tools, and SimulationX.

The general underlying formalism and the types of analyses performed typically gravitate MBDA techniques towards two leading paradigms [56]. In Failure Logic Synthesis and Analysis (FLSA) the fault tree or other failure model is automatically constructed from the information stored in the system model. The aforementioned FPTN, FPTC and HiP-HOPS all belong to this category. The other approach is Behavioural Fault Simulation (BFS), where faults are injected into the model that simulates the normal system behaviour.

Safety Analysis Modelling Language (SAML) [60] is a BFS technique. It utilises finite state automata to describe system models. A model consists of modules that have state variables and transition rules to update them. SAML can be used as an intermediate language for MBDA techniques, but there exists also Software-intensive Systems Specification Environment (S^3E) [65] for design and verification of SAML models.

State Machines for Automation of Reliability-related Tasks using Information FLOWs (smartIflow) [61] is a modelling language that has been specially designed for the purpose of automating the safety analysis process. In smartIflow components are considered as finite state machines. The information exchange between the components is undirected, which differs from all previously mentioned techniques. Using directed connections often leads to problems in failure situations where the cause-effect relationship reverses [66]. The direction of an undirected connection is determined automatically, which makes the definition of the direction unnecessary for the model creator. Another difference is that smartIflow uses object-oriented approach to characterise the component types by using model classes.

Another technique that uses undirected connections is AltaRica 3.0 [62]. It is a high-level modelling language, which is dedicated to safety analysis. In AltaRica the relations are defined in detail by creating connections between model element attributes. This is done by using transitions and assertions [67]. The earlier versions of the language are AltaRica LaBRI and AltaRica Data-Flow. Like smartIflow, they all use an object-oriented paradigm to organise large models into hierarchies. A library of reusable components can be created by defining classes. A class is a generic component, which can be instantiated in the model. AltaRica 3.0 enriches the model structure with blocks, which are similar to prototypes from prototype-based [68] programming languages. A block is like a class that is automatically instantiated as an unique occurrence in the model.

The underlying mathematical formalism of AltaRica 3.0 is Guarded Transitions Systems (GTS) [69]. It is a state automata, which generalises RBDs, Markov chains and SPNs. GTS can represent instant loops and acausal components. The direction of the flow propagation is determined at run time. AltaRica 3.0 compiler translates models into GTS by flattening the structure of classes and blocks. Also Dynamic Fault Tree (DFT) can be translated into GTS [70], and automatic conversion from SAML models has been planned [71]. A fault tree compiler, a stochastic simulator, and various other assessment tools exists for GTS [72]. Both AltaRica 3.0 and GTS have a strictly defined grammar [67].

Unlike smartIflow and various other techniques, AltaRica is a standalone technique without direct connections to design models. Creation of a dedicated dependability model requires more work, but it allows using the optimal level of abstraction and inclusion of only the needed details from reliability and risk analysis point of views. Another standalone technique is Figaro [63], which is a reasoning system that is based on a Scala programming language. The models can include Scala and also Java programs. Figaro is a description language of KB3 [73] workbench, which enables automated dependability analysis of complex systems.

Table 2.2 summarises the classification of the referred MBDA techniques by using four previously mentioned disquisition criteria. The table includes also a characterisation of the approach that is presented in this thesis. The AoT framework uses standalone models, which are dedicated to risk and performance analysis. BFS can be made by using AoT models. Basic modelling techniques, such as FTA and Markov analysis, use basic directed relations in AoT, but advanced relation definition by creating connections between model element attributes is also enabled. The basic directed relations can be seen as shortcuts that encapsulate certain more detailed connection definitions. The object-oriented paradigm is used as a basis for AoT models.

Table 2.2 Model-Based Dependability Assessment (MBDA) formalisms

	Use of design models	Underlying formalism	Relation modelling	Object-oriented
FPTN	standalone	FLSA	basic	no
FPTC	yes	FLSA	basic	no
HiP-HOPS	yes	FLSA	basic	no
SAML	yes	BFS	basic	no
smartIflow	yes	BFS	advanced	yes
AltaRica	standalone	BFS	advanced	yes
Figaro	standalone	BFS	both	yes
AoT	standalone	BFS	both	yes

Based on the four classification criteria, AoT shares the most similarities with Figaro. However, at least two clear differences can be recognised between AoT and other techniques. The first is the use of tabular model definition format, which is not used in any other approach. Tables was selected for AoT because an average modeller usually understands tabular format easier than any markup language, such as Extensible Markup Language (XML) [74]. Figaro uses a programming language that is based on Scala, which might create a high threshold for a modeller.

The other difference is the separation of the modelling technique definition from the model creation. AoT is a framework that includes several built-in techniques and permits their customisation. Each technique is defined with the same tabular format, which allows using built-in techniques as a basis for a new technique. Other approaches use only fixed techniques and possibly allow their combination.

The AoT framework improves the OpenMARS [25, 26] approach. In OpenMARS the model definition is divided to five different tables. Each table forms a clear modelling step, which helps basic users to understand the use of the format. Separate tables also simplify the definition of privileges, because edit of only a certain table can be allowed for a user. The AoT framework uses the Triplets data format, which emphasises less the features that simplify the manual model definition for a basic system engineer. The most important design principle of the Triplets format was efficiency. All definitions are made in a single table. This simplifies the storing of the model data into databases and the transferring of the models between different tools. When compared to OpenMARS, the Triplets format is more plain and compact, but still compatible with the original OpenMARS format. The definition of the Triplets format and the AoT framework follows as strictly as possible the terminology that is used with object-oriented paradigm and graph theory. These theoretically more suitable terms might not be the most familiar for a basic system engineer, which was considered more with the OpenMARS approach.

3 OBJECT-ORIENTED METHODOLOGY FOR RISK AND PERFORMANCE ASSESSMENT

This chapter specifies the methodology of the Analysis of Things (AoT) framework. All the definitions are made by using tabular Triplets data format. Section 3.1 defines the syntax of the format and explains how different type of definitions are made. Section 3.2 introduces the fundamental classes, which form the basis for all modelling techniques. The models are analysed with stochastic discrete event simulation, which principles are presented in Section 3.3. The next Chapter 4 gives examples on how to apply the methodology that is specified in this chapter.

3.1 Triplets data format for object-oriented model definition

The AoT models consist of objects. Each object is an instance of a *class*, which defines the type of the object. The modelling techniques are defined by declaring a catalogue of classes. A new class can be declared if a technique requires a new type of model objects for certain special need. Several models can use the same technique. Each model can contain any number of objects, which use all or some of the classes provided by the technique.

A class inherits all *attribute* declarations of its super class, and can extend them by declaring new attributes. An attribute is declared by defining its type and name. The type of an attribute is a class. For example, a class can declare an attribute, which type is `Number` and name is ”level”. An object assigns attribute values, which are objects. For example, an object can assign that the ”level” is 5. The model objects that can contain attributes are called ”elements”.

Triplets data format uses a single table for defining the modelling technique, the structure of model objects, and the parameter values. The plain and compact data format is designed to enable efficient manual model editing while permitting direct storing to databases. The simple example tables shown in this thesis can be easily manually edited. When the size of the model increases, the use of a GUI or a direct import from management systems or other databases is preferred. The Triplets data format open and non-proprietary, which allows using it with any model definition software or calculation tool.

The use of a single table does not exclude the possibility for dividing the model definition to separate modules. For example, all models that use the same modelling technique can include the same predefined module table. The inclusion is performed simply by including the rows of the module tables in the handled model table. This ensures efficient utilising of the previously implemented similar solutions. New modelling techniques and customised special features can be easily shared for public use with any platform-independent file or data format that supports storing tabular information.

3.1.1 Different definition rows

The Triplets data table has three columns. Because the same table is used for the entire model definition, the content of each table column slightly differs based on the made definition. Headings A, B and C are used for distinguishing the columns. Following list gives a general idea of the nature of each column:

- COLUMN A: The object of the definition
- COLUMN B: A keyword that indicates the type of the definition
- COLUMN C: The value defined for the object

In this thesis the definition tables have two additional columns. The first column shows the row number, which can be used as an index when certain definition needs to be referred from a table. The last column contains comments to explain the meaning of the definition. Table 3.1 illustrates how different type of Triplets data format definitions are presented in this thesis.

Following names are used for the different definition types:

Class declaration Keyword ”class” is used in column B to declare a new class. The name of the new class is defined in column A. The class inherits the features of a super class that is defined in column C. (see row 1 in Table 3.1)

Table 3.1 Examples to illustrate different type of definitions made with the Triplets format

	A	B	C	COMMENTS
1	ClassName	class	ClassSuper	Declaration of a class
2	ClassName	container	ClassMember	Declaration of a container
3	ClassName/attributeName	attribute	ClassAttr	Declaration of a class attribute
4	container/instanceName	instance	ClassType	Instance creation
5	element1/connectionName	connect	element2	Element connection
6	element/attributeName	=	value	Attribute value assignment
7	element/attributeName	add	value	Attribute value add to list

Container declaration Keyword ”container” is used in column B to declare a container class. Column A defines the class, which instances are containers of objects. The type of the contained objects is defined in column C. (row 2)

Attribute declaration Keyword ”attribute” is used in column B to declare an attribute for a class. Column A defines the name of the class and the name of the attribute. A slash (/) symbol is used as a delimiter. Column C defines the type of the declared attribute. (row 3)

Instance creation Keyword ”instance” is used in column B to create new model objects. Column A defines the container and the name of the new object. A slash (/) symbol is used as a delimiter. Column C defines the type of the created object. (row 4)

Connection adding Keyword ”connect” is used in column B to add a connection between model elements. Column A defines the connecting element and the name of the connection. A slash (/) symbol is used as a delimiter. Column C defines the connected element. (row 5)

Value assignment Equals sign ”=” is used in column B to assign a value to an element attribute. Column A defines the element and the name of the attribute. A slash (/) symbol is used as a delimiter. Column C defines the value that is assigned to the attribute. (row 6)

Value add Keyword ”add” is used in column B to add a value to an element attribute. Column A defines the element and the name of the attribute. A slash (/) symbol is used as a delimiter. Column C defines the value that is added to the attribute. (row 7)

3.1.2 Naming conventions, restrictions and an array definition

Rules 3.1 - 3.3 summarise the naming conventions that are used with the Triplets data format. The restrictions help avoiding potential problems caused by names containing special characters. The rules also help the error detection. Based on the name the modeller can immediately recognise whether the definition is for a class or an instance. The rules follow the Java naming conventions [75]. In addition, Rule 3.4 about duplicate definitions is added to help avoiding potential mistakes. An error message should be shown if the modeller unwittingly makes the same definition more than once.

Rule 3.1. The class, instance and attribute names shall only consist of a-z, A-Z, 0-9 and underscore (_) characters.

Rule 3.2. Each class name starts with an upper-case letter.

Rule 3.3. Each instance and attribute name starts with a lower-case letter.

Rule 3.4. It is not allowed to declare the same class, container nor attribute, create the same instance, nor assign a value to the same attribute more than once.

Rule 3.4 does not exclude the possibility to assign an array of values to an attribute instead of a single value. In Triplets data format all attributes can be used as arrays if more than one value needs to be defined for the same attribute. An array is actually an ordered map, which allows associating values to keys. An array definition is made by adding the key inside square brackets ”[]” as a suffix of the defined attribute name. This creates a syntax ”attribute[key]”. In addition to integer, the key can be also a string, which follows the same naming conventions as instance and attribute names (see Rules 3.1 and 3.3). The basic use of an attribute name without a key is considered as an array definition with an empty string as a key.

The key can be omitted if keyword ”add” is used instead equals sign in column B. The value add appends the value to the first available integer key. Usually in programming languages zero is the first index of an array, but in AoT the first integer key in value adding is ”1”. For example, if value add is used twice for ”attributenName”, the values are assigned to ”attributeName[1]” and ”attributeName[2]”. The keyword ”add” supports adding of more than one values in a single definition by using a comma (,) symbol as a delimiter. The ”add” keyword helps defining the model in situations where the number of the added values is not fixed.

Table 3.2 gives examples of erroneous and allowed assignments of more than one value for the same attribute.

Table 3.2 Definition of more than one value for the same attribute

	A	B	C	COMMENTS
1a	element/attributeName	=	val1	Duplicate assignment to attribute -> Error
1b	element/attributeName	=	val2	Duplicate assignment to attribute -> Error
2a	element/attributeName[x]	=	val1	Duplicate assignment with key "x" -> Error
2b	element/attributeName[x]	=	val2	Duplicate assignment with key "x" -> Error
3a	element/attributeName[one]	=	val1	Assign value *val1* to attribute with key "one" -> OK
3b	element/attributeName[two]	=	val2	Assign value *val2* to attribute with key "two" -> OK
4a	element/attributeName	add	val1	Append new attribute value *val1* -> OK
4b	element/attributeName	add	val2	Append new attribute value *val2* -> OK
4c	element/attributeName	add	val3,val4	Append new attribute values *val3* and *val4* -> OK

3.1.3 Referring to objects

All model objects have an Unique Identifier (UID). It is formed by combining the UID of the container and the name of the object. A slash (/) symbol is used in the UID as a delimiter between the container and the name. This recursive definition ends always to the base folder of the model (see Table 3.10 on page 51), which contains all model objects either directly or via some sub container. The UID of the base folder is an empty string, which leads to the UID format that starts always with a slash symbol. For example, the UID */folder/subfolder/element* is for an element in a "subfolder" of a "folder", which is located in the model base folder. The UIDs are formed similarly for attributes of elements.

The slash symbols split the UID to a list of names. The first name is always an empty string, which refers to the base folder of the model. After that, each name refers uniquely to exactly one attribute of the container object that was referred by the previous names. For example, an UID */folder/subfolder/element* splits to four names: "" (an empty string), "folder", "subfolder" and "element". To ensure that the name always refers to exactly one attribute, the name contains also the key of an array definition whenever needed.

A *path* is similar to UID, but it can refer to more than one object. The slash symbols split also the path to a list of names. If a name of a path omits the array key, it refers to all keys of the attribute. A path can also define a comma separated list or an interval of keys. Intervals are allowed for both integers and characters. Table 3.3 shows examples of using a path to refer multiple attributes in a value assignment.

Table 3.3 Referring to multiple attributes in a value assignment

	A	B	C	COMMENTS
1	element/attributeName[1, 2]	=	val	Assign to "attributeName[1]" and "attributeName[2]"
2	element/attributeName[one, two]	=	val	Assign to "attributeName[one]" and "attributeName[two]"
3	element/attributeName[1–4]	=	val	The same as the definition "attributeName[1, 2, 3, 4]"
4	element/attributeName[a–d]	=	val	The same as the definition "attributeName[a, b, c, d]"
5	element/attributeName	=	val	Assigns values to all keys of "attributeName"

In Triplets data format the container in an instance creation, and the element in a value assignment (see Table 3.1 on page 43) are paths or UIDs. An UID can be seen as a special case of a path that refers to exactly one object. If a path starts with a slash symbol, it is called an *absolute path*. Otherwise it is a *name path*, which first name refers to all objects that have the defined name. This allows making definitions only to an element in certain container or to all elements with the defined name. Table 3.4 gives examples of different path definitions.

Table 3.4 Examples of different path definitions

	A	B	C	COMMENTS
1	/folderName/elementName/attributeName	=	val	The folder *folderName* in the model base folder
2	folderName/elementName/attributeName	=	val	All *folderName* folders in any container
3	/elementName/attributeName	=	val	The element *elementName* in the base folder
4	elementName/attributeName	=	val	All *elementName* elements in any container

The number of objects that a path refers to depends on the previous definitions. This kind of dynamic definition helps making changes to the model. Changing of the number of similar elements that have the same name does not require changing the element definitions if they are made by using name paths. Table 3.5 gives an example of a model that contains various similar systems and items. The number of systems can be updated in row 1 without a need for updating the rows 2 and 3, which define the items of each system and the attribute values of the items. Similarly, if in row 2 the number of items in each system is changed, any changes are not needed to row 3.

Table 3.5 An assignment can be made irrespective of the number of the items

	A	B	C	COMMENTS
1	/system[1–3]	instance	Folder	One folder for each system
2	system/item[a, b]	instance	Element	Two items in each system
3	item/attributeName	=	val	The same value assigned for all items

3.1.4 Connections

The adding of connections by using the keyword ”connect” is similar to the value adding with the keyword ”add”. The connected element is added to the first available integer index of the connection attribute. However, the difference is that with connections the path definitions are used for both columns A and C. Handling of both path definitions independently would lead adding connections between all elements that are defined by both columns. Simultaneous handling of both name paths and array definitions enables more possibilities for defining the connections. This is especially useful in vast models that contain various similar structures.

Table 3.6 gives examples of different possibilities in connection adding. The connections are added between the components that were created in previous Table 3.5. The connection attribute name ”to” is used in each definition. The modelling techniques can define any name, such as ”source”, ”target”, ”child”, ”parent”, ”input” or ”output”, ”in” or ”out”, for the connection attribute, which allows creating several type of connections between elements.

Table 3.6 Examples to illustrate different type of connection definitions

	A	B	C	COMMENTS
1	/system[1]/item[a]/to	connect	/system[2]/item[b]	Exact definition of both elements
2	/system[1-3]/item[a]/to	connect	/system[1-3]/item[b]	From each *item[a]* to each *item[b]*
3	/system/item[a]/to	connect	/system/item[b]	Connection inside same system
4	item[a]/to	connect	item[b]	Connection inside all common containers
5	/system[1]/item[a]/to[X]	connect	/system[3]/item[b]	Key X distinguishes the connection

Following list describes the different type of connection definitions:

- In simplest case the connection is added between two elements, which are both defined by using an UID (see row 1 in Table 3.6).

- The intervals and lists are handled independently for both columns, which makes it possible to add all combinations of connections between elements (row 2).
- If an array keys are omitted, the connections are made only if the key is common for both elements (row 3).
- If only element names are used for both connected elements, the connection is added only if the two names are found in the same container (row 4).
- The key of an array definition can be used to distinguish different connections that are made with the same connection attribute (row 5). For example, with function models (see Section 4.4) the dividend can be distinguished from other connected operands of a division function by using a key of the connection attribute.

3.1.5 Prototypes and inclusion

Previous sections introduced how the name path can be used to refer all elements with a same name. This can be used to create several similar structures by making the definitions only once. However, the creation of the structure needs to be made after all the similar elements have been created. This order is not convenient when modelling techniques provide template structures. The definition of a technique needs to be before the model structure definition because the classes must be declared before the element creation. This means that also the template structures should be defined before the creation of the elements that use them.

A *prototype* is an answer to the need of defining template structures in a modelling technique definition. Prototypes are elements that have a class as their container. A modelling technique can create any number of prototypes for a class and also add connections between them. The instances of the class are containers of elements that are automatically created based on the prototypes. The connections are added to the created elements based on the connection definitions of the prototype. Also the connections from an attribute to a class prototype are allowed.

Table 3.7 gives an example of a template structure definition. The new declared `MyElement` class has two prototypes. A special keyword ”prototype” is used for the prototype definitions. A connection is added between the two prototypes. The class has also an attribute ”first”, which is defined to refer to the prototype ”tempY”.

The creation of an instance */myObject* creates automatically also */myObject/tempX* and */myObject/tempY*. Based on the defined template structure, both attributes */myObject/tempX/to* and */myObject/first* refer to */myObject/tempY*.

Table 3.7 An example to illustrate the prototype definition

	A	B	C	COMMENTS
1	MyElement	class	Element	Declaration of a special element class
2	MyElement/tempX	prototype	Element	A template element for the class
3	MyElement/tempY	prototype	Element	A template element for the class
4	MyElement/tempX/to	connect	MyElement/tempY	A connection between template elements
5	MyElement/first	attribute	Element	An attribute for the class
6	MyElement/first	connect	MyElement/tempY	Connect a template element to attribute
7	/myObject	instance	MyElement	Create an instance of the class

If the name of a prototype is used when new instances are created for the container element, the connections defined for the prototype are added also to the created element. The array definition is used to give a different key for the created element. For example, Table 3.8 creates an instance */myObject/tempX[key]* by using the name "tempX", which was in previous Table 3.7 used as a prototype name. In this case a "to" connection to the element */myObject/tempY* is created automatically also for the instance */myObject/tempX[key]*. This feature is needed, for example, when creating mode-dependent events (see Section 4.2.4).

Table 3.8 Using an array definition with the name of a prototype

	A	B	C	COMMENTS
8	/myObject/tempX[key]	instance	Element	A new element has the same name with a prototype

Another special keyword is "include", which permits inclusion of attributes of a class for an object after the creation of the instance. The need of the inclusion is related to the use of template structures that are defined by prototypes. In basic instance creation, the correct class can be defined directly when the object is created. The creation of instances automatically based on prototypes does not have similar possibility, which means that the proper class needs to be defined after the element creation. This feature is needed, for example, when defining the delay distribution of an event after the creation of the state-event template structure (see Section 4.1).

To avoid the problems of multiple inheritance [76], a Rule 3.5 is used. The rule causes that the inclusion can't be used to combine two arbitrary classes. It can be used only to extend the features of an element.

Rule 3.5. An object can include only sub classes of its own class.

The inclusion is allowed also for prototypes, which can be seen as default inclusion for all the automatically created elements. Because of Rule 3.5, the use of the included class directly when the prototype is created would restrict the possibilities to include classes to the objects, which are automatically created based on the prototype.

3.2 Fundamental classes of the AoT framework

The AoT framework contains a special `ModelObject` class, which is a built-in root of the class hierarchy. All other classes have exactly one super class. A `Primitive` is a class, which instances store only a single value, such as a string or a number. Primitives do not contain any attributes. Other objects are instances of an `Element` class. A `Vertex` and an `Edge` classes specify two main categories of elements. The model structures that are formed by interconnected vertices and edges are similar to graphs in graph theory [77, 78]. Different modelling techniques declare their own `Vertex` and `Edge` sub classes. AoT models can contain also special elements, which are instances of `Tool` or `Folder` classes. Table 3.9 shows the declaration of these fundamental classes. They form the basis for all modelling techniques of the AoT framework. Figure 3.1 illustrates the class hierarchy that is formed by the fundamental model object types.

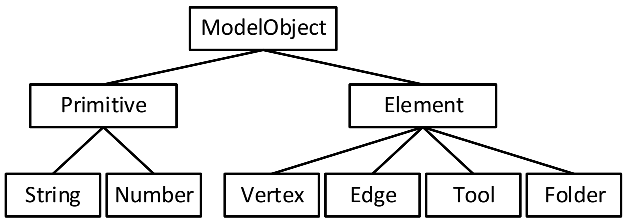

Figure 3.1 The class hierarchy that is formed by the fundamental model object types

Table 3.9 The declaration of the fundamental model object classes

	A	B	C	COMMENTS
1	Primitive	class	ModelObject	Does not contain attributes
2	String	class	Primitive	Unformatted text (a sequence of characters shown in a single line)
3	Number	class	Primitive	Any number
4	Element	class	ModelObject	Contains attributes, which are either elements or primitives
5	Vertex	class	Element	An element that is related to modelling of a state
6	Edge	class	Element	An element that models the relationships between states
7	Tool	class	Element	An element that handles or analyses the models
8	Folder	class	Element	A container, which can group elements

In addition to strict attribute declarations with a name and a type, AoT permits also declaring a class as a container of certain type of objects. For example, `Folder` is defined as a container of `Element` instances. Folders can contain any number of elements, which in AoT are seen as attributes of the folder. The name of the contained element is also the name of the attribute the folder has. The contained elements are called *members* of the container to distinguish them from elements that are attributes with a predefined name. All model objects are created as members or assigned to predefined attributes of a container.

A hierarchy can be formed based on the containers of the model objects. This structure helps handling of large models. The root of the container hierarchy is a *base folder* of the model. It is a special built-in container of independent model objects, which do not belong to any other container of the model. Table 3.10 shows the creation of a model base folder instance, which is used as a default container of otherwise independent model objects. It is also possible to use, for example, a `Node` instance as a default container. This might be useful with simple models that contain only `State` and `Event` elements.

Table 3.10 Creation of the default base folder

	A	B	C	COMMENTS
1	/	instance	Folder	Creation of the default base folder

Each element can contain other elements and primitives, which are the leafs of the tree formed by the container hierarchy. The structure enables grouping of the model elements, which helps handling of large models. For example, certain folder can contain all elements of a sub model. Figure 3.2 illustrates the container hierarchy of a model.

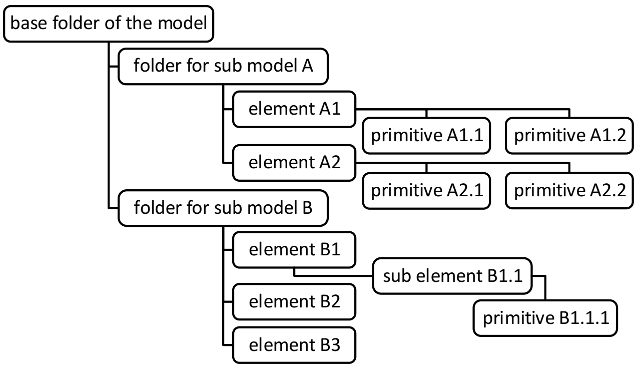

Figure 3.2 An example of a container hierarchy of model objects

Tools are special objects, which store the attributes needed for handling of the models. For example, a simulator tool (see Chapter 3.3) needs to store the number of simulated rounds and the simulation time period. These attributes are stored to a separate tool instance, because they do not belong to any element of the model structure. In addition to analysis of models, tools can store the attributes related to the visual presentation of the model. For example, a software that is used for GUI of model creation and edit can include a tool element in the model. The attributes that are stored to the GUI element can be for example the used view, zoom or window location. Also settings related to reports might need storing of attribute values.

3.2.1 Primitives store the parameter values

Primitive classes are used for storing basic string and number values as element attributes. Table 3.11 presents the declaration of the `String` sub classes. Each class restricts the format that is allowed for the values. For example, all strings are not proper Uniform Resource Locators (URLs) [79]. The most strict possible definition for an attribute type helps the use of the model because the user can understand the required format, and errors messages can be shown directly if the given value does not have a proper format.

Table 3.11 Primitive classes for storing string values

	A	B	C	COMMENTS
1	Text	class	String	Formatted text (possibly contains line breaks etc.)
2	Name	class	String	A string with only characters a-z, A-Z, 0-9 and _ allowed
3	Path	class	String	A string, which is proper path to refer model objects
4	URL	class	String	Reference to external location
5	Boolean	class	String	Either true or false

Table 3.12 presents the declaration of `Number` sub classes. Sometimes only an integer is allowed or the proper values must be between certain interval. More `Number` sub classes can be declared to make the needed restrictions for the input values. `Duration` and `Rate` are examples of classes that extend the available number value input format.

Table 3.12 Primitive classes for storing number values

	A	B	C	COMMENTS
1	Integer	class	Number	A whole number without fractional component
2	Probability	class	Number	Allows number from closed interval between 0 and 1)
3	Duration	class	Number	Time interval given with value and time unit
4	Rate	class	Duration	Value per time unit

The `Duration` class allows using time units in number values, which simplifies the assignment of durations. Table 3.13 lists the time units and the corresponding factors, which are used to convert a duration to a number. The factor shows the units magnitude compared to 1 hour, which is the base unit in this framework. The definition of the duration is also possible by using multiple value-unit pairs. For example, a value *3d 5h 15m* is converted to number *77.25*. As a special definition, an empty string is converted to infinite duration, which gives a simple way to define that something never happens.

The time unit of `Rate` values is defined with a slash (/) character as a prefix for the time unit. The base unit for rates is *1/h*. For example, a rate value *3/d* is converted to number *0.125*. If needed, it is possible to declare new classes that use other base unit than hour.

Table 3.13 The time units and the corresponding factors

Unit	Name	Factor
s	second	1/3600
m	minute	1/60
h	hour	1
d	day	24
a	year, annus	8766

Unlike elements, the primitives do not contain their own attributes. There exists one exception, which enables straightforward reading of values from external locations. Table 3.14 shows how attributes for URL definition are introduced for primitives. In addition to the ”url” attribute, the ”urlQuery” can be used to define components of the URL [79].

Table 3.14 Attributes for reading attribute values from external locations

	A	B	C	COMMENTS
1	Primitive/url	attribute	URL	The URL of the external location which contains the value
2	Primitive/urlQuery	attribute	String	The definition of URL query string

Table 3.15 shows examples how the reading of attribute values from external locations can be defined. The value for the attribute ”url” can be directly an URL of a file that contains the assigned value. The ”urlQuery” attribute enables including more information how the value is found from the read location. For example, if one Excel file contains values for several data, the file can be defined for all instances by using class default value definition. After that, each instance of the class needs only to use URL query for the definition of the cell that contains the value that belongs to the instance. Naturally it is also possible to define the information of the URL query directly to the ”url” attribute.

3.2.2 Vertices and edges form the model structure

The model structures are formed by vertices and edges, which have connections. Usually each connection has a direction, but some modelling techniques can use

Table 3.15 Examples of reading attribute values from external locations

	A	B	C	COMMENTS
1	objA/attrName/url	=	http://.../data.txt	Read value from a file
2	ClassB/attrName/url	=	http://.../data.xlsx	Define a file for all instances
3	objB/attrName/urlQuery[cell]	=	B2	Instance defines only a cell
4	objC/attrName/url	=	http://.../data.xlsx?cell=C2	Define also a query in URL

also undirected connections. The difference between `Vertex` and `Edge` objects is that vertices model certain system state, and edges the relationships between them. Different modelling techniques, which can use binary, discrete, or continuous states, follow this general categorisation when they declare the model element classes.

This thesis uses simplified visualisation of model structures. Rounded blue rectangles are used for vertices and grey rectangles (without rounded corners) for edges. Arrows are used to visualise the connections between elements. The use of constant symbols emphasises the similarities between structures of different modelling techniques. The use of only rectangular shapes maximises the space for adding of descriptive text inside the elements. Figure 3.3 illustrates the used symbols.

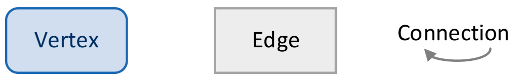

Figure 3.3 Symbols that are used in this thesis to visualise model structures

Table 3.16 declares the four main `Vertex` classes, which each have a corresponding `Edge` class. These elements form the basis for the application of the AoT framework with different modelling techniques (see Chapter 4).

The `Event` class defines the simplest edges of the AoT models. An event has a delay of its activation, which can be stochastic. By default the delay is positive infinity, which means that the event never activates. The sub classes of `Event` can define any distribution function for the delay. Table 3.17 shows the declaration of the ”delay” attribute. Similar delay can be included in `Operator` and `Transition` edges. The `Function` edge does not include delay, which means that all calculations are made immediately.

Table 3.16 The declaration of the four vertex classes and the corresponding edges

	A	B	C	COMMENTS
1	State	class	Vertex	Models a particular set of circumstances
2	Node	class	Vertex	Models an item with finite number of distinct states
3	Place	class	Vertex	State is defined by an integer (number of tokens)
4	Variable	class	Vertex	State is defined by a numeric value
5	Event	class	Edge	Changes an active state
6	Operator	class	Edge	Makes updates to nodes
7	Transition	class	Edge	Makes updates to places
8	Function	class	Edge	Makes updates to variables

Table 3.17 The declaration of basic attributes of Event class

	A	B	C	COMMENTS
1	Event/delay	attribute	Duration	A delay before an event is activated
2	Event/delay	=		The default the event never activates

Each `State` vertex models a circumstance of the studied subject, which is either active (/true) or not. A `Node` vertex combines predefined number of states to model different circumstances related to an item. In addition to states, the nodes contain also events. `Place` and `Variable` classes are needed when the number of distinct states is not restricted. A place has an integer, which models the state. With variables the number of possible states in uncountable. The state is modelled by using a number value. Table 3.18 shows the declaration of attributes that are used for modelling of states.

Table 3.18 The declaration of basic vertex attributes

	A	B	C	COMMENTS
1	Node	container	State	Nodes contain predefined states
2	Node	container	Event	Nodes contain events to model state changes
3	Node/initialState	attribute	State	Reference to an initial state
4	Place/tokens	attribute	Integer	An integer to model the state of a place
5	Place/initialTokens	attribute	Integer	Initial state of a phase
6	Variable/value	attribute	Number	A number to model the state of a variable
7	Variable/initialValue	attribute	Number	Initial state of a variable

To avoid confusions, this thesis uses dedicated terms for connections of different type of model structures. An `Event` edge models a change of a state between *source* and *target* states. Similarly, an `Operator` edge models directed connections between *child* and *parent* nodes, an `Transition` edge between *input* and *output* places, and `Function` edge between *in* and *out* variables. Table 3.19 shows the declaration of attributes of edges that are used to create connections between vertices.

Table 3.19 The declaration of connection attributes

	A	B	C	COMMENTS
1	Event/source	attribute	State	Event connection from a source state
2	Event/target	attribute	State	Event connection to a target state
3	Operator/child	attribute	Node	Operator connection from a child node
4	Operator/parent	attribute	Node	Operator connection to a target node
5	Transition/input	attribute	Place	Transition connection from an input place
6	Transition/output	attribute	Place	Transition connection to an output place
7	Function/in	attribute	Variable	Function connection from an in variable
8	Function/out	attribute	Variable	Function connection to an out variable

AoT includes a concept of symmetry, which gives more possibilities to model structure creation. In previous table the connection attributes were defined for `Edge` classes, but sometimes it would be more natural to define to a vertex that it is connected to an edge. The symmetry enables that when a connection is defined to either one of the connected elements, they both will automatically have a value in a connection attribute. Table 3.20 shows the symmetry definitions for the previously defined edge connections. The name of the connection attribute is given as a key of the "symmetry" assignment and the given value defines the name of the attribute that is used when the connection is added also to the connected element. The symmetry definition declares automatically the needed connection attribute for the connected class. If symmetry is defined for an undirected connection, both the attribute key and the given value have the same connection attribute name.

As an example, if an user wants to connect an event to a state, the symmetry definition enables that either a target connection to the event, or a source connection to the state can be added. Table 3.21 illustrates the situation by first creating the state and element instances, and then by showing the two alternative connection definitions in rows 3a and 3b. The symmetry of "source" and "target" connections ensures that the both definitions create the same relation between the two elements.

Table 3.20 Symmetry definitions for the connection attributes

	A	B	C	Comments
1	Edge/symmetry	attribute	name	An attribute for symmetry definitions
2	Event/symmetry[source]	=	target	If X is a source of Y then Y is a target of X
3	Event/symmetry[target]	=	source	If X is a target of Y then Y is a source of X
4	Operator/symmetry[child]	=	parent	If X is a child of Y then Y is a parent of X
5	Operator/symmetry[parent]	=	child	If X is a parent of Y then Y is a child of X
6	Transition/symmetry[input]	=	output	If X is an input for Y then Y is an output for X
7	Transition/symmetry[output]	=	input	If X is an output for Y then Y is an input for X
8	Function/symmetry[in]	=	out	If X is in to Y then Y is out from X
9	Function/symmetry[out]	=	in	If X is out from Y then Y is in to X

Table 3.21 An example of symmetry definitions

	A	B	C	Comments
1	/eventX	instance	Event	Create an event instance
2	/stateY	instance	State	Create a state instance
3a	/eventX/target	connect	/stateY	Basic connection adding
3b	/stateY/source	connect	/eventX	An alternative way enabled by symmetry definition

In addition to direct connections between elements, the AoT framework includes *modes* and *messages* to enable interactions between distinct elements. Each message and mode is identified with a name, which creates a communication interface. This is especially suitable for a composite model that is comprised of interconnected sub models. The communication interface obviates the need of knowing the exact UIDs of elements when connections between sub models are added.

The messages are defined to `Event` elements. A message is sent when the event activates. Similarly, the modes are defined to `State` elements. The mode is started when the state activates and ended when the state activation ends. As defined by Rule 3.6, messages are created when a mode starts and ends. All elements can listen the messages. For example, an event with a listener activates if a message with a proper name is sent. Elements can listen also mode changes, which means that the elements are noticed when certain mode starts or ends. Table 3.22 presents the declaration of the communication interface attributes.

Table 3.22 The declaration of communication interface attributes

	A	B	C	COMMENTS
1	Event/message	attribute	Name	Messages are sent when the event activates
2	State/mode	attribute	Name	Modes are started and ended based on the state activations
3	Element/listener	attribute	Name	All elements can listen messages and mode changes

Rule 3.6. Each mode start creates a message with a suffix ”Start”, and each mode end a message with a suffix ”End”.

As defined by Rule 3.7, elements can listen the complement mode, which means that they are noticed by the starts and ends of modes in the opposite order. This can be useful if there exists more than two states but only two modes (see Table 5.12 on page 135).

Rule 3.7. Each mode has a complement mode, which is denoted with a suffix ”Not”. The complement mode is started when the original mode ends, and ended when the original mode starts.

As an example, Table 3.23 defines elements from two distinct models. An event of model 1 sends messages and a state has a mode. The events from model 2 listen the messages and mode changes. A connection is created between elements of distinct models without needing to know the actual UIDs of the elements. Figure 3.4 illustrates the visualisation of modes and messages that is used in this thesis.

Table 3.23 An example of using modes and messages

	A	B	C	COMMENTS
1	model1/eventA/message	=	messageName	An event of model 1 sends messages
2	model1/stateB/mode	=	modeName	A state of model 1 has a mode
3	model2/eventX/listener	=	messageName	An event of model 2 listens messages
4	model2/eventY/listener	=	modeNameStart	An event of model 2 listens mode start

3.2.3 Reading the object-oriented model from Triplets data format

Table 3.1 on page 43 illustrated seven different definition types for Triplets data format. The Section 3.1.5 introduced the special ”prototype” and ”include” definitions.

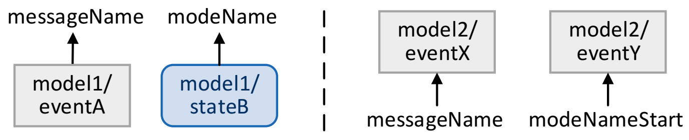

Figure 3.4 Visualisation of modes and messages

The nine different keywords divide the reading of the Triplets data format definitions to nine different cases. The use of strictly defined keywords enables straightforward reading of the Triplets data format definitions. The use of exactly three columns for whole model definition simplifies the handling of the model data further.

If the keyword in Column B matches to ”class”, a new class is being declared. Following checking is made to validate that the definition is proper:

- Column C must contain a previously declared class name, or a built-in class `ModelObject`, which will bee the super class of the declared new class.
- Column A must not contain a previously declared class name (see Rule 3.4).
- The class name in Column A must be a proper class name (see Rule 3.1).
- The class name in Column A must start with an upper case character (see Rule 3.2).

If the keyword in Column B matches to ”container”, a container declaration is being made for a class. Following checking is made to validate that the definition is proper:

- Column A must contain a previously declared class name, which will be defined as a container class.
- Column C must contain a previously declared class name, which will be defined as a type of the contained objects.
- The class of the container must have not been previously declared to contain the contained type of objects. This means that the container class or any of its super classes must not have been declared to contain any super class of the contained type of objects (see Rule 3.4).

If the keyword in Column B matches to ”attribute”, an attribute is being declared for a class. Following checking is made to validate that the definition is proper:

- Column A must split to exactly two name strings by using a slash (/) symbol as a delimiter. The first name is considered as a class name and the last as an attribute name.
- The class name in Column A must match a previously declared class name, which will be defined as a owner class of the attribute.
- The name in Column A must be a proper attribute name (see Rule 3.1).
- The attribute name in Column A must start with a lower case character (see Rule 3.3).
- The owner class must not contain a previously declared attribute with the same name (see Rule 3.4).
- Column C must contain a previously declared class name, which will be defined as a type of the attribute objects.
- If a super class of the owner class contains an attribute with the same name, its type must be a super class of the attribute type. This permits declaration of sub classes that have more strict rules for the types of its attributes.

If the keyword in Column B matches to ”prototype”, a prototype is being created to a class. Following checking is made to validate that the definition is proper:

- Column A must split to exactly two name strings by using a slash (/) symbol as a delimiter. The first name is considered as a class name and the last as a prototype name.
- The class name in Column A must match a previously declared class name, which will be defined as a owner class of the prototype.
- The name in Column A must be a proper instance name (see Rule 3.1).
- The prototype name in Column A must start with a lower case character (see Rule 3.3).
- The owner class must not contain a previously created prototype with the same name (see Rule 3.4).

- Column C must contain a previously declared class name, which will be defined as a type of the prototype.

If the keyword in Column B matches to ”instance”, a model object is being created. Following checking is made to validate that the definition is proper:

- Column C must contain a previously declared class name, which will be defined as a type class of the created object.
- Column A must be a proper path, which defines a container and the name of the created object. If the path is just a name, it is used as an object name and the model base folder is used as a container. Otherwise the path is split to a list of names by using a slash (/) symbol as a delimiter, where the first names are considered as a container path and the last as an object name.
- The object name in Column A must be a proper instance name (see Rule 3.1).
- The object name in Column A must start with a lower case character (see Rule 3.3).
- The container path in Column A must refer to at least one container, which are considered as container objects.
- The class of the container objects must have been previously declared as a container of the type class objects.
- Objects with the same name must have not been created for the same container objects previously (see Rule 3.4).

If the keyword in Column B matches to ”include”, a class is being included to an element or a prototype. Following checking is made to validate that the definition is proper:

- Column C must contain a previously declared class name, which will be considered as the included class.
- Column A must be a proper path, which refers to at least one object or a prototype.
- The object in Column A must be an instance of a class that is a super class of the included class.
- The object in Column A must have not already included a class (see Rule 3.4).

If the keyword in Column B matches to ”connect”, a connection is being added between elements or prototypes. Following checking is made to validate that the definition is proper:

- Column A must be a proper path, which defines the connection elements and the name of the connection attribute. The path is split to a list of names by using a slash (/) symbol as a delimiter, where the first names are considered as the connection element path and the last as the connection attribute name.
- The connection element path in Column A must refer to at least one element or a prototype.
- Column C must be a proper path, which defines at least one connected elements or a prototype.
- There must be at least one pair of a connection and connected elements that share properly a common container (see Section 3.1.4).
- For all found connection and connected element pairs, the connection attribute name in Column A must have been declared as an attribute of the class of the connection element, and the type of the attribute must be a super class of the type of the connected element.

If the Column B matches to equals ”=” sign, a value is being either assigned to an attribute. Following checking is made to validate that the definition is proper:

- Column A must be a proper path, which defines an element and the name of the assigned attribute. The path is split to a list of names by using a slash (/) symbol as a delimiter, where the first names are considered as an element path and the last as an attribute name.
- The element path in Column A must refer to at least one element. The element path can also refer to a class or a prototype, which changes the definition from an element value assignment to a default value assignment, which can be handled similarly.
- The attribute name in Column A must have be a declared as an attribute of the class of the element, and the a type of the attribute must be a sub class of a `Primitive`.

- If the type class of the attribute is a `Number`, it must be possible to parse the content of Column C to a number value. Special parsing could be included in the tool that reads the models for certain sub classes of a `Number`, such as `Duration`.
- The same attribute must have not been assigned for the same element previously (see Rule 3.4).

If the keyword in Column B matches to "add", a value is being either added to an attribute. Following checking is made to validate that the definition is proper:

- Column A must be a proper path, which defines an element and the name of the added attribute. The path is split to a list of names by using a slash (/) symbol as a delimiter, where the first names are considered as an element path and the last as an attribute name.
- The element path in Column A must refer to at least one element. The element path can also refer to a class or a prototype, which changes the definition from an element value add to a default value add, which can be handled similarly.
- The attribute name in Column A must have be a declared as an attribute of the class of the element, and the a type of the attribute must be a sub class of a `Primitive`.
- If the type class of the attribute is a `Number`, it must be possible to parse the content of Column C to a number value. Special parsing could be included in the tool that reads the models for certain sub classes of a `Number`, such as `Duration` (see Section 3.2.1).

If the Column B does not match to any of the nine keywords, the definition is erroneous. In addition, all rows that do not pass the validity check that was listed above are considered as erroneous. The reading of the model data may continue after an erroneous definition has been skipped, but a clear error message must be shown to the user.

3.3 Stochastic discrete event simulation of the model

The AoT models are decoupled from calculation. The Triplets data format forms a platform-independent interface between the model definition and the tools that analyse the models. A tool can be an analytical solver for certain modelling technique, such as a calculator of the minimal cut sets of FTA models, or a simulator for various techniques. This thesis presents a generic stochastic Discrete Event Simulation (DES) [80] framework for analysis of any type of AoT models. The flexible approach can be configured to support any modelling technique and their combination.

3.3.1 Dynamic compilation of the simulation algorithm

The flexibility of the stochastic DES framework is achieved by using configurations that can be customised to define the used simulation algorithm for each technique. Figure 3.5 illustrates how a calculation engine, a simulator tool and configurations of modelling techniques are dynamically compiled to a Java [75] simulation program. The dynamic compilation ensures that only the procedures that are needed by the analysed model are included in the simulation algorithm. This increases the efficiency of the analysis process because the used algorithm is always as simple as possible.

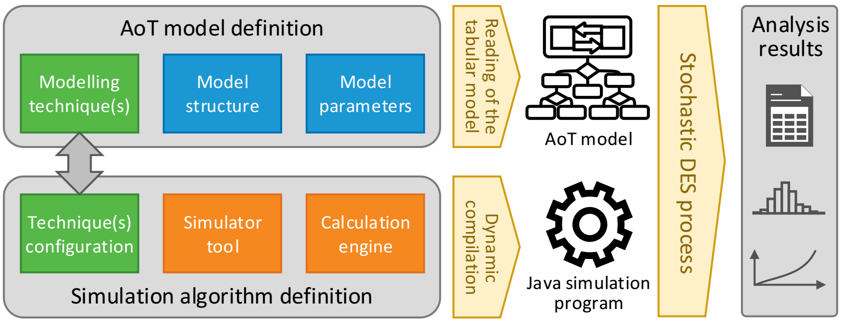

Figure 3.5 The dynamic compilation of the Java simulation program

The Java simulation program is built by using a template method pattern [81], which is an example of an Inversion of Control (IoC) design principle [82]. In object-oriented programming, the IoC is used to increase the modularity of the program. In traditional programming the custom code calls for static libraries, but with IoC, it is the generic framework that calls task-specific codes. The calculation engine calls a `calculationProcess()` template method of the simulator tool, which implements the method by defining the simulation algorithm core. Similarly, the classes of different modelling techniques implement the template methods that are called by the simulator tool during the execution of the simulation algorithm.

The pre-processing of the modelling techniques' configuration is handled by the calculation tool. It reads the procedures of the template methods and translates them to a Java code that is suitable for dynamic compilation. The DES process requires efficient handing of events queue, which is provided by the calculation tool. The pre-processing and the event handing are the only static parts of the process. Other parts are defined by the configuration, which makes the approach highly customisable.

3.3.2 Definition of the simulation algorithm by using Triplets format

The definition of the simulation algorithm includes the creation of a simulator tool and the configuration of the modelling techniques' template methods. Like the elements of the model, the simulator tool is an instance of a class. Table 3.24 presents the use of Triplets data format for declaration of a general DES simulator tool class and creation of a tool instance.

Table 3.24 The definition of a simulator tool

	A	B	C	COMMENTS
1	DES	class	Tool	Declaration of simulator tool class
2	DES/maxRounds	attribute	Integer	The rounds limit parameter of the simulator
3	DES/period	attribute	Duration	The time period parameter
4	DES/maxRounds	=	1000	By default 1000 rounds are simulated
5	DES/period	=	10a	By default the time period of each round is 10 years
6	/simulator	instance	DES	The creation of a simulator tool instance (UID: /simulator)

The attributes of the simulator tool define a limit to the total simulated time. The attributes that are declared for the modelling technique and assigned for the model elements are used as other parameters of the simulation. The analysis of the model should not update the values of the model attributes. The configuration declares variables, which update is allowed during the simulation process. The variables can be divided to three categories:

Status variables store the current situation of the simulation process. For example, a variable that stores the number of already simulated rounds and a variable that stores the current state of a vertex are status variables.

Statistics variables collect the statistics data during the simulation process. For example, a variable that stores the cumulative count of state activations is a statistics variable.

Result variables are updated after the simulation process. They are calculated based on statistics variables. For example, a mean number of state activations is a result variable.

The configuration declares the variables to the simulator tool and to the classes declared by the modelling techniques. Table 3.25 presents the basic status, statistics and result variables declaration of the general DES simulator tool.

Table 3.25 Basic status, control and result variables declaration for DES tool

	A	B	C	COMMENTS
1	DES/currentRound	attribute	Number	Status: Current simulation round
2	DES/currentTime	attribute	Duration	Status: Current simulation clock time
3	DES/stepLength	attribute	Duration	Status: The length of current simulation step
4	DES/eventsHandled	attribute	Integer	Statistics: The cumulative event activations
5	DES/stepsTaken	attribute	Integer	Statistics: The number of steps taken
6	DES/simulatedTime	attribute	Duration	Result: Store the total simulated time

The procedures that read the parameters and update the variables are defined for template methods. The first implemented procedure is the `calculationProcess()`, which is called by the calculation engine. The simulator tool implements the method by defining a simulation algorithm core, which calls template methods of the model elements. The configuration defines the procedure codes of the template methods by using a programming language that is based on Java. The use of any Java libraries and data structures is enabled.

For implementation of the template method program codes, an attribute with name ”procedure” and type `Text` is declared for all elements. There exists also three type of functions, which return either a number, a Boolean or an element. Each function type has its own attribute name. Table 3.26 presents the declaration of the program code definition attributes.

Table 3.26 The declaration of procedure attributes

	A	B	C	COMMENTS
1	Element/procedure	attribute	Text	An attribute to define procedures
2	Element/function	attribute	Text	An attribute to define number functions
3	Element/logic	attribute	Text	An attribute to define Boolean functions
4	Element/select	attribute	Text	An attribute to define element functions

The codes can be assigned directly as an attribute value or read from an external location (see Table 3.15 on page 55). The key of the array definition defines the name of the assigned procedure or function. A template method can call any procedures and functions, which codes are assigned by using the corresponding attribute key. Table 3.27 presents examples of procedure and function definitions.

Table 3.27 Examples of using the procedure attributes

	A	B	C	COMMENTS
1	ClassName/procedure[template]	=	// procedure code	An example of a procedure definition
2	ClassName/function[value]	=	return 0;	An example of a number function
3	ClassName/logic[test]	=	return true;	An example of a logic function
4	ClassName/select[find]	=	return THIS;	An example of a select function

Instead of using the Triplets format, this thesis presents the procedure and function code definitions by using listings. The used format is similar to Java method definition. The use of listings helps the reading of the procedure and function codes that have multiple lines. The return value of the method defines whether the code is a procedure or certain type of function. The name of the model element class is added as prefix of the method name. To illustrate the format that is used in this thesis, Listing 3.1 presents the same example program codes that were assigned in Table 3.27.

```
void ClassName.template() {
  // procedure code
}

double ClassName.value() {
  return 0;
}

boolean ClassName.test() {
  return true;
}

Element ClassName.find() {
  return THIS;
}
```

Listing 3.1 The format of the procedure and function definitions of this thesis

To enable the most straightforward procedure code definition, a special syntax has been included in the used programming language. The following keywords can be used for basic handling of AoT model elements and attributes:

- `THIS`: Enables an access to a Java object that represents the element itself.
- `CALCULATION`: Enables an access to a Java object of the calculation tool.
- `MODEL`: Enables an access to a Java object that represents the whole model.
- `ACTIONS`: Enables an access to a Java object that handles the actions queue.
- `RANDOM`: Enables an access to a Java object that generates random numbers.

All AoT model elements and the simulator tool are handled in procedure code as they were Java objects. A Java class `Element` represents the AoT model elements. The container of an element and all attributes of type `Element` are handled with the methods that are defined for the Java object. The following built-in methods are available for the `Element` class:

- `String Element.getUID()`: Returns the UID of the element.
- `String Element.getName()`: Returns the name of the element.
- `String Element.getKey()`: Returns the key of the array definition or an empty string if array is not used with the element.
- `Element Element.getContainer()`: Returns the container of the element.

- `Element Element.getElement(String name)`: Returns an element that is assigned or connected by using the given name. The first element is returned if more than one assignments or connections have been made with the same name. Null is returned if no elements exist with the given name.

- `void Element.setElement(String name, Element element)`: Assigns an element attribute with the given name to the element.

The AoT attributes of type `Primitive` are handled in procedure code as `String`, `double` and `long` instance variables of the Java object. The AoT `Number` attributes are translated to doubles and `Integer` attributes to longs. The use of Java primitive data types permits the use of assignment operators, such as '=', '+=' and '++'.

The handling of all AoT features, such as the container declaration and the array definition, is not directly possible with Java instance variables. To help the needed pre-processing, the upper case version `THIS` is used in procedure code instead of the Java keyword `this`. For example, the expression `THIS.attrName = 2;` assigns value 2 to attribute "attrName". Also the key of the array assignment can be given directly. For example, the expression `THIS.attrName[key]++;` adds the value that is stored to key "key" of the attribute "attrName".

The pre-processing of the procedure and function codes enable similar simpler handling for element attributes. The procedure code `THIS.name` uses the method `THIS.getElement(String name)` if the "name" is an attribute of type `Element`. Similarly, the procedure code `THIS.name = element` uses the built-in method `THIS.setElement(String name, Element element)`.

Sometimes there is a need to have an access to all attributes that are defined with certain name. This is needed, for example, when an operation is made for all connected elements by using a for-loop. Following built-in methods are available for the `Element` class to handle situations where values with all keys needs to be accessed:

- `List<Element> Element.getElements(String name)`: Returns a list of elements that have been assigned for the element by using the given attribute name (with any array key). An empty list is returned if the element does not contain any elements with that attribute name.

- `List<String> Element.getStrings(String name)`: Returns a string list that have been assigned for the element by using the given attribute name (with any array key). An empty list is returned if the element does not contain any strings with that attribute name.

- `List<Number> Element.getNumbers(String name)`: Returns a list of numbers that have been assigned for the element by using the given attribute name (with any array key). An empty list is returned if the element does not contain any numbers with that attribute name.

The procedures and functions are called like methods of Java objects. The pre-processing is also needed when a method is called, because different type of elements do not have always the same methods. To help the pre-processing, the keyword `THIS` is always used with the methods. For example, `THIS.activate()` calls a procedure of name `activate` for the element.

The procedure codes have an access to the calculation tool with the keyword `CALCULATION`, which in this case is a simulator tool. The previously listed element methods can be used for handling the simulator attributes.

The keyword `MODEL` enables accessing the other model elements from any procedure and function code. The get and set of any number attribute value is also enabled. The `MODEL` object has the following methods:

- `Element MODEL.getElement(String path)`: Returns the element based on the given path.
- `List<Element> MODEL.getElements()`: Returns the list of all elements of the model.
- `List<Element> MODEL.getElements(String... className)`: Returns a list of elements of given class or classes.
- `double MODEL.getNumber(String path)`: Returns the value of a number attribute based on the given path.
- `void MODEL.setNumber(String path, double number)`: Sets the number attribute value based on the given path.

The DES needs the handling of events, which in this case are called more generally *actions*. An action triggers certain event or activates other element. The `ACTIONS` object is a queue that stores actions in a chronological order. The first action in the queue is always the next that will be handled. The calculation engine implements the `ACTIONS` object, which provides following methods:

- `Action ACTIONS.add(double time, Element target)` Actions are added when the simulation round starts and during the handling of other actions. Each action has a time of its occurrence and a target element. To keep the actions queue in the chronological order, each new action is inserted to the correct location based on the occurrence time. The adding method returns the new created action.

- `Action ACTIONS.add(Element target)` Same as above, but adds an immediate action. The current simulation time (`MODEL.currentTime`) is used as a time of the added action.

- `double ACTIONS.getFirstTime()` Peeks the first action from the queue and returns the time of its occurrence. This is needed when the calculation decides whether an action handing can be started or a simulation step needs to be taken. Positive infinity is returned if no actions exist.

- `Element ACTIONS.getFirstTarget()` Peeks the first action from the queue and returns the target element. This is needed to find the correct handler for the action. Null is returned if no actions exist.

- `double ACTIONS.removeFirst()` The first action is removed from the queue after it has been properly handled. The occurrence time of the next action is returned. The return value is positive infinity if the last action of the list was removed.

- `Action ACTIONS.remove(Element target)` Remove an action that has the given target element from actions queue. Returns the removed action, or null if actions with the defined target do not exist.

- `void ACTIONS.clear()` Clears all actions from the queue. The actions from a previous round needs to be cleared before a new simulation round starts.

An `Action` is a basic Java object, which stores the time and the target of the action. The stored values are assigned by using the `add` methods of the `ACTIONS` queue. The `Action` Java object provides a method for reading of each stored value:

- `double Action.getTime()` Returns the time of occurrence.

- `Element Action.getTarget()` Returns the target element of the action.

Random numbers are frequently needed in stochastic DES. The implementation of a random number generation can be done with the help of Java Random class. The `RANDOM` object provides following methods to give an access to the generation of Java pseudo-random numbers:

- `void RANDOM.init()` Initialises the random number generator.
- `void RANDOM.init(long seed)` Initialises the random number generator using a certain seed. This allows repeating the same simulation results, which is useful for debugging purposes.
- `double RANDOM.prob()` Returns uniformly distributed probability value between 0.0 and 1.0. This random value can be used for different distribution functions.
- `double RANDOM.exp(double mean)` Returns exponentially distributed value with given mean.

3.3.3 The configuration of a generic DES tool

The calculation engine calls a `calculationProcess()` template method, which is implemented by each analysis tool. This thesis presents a general DES tool, which can be applied with various modelling techniques. The DES algorithm repeats defined number of rounds and calculates the analysis results based on them. Defined simulation time period is handled during each round. At first the model elements create initial actions, which are stored to a chronologically ordered list. An `Action` has a time of occurrence and a target model object that handles the event. The continuous time DES takes an adaptive time step to the first action time, which is handled by the target model object. New actions are possibly created and inserted to the list. This process is repeated until the simulation period is reached.

Listing 3.2 shows the procedure code of the simulation algorithm skeleton, which divides the DES process to template methods. The definition is made by using the configuration format that was presented in the previous section (see Listing 3.1 on page 69). Figure 3.6 illustrates the defined algorithm skeleton. Other calculation tools can implement similarly their own `calculationProcess()` template method. Different algorithm core can be used, for example, if times between the events are not considered or if only basic calculations are made (see Section 4.6).

```
void DES.calculationProcess() {
  THIS.simulationStart();
  do {
    THIS.roundStart();
    while (true) {
      while (THIS.actionStart()) {
        THIS.actionHandle();
        THIS.actionEnd();
      }
      if (THIS.roundEnd()) {
        break;
      }
      THIS.step();
    }
  } while (!THIS.simulationEnd());
  THIS.createResults();
}
```

Listing 3.2 The simulation algorithm skeleton of the general DES tool

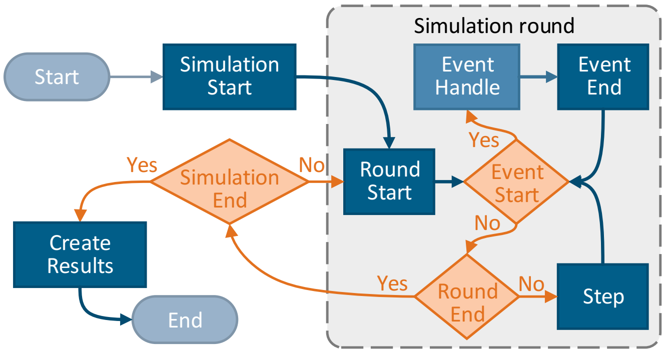

Figure 3.6 The algorithm skeleton that divides the DES process to template methods

Each template method of the simulation algorithm skeleton is defined by the configuration of the simulator tool. The configurations of the modelling techniques define similar methods for the model objects. In simplest case the simulator tool only calls the each template method every time for all other model objects, but there can be also other logic rules and operations defined for the simulator tool. Special definition is needed at least for action handling, which is called only for the target element of the handled action. Chapter 4 presents how the classes of the modelling techniques define the template methods that are called from the simulator tool.

The method simulationStart() is called at the beginning of each simulation. Here a reset is made for the variables that control the simulation process and collect the statistics data. For example, the simulator tool resets the variable that stores the current simulation round. The simulator also resets here the random number generator of the calculation engine. Listing 3.3 shows the procedure code of the simulationStart() template method.

```
void DES.simulationStart() {
  RANDOM.init(); // parameter can be added if fixed seed is needed
  THIS.currentRound = 0; // reset the status variable:  currentRound
  THIS.eventsHandled = 0; // reset the statistics variable:  eventsHandled
  THIS.stepsTaken = 0; // reset the statistics variable:  stepsTaken
  for (Element element : MODEL.getElements()) {
    element.simulationStarted(); // call the template method for all elements
  }
}
```

Listing 3.3 The DES tool definition of the simulationStart() template method

The method simulationStart() calls simulationStarted() template method for each model element. The method can be used to reset the different statistics variables of each element class. For example, each event instance has a variable that stores the cumulative number of times the event activates. When this variable is divided by the number of simulated rounds, the result value gives the mean number of the activations of the event.

The method simulationEnd() is called after a simulation round has ended. If the rounds limit is reached, the next phase is the creation of the analysis results. If more simulation rounds are still needed, the next phase is the start of a new simulation round. Listing 3.4 shows the procedure code of the simulationEnd() template method.

```
boolean DES.simulationEnd() {
  return THIS.currentRound == THIS.maxRounds;
}
```

Listing 3.4 The DES tool definition of the simulationEnd() template method

The method roundStart() is called before the handling of a new round starts. The simulator tool resets the simulation clock and clears the actions list. The reset is needed also for some model object variables. For example, the initial state of a node is activated, which creates the actions of initial events. Listing 3.5 shows the procedure code of the roundStart() template method.

```
void DES.roundStart() {
  THIS.currentTime = 0; // reset the status variable:  currentTime
  ACTIONS.clear(); // reset the events queue
  for (Element element : MODEL.getElements()) {
    element.roundStarted(); // call the template method for all elements
  }
}
```

Listing 3.5 The DES tool definition of the roundStart() template method

The method `roundEnd()` is called before simulation steps are taken. The current round is ended if the simulation clock equals the time period of each round. Otherwise an adaptive time step can be taken. The elements can do operations here if, for example, some update is required for statistics variables. The simulator tool increases here the current simulation round variable. Listing 3.6 shows the procedure code of the `roundEnd()` template method.

```
boolean DES.roundEnd() {
  if (THIS.currentTime == THIS.period) {
    for (Element element : MODEL.getElements()) {
      element.roundEnded(); // call the template method for all elements
    }
    THIS.currentRound++;  // increase the status variable:  currentRound
    return true; // a round has been simulated
  }
  return false; // a time step needs to be still taken
}
```

Listing 3.6 The DES tool definition of the roundEnd() template method

The method `actionStart()` is called after each action handling and after taking a time step. A new action is handled if the simulation clock equals the time of the first action. If no actions occur at the current time, a step needs to be taken. Listing 3.7 shows the procedure code of the `actionStart()` template method.

```
boolean DES.actionStart() {
  return ACTIONS.getFirstTime() == THIS.currentTime;
}
```

Listing 3.7 The DES tool definition of the actionStart() template method

The method `actionHandle()` is called if an action needs to be handled. The handing activates the target element, which has a template method `activate()` that defines the procedure of the activation. Listing 3.8 shows the procedure code of the `actionHandle()` template method.

```
void DES.actionHandle() {
  ACTIONS.getFirstTarget().activate(); // action activates the target element
}
```

Listing 3.8 The DES tool definition of the actionHandle() template method

The method `actionEnd()` is called after an action has been handled. The elements can update the statistics variables here. After all procedures related to the action are made, the simulator removes the handled action from the actions queue. Listing 3.9 shows the procedure code of the `actionEnd()` template method.

```
void DES.actionEnd() {
  for (Element element : MODEL.getElements()) {
    element.actionEnded(); // call the template method for all elements
  }
  ACTIONS.removeFirst(); // the handled event can be removed from the queue
}
```

Listing 3.9 The DES tool definition of the actionEnd() template method

The method `Step` is called if there exists no action at current time. The step is taken to the occurrence time of the next action or to the end of the simulation period if no actions occur before it. The simulator tool calculates the length of the step that is taken and stores it to a variable, which can be used by other elements when they update statistics variables. For example, the length of the step is added to a cumulative time variable of a state element if the state is active during the step. When this variable is divided by the the total simulated time, the result value gives the probability of the state. Listing 3.10 shows the procedure code of the `step()` template method.

```
void DES.step() {
  double nextTime = ACTIONS.getFirstTime();
  if (nextTime > THIS.period) {
    nextTime = THIS.period; // step to period end if no events before it
  }
  THIS.stepLength = nextTime - THIS.currentTime; // assign status variable
  for (Element element : MODEL.getElements()) {
    element.stepTaken(); // call the template method for all elements
  }
  THIS.stepsTaken++; // increase the statistics variable:  stepsTaken
  THIS.currentTime = nextTime; // a time step has been taken
}
```

Listing 3.10 The DES tool definition of the step() template method

The method `createResults()` is called when all simulation rounds have been finished. Here the analysis results are calculated based on the statistics variables. Listing 3.11 shows the procedure code of the `createResults()` template method.

```
void DES.createResults() {
  THIS.simulatedTime = THIS.period * THIS.currentRound; // a result value
  for (Element element : MODEL.getElements()) {
    element.createResult(); // call the template method for all elements
  }
}
```

Listing 3.11 The DES tool definition of the createResults() template method

All classes do not necessarily implement all template methods, but each method can be always called to all elements. The code pre-processing ensures that the call of an undefined template method is just skipped. Avoiding all unnecessary procedure code executions is the most significant mean for improving the efficiency of the simulation process (see Section 3.4).

The elements' template methods form the interface between the simulation algorithm and the configuration of the modelling techniques. The definition of the template methods is configured for each class of the technique. The presented simulation algorithm core calls following template methods for all element classes:

simulationStarted() initialises the statistics variables of the element. For example, a value zero is initialised to active time and activation count variables of each state element.

roundStarted() resets the status and statistics variables that are related to a simulation rounds. For example, an initial state of a node is activated.

roundEnded() updates the statistics variables that are related to a simulation rounds. For example, the active time of the simulated round is stored.

activate() handles the activation of the element. For example, an activated event changes the active state of a node.

actionEnded() updates the statistics variables that are related to actions. For example, the activation count of activated state elements is increased.

stepTaken() updates the statistics variables that are related to a simulation steps. For example, the step length is added to the activation time of active states.

createResult() updates the result variables of the element. For example, the mean number of activations of a state is calculated.

3.3.4 The configuration of the state changes

The configuration of Event, State and Node classes define the core of the DES process action handling. The modelling techniques extend the configuration to define the action handling of Operator and other element classes (see Chapter 4). Basic simulation steps are taken between event activations, which update the currently active state of a node. Each activation can trigger activations to other model elements.

Table 3.28 presents the declaration of the basic status and statistics variables that are needed in the core procedures of the action handling.

Table 3.28 Variable declarations for the core of the action handling

	A	B	C	COMMENTS
1	Node/currentState	attribute	State	Status: Store the currently active state
2	State/activeTime	attribute	Duration	Statistics: Store the active time of the state
3	State/activationCount	attribute	Integer	Statistics: Store the number of activations

The core status variables are handled by the procedures of the Event, State and Node classes. For the calculation of basic results, the simulation process collects cumulative statistics variables for the time the state is active and for the number of the state activations. Listing 3.12 shows the reset of the used two statistics variables.

```
void State.simulationStarted() {
  THIS.activeTime = 0; // reset the statistics variable: activeTime
  THIS.activationCount = 0; // reset the statistics variable: activationCount
}
```

Listing 3.12 The reset of the statistics variables of the State class

Each simulation round starts with the activation of the initial node states. Listing 3.13 shows the procedure of a Node class that activates the initial state when a round starts.

```
void Node.roundStarted() {
  THIS.currentState = null; // reset the current state
  ACTIONS.add(THIS.initialState); // activate the initial state
}
```

Listing 3.13 The activation of the node initial state

Listing 3.14 shows the definition of the action handling for the State class. A state activation updates the ”currentState” status variable of a node. After the state change, an action is created for the container node to activate, for example, the operators that are connected to the node. The applied modelling technique defines the handling of the node action and creation of the actions for the connected elements (see Chapter 4).

```
 1  void State.activate() {
 2    if (THIS.getContainer().currentState != null) {  // not done at round start
 3      THIS.getContainer().currentState.sendEndMessages();
 4    }
 5    THIS.getContainer().currentState = THIS; // current container node state
 6    THIS.sendStartMessages();
 7    THIS.activationCount++;
 8    ACTIONS.add(THIS.getContainer()); // check node
 9    THIS.createNextAction(); // find the first event to activate
10  }
```

Listing 3.14 The activation handling of the State class

The method `sendEndMessages()` is called before a state activation ends. The call is not made at the beginning of each simulation round when the ”currentState” is not defined for the container node (see line 2 in Listing 3.14), but all other state changes call the method. Listing 3.15 presents the procedure code of the method.

```
void State.sendEndMessages() {
  for (String mode : THIS.getStrings("mode")) {
    for (Element element : MODEL.getElements()) {
      if (element.getStrings("listener").contains(mode + "End")) {
        ACTIONS.add(element); // activate immediately the listener
      }
      if (element.getStrings("listener").contains(mode)) {
        element.modeEnd(); // notice an end of a mode
      }
      if (element.getStrings("listener").contains(mode + "Not")) {
        element.modeStart(); // notice a start of a complement mode
      }
    }
  }
}
```

Listing 3.15 Sending the mode end messages of the State class

The method handles the message sending that is related to the modes of the state (see Table 3.22 on page 59). The method calls `modeEnd()` and `modeStart()` template methods, which can be implemented by the listener elements (see Section 4.2.4). The method handles also the complement modes (see Rule 3.7).

Similarly, the method sendStartMessages() is called after an activation of a new state. Listing 3.16 presents the procedure code of the method.

```
void State.sendStartMessages() {
  for (String mode : THIS.getStrings("mode")) {
    for (Element element : MODEL.getElements()) {
      if (element.getStrings("listener").contains(mode + "Start")) {
        ACTIONS.add(element); // activate immediately the listener
      }
      if (element.getStrings("listener").contains(mode)) {
        element.modeStart(); // notice a start of a mode
      }
      if (element.getStrings("listener").contains(mode + "Not")) {
        element.modeEnd(); // notice an end of a complement mode
      }
    }
  }
}
```

Listing 3.16 Sending the mode start messages of the State class

Each state change creates a new action for a target event, which activates a next state after certain delay. The procedure createNextAction() handles the selection of the created action. Because only one state is active, an action is created only for the event that has the shortest delay. Listing 3.17 presents the procedure code of the method.

```
void State.createNextAction() {
  Element firstTarget = null;
  double firstTime = Double.POSITIVE_INFINITY;
  for (Element target : THIS.getElements("target")) {
    double targetTime = target.getEventTime();
    if (firstTime > targetTime) {
      firstTarget = target;
      firstTime = targetTime;
    }
  }
  if (firstTarget != null) {
    ACTIONS.add(firstTime, firstTarget);
  }
}
```

Listing 3.17 The next event action creation the State class

Listing 3.18 presents the code of getEventTime() and getDelayTime() functions, which are used for defining a delay before event activates. The Event sub classes can overwrite the getDelayTime() function (see Section 4.1) for defining stochastic delays. Overwriting the `getEventTime()` is needed, for example, if certain factor updates the event time (see Listing 4.25 on page 111).

```
double Event.getEventTime() {
  return CALCULATION.currentTime + THIS.getDelayTime();
}

double Event.getDelayTime() {
  return THIS.delay;
}
```

Listing 3.18 The delay time of the Event class

Listing 3.19 shows how the event activation is configured for the `Event` class. At first an event finds an active state from the event sources. An activated event activates a target state only if one of its sources is active. Otherwise the event activation is ignored. This ensures that always exactly one state of a node is active.

```
void Event.activate() {
  for (Element source : THIS.getElements("source")) { // find active source
    if (THIS.getOwner().currentState == source) {
      EVENTS.add(THIS.target); // activate the state
      THIS.sendMessages();
      break;
    }
  }
}
```

Listing 3.19 The definition of the activation of the Event class

In previous Listing 3.19, a call of the `sendMessages()` procedure was added after the state activation. It configures the handling of the message sending (see Table 3.22 on page 59). An action is added for all events that listen the sent message. Listing 3.20 presents the code of the method. The procedure is similar to sending the messages of mode start and end (see Listings 3.15 and 3.16 on page 80)

```
void Event.sendMessages() {
  for (String message : THIS.getStrings("message")) {
    for (Element element : MODEL.getElements()) {
      if (element.getStrings("listener").contains(message)) {
        ACTIONS.add(element); // activate immediately the listener
      }
    }
  }
}
```

Listing 3.20 Sending the messages of the Event class

3.3.5 Calculation of basic analysis results

The analysis results are calculated based on the simulation data, which is collected by using the statistics variables. For example, the result values for instances of a `State` class are calculated based on ”activationCount” and ”activeTime” statistics variables. The ”activationCount” is updated during the action handling (see Listing 3.14 on page 80). The update of the ”activeTime” statistics variable is presented in Listing 3.21. The template method is called for all elements during simulation step (see Listing 3.10 on page 77).

```
void State.stepTaken() {
  if (THIS.getContainer().currentState == THIS) { // if active
    THIS.activeTime += CALCULATION.stepLength;
  }
}
```

Listing 3.21 The reset of the main element statistics variables

The values, which are calculated to the result variables, are stored to element attributes. For example, Table 3.29 presents basic result variable declarations for the `State` class. These mean and probability variables can be used for the calculation of results that are related to nodes. For example, the unavailability of a fault node is the ”activeProb”, and the MTTF is the ”activationMTB” of the ”fault” state.

Table 3.29 The declaration of basic result variables for states

	A	B	C	COMMENTS
1	State/activeTimeMean	attribute	Duration	Result: Mean cumulative active time
2	State/activationCountMean	attribute	Number	Result: Mean number of activations
3	State/activeProb	attribute	Probability	Result: Probability that the state is active
4	State/activationDuration	attribute	Duration	Result: Mean duration of one activation
5	State/activationMTB	attribute	Duration	Result: Mean time between activations

Listing 3.22 shows the procedure code for calculating the basic result values for instances of the `State` class. The `createResult()` template method is called for all elements when the simulation ends (see Listing 3.11 on page 78). The calculation uses the statistics variables of the state and the simulator. Because only cumulative statistics variables are needed, these simple results can be calculated with minimal memory consumption. In this case the higher number of simulated rounds does

not increase the memory consumption. More detailed results require more statistics variables for collecting the needed data.

```
void State.createResult() {
  THIS.activeTimeMean = THIS.activeTime / CALCULATION.currentRound;
  THIS.activationCountMean =
      THIS.activationCount / CALCULATION.currentRound;
  THIS.activeProb = THIS.activeTime / CALCULATION.simulatedTime;
  THIS.activationDuration = THIS.activeTime / count;
  THIS.activationMTB = THIS.simulatedTime / THIS.activationCount;
}
```

Listing 3.22 The calculation of basic analysis results for states

3.4 Improving the efficiency of the simulation process

The procedure codes that are presented in this thesis are not optimised from the efficiency point of view. It is possible to obtain similar analysis results with an improved algorithm that uses less calculation time. Similarly, improvements are needed for calculating more detailed results. This thesis presents the basic procedure codes, which can be used as a basis for optimising the configuration.

3.4.1 Removing unnecessary method calls

Removing the unnecessary method calls is the most significant way for improving the efficiency of the simulation process. A template method is called for all model elements after each activation has ended (see Listing 3.9 on page 77) and during each simulation step (see Listing 3.10 on page 77). Both of these template methods are repeated several times during the simulation process, which means that even a minor improvement can have a significant effect on the efficiency. The pre-processing of the simulation algorithm can update the code to skip the call of a template method for elements that never need any operation in the situation. For example, the calculation engine can automatically update each `for` loop to go through only the elements that have implemented the template method. More complex improvements require manual updates to the defined configuration.

The number of method calls can be reduced by changing the situation when the update is made. New status variables might be needed to make the update differently. For example, in basic situation the cumulative active time of a state is added during each simulation step (see Listing 3.21 on page 83). An alternative approach is to update the cumulative active time only once after an activation ends. This requires a new status variable, which declaration is presented in Table 3.30.

Table 3.30 The declaration of the previous activation status variable

	A	B	C	COMMENTS
1	State/activationStart	attribute	Duration	Status: Store the start time of an activation

The use of the new ”activationStart” status variable is added to the original activation handling of the `State` class (see Listing 3.14 on page 80). This change makes the `stepTaken()` template method obsolete for state elements, which reduces significantly the number of procedure calls that are made during the simulation process. Listing 3.23 presents the updated activation handling of the `State` class. The ”activeTime” of a previous active state is increased when the new state is activated.

```
void State.activate() {
  if (THIS.getContainer().currentState != null) {  // not done at round start
    THIS.getContainer().currentState.activeTime +=
        CALCULATION.currentTime -
        THIS.getContainer().currentState.activationStart; // use the variable
    THIS.getContainer().currentState.sendEndMessages();
  }
  THIS.getContainer().currentState = THIS; // current state of node
  THIS.getContainer().currentState.sendStartMessages();
  THIS.activationCount++;
  THIS.activationStart = CALCULATION.currentTime; // update the variable
  ACTIONS.add(THIS.getContainer()); // check node
  THIS.createNextAction();
}
```

Listing 3.23 The updated activation handling of the State class

Sometimes it is possible to calculate a value based on other values instead of updating it separately. For example, if a node contains only two states, it is sufficient to add the cumulative active time only for one of them. Because always exactly one node state is active, the difference between the total simulated time and the cumulative active time of the other state gives the statistics variable value for the other. This improvement requires a special class for nodes that always have exactly two states and a configuration that makes the needed updates for the statistics variables of the other state when the activation changes. This example shows that new special sub classes might be needed for the simulation efficiency purposes even if the modelling of the same situation would also be possible with a general super class.

3.4.2 Generic statistics and result variables handling

A basic result creation procedure (see Section 3.3.5) calculated mean values, which can be obtained by using only cumulative statistics variables. If more detailed result values, such as minimum, maximum, quantiles or distributions, are needed, also more statistics variables need to be stored. This requires new procedure code for updating the variables. Because the calculation of the result values is similar for various variables, a generic calculation procedures can be created for avoiding the need of writing the same procedure codes several times. In addition, the generic variable handing improves the efficiency of the calculation process.

The generic variable handling requires including a new Java object in the calculation engine. The keyword `VARIABLES` is used in procedure codes for for accessing the object. Following methods are available for the generic variable handing:

- `void VARIABLES.add(Element element, String name)` Initialises an element attribute with a given name as a statistics variable.
- `void VARIABLES.round()` Stores the round end value of each statistics variable.
- `void VARIABLES.results()` Calculates the result values for each statistics variable.

The element classes initialise the variables when the simulation starts. For example, Listing 3.24 shows the adding of basic statistics variables for instances of the `State` class. The original initialisation was only a reset of the variable value (see Listing 3.12 on page 79). After the change the update of the statistics variable values is still made similarly during the simulation process. The `VARIABLES` object only automatises the calculation of various result values.

```
void State.simulationStarted() {
  VARIABLES.add(THIS, "activeTime");
  VARIABLES.add(THIS, "activationCount");
}
```

Listing 3.24 The reset of the statistics variables of the state class

The method `VARIABLES.round()` is called after each simulation round (see Listing 3.6 on page 76). The result values of all variables are calculated when the method `VARIABLES.results()` is called after all simulation rounds are finished (see Listing 3.11 on page 78). Various result values are calculated based on the stored statistics variables of each simulated round. Each result value has its own suffix, which is added to the variable name to get a name of the result value attribute. Following list shows examples of the result values that are calculated for a variable ”activeTime”:

- ”activeTime_MEAN”: The mean active time.
- ”activeTime_MIN”: The minimum active time.
- ”activeTime_MAX”: The maximum active time.
- ”activeTime_QUANTILE5”: The 5% quantile of the active time.
- ”activeTime_QUANTILE95”: The 95% quantile of the active time.

For example, the result value that is stored to an attribute ”activeTime_MEAN” can be used instead of the separately declared (see Table 3.29 on page 83) and manually calculated (see Listing 3.22) attribute ”activeTimeMean”. It is simple to calculate the mean value, so this change does not simplify the procedure code much, but with more detailed result values the change is more significant. For example, the quantiles can be calculated based on the values that are stored when `VARIABLES.round()` is called after each round. After ordering the values, the 5% quantile is obtained from index, which is the number of values divided by `20`. Similarly, the 95% quantile is obtained from index, which is the last index minus the number of values divided by `20`. For nodes that represent faults these quantiles calculate the time periods for 5% and 95% reliability and unreliability.

The values that are stored after each round form a distribution, which gives even more detailed result than the mean and quantile values. Sometimes similar results are needed from other parts of the simulation period. This can be done by storing values always after certain interval. The result can be list of mean values that describe how the value evolves in different parts of the simulation period, or a full distribution and quantiles for each interval.

In all of the previously mentioned situations, the generic result calculation simplifies the definition of the configuration and makes the storing of the needed result values more efficient. The most suitable level of details in result values can be selected based on the needs of the studied case and the possible restrictions related to the memory consumption.

3.4.3 Parallel computing

The simulation process repeats several rounds, which are independent. The possibility to calculate each round independently makes the process suitable for parallel calculation. Powerful laptops and desktop computers have usually more than one core, which should be all used for optimising the calculation speed. The parallelisation is especially important if the simulation is made in a computing cluster that can have a high number of cores.

The division of the simulation process to cores is made before the call of the template method `calculationProcess()`. At first the identical models are loaded to all cores. The attribute values of each model are the same with the original model, but the division assigns a new value for the "maxRounds" attribute of the simulator tool of each model. It is not necessarily possible to divide the simulated rounds equally to all cores. The value for the "maxRounds" attribute of a core can be calculated by using a function

$$rounds(i) = \begin{cases} \lfloor N/m \rfloor + 1 & \text{if } N \bmod m > i \\ \lfloor N/m \rfloor & \text{if } N \bmod m \leq i\,, \end{cases} \tag{3.1}$$

where the total number of simulated round is N, the number of available cores is m, and the core index $i = 0, 1, ..., m-1$.

After the number of simulated rounds is defined, the simulation process is executed normally in each core. The statistics variables of all cores need to be combined before the results are calculated by calling the template method `createResults()`. If all the statistics variables are handled by using the `VARIABLES` object, the combination is done by merging the objects of each core. The statistics variable values are stored to the `VARIABLES` object after each round, so the merge of two objects just appends all the stored round values of each handled statistics variable from one object to other. After the combination, the template method `createResults()` is executed by using the combined `VARIABLES` object that contains data from all cores.

4 APPLICABILITY OF AOT WITH BASIC MODELLING TECHNIQUES

The AoT framework is created for customisation and combination of risk and performance assessment modelling techniques. This chapter uses the methodology that was specified in Chapter 3 to apply the AoT framework with basic modelling techniques. The declaration of the classes and attributes, and the configuration of the simulation algorithm are presented for each technique. The same approach can be used for definition of an arbitrary technique, which is the most suitable for certain special type of case. In Chapter 5 an example model is created by using the techniques that are presented in this chapter.

4.1 Delay distributions of state changes

An `Event` is a fundamental element class that handles the basic state changes in AoT models. Events have a probability and a delay (see Table 3.17 on page 56), which define when the event changes an active state of a node. If the delay is stochastic, the sub classes of `Event` define how the delay of each activation is obtained.

Table 4.1 lists the `Event` sub classes that are presented in this thesis. A new class can be declared similarly for any distribution function. Adding a new distribution requires declaring the parameter attributes for the new class and configuring the calculation for the simulation algorithm. As a naming convention, a prefix `Event` is used with all sub class names.

These `Event` classes are available in all modelling techniques. Following sections present the declaration of the classes. To improve the efficiency of the delay definition, the presented sub classes ignore the ”prob” attribute, which is used by the `Event` class (see Listing 3.18 on page 82).

Table 4.1 Basic built-in Event classes

	A	B	C	COMMENTS
1	EventExp	class	Event	Exponential distribution
2	EventWeibull	class	Event	Weibull distribution
3	EventNorm	class	Event	Normal distribution
4	EventLognorm	class	Event	Log-normal distribution
5	EventHistory	class	Event	Distribution class that is used when there is history data available

4.1.1 Definition of a event distribution

Each `Event` sub class can redefine the function `getEventTime()` or `getDelayTime()` to define the stochastic delay (see Listing 3.18 on page 82). The definition is made with the help of distribution functions. Table 4.2 presents the notation that is used in this section to specify various distribution functions. Each function is defined by declaring an `Event` sub class and by configuring its simulation algorithm by using the procedure code. Basic Java operations and a Math library class are used for numeric operations in the code.

Table 4.2 A general notation for the definition of event distributions

Symbol	Description
F_X	Cumulative distribution function of event class X
Q_X	Generalised inverse distribution function of event class X
U	Next random event time from the distribution
t_c	Current simulation clock time
μ	Mean or expected value
σ	Standard deviation
u	Uniform random variable in the interval [0, 1]
$\exp(x)$	Natural exponential function e^x
$\ln(x)$	Natural logarithm function $\log_e(x)$

The basis of the definition is a cumulative distribution function

$$F_X(x) = \mathrm{P}(X \leq x) = u\,, \tag{4.1}$$

which gives the probability u that the event X activates before (or equal to) certain duration x.

During the simulation process random variables are generated from specified distributions. The variable generation is made by using generalised inverse distribution function

$$Q_X(u) = \inf_{x\in\mathbb{R}} \{F_X(x) \geq u\}\,, \tag{4.2}$$

as the quantile function

$$F_X^{-1}(u) = Q_X(u), \text{ if } F_X \text{ is continuous and strictly monotonically increasing}, \tag{4.3}$$

is not always defined. Next activation times are calculated by adding the delay time obtained from (4.2) to current simulation clock time:

$$U(u) = t_c + Q_X(u) \tag{4.4}$$

In AoT, the function `getEventTime()` defines the $U(u)$. An `Event` sub class can redefine also the procedure `createEvent()`, if needed.

4.1.2 Exponential distribution

Exponential distribution assumes a constant failure rate $(1/\mu)$. The key property of the distribution is that it is memoryless. It is also a simple distribution with only one parameter. Table 4.3 presents the declaration of the "mean" (μ) attribute for the `EventExp` class.

Table 4.3 The attribute declaration of the exponentially distributed events

	A	B	C	COMMENTS
6	EventExp/mean	attribute	Duration	The mean (μ) of the exponential distribution

The mean delay time (μ) is used as a parameter in the cumulative distribution function

$$F_{exp}(x) = 1 - \exp\left(\frac{-x}{\mu}\right) \tag{4.5}$$

and in the generalised inverse distribution function

$$Q_{exp}(u) = -\mu \ln(1-u). \tag{4.6}$$

Listing 4.1 shows the redefinition of the getEventTime() function, which uses the "mean" attribute value as a parameter.

```
double EventExp.getDelayTime() {
  return -THIS.mean * Math.log(1 - RANDOM.prob());
}
```

Listing 4.1 The redefinition of the delay function for the EventExp class

4.1.3 Weibull distribution

Weibull distribution is defined with scale (α) and shape (β) parameters. Additionally, a user can define an activation free time or location (γ), which fixes the point in time from which the activations begin to occur. Table 4.4 presents the declaration of the `EventWeibull` class attributes.

Table 4.4 The attribute declaration of the Weibull distributed events

	A	B	C	Comments
7	EventWeibull/scale	attribute	Duration	Scale parameter of the Weibull distribution (α)
8	EventWeibull/shape	attribute	Number	Shape parameter the Weibull distribution (β)
9	EventWeibull/location	attribute	Duration	End of failure free time (γ)

The scale (α), shape (β) and location (γ) are used as parameters in the cumulative distribution function

$$F_w(x) = \begin{cases} 1 - \exp\left(-\left(\frac{x-\gamma}{\alpha}\right)^{\beta}\right) & \text{if } x \geq \gamma \\ 0 & \text{if } x < \gamma \end{cases} \tag{4.7}$$

and in the generalised inverse distribution function

$$Q_w(u) = \gamma - \alpha \ln(1-u)^{1/\beta}. \tag{4.8}$$

Equations (4.7) and (4.8) reduce to basic two parameter Weibull distribution functions if the location parameter $\gamma = 0$. If also the shape parameter $\beta = 1$, the Weibull distribution reduces to an exponential distribution.

Listing 4.2 shows the redefinition of the getEventTime() function, which uses the "scale", "shape" and "location" attribute values as parameters.

```
double EventWeibull.getDelayTime() {
  return THIS.location + THIS.scale *
      Math.pow(-Math.log(1 - RANDOM.prob()), 1 / THIS.shape);
}
```

Listing 4.2 The redefinition of the delay function for the EventWeibull class

4.1.4 Normal distribution

Normal distribution is defined with mean (μ) and standard deviation (σ) parameters. Table 4.5 presents the declaration of the `EventNorm` class attributes.

Table 4.5 The attribute declaration of the normal distributed events

	A	B	C	COMMENTS
10	EventNorm/mean	attribute	Duration	Mean of the normal distribution (μ)
11	EventNorm/dev	attribute	Number	Standard deviation of the normal distribution (σ)

The random variable is defined with the generalised Box-Muller transform [83], which uses two independent random variables. The mean (μ) and standard deviation (σ) parameters are in the generalised inverse distribution function

$$Q_{norm}(u_1, u_2) = \mu + \sigma\sqrt{-2\ln u_1}\cos(2\pi u_2). \tag{4.9}$$

Listing 4.3 shows the redefinition of the `getEventTime()` function, which uses the "mean" and "dev" attribute values as parameters.

```
double EventNorm.getDelayTime() {
  return THIS.mean + THIS.dev * Math.sqrt(-2 * Math.log(RANDOM.prob())) *
      Math.cos(2 * Math.PI * RANDOM.prob());
}
```

Listing 4.3 The redefinition of the delay function for the EventNorm class

4.1.5 Log-normal distribution

Log-normal distribution is defined with scale (α) and shape (β) parameters. Additionally, a user can define an activation free time or location (γ), which fixes the point in time from which the activations begin to occur. Table 4.6 presents the declaration of the `EventLognorm` class attributes.

Table 4.6 The attribute declaration of the log-normal distributed events

	A	B	C	COMMENTS
12	EventLognorm/scale	attribute	Duration	Scale parameter of the log-normal distribution (μ)
13	EventLognorm/shape	attribute	Number	Shape parameter of the log-normal distribution (σ)
14	EventLognorm/location	attribute	Duration	End of activation free time (γ)

The random variable is defined by using the Equation (4.9). The scale α, shape β and location γ are used as parameters in the generalised inverse distribution function

$$Q_{lognorm}(u_1, u_2) = \gamma + \exp(Q_{norm}(u_1, u_2)), \tag{4.10}$$

where the scale (α) of the log-normal is the mean (μ) parameter of the normal distribution, and similarly the shape (β) of the log-normal is the standard deviation (σ) parameter of the normal distribution.

Listing 4.4 shows the redefinition of the `getEventTime()` function, which uses the "scale", "shape" and "location" attribute values as parameters.

```
double EventLognorm.getDelayTime() {
  return THIS.location +  Math.exp(THIS.scale + THIS.shape *
      Math.sqrt(-2 * Math.log(RANDOM.prob())) *
      Math.cos(2 * Math.PI * RANDOM.prob()));
}
```

Listing 4.4 The redefinition of the delay function for the EventLognorm class

4.1.6 History data fitting

Instead of estimating the parameters directly, the distribution can be defined by using history data fitting. Table 4.7 presents the declaration of the `EventHistory` class attributes, which can be used for storing the history data. In addition to the event times, a weight and a type can be stored to each history data value. The weight allows defining several similar data values by using one definition. The type enables including also other information in the data than the basic activation times. For example, if there is information that an item has been in use for certain time without a failure, the type attribute can be used to indicate that the assigned data value is different than the basic failure time data. The `EventHistory` class contains also a prototype event that implements the fitting.

Table 4.7 The attribute declaration of the history data fitting events

	A	B	C	COMMENTS
15	EventHistory/data	attribute	Duration	Event time values
16	EventHistory/weight	attribute	Number	Weights for history data values if needed
17	EventHistory/weight	=	1	Default weight of each history data value is 1
18	EventHistory/type	attribute	Number	Types for history data values if needed
19	EventHistory/type	=	1	Default type of each history data value is 1
20	EventHistory/fitting	prototype	Event	The event that implements the fitting

As an example of a history data, Figure 4.1 shows a result of a year long reliability test where two items survived the testing. Table 4.8 presents the history data in a data matrix. The two similar data values are merged to a data row with weight value 2. The type is 1 if the data defines a time to failure. The type 0 indicates that the data gives information about the use of the item without a failure.

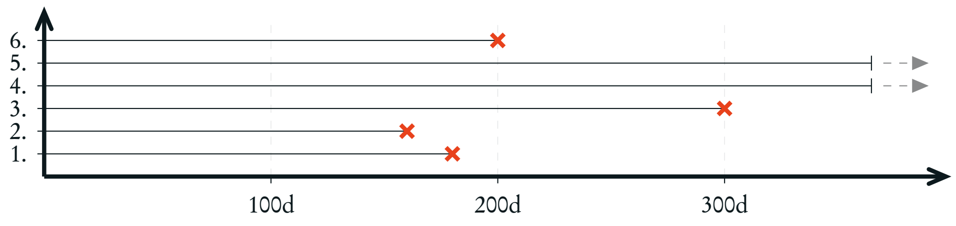

Figure 4.1 A result of a year long reliability test

Table 4.8 The failure history from the Figure 4.1 in a data matrix

Item	Time	Weight	Type
1.	180d	1	1
2.	160d	1	1
3.	300d	1	1
4. & 5.	1a	2	0
6.	200d	1	1

Table 4.9 shows the creation of a `EventHistory` and the assignment of the data of Figure 4.1 by using the Triplets data format and the attributes of the `EventHistory` class. The keyword ”add” is used for assignment, which permits the adding of several comma separated values by using only one definition row.

Table 4.9 An example of an assignment of a fitting from history data

	A	B	C	Comments
1	/exampleHist	instance	EventHistory	Create a history event instance
2	/exampleHist/fitting	include	EventWeibull	Use Weibull fitting
3	/exampleHist/data	add	180d, 160d, 300d, 1a, 200d	List of history duration values
4	/exampleHist/weight	add	1,1,1,2,1	Weights for the history data values
5	/exampleHist/type	add	1,1,1,0,1	Types for the history data values

The history data is fitted to a distribution when the simulation starts. The class of the ”fitting” attribute event defines the used fitting algorithm. Listing 4.5 presents the call of the procedure `fitFromData()` for the fitting event. There exist various algorithms to fit a data to different distributions. For example, with a two parameter Weibull fitting, the history data results in a distribution with scale parameter value 332 days and shape value 2.375 [84]. The definition of the distribution parameters from the history data can also be done externally with a separate tool.

```
void EventHistory.simulationStarted() {
  THIS.fitting.fitFromData();
}
```

Listing 4.5 The fitting of the history data when the simulation starts

After the fitting, the `EventHistory` class can use the event from the attribute ”fitting” to get the delay times for the event activations. Listing 4.6 shows the redefinition of the `getEventTime()` function.

```
double EventHistory.getDelayTime() {
  return THIS.fitting.getDelayTime();
}
```

Listing 4.6 The redefinition of the delay function for the EventHistory class

4.2 Advanced Fault Tree Analysis modelling technique

The advanced Fault Tree Analysis (FTA) modelling technique [19] is an extended version of the standard FTA [2]. The advanced FTA extends traditional FTA with modelling of repair (Section 4.2.1), delays and alternative consequence (4.2.2), cost risk (4.2.3), mode-dependent events (4.2.4) and maintenance actions (4.2.5).

4.2.1 Advanced FTA: Basic features

The advanced FTA technique models consist of fault nodes, which have two states (normal, fault) and two events (failure, restoration). Table 4.10 presents the declaration of the `Fault` node class. These declarations extend the basic `Node` class attributes (see Table 3.18 on page 56).

Table 4.10 The Fault node declaration of the advanced FTA modelling technique

	A	B	C	COMMENTS
1	Fault	class	Node	Node with only two states and two events
2	Fault/normal	prototype	State	Normal state of the node
3	Fault/fault	prototype	State	Fault state of the node
4	Fault/failure	prototype	Event	Event from normal to fault
5	Fault/restoration	prototype	Event	Event from fault to normal
6	Fault/failure/source	connect	Fault/normal	Connect source state to event
7	Fault/failure/target	connect	Fault/fault	Connect event to target state
8	Fault/restoration/source	connect	Fault/fault	Connect source state to event
9	Fault/restoration/target	connect	Fault/normal	Connect event to target state
10	Fault/initialState	connect	Fault/normal	Start from a normal state

Figure 4.2 illustrates prototype structure of fault nodes. The state of the node changes between ”fault” and ”normal”. An event from ”fault” to ”normal” is ”failure” and the event to opposite direction is ”restoration”.

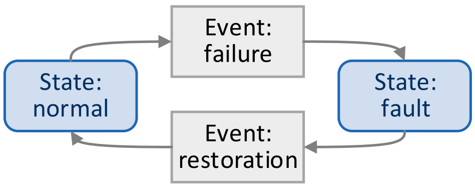

Figure 4.2 Fault nodes have two states (normal and fault) and events (failure and restoration)

Gate operators define the connection logic between the fault nodes. Table 4.11 presents the declaration of the `Gate` operator class and various sub classes. Each logic rule is implemented by a sub class. More gate classes can be created for special needs, such as implementing priority AND and other dynamic gates.

Table 4.11 The declaration of Gate operators of the advanced FTA modelling technique

	A	B	C	COMMENTS
11	Gate	class	Operator	Operator that handles Boolean logic rules
12	OR	class	Gate	Logic operator for rule ”At least one”
13	AND	class	Gate	Logic operator for rule ”All”
14	Vote	class	Gate	”At least m” or ”k out of n” (m = n-k+1)
15	Vote/atLeast	attribute	Integer	The number child faults needed at least
16	XOR	class	Gate	Logic operator for rule ”Exactly one”
17	Limits	class	Gate	Rule ”At least m but at most h”
18	Limits/atLeast	attribute	Integer	The number of children needed at least
19	Limits/atMost	attribute	Integer	The number of children allowed at most
20	Never	class	Gate	Logic operator for rule ”Never”
21	Always	class	Gate	Logic operator for rule ”Always”
22	Cond	class	Gate	Logic operator for a probability condition
23	Cond/prob	attribute	Probability	Probability of the parent after child
24	Ignore	class	Gate	Gate is ignored in simulation

Connections between `Fault` and `Gate` elements are made by using ”child” and ”parent” attributes (see Table 3.19 on page 57). The connection attributes are symmetric (see Table 3.20 on page 58), which means that the connection is always created to both directions when either one of the attributes is assigned.

The analysis of the advanced FTA technique is configured by implementing the template methods of the simulation algorithm core (see Figure 3.6 on page 74). The handling of state changes was presented in Section 3.3.4. Each state change creates an activation of the container node (see Listing 3.14 on page 80). The activation of `Fault` nodes creates new activations for the connected ”parent” `Gate` elements. The procedure code of the `activate()` method is presented for `Fault` class in Listing 4.7.

```
void Fault.activate() {
  for (Element parent : THIS.getElements("parent")) {
    ACTIONS.add(parent);
  }
}
```

Listing 4.7 The activation of fault nodes

The procedure code of the `activate()` template method is presented for `Gate` class in Listing 4.8. The gates contain a logic rule, which defines the update that is needed for the ”parent” `Fault` elements. A template method `checkTrue()` is defined by the sub classes of the gate. The procedures `toTrue()` and `toFalse()` change the state of the parent nodes.

```
void Gate.activate() {
  if (THIS.checkTrue()) {
    for (Element parent : THIS.getElements("parent")) {
      parent.toTrue();
    }
  }
  else {
    for (Element parent : THIS.getElements("parent")) {
      parent.toFalse();
    }
  }
}
```

Listing 4.8 The activation of gate operators

The fault and gate activations are repeated until the top node of the fault tree is reached, or when no changes are needed. Listing 4.9 presents the codes of the `toTrue()` and `toFalse()` procedures.

```
void Fault.toTrue() {
  if (THIS.currentState != THIS.fault) { // change only if needed
    if (THIS.currentState != null) {  // not done at round start
      THIS.currentState.sendEndMessages(); // send end messages of the state
      THIS.failure.sendMessages(); // send also event messages
    }
    THIS.currentState = THIS.fault;
    THIS.currentState.sendStartMessages();
    THIS.currentState.activationCount++;
    ACTIONS.add(THIS); // notice also the connected gates about the change
  }
}

void Fault.toFalse() {
  if (THIS.currentState != THIS.normal) { // change only if needed
    if (THIS.currentState != null) {  // not done at round start
      THIS.currentState.sendEndMessages(); // send end messages of the state
      THIS.restoration.sendMessages(); // send also event messages
    }
    THIS.currentState = THIS.normal;
    THIS.currentState.sendStartMessages();
    THIS.currentState.activationCount++;
    ACTIONS.add(THIS); // notice also the connected gates about the change
  }
}
```

Listing 4.9 The change of a fault node state

The changing of the state is made directly, which means that the use of `Gate` operators overrides the basic state activations of nodes (see Listing 3.14 on page 80). The update of the statistics variable and the message sending are made in the procedure like in basic state activation, but the action creation of the target events is skipped. The skipping of irrelevant procedures improves the efficiency of the simulation process.

The Boolean `checkTrue()` functions of gates use the same function of fault nodes to check the state of each "child" node. Listing 4.10 presents code of the function, which returns true if "fault" state is currently active. Special `Fault` node classes with more that two states can redefine this template method.

```
boolean Fault.checkTrue() {
  return THIS.currentState == THIS.fault;
}
```

Listing 4.10 The checking of a fault node state

As an example, Listings 4.11 – 4.14 present the implementation of the function `checkTrue()` for `OR`, `AND`, `Vote` and `Cond` gates. The algorithms check the states of all "child" nodes and the logic rule of the gate. Other `Gate` sub classes with different logic rules can be configured similarly.

```
boolean OR.checkTrue() {
  for (Element child : THIS.getElements("child")) {
    if (child.checkTrue()) {
      return true;
    }
  }
  return false;
}
```

Listing 4.11 The logic rule check for the OR gate

```
boolean AND.checkTrue() {
  for (Element child : THIS.getElements("child")) {
    if (!child.checkTrue()) {
      return false;
    }
  }
  return true;
}
```

Listing 4.12 The logic rule check for the AND gate

```
boolean Vote.checkTrue() {
  int count = 0;
  for (Element child : THIS.getElements("child")) {
    if (child.checkTrue()) {
      count++;
      if (count >= THIS.atLeast) {
        return true;
      }
    }
  }
  return false;
}
```

Listing 4.13 The logic rule check for the Vote gate

```
boolean Cond.checkTrue() {
  for (Element child : THIS.getElements("child")) {
    if (child.checkTrue()) {
      return RANDOM.prob() < THIS.prob;
    }
  }
  return false;
}
```

Listing 4.14 The logic rule check examples for the Cond gate

The basic analysis result of advanced FTA technique models can be obtained from the `State` class result values. For example, the MTTF can be calculated by dividing the "activeTime" of the "fault" state by its "activationCount". Similarly, the unavailability is calculated by dividing the "activeTime" by the total simulated time. The calculation of basic result values was presented in Section 3.3.5.

4.2.2 Advanced FTA: Delays and alternative consequences

The basic FTA connections are immediate. Table 4.12 presents the declaration of new `Gate` sub classes for inclusion of `Delay` connections. Instead of updating the connected nodes directly, an action is created to a `Concequence` element, which activates later the target nodes of the original gate.

Listing 4.15 configures the activation handling of the `Delay` gate. The procedure overwrites the basic `Gate` template method (see Listing 4.8 on page 99). Like events, delay gates use the `getDelayTime()` function to get the delay of the created action. This enables defining stochastic delays by declaring sub classes, such as `DelayExp` and `DelayWeibull`, which redefine the function (see Section 4.1).

Table 4.12 The declaration of the Delay gate

	A	B	C	COMMENTS
1	Delay	class	OR	Logic operator for a delay
2	Delay/delay	attribute	Duration	The delay after a source node fault
3	Consequence	class	Gate	A special element for activation after the delay
4	Delay/conseq	prototype	Consequence	An element is create for each delay gate

```
void Delay.activate() {
  if (THIS.checkTrue()) {
    ACTIONS.add(CALCULATION.currentTime + THIS.getDelayTime(),
        THIS.conseq);
  }
  else {
    for (Element parent : THIS.getElements("parent")) {
      parent.toFalse();
    }
  }
}

double Delay.getDelayTime() {
  return THIS.delay;
}
```

Listing 4.15 The activation of delay gates

Listing 4.16 configures the activation of the Consequence element, which is similar to the procedure of a basic Gate class (see Listing 4.8 on page 99), but the activated nodes are parents of the container delay gate. The activation is made only if the logic rule is still true. The Delay extends OR gate, which implements the checkTrue() template method (see Listing 4.11 on page 100) for handling of the situations where more than one "child" nodes are connected for the delay gate.

```
void Consequence.activate() {
  if (THIS.getContainer().checkTrue()) {
    for (Element parent : THIS.getContainer().getElements("parent")) {
      parent.toTrue();
    }
  }
}
```

Listing 4.16 The activation of a consequence of delay gates

Modelling of alternative consequences is another special situation that handling is not possible with basic gates. By connecting a fault node to two or more gate operators, it is possible to model the situation where an event has multiple consequences. For example, a common cause failure is an event that causes two or more items to fail. However, this approach is not suitable for handling a situation where the consequences are exclusive. Table 4.13 declares `Select` gate, which is used for modelling of the situation where only one of the many consequences is selected to occur.

Table 4.13 Declaration of the Select gate for modelling of alternative consequences

	A	B	C	COMMENTS
5	Select	class	OR	A special gate for handling of alternative consequences
6	Select/prob	attribute	Probability	Probabilities of different consequences

The array definition is used to assign probabilities of each consequence. Each ”parent” connection has its own probability. The keys of the array definition are used to connect the parent node to the probability. Table 4.14 illustrates how probabilities can be assigned for to connected parent nodes.

Table 4.14 An example of the alternative consequence definition

	A	B	C	COMMENTS
1	parentNodeA/child[a]	connect	selectGate	Parent node A is connected to a select gate
2	parentNodeB/child[b]	connect	selectGate	Parent node B is connected to a select gate
3	selectGate/prob[a]	=	0.6	Parent node A has probability 0.6
4	selectGate/prob[b]	=	0.4	Parent node B has probability 0.4

The sum of the probabilities must equal to 1.0. If this is not the case, the values are considered as weights of the consequences. Listing 4.17 shows a procedure to ensure that the probabilities are proper.

```
void Select.simulationStarted() {
  double sum = 0;
  for (Element parent : THIS.getElements("parent")) {
    sum += THIS.prob[parent.getKey()];
  }
  if (sum != 1.0) {
    for (Element parent : THIS.getElements("parent")) {
      THIS.prob[parent.getKey()] /= sum;
    }
  }
}
```

Listing 4.17 The reset of the consequence probabilities

Listing 4.18 configures the activation handling of the `Select` gate. The procedure overwrites the basic `Gate` template method (see Listing 4.8 on page 104). A random probability number is used to select the "parent" node that is activated.

```
void Select.activate() {
  if (THIS.checkTrue()) {
    double prob = RANDOM.prob();
    for (Element parent : THIS.getElements("parent")) {
      prob -= THIS.prob[parent.getKey()];
      if (prob < 0) { // select this parent
        parent.toTrue();
        return;
      }
    }
  }
  else {
    for (Element parent : THIS.getElements("parent")) {
      parent.toFalse();
    }
  }
}
```

Listing 4.18 The activation of select gates

4.2.3 Advanced FTA: Cost risk analysis

Cost can be associated to the number of state activations and to the duration a state is activate. The cost risk analysis can assess both negative and positive impacts. The cost of a repair each time a failure occurs, or the loss of a production associated to the system downtime are examples of negative impacts. Positive impact can be, for example, the revenue of the production. The cost definitions allow estimating the risk, which is the mean cumulative cost caused by the state activations and active times during the analysis period. In addition to the mean result, the distribution of risk should be calculated to understand the probabilities of high (or low) costs.

The costs are defined by using the attributes that are declared in Table 4.15. Different type of risks can be modelled by using the array definition, where the name of the cost type is given as the key of the array attribute. All cost definition values must use the same currency or other cost unit. This thesis uses Euro (€) as a cost unit.

The cost attribute values are combined with statistics variables to calculate the risk results. In the simplest case, the mean total cost result is calculated for each

Table 4.15 The declaration of the cost attributes for the advanced FTA modelling technique

	A	B	C	COMMENTS
1	State/costCount	attribute	Number	A cost for each time the event activates
2	State/costTime	attribute	Rate	A cost per time unit the state is active

`Fault` node. Table 4.16 presents the declaration of the result value. The costs of the ”normal” state are considered positive, and the costs of the ”fault” state negative. Listing 4.19 presents the calculation of the cost risk result. The total cost of the whole simulation is divided by the number of simulated round to get the mean cost risk.

Table 4.16 The declaration of a cost risk result value

	A	B	C	COMMENTS
3	Fault/meanCost	attribute	Number	The mean total cost of a fault node

```
void Fault.createResult() {
  THIS.meanCost = 0;
  for (Number cost : THIS.normal.getNumbers("costCount")) {
    THIS.meanCost += cost * THIS.normal.activationCount;
  }
  for (Number cost : THIS.normal.getNumbers("costTime")) {
    THIS.meanCost += cost * THIS.normal.activeTime;
  }
  for (Number cost : THIS.fault.getNumbers("costCount")) {
    THIS.meanCost -= cost * THIS.fault.activationCount;
  }
  for (Number cost : THIS.fault.getNumbers("costTime")) {
    THIS.meanCost -= cost * THIS.fault.activeTime;
  }
  THIS.meanCost /= CALCULATION.currentRound;
}
```

Listing 4.19 The calculation of the cost risk results for fault nodes

For more detailed cost results, a new cost statistics variable can be added and updated during the simulation process. The procedure of the update would be added to the `toTrue()` and `toFalse()` template methods (see Listing 4.9 on page 99). It is possible to calculate quantiles and distributions of the cost risk by using the generic result calculation (see Section 3.4.2) for the cost statistics variable.

4.2.4 Advanced FTA: Mode-dependent events

The delay distributions can change based on a mode. Modes are defined by certain active states (see Table 3.22 on page 59). This situation is called *mode-dependency*. Table 4.17 presents the declaration of a special `EventMode` class, which handles the mode-dependent failure rates.

Table 4.17 The declaration of the EventMode class for handling of mode-dependency

	A	B	C	COMMENTS
1	EventMode	class	Event	A special class for handling of mode-dependency
2	EventMode/waitTime	attribute	Duration	An attribute for storing the event time in wait mode
3	EventMode/isWaiting	attribute	Boolean	Value is true if the mode is not active

Table 5.12 on page 135 presents an example of defining a mode-dependent failure rate. Generally, the idea is that there can be more than one "failure" events from "normal" to "fault" state, which are active based on the operating mode. The "restoration" event can be mode-independent. Figure 4.3 illustrates a structure of a mode-dependent fault node.

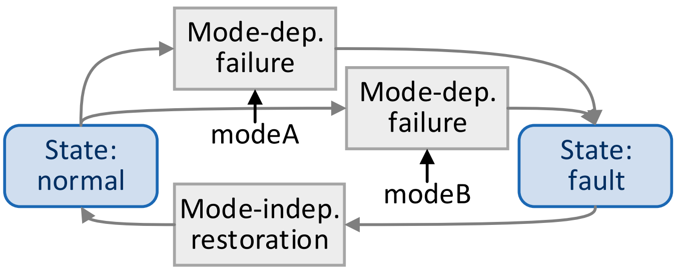

Figure 4.3 A fault node with more than one mode-dependent failure events

The template methods `modeEnd()` and `modeStart()` are called after a change of a mode (see Listings 3.15 and 3.16 on page 80). Listing 4.20 presents the procedure codes of the methods for the `EventMode` class. If the event is not in use, the mode change only updates the value of the "isWaiting" attribute. Otherwise an action is either removed from or added to the `ACTIONS` queue.

```
void EventMode.modeEnd() {
  Action future = ACTIONS.remove(THIS);
  if (future != null) { // store the time to activation if action exist
    THIS.waitTime = future.getTime() - CALCULATION.currentTime;
  }
  THIS.isWaiting = true;
}

void EventMode.modeStart() {
  if (THIS.waitTime > 0) { // add new action if wait time is defined
    ACTIONS.add(CALCULATION.currentTime + THIS.waitTime, THIS);
    THIS.waitTime = 0;
  }
  THIS.isWaiting = false;
}
```

Listing 4.20 The procedure code of the end and start of a mode

The event is in use only when its ”source” state is active. The `getEventTime()` is called from `createNextAction()` procedure (see Listing 3.17 on page 81) when the event is taken into use. This creates a future action and adds it to the `ACTIONS` queue. Listing 4.21 presents the redefined `getEventTime()` function of the `EventMode` class. If ”isWaiting” is true, the delay time is stored to ”waitTime” attribute. The delay time is used for creating an action when the mode is later started. A positive infinity is returned if the mode is not active, which skips the action creation.

```
double EventMode.getEventTime() {
  if (THIS.isWaiting) { // if currently waiting, store the delay to waitTime
    THIS.waitTime = THIS.getDelayTime();
    return Double.POSITIVE_INFINITY;
  }
  return CALCULATION.currentTime + THIS.getDelayTime();
}
```

Listing 4.21 The redefined method to get mode-dependent event time

A reset of the ”isWaiting” needs to be made when a simulation round starts. If a ”listener” has not been defined or if a complement mode (see Rule 3.7 on page 59) is listened, the mode is active when a round starts. Listing 4.22 presents the procedure code of the `roundStarted()` template method.

```
void EventMode.roundStarted() {
  THIS.isWaiting =
      THIS.listener != null && !THIS.listener.endsWith("Not");
}
```

Listing 4.22 A reset of the mode state when a simulation round starts

4.2.5 Advanced FTA: Maintenance actions

Maintenance actions can be included in advanced FTA models. Table 4.18 declares an `Maintenance` class, which handles different type of service operations. In AoT, a maintenance is a special type of element, which has a time interval and a cost of its activation. The start and end times of the maintenance define when the interval is active.

Table 4.18 The declaration of the Maintenance class for handling of service operations

	A	B	C	COMMENTS
1	Maintenance	class	Element	A special element to model maintenance actions
2	Maintenance/interval	attribute	Duration	The maintenance interval (e.g. once per year)
3	Maintenance/interval	=		Default is infinity so without defining there is no actions
4	Maintenance/start	attribute	Duration	The first time to make the maintenance the action
5	Maintenance/start	=		By default the interval defines the start time
6	Maintenance/end	attribute	Duration	The last time to make the maintenance action
7	Maintenance/end	=		By default there is no end limit
8	Maintenance/cost	attribute	Number	The cost of the action

Listing 4.23 presents the procedure of the `roundStarted()` method, which creates the first maintenance actions at the beginning of each simulation round. If the ”start” value is not defined, the ”interval” defines the time of the first action. The default value is positive infinity, which means that the attribute value has not been assigned. Either the ”start” or ”interval” needs to be defined to create the first action.

```
void Maintenance.roundStarted() {
  if (THIS.start == Double.POSITIVE_INFINITY) {
    ACTIONS.add(THIS.interval, THIS);
  }
  else {
    ACTIONS.add(THIS.start, THIS);
  }
}
```

Listing 4.23 The definition of the roundStart() template method of the Maintenance class

Listing 4.24 presents the procedure of the `activate()` template method, which handles the maintenance action. The time of the next event is defined, if the end limit is not yet reached. The template method `handleMaintenance()` is implemented by sub classes.

```
void Maintenance.activate() {
  THIS.handleMaintenance(); // defined by sub classes
  if (CALCULATION.currentTime + THIS.interval <= THIS.end) {
    ACTIONS.add(CALCULATION.currentTime + THIS.interval, THIS); // add next
  }
}
```

Listing 4.24 The handling of a maintenance action

Table 4.19 declares sub classes of the `Maintenance` class. Different type of actions have their own attributes to define the effect of the maintenance action. The AoT approach includes following classes to model different type of maintenance actions:

- `Preventive` maintenance is carried out to reduce the probability of failure. The maintenance causes that the ageing of the component changes according to an effect factor. For example, with value 0.75 the element ages 75 % of the calendar time, which means that the time to next failure grows by a factor of 1/0.75. If the interval or the effect of a maintenance action changes, a new factor value must be assigned.
- `Inspection` can detect symptoms of developing failures. If the inspection is made during the symptom time, it has certain probability to detect the developing failure and to avoid it. When compared to a repair after a failure, the restoration time can be shorter and the cost lower if the failure can be detected and repair started before it occurs.
- `Improvement` is a maintenance that is carried out to reduce the degradation, which could lead to a wear-out or ageing failure. After the improvement, the component is in better condition than before the maintenance action, which postpones the the next failure. The improvement is defined by an effect factor. For example, with a value 0.75, the component is 75 % ”younger” than before the maintenance action.
- `Replacement` replaces the element to be as good as new. The restoration time of a scheduled replacement can be shorter and the cost lower when compared to a repair after a failure.
- `Finding` detects and repairs latent failures, which are not found in normal operations. A probability is defined to define if the finding can detect an existing latent failure when the action is made.

Table 4.19 The declaration of the maintenance action sub classes

	A	B	C	COMMENTS
9	Preventive	class	Maintenance	Preventive maintenance without any special action
10	Preventive/effect	attribute	Number	Ageing effect factor
11	Preventive/effect	=	1.0	Default factor is 1.0 (no effect)
12	Inspection	class	Maintenance	Detection of symptoms to avoid a failure
13	Inspection/symptom	attribute	Duration	Time before the failure when detection is possible
14	Inspection/symptom	=		By default inspection finds all faults
15	Inspection/prob	attribute	Probability	The probability that the inspection is successful
16	Inspection/prob	=	1.0	By default inspection is always successful
17	Improvement	class	Maintenance	Improves the condition of the component
18	Improvement/effect	attribute	Number	How much the action affects to the component age
19	Improvement/effect	=	1.0	Default factor is 1.0 (no effect)
20	Improvement/minAge	attribute	Duration	Minimum age to make the improvement
21	Improvement/minAge	=	0	No minimum age limit by default
22	Replacement	class	Maintenance	Replace the element to be as good as new
23	Replacement/minAge	attribute	Duration	Minimum age to make the replacement
24	Replacement/minAge	=	0	No minimum age limit by default
25	Finding	class	Maintenance	Finding of hidden failures
26	Finding/prob	attribute	Probability	The probability to find the hidden fault
27	Finding/prob	=	1.0	By default finding action finds fault always

The maintenance actions are added as attributes of a special `FaultMaint` class. It includes `EventMaint` for ”failure” event to handle maintenance effects. The ”factor” attribute models the changes to the failure rate. The declaration of the classes is presented in Table 4.20. It would be also possible to declare a special `Fault` class that contains a ”service” state. Like the down state ”fault” is active during the restoration after a failure, the ”service” state is active during the maintenance actions.

Table 4.20 The declaration of the FaultMaint node class

	A	B	C	COMMENTS
28	FaultMaint	class	Fault	Special fault node that can handle maintenance actions
29	FaultMaint	container	Maintenance	Maintenance fault nodes are containers of actions
30	EventMaint	class	Event	Special event that can handle maintenance actions
31	FaultMaint/failure	include	EventMaint	Use special event for failure
32	EventMaint/factor	attribute	Number	An effect factor from preventive maintenance

Preventive maintenance defines its effect by updating a factor that changes the failure times. Listing 4.25 presents the reset of the EventMaint ”factor” attribute, and how the EventMaint overwrites the getEventTime() function (see Listing 3.18 on page 82) to include the factor on event creation.

```
void EventMaint.simulationStarted() {
  THIS.factor = 1; // reset the factor before checking actions
}

void Preventive.simulationStarted() {
  THIS.getContainer().failure.factor *= THIS.effect; // failure event factor
}

double EventMaint.getEventTime() {
  return CALCULATION.currentTime + THIS.getDelayTime() / THIS.factor;
}
```

Listing 4.25 Including the effect factor in event time

Other Maintenance sub classes implement the handleMaintenance() template method to define the effect of the maintenance actions. Listing 4.26 implements the method for Inspection action. If a future action exists for the failure event of the node, the symptom time and probability of success is checked. If the failure is prevented, a new event is created. Otherwise the same future event is added back.

```
void Inspection.handleMaintenance() {
  Action future = ACTIONS.remove(THIS.getContainer().failure);
  if (future != null) { // if a future failure action exists
    if (future.getTime() - CALCULATION.currentTime < THIS.symptom &&
        RANDOM.prob() < THIS.prob) { // inspection prevents failure
      THIS.getContainer().failure.createAction(); // create new failure
    }
    else { // add the failure back
      ACTIONS.add(future.getTime(), future.getTarget());
    }
  }
}
```

Listing 4.26 The handling of an inspection action

Listing 4.27 presents the procedure code of the handleMaintenance() template method of the Improvement action. The item is improved only if certain minimum age is reached. The check is made with the help of a ”activationStart” attribute (see Table 3.30 on page 85) of the ”normal” state, which is updated each time a ”failure” action is created. The time of the found future action is updated based on the ”effect” factor.

```
void Improvement.handleMaintenance() {
  if (CALCULATION.currentTime -
      THIS.getContainer().normal.activationStart > THIS.minAge) {
    Action future = ACTIONS.remove(THIS.getContainer().failure);
    if (future != null) { // if a future failure action exists
      ACTIONS.add((future.getTime() - CALCULATION.currentTime) /
          THIS.effect + CALCULATION.currentTime, future.getTarget());
    }
  }
}
```

Listing 4.27 The handling of an improvement action

Listing 4.28 presents the implementation of the `handleMaintenance()` template method for the `Replacement` action. The procedure is similar to `Improvement`, but instead of updating the action time, a new failure is created.

```
void Replacement.handleMaintenance() {
  if (CALCULATION.currentTime -
      THIS.getContainer().fault.activationStart > THIS.minAge) {
    Action future = ACTIONS.remove(THIS.getContainer().failure);
    if (future != null) { // if a future failure action exists
      THIS.getContainer().failure.createAction(); // create new failure
    }
  }
}
```

Listing 4.28 The handling of a replacement action

Listing 4.29 implements the method `handleMaintenance()` for `Finding` action. If the node state is currently ”fault” and the finding is successful, a new immediate restoration action is created.

```
void Finding.handleMaintenance() {
  if (THIS.getContainer().currentState == THIS.getContainer().fault &&
      RANDOM.prob() < THIS.prob) {
    ACTIONS.add(THIS.getContainer().restoration);
  }
}
```

Listing 4.29 The handling of a finding action

The cost definition is common for all type of maintenance actions. In simplest case the calculation of costs is done after the simulation ends. By using the intervals and the defined cost of an action, the maintenance costs can be added to the total costs of a `Fault` node (see Listing 4.19 on page 105). For more detailed results, a cost statistics variable needs to be added an updated each time a maintenance action is handled.

4.3 Reliability Block Diagram modelling technique

The Reliability Block Diagram (RBD) [5] modelling technique connects fault nodes to form a diagram of success logic. The structure is created inside a special `RBD` node, which contains predefined start and end nodes. The `RBD` node is at normal state if there is a path from start to end that does not contain nodes that are at fault state. Otherwise the `RBD` node is at fault state. Parallel and serial fault logics are modelled by connecting the fault nodes in the diagram. The redundancy structures are modelled with a specific ”atleast” attribute, which defines the number of source connections required. Table 4.21 declares the RBD modelling technique classes and introduces their attributes.

Table 4.21 The RBD modelling technique

	A	B	C	COMMENTS
1	FaultRBD	class	Fault	A fault node in a RBD
2	RBD	class	FaultRBD	A fault node described by RBD
3	RBD/start	prototype	FaultRBD	Predefined start node for the diagram
4	RBD/end	prototype	FaultRBD	Predefined end node for the diagram
5	RBD	container	FaultRBD	The intermediate fault nodes of the RBD
6	FaultRBD/target	attribute	FaultRBD	An attribute for creating connections
7	FaultRBD/symmetry[target]	=	source	If X is a source of Y then Y is a target of X
8	FaultRBD/atLeast	attribute	Integer	The number of source connections required
9	FaultRBD/atLeast	=	1	By default one source connection is enough

A special `FaultRBD` class is used in the `RBD` to allow defining when the check of state change is needed. Listing 4.30 presents the procedure code of the method `activate()`, which overwrites the original procedure that was defined for `Fault` class (see Listing 4.7 on page 98). The only difference is that each activation calls a `checkState()` template method of the container `RBD` node.

```
void FaultRBD.activate() {
  getContainer().checkState(); // notice the possible RBD node state change
  for (Element parent : THIS.getElements("parent")) {
    ACTIONS.add(parent);
  }
}
```

Listing 4.30 The activation of a RBD fault node

Listing 4.31 presents the procedure code of the RBD node checkState() method. The possible state change is made by using the toTrue() or toFalse() methods of the Fault node (see Listing 4.9 on page 99). They update the RBD node to the ”normal” state if a path exists. Otherwise the RBD node is updated to ”fault” state.

```
void RBD.checkState() {
  if (THIS.pathExists()) {
    THIS.toFalse(); // to normal state
  }
  else {
    THIS.toTrue(); // to fault state
  }
}
```

Listing 4.31 The check of a RBD state

The pathExists() function is presented in Listing 4.32. It uses breadth-first search to check if the RBD model has a path from the ”start” to the ”end” node. At first the ”start” node is added to a search list, which contains all the nodes that are successfully connected to the start node. If the list contains a node that has a connection to the ”target” node, the RBD model path is OK.

```
boolean RBD.pathExists() {
  List<Element> pathOK = new ArrayList<Element>();
  pathOK.add(THIS.start);
  int index = 0;
  while (index < pathOK.size()) {
    for (Element target : pathOK.get(index).getElements("target")) {
      if (target == THIS.end) {
        return true;
      }
      if (!target.checkTrue() && !pathOK.contains(target)) {
        int sourcesCount = 0;
        for (Element source : target.getElements("source")) {
          if (pathOK.contains(source)) {
            sourcesCount++;
          }
        }
        if (sourcesCount >= target.atLeast) {
          pathOK.add(target);
        }
      }
    }
    index++;
  }
}
```

Listing 4.32 The check if a path exists

Before a new node is added to the search list, the `checkTrue()` function of the `Fault` class is used to check the node state (see Listing 4.10 on page 100). The node must also have enough connected `source` nodes. By default only one connected source node is needed, but by using the ”atLeast” attribute it is possible to define that more connections are required.

The classes of the RBD are based on a `Fault` class, which allows including them in advanced RBD models. The result value calculation of a `Fault` node can be used to get basic results. If needed, the `RBD` and `FaultRBD` classes can declare their on statistics variables and configure calculation of special result values. For example, the probability that there exists an OK path that reaches the `FaultRBD` node might be an interesting result value in some situations.

4.4 Function modelling technique

Function modelling technique allows to create an environment for visual programming of mathematical operations. `Variable` and `Function` element classes (see Table 3.16 on page 56) are used for creation of model structures. Attributes ”in” and ”out” are used for connections of variables and functions (see Table 3.19 on page 57).

Table 4.22 declares special `Variable` classes for connecting the function models to attributes of other modelling techniques. There exist also a `Random` variable class, which value is get each time from a random generator, and a `StoreTime` variable class, which can be used to store current simulation time.

Table 4.22 The declaration of the Variable classes for the function modelling technique

	A	B	C	COMMENTS
1	VariableRead	class	Variable	A variable which reads its value from an attribute
2	VariableRead/path	attribute	Path	An attribute path to read the value from
3	VariableWrite	class	Variable	A variable which writes its value to an attribute
4	VariableWrite/path	attribute	Path	An attribute path to write the value to
5	VariableSync	class	Variable	A variable is read from and written to an attribute
6	VariableSync	include	VariableRead	Synchronized variable is read from an attribute
7	VariableSync	include	VariableWrite	Synchronized variable is written to an attribute
8	Random	class	Variable	A random value between 0 and 1
9	StoreTime	class	Variable	A variable that can store the current simulation time

Listing 4.33 presents the `activate()` method of the `Variable` class. The methods `updateIn()` and `updateOut()` update the value from an ”in” connected function, and ask all ”out” functions to make the needed changes after the updated value.

```
void Variable.activate() {
  THIS.updateIn(); // check value of an "in" connected function
  THIS.updateOut(); // update "out" connected functions
}

void Variable.updateIn() {
  if (THIS.getElement("in") != null) { // if "in" function is defined
    THIS.getElement("in").updateIn();  // update its value
    THIS.value = THIS.getElement("in").getValue();  // and assign it
  }
}

void Variable.updateOut() {
  for (Element out : THIS.getElements("out")) {
    out.updateOut(); // update "out" connected functions
  }
}
```

Listing 4.33 The configuration of the handling of a variable action

Listing 4.34 presents the changes that are needed for classes `VariableWrite` and `StoreTime`. In addition to updating the connected functions, the `VariableWrite` class writes the value to the defined attribute. The class `StoreTime` stores the current time before making the updates.

```
void VariableWrite.updateOut() {
  MODEL.setNumber(THIS.path, THIS.value);  // assignment to defined path
  for (Element out : THIS.getElements("out")) {
    out.updateOut();
  }
}

void StoreTime.activate() {
  THIS.value = CALCULATION.currentTime;
  THIS.updateOut();
}
```

Listing 4.34 The changes needed by the special variable classes

Listing 4.35 presents the basic configuration of the `Variable` class value handling. Variables reset the initial value when simulation rounds start. The `getValue()` function is used when the value of the variable is needed. The `VariableRead` and `Random` sub classes rewrite the function to define special value reading and generation.

```
void Variable.roundStarted() {
  THIS.value = THIS.initialValue;
}

double Variable.getValue() {
  return THIS.value;
}

double VariableRead.getValue() {
  return MODEL.getNumber(THIS.path);
}

double RANDOM.getValue() {
  return RANDOM.prob();
}
```

Listing 4.35 The configuration of the value handling of variables

Listing 4.36 presents the `activate()` template method of the `Function` class. The `updateIn()` method updates the ”in” variables before the calculation of the function value. The `updateOut()` method calculates the result value and changes the value of the connected ”out” variables. The updates are not needed if the variable value does not change.

```
void Function.activate() {
  THIS.updateIn(); // update all input variables
  THIS.updateOut(); // calculate value and update all output variables
}

void Function.updateIn() {
  for (Element in : THIS.getElements("in")) {
    in.updateIn(); // update all input variables
  }
}

void Function.updateOut() {
  double result = getValue();  // calculate value
  for (Element out : THIS.getElements("out")) {
    double outValue = out.getValue();
    if (outValue != result) {  // update if changed
      out.value = result;
      out.updateOut();
    }
  }
}
```

Listing 4.36 The configuration of the function value calculation

Table 4.23 presents the declaration of basic built-in `Function` sub classes. Listings 4.37 – 4.40 show how they implement the `getValue()` method. The minuend in `Subtraction` and the dividend in `Division` class procedures are defined by using a special ”in[minuend]” and ”in[dividend]” connections.

Table 4.23 The declaration of basic built-in functions for the function modelling technique

	A	B	C	COMMENTS
10	Addition	class	Function	Built-in sum function (+)
11	Subtraction	class	Function	Built-in difference function (−)
12	Multiplication	class	Function	Built-in product function (∗)
13	Division	class	Function	Built-in quotient function (/)

```
double Addition.getValue() {
  double result = 0;
  for (Element in : THIS.getElements("in")) {
    result += in.getValue();
  }
  return result;
}
```

Listing 4.37 The configuration of the value calculation for addition

```
double Subtraction.getValue() {
  double result = 0;
  for (Element in : THIS.getElements("in")) {
    if (in.getKey().equals("minuend")) {
      result += in.getValue();
    }
    else {
      result -= in.getValue();
    }
  }
  return result;
}
```

Listing 4.38 The configuration of the value calculation for subtraction

```
  double Multiplication.getValue() {
  double result = 1;
  for (Element in : THIS.getElements("in")) {
    result *= in.getValue();
  }
  return result;
}
```

Listing 4.39 The configuration of the value calculation for multiplication

```
double Division.getValue() {
  double result = 1;
  for (Element in : THIS.getElements("in")) {
    if (in.getKey().equals("dividend")) {
      result *= in.getValue();
    }
    else {
      result /= in.getValue();
    }
  }
  return result;
}
```

Listing 4.40 The configuration of the value calculation for division

Corresponding assignment operators are declared in Table 4.24. As presented for `AddAssignment` in Listing 4.41, their first result value is obtained from an ”out” connected variable. The assignment operators help avoiding loops in function models, which simplifies the update algorithms. A new `Function` class and the `getValue()` template method can be defined similarly for any mathematical operation.

Table 4.24 The declaration of basic built-in assignment functions

	A	B	C	COMMENTS
14	AddAssignment	class	Addition	Built-in sum assignment (+=)
15	SubAssignment	class	Subtraction	Built-in difference assignment (−=)
16	MulAssignment	class	Multiplication	Built-in product assignment (*=)
17	DivAssignment	class	Division	Built-in quotient assignment (/=)

```
double AddAssignment.getValue() {
  double result = THIS.getElement("out").getValue();
  for (Element in : THIS.getElements("in")) {
    result += in.getValue();
  }
  return result;
}
```

Listing 4.41 The configuration of the value calculation for addition assignment

Listing 4.42 presents how the `activate()` template method is called for all elements of `VariableWrite` class when the simulation starts. This allows using function models for calculation and update of attribute values before they are used in the simulation. The activation updates the value from an ”in” connected operator and assigns it to the defined attribute (see Listings 4.33 and 4.34 on page 116).

```
void VariableWrite.simulationStarted() {
  THIS.activate();
}
```

Listing 4.42 The calculation of the function values when the simulation starts

By adding a ”listener” (see Table 3.22 on page 59) for a `Variable` element, it is possible to update the values during the simulation process. A message needs to be created when the attribute value is updated. The listener notices this message and updates the change to all connected functions.

The calculation of basic function result values is made with the help of the generic statistics handling (see Section 3.4.2). New `Variable` classes can be declared if certain special result values are needed. Table 4.25 declares a `VariableResult` class, which presents results related to the variable value at each round end. A `VariableRate` class is declared for generic production results.

Table 4.25 Declaration of variable that collects result values

	A	B	C	COMMENTS
18	VariableResult	class	Variable	A variable which round end result value is shown
19	VariableRate	class	Variable	A variable models a rate of a general production
20	VariableRate/production	attribute	Number	The production result values

Listing 4.43 shows how the ”value” and ”production” attributes are added to generic statistics handling for the `VariableResult` and `VariableRate` classes. The `stepTaken()` method (see Listing 3.10 on page 77) increases the production based on the current ”value” attribute. This configuration automatically creates various result attributes, such as ”value_MEAN” and ”production_QUANTILE_5”.

```
void VariableResult.simulationStarted() {
  VARIABLES.add(THIS, "value");
}

void VariableRate.simulationStarted() {
  VARIABLES.add(THIS, "production");
}

void VariableRate.stepTaken() {
  THIS.production += CALCULATION.stepLength * THIS.value;
}
```

Listing 4.43 The configuration of the variable result values

4.5 Petri net modelling technique

The Petri net [5] modelling technique contains places and transitions, which are connected with arcs. The number of tokens defines a state of a place, which in AoT are defined as an attribute of a main element class `Place` (see Tables 3.16 and 3.18 on page 56). The upstream and downstream arcs are modelled as ”input” and ”output” connection attributes (see Table 3.19 on page 57) of a `Transition` class. Table 4.26 declares the attributes for inclusion of transition delay, weights and negative logic.

Table 4.26 The declaration of Petri net transition attributes

	A	B	C	COMMENTS
1	Transition/delay	attribute	Duration	Delay before the transition is fired
2	Transition/inputWeight	attribute	Integer	Weights of the input connections
3	Transition/inputWeight	=	1	Default weight is 1
4	Transition/outputWeight	attribute	Integer	Weights of the output connections
5	Transition/outputWeight	=	1	Default weight is 1
6	Transition/inhibit	attribute	Boolean	Possibility to define not logic

A place activation creates transitions, which in timed Petri net are fired after certain delay. Firing of a transition removes a token from ”input” places and adds it to ”output” places. Attributes ”inputWeight” and ”outputWeight” are used if the number of tokens is other than one. The key of the array definition connects the weight to the connected place. A transition is valid if the number of tokens at ”input” connected places is at least or equal to the value of the corresponding ”inputWeight” attribute. If the ”inhibit” is true for the arc, the number of tokens at the source places must be strictly lower than the weight. After the delay, the number tokens of the target places is updated based on the corresponding ”outputWeight” values.

The handling of tokens is analogous with the state changes (see Section 3.3.4). Listing 4.44 presents the configuration of the `roundStarted()` template method for the `Place` class. When a simulation round starts, the initial number of tokens is assigned. A first action is added automatically to start the transition process.

```
void Place.roundStarted() {
  THIS.tokens = THIS.initialTokens; // reset the tokens
  ACTIONS.add(THIS); // create the transitions
}
```

Listing 4.44 The configuration of the round start template method for the Place class

Listing 4.45 presents the `activate()` method of the `Place` class. When a place activates, it creates actions for all "output" connected transitions.

```
void Place.activate() {
  for (Element output : THIS.getElements("output")) {
    output.createAction(); // create new actions for output transitions
  }
}
```

Listing 4.45 The configuration of the activation template method for the Place class

Listing 4.46 configures the validity check for the `createAction()` method of the `Transition` class. If the transition is not valid, the possibly already existing action is removed. Otherwise, if there was not already an action in the future `ACTIONS` queue, the delay time of the transition is used to create a new action.

```
void Transition.createAction() {
  boolean valid = true;
  for (Element input : THIS.getElements("input")) {
    if (THIS.inhibit !=
        (input.tokens < THIS.inputWeight[input.getKey()])) {
      valid = false;
      break;
    }
  }
  Action future = ACTIONS.remove(THIS);
  if (valid) {
    if (future != null) { // use the previously created action if possible
      ACTIONS.add(future.getTime(), THIS);
    }
    else {
      ACTIONS.add(CALCULATION.currentTime + THIS.getDelayTime(), THIS);
    }
  }
}
```

Listing 4.46 Basic delay handling of the Transition class

Listing 4.47 configures the activation of transitions, which occurs when the transition is fired after the delay. An action is created for each updated "output" place to notice the change for the connected transitions.

```
void Transition.activate() {
  for (Element output : THIS.getElements("output")) {
    output.tokens += THIS.outputWeight[output.getKey()];
    ACTIONS.add(output);
  }
}
```

Listing 4.47 The configuration of the situation when a transition is fired

Like events in state change handling (see Section 4.1), and delay gates in advanced FTA (see Listing 4.15 on page 102), the transitions use `getDelayTime()` function to get the delay before the transition is fired. This enables defining stochastic delays with sub classes, such as `TransitionExp` and `TransitionWeibull`. Listing 4.48 presents the basic delay handling of the `Transition` class. The sub classes can redefine the template method to return a stochastic delay time that is based on a certain distribution function.

```
double Transition.getDelayTime() {
  return THIS.delay;
}
```

Listing 4.48 Basic delay handling of the Transition class

Predicates and assertions can be included in transitions by defining them in string format. Array definitions can be used if more than one predicates or assertions are needed. Their functional implementation is not specified in this thesis. The simplest approach for the calculation engine that is presented in this thesis is to create `handleAssertions()` and ”checkPredicates()” methods, which allow defining any procedure codes for the special assertion and predicate rules.

With the help of the generic statistics handling (see Section 3.4.2), the calculation of basic Petri net result values is simple. Listing 4.49 shows how the ”tokens” attribute is added to generic statistics handling. This automatically creates various result attributes, such as ”tokens_MEAN” and ”tokens_QUANTILE_5”.

```
void Place.simulationStarted() {
  VARIABLES.add(THIS, "tokens");
}
```

Listing 4.49 The Petri net tokens attribute is also a statistics variable

4.6 Using other than discrete event simulation tool

Section 3.3 presented a DES tool, which can be configured for analysis of all the modelling techniques that were defined in previous sections of this chapter. However, the discrete event simulation is not always the most suitable approach. AoT enables defining other calculation tools for such cases. This section presents examples of techniques, which use other than the DES tool.

4.6.1 Markov modelling technique

The Markov [6] modelling technique uses a model, which consist of states and state transitions. In AoT a transition is modelled by an `Event` element. The basic state handling (See Section 3.3.4) forms a continuous-time semi-Markov process. Because other than exponential distributions are allowed for delays of state changes, the Markovian property is valid only when the states start.

Discrete-time Markov process defines probabilities of state changes without considering delays. This simplification ensures that the Markovian property is always valid. Table 4.27 shows the declaration of `StateMarkov` and `EventMarkov` classes for creating discrete-time Markov models. The probabilities of `EventMarkov` elements define the next activated state.

Table 4.27 The Markov modelling technique

	A	B	C	COMMENTS
1	StateMarkov	class	State	Special state class for the Markov model
2	EventMarkov	class	Event	Special event class for the Markov model
3	EventMarkov/prob	attribute	Probability	Probability of a state change

The sum of probabilities must equal to 1.0. Like with alternative consequences (see Section 4.2.2), the given values are considered as weights of the consequences. Listing 4.50 shows how to translate the weights to probabilities.

```
void StateMarkov.simulationStarted() {
  double sum = 0;
  for (Element target : THIS.getElements("target")) {
    sum += THIS.prob[target.getKey()];
  }
  if (sum != 1.0) {
    for (Element target : THIS.getElements("target")) {
      target.prob[target.getKey()] /= sum;
    }
  }
}
```

Listing 4.50 The reset of the Markov state transition probabilities

Listing 4.51 configures the selection of an event to activate. The `StateMarkov` rewrites the `createNextAction()` method of the `State` class (see Listing 3.17 on page 81). A random probability number selects the ”target” event that is activated.

```
void StateMarkov.createNextAction() {
  double prob = RANDOM.prob();
  for (Element target : THIS.getElements("target")) {
    prob -= target.prob;
    if (prob < 0) { // select this target
      ACTIONS.add(TARGET);
      return;
    }
  }
}
```

Listing 4.51 The activation of the select gate

When compared to continuous-time analysis, the algorithm for discrete-time simulation is simpler. Table 4.28 declares a tool, which repeats at most ”maxSteps” number of discrete time steps for each round. Depending on the significance of the initial state of the model, the analysis can simulate only one simulation round with a large number of ”maxSteps”, or several rounds with less steps in each round.

Table 4.28 The definition of a simulator tool for discrete-time simulation

	A	B	C	COMMENTS
1	Discrete	class	DES	Declaration of a simulator tool for discrete-time simulation
2	Discrete/maxSteps	attribute	Integer	The steps limit of each round
3	/simulator	instance	Discrete	The creation of a simulator tool instance (UID: /simulator)

Listing 4.52 presents the simulation algorithm core, which simplifies the procedure of DES tool (see Listing 3.2 on page 74). The algorithm core does not need the `step()` template method as the lengths of the time steps are ignored during the simulation process. Also the ending of a simulation round is handled differently.

```
void Discrete.calculationProcess() {
  THIS.simulationStart();
  do {
    THIS.roundStart();
    while (THIS.actionStart()) {
      THIS.actionHandle();
      THIS.actionEnd();
    }
  } while (!THIS.simulationEnd());
  THIS.createResults();
}
```

Listing 4.52 The simulation algorithm skeleton of the discrete-time simulation tool

Listing 4.53 presents the changes needed to the `actionStart()` method of the `DES` tool (see Listing 3.7 on page 76). The "currentTime" attribute counts the handled steps. A simulation round ends if the actions queue is empty or when the "maxSteps" limit is reached.

```
boolean Discrete.actionStart() {
  if (ACTIONS.getFirstTarget() != null &&
      THIS.currentTime < THIS.maxSteps) {
    THIS.currentTime++;
    return true;
  }
  for (Element element : MODEL.getElements()) {
    element.roundEnded(); // call the template method for all elements
  }
  THIS.currentRound++;  // increase the status variable:  currentRound
  return false;
}
```

Listing 4.53 The redefinition of the actionStart() for discrete-time simulator

4.6.2 Failure Modes and Effects Analysis modelling technique

The Failure Modes and Effects Analysis (FMEA) [7] modelling technique is an example for including qualitative information in AoT models. Table 4.29 declares a `FMEA` fault node and attributes for severity, occurrence and detection of the failure.

Table 4.29 The attributes for the FMEA modelling technique

	A	B	C	COMMENTS
1	FMEA	class	Fault	A special fault for including qualitative information
2	FMEA/severity	attribute	Text	Text to describe severity of the failure
3	FMEA/occurrence	attribute	Text	Text to describe occurrence rate of the failure
4	FMEA/detection	attribute	Text	Text to describe detectability of the failure
5	FMEA/sev	attribute	Integer	Integer between 0 and 10 to rate the severity
6	FMEA/occ	attribute	Integer	Integer between 0 and 10 to rate the occurrence likelihood
7	FMEA/det	attribute	Integer	Integer between 0 and 10 to rate the detectability
8	FMEA/rpn	attribute	Integer	The risk priority number (RPN)

The FMEA procedure can be continued by defining actions for failure mitigation that improve the current situations. Table 4.30 declares a special `Action` class, which can be used to include actions in a `FMEA` fault nodes.

Table 4.30 The attributes of the Action class of the FMEA modelling technique

	A	B	C	Comments
9	Action	class	Element	A special class for including FMEA actions
10	FMEA	container	Action	The FMEA fault node contains actions
11	Action/what	attribute	Text	Text to describe recommended action to reduce the risk priority
12	Action/who	attribute	Text	Person responsible for the recommendation and the actions
13	Action/date	attribute	Time	Target date to complete the recommended actions
14	Action/sev	attribute	Integer	The reduced severity if recommended mitigation is applied
15	Action/occ	attribute	Integer	The reduced occurrence likelihood recommended action is applied
16	Action/det	attribute	Integer	The improved detectability if recommended action is applied
17	Action/rpn	attribute	Integer	The updated priority number if recommended action is applied

The value of the attribute ”rpn” is obtained by multiplying the values of the ”sev”, ”occ” and ”det” attributes. In AoT, it is possible to create a tool that automatically calculates and assigns attribute values based on other attributes. Table 4.31 declares a `Calculator` tool, which is much simpler that the previously defined `Discrete` and `DES` simulation tools.

Table 4.31 The declaration of a simple calculator tool

	A	B	C	Comments
1	Calculator	class	Tool	Calculator tool for automatic value assignment

Listing 4.54 presents the `calculationProcess()` method of `Calculator` tool. Instead of defining a simulation algorithm core (see Listing 3.2 on page 74), it only calls the `calculate()` template method for all model elements.

```
void Calculator.calculationProcess() {
  for (Element element : MODEL.getElements()) {
    element.calculate(); // call the template method for all elements
  }
}
```

Listing 4.54 The calculation process of a simple Calculator tool

Listing 4.55 presents the implementation of the `calculate()` template method for the FMEA modelling technique. The ”rpn” attribute value is assigned automatically for all elements of `FMEA` and `Action` classes.

```
void FMEA.calculate() {
  THIS.rpn = THIS.sev * THIS.occ * THIS.det;
}

void Action.calculate() {
  THIS.rpn = THIS.sev * THIS.occ * THIS.det;
}
```

Listing 4.55 The risk priority number calculation for FMEA modelling technique

5 EXAMPLE MODELS OF DIFFERENT AOT MODELLING TECHNIQUES

This chapter presents example models that apply the modelling techniques that were specified in Chapter 4. The examples illustrate how, after the declaration of a modelling technique, the model structure can be created and the attribute values assigned with a limited number of definition rows. A similar approach is applicable to modelling with arbitrary techniques. With the most suitable technique the definition of the model is as simple as possible.

5.1 Advanced Fault Tree Analysis example model

This section presents an example model, which uses the advanced FTA modelling technique (see Section 4.2). Figure 5.1 illustrates the system model, which consist of a power input with a backup generator, and three similar pumping units. Each unit has two redundant pumps.

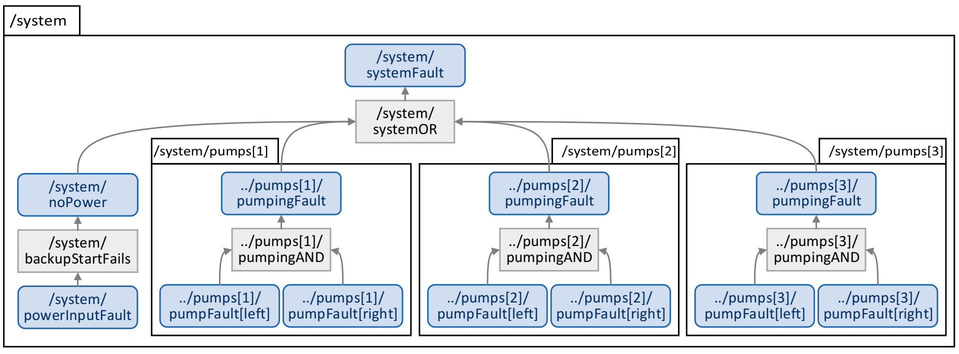
/system
/system/ systemFault
/system/ systemOR
/system/ noPower
/system/ backupStartFails
/system/ powerInputFault
/system/pumps[1]
../pumps[1]/ pumpingFault
../pumps[1]/ pumpingAND
../pumps[1]/ pumpFault[left]
../pumps[1]/ pumpFault[right]
/system/pumps[2]
../pumps[2]/ pumpingFault
../pumps[2]/ pumpingAND
../pumps[2]/ pumpFault[left]
../pumps[2]/ pumpFault[right]
/system/pumps[3]
../pumps[3]/ pumpingFault
../pumps[3]/ pumpingAND
../pumps[3]/ pumpFault[left]
../pumps[3]/ pumpFault[right]

Figure 5.1 An advanced FTA modelling technique example model structure

The definition of the model is divided into three steps: Creation of the model elements, adding of the connections, and assignment of the attribute values. The steps are presented in separate tables, which together define the entire model. In addition, the inclusion of cost risks and maintenance actions is presented in this section.

Table 5.1 creates 17 elements with 10 definition rows. Table 5.2 forms the model structure by adding 16 connections with 7 rows. The elements of the three similar pumping units can be defined without a need to repeat a definition for each unit. The number of elements and connections would be higher if the automatically created structures of states and events inside each fault node are included.

Table 5.1 The model structure creation of the advanced FTA example model

	A	B	C	COMMENTS
1	/system	instance	Folder	Folder for the fault tree of the system
2	system/systemFault	instance	Fault	Fault tree top node
3	system/systemOR	instance	OR	Logic OR operator of the system
4	system/noPower	instance	Fault	No power to the system
5	system/backupStartFails	instance	Cond	Backup starts fails at certain probability
6	system/powerInputFault	instance	Fault	Power input fault of the system
7	system/pumps[1-3]	instance	Folder	Three similar pump model folders
8	pumps/pumpingFault	instance	Fault	A pumping fault for each folder
9	pumps/pumpingAND	instance	AND	AND operator for each pumping folder
10	pumps/pumpFault[left,right]	instance	Fault	Two similar pump faults for each folder

Table 5.2 The connections of the advanced FTA example model

	A	B	C	COMMENTS
11	system/child	connect	system/systemOR	From OR gate to system fault
12	systemOR/child	connect	noPower	From no power to system OR gate
13	noPower/child	connect	backupStartFails	From backup fault to no power
14	backupStartFails/child	connect	powerInputFault	From power input fault to backup
15	systemOR/child	connect	pumps/pumpingFault	From each pumping fault to OR gate
16	pumpingFault/child	connect	pumpingAND	From pumping AND to pumping fault
17	pumpingAND/child	connect	pumpFault	From both pump faults to AND gate

Table 5.3 assigns the attribute values of the advanced FTA example model. The failures of the pumps are frequent but the redundant pump reduces the consequences

of a failure. The power input has also frequent failures, but the possibility to start a backup generator helps avoiding the system failures.

Table 5.3 The attribute values of the advanced FTA example model

	A	B	C	COMMENTS
18	Fault/failure	include	EventExp	All failures are exponentially distributed
19	Fault/restoration	include	EventExp	Exponentially distributed restorations
20	backupStartFails/prob	=	0.1	10% probability of backup startup failure
21	powerInputFault/failure/mean	=	50d	MTTF of power input is 50 days
22	powerInputFault/restoration/mean	=	1d	MTTR of power input is 1 day
23	pumpFault/failure/mean	=	10d	MTTF of pump is 10 days
24	pumpFault/restoration/mean	=	5h	MTTR of pump is 5 hours

For the cost risk analysis (see Section 4.2.3), Table 5.4 presents an example of the assignment of a repair cost for each component of the the example model. The repair cost can be defined for all 6 pumps by using only one definition row. In addition, a downtime loss is assigned for the system fault. All costs are in this case added to the ”fault” states of the elements. It would be also possible, for example, to add a positive production cost to the ”normal” state of the system fault node.

Table 5.4 The inclusion of cost risks in the advanced FTA example model

	A	B	C	COMMENTS
25	powerInputFault/fault/costCount[repair]	=	3000	Power input repair cost: 3000€
26	pumpFault/fault/costCount[repair]	=	800	Pump repair cost: 800€
27	system/fault/costTime[loss]	=	2000/h	System downtime loss: 2000€ per hour

For the inclusion of maintenance actions (see Section 4.2.5), Table 5.5 presents how an inspection is added for the power input and preventive maintenance for the pumps. The power input is inspected once per week. The symptom time is only a day so the inspection can only detect and stop about one out of seven failures. The pumps have a preventive maintenance action once a month, which reduces the failure rate by 50 %.

Table 5.5 The inclusion of maintenance actions in the advanced FTA example model

	A	B	C	COMMENTS
28	powerInputFault	include	FaultMaint	Power input fault has maintenance
29	powerInputFault/insp	instance	Inspection	Inspection for power input fault
30	powerInputFault/insp/interval	=	7d	Inspection is done weekly
31	powerInputFault/insp/symptom	=	1d	Symptom time is only one day
32	pumpFault	include	FaultMaint	Pump faults have maintenance
33	pumpFault/prev	instance	Preventive	Preventive maintenance for pumps
34	pumpFault/prev/prob	=	0.5	Maintenance halves the failure rate
35	pumpFault/prev/interval	=	30d	Action is done once per month
36	pumpFault/prev/cost	=	200	An action costs 200€

5.2 An example of including modelling of phase changes

This section presents a phase change example model. The definition of the model can be made by using basic AoT state change modelling (see Section 3.3.4). The model defines a continuous-time semi-Markov process (see Section 4.6.1). As in previous example, the definition of the model is divided into three steps: Creation of the model elements, adding of the connections, and assignment of the attribute values. The steps are presented in separate tables, which together define the entire model. In addition, the inclusion of connections from a distinct model is presented in this section.

Figure 5.2 illustrates a phase change model, which consist of three phases and their connections. Table 5.6 creates the elements of the example model.

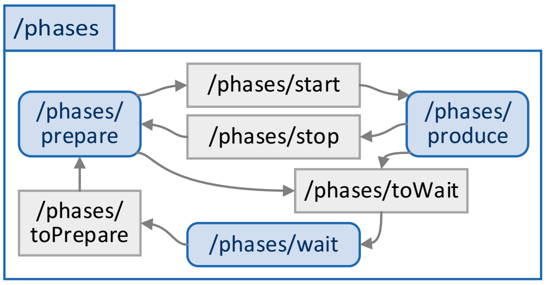

Figure 5.2 An example of a phase change model structure

Table 5.6 The element creation of the phase change example model

	A	B	C	COMMENTS
1	/phases	instance	Node	A node to model phase changes
2	phases/prepare	instance	State	The prepare state of the phase change model
3	phases/produce	instance	State	The produce state of the phase change model
4	phases/wait	instance	State	The wait state of the phase change model
5	phases/start	instance	Event	The produce start event of the phase change model
6	phases/stop	instance	Event	The produce stop event of the phase change model
7	phases/toWait	instance	Event	The wait start event of the phase change model
8	phases/toPrepare	instance	Event	The prepare start event of the phase change model

Table 5.7 adds the connections of the phase change model. The event ”toWait” has two sources, which means that the ”wait” state can start irrespective of the current state. After a wait, the basic operation restarts from the ”prepare” phase.

Table 5.7 The adding of the phase change example model connections

	A	B	C	COMMENTS
9	start/source	connect	prepare	From prepare to start
10	start/target	connect	produce	From start to produce
11	stop/source	connect	produce	From produce to stop
12	stop/target	connect	prepare	From stop to prepare
13	toWait/source	connect	prepare, produce	From prepare or produce to wait
14	toWait/target	connect	wait	From wait transition to state
15	toPrepare/source	connect	wait	From wait to prepare transition
16	toPrepare/target	connect	prepare	From prepare transition to state

Table 5.8 assigns the attribute values of the model. In basic situation the phase changes between preparation and production. There is a delay of 2 days before each start of the ”produce” phase, and the length of each production is 1 day. The initial state of the model is the ”prepare” phase.

Table 5.8 The assignment of the phase change example model attribute values

	A	B	C	COMMENTS
17	phases/initialState	connect	phases/prepare	The initial state is prepare
18	start	delay	2d	The delay to start the production
19	stop	delay	1d	The duration of the production

In this example, external actions cause an interruption of the basic operation cycle. The AoT framework includes messages and listeners for modelling of connections between distinct models (see Table 3.22 on page 59). The ”wait” phase is started when an external *systemFailure* message is sent. Similarly, the ”wait” phase ends after a *systemOK* message. Table 5.9 presents the assignment of event listeners.

Table 5.9 The assignment of external message listeners of the phase change example model

	A	B	C	COMMENTS
20	toWait/listener	=	systemFailure	Wait state starts after an external message
21	toPrepare/listener	=	systemOK	Wait state ends after an external message

The external messages are sent from an distinct model, such as the advanced FTA model that was presented in Section 5.1. In this case, the messages are added to the events of the *systemFault* node. Table 5.10 presents the assignment of the message sending attributes. The table continues the definitions that was made in Section 5.1 for the advanced FTA example model. Figure 5.3 illustrates the the connections of the two models. The state and event structure of the *systemFault* node, and the assigned messages and listeners are shown.

Table 5.10 The assignment of the message sending of the advanced FTA example model

	A	B	C	COMMENTS
37	systemFault/failure/message	=	systemFailure	Send message when system fault starts
38	systemFault/restoration/message	=	systemOK	Send message when system fault ends

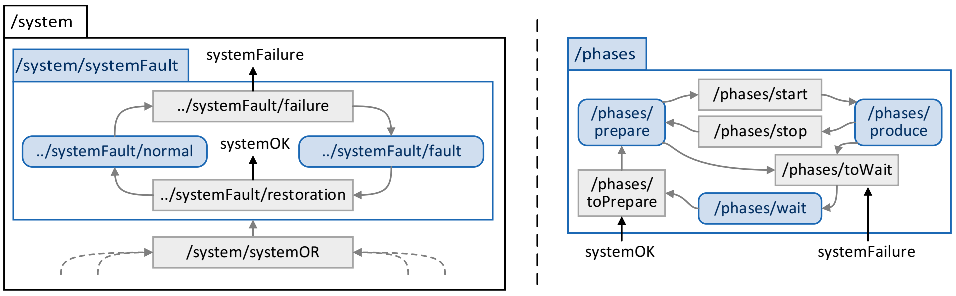

Figure 5.3 An example of distinct models that are connected with messages and listeners

The separate models can also have connections to the opposite direction. For example, the active state of the phase change model can affect the failure rate in the fault model. Modes (see Table 3.22 on page 59) are used as an interface for defining such connections. Table 5.11 presents the assignment of a *prep* mode for the "prepare" state and a *prod* mode for the "produce" state.

Table 5.11 The assignment of a production mode of the phase change example model

	A	B	C	COMMENTS
22	prepare/mode	=	prep	A prep mode is defined for the prepare state
22	produce/mode	=	prod	A prod mode is defined for the produce state

The mode-dependent events (see Section 4.2.4) can listen the mode changes. Table 5.12 continues the definitions that was made in Section 5.1 for the advanced FTA example model. It changes the pump faults to be mode-dependent. The original failure event listens the *prod* mode. A new event is created for the "prep" mode. The delays in both models are exponentially distributed. During the "produce" state the mean delay is 10 days (see Table 5.3 on page 131) and during the "prepare" state 20 days. Because the modes are defined only for the "prepare" and "produce" states, during the "wait" state there are no pump failures. These definitions do not affect to the restoration time, which is not mode-dependent.

Table 5.12 A mode-dependent failure rate for the advanced FTA example model

	A	B	C	COMMENTS
39	pumpFault/failure	include	EventMode	The original event is mode-dependent
40	pumpFault/failure/listener	=	prod	Active when producing
39	pumpFault/failure[prep]	instance	Event	Create new event instance
39	pumpFault/failure[prep]	include	EventMode	The new event is mode-dependent
39	pumpFault/failure[prep]	include	EventExp	The new event is exponential
39	pumpFault/failure[prep]/listener	=	prep	Active when preparing
39	pumpFault/failure[prep]/mean	=	20d	Less failures when not producing

This example connects phase change and advanced FTA models by using modes and messages. It is also possible to connect a node that models an arbitrary semi-Markov process as a child of a gate in an advanced FTA model. This requires a declaration of a node sub class, which configures the `isTrue()` method (see Listing 4.10 from page 100) to define the state of the model that represents the fault situation.

5.3 Using Reliability Block Diagram to model success logic

This section presents an example model, which uses the Reliability Block Diagram (RBD) modelling technique (see Section 4.3). The success logic of RBD makes modelling of certain situations simpler when compared to failure logic, which is used in FTA. The first part of the example is a bridge circuit model [2, Sec. 7.5.5], which is illustrated in Figure 5.4. The second part defines a sub model for each block of the first model, which illustrates the possibilities to divide the model to distinct levels.

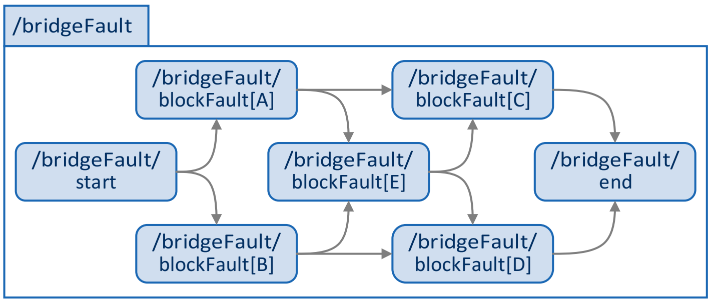
/bridgeFault
/bridgeFault/ start
/bridgeFault/ blockFault[A]
/bridgeFault/ blockFault[B]
/bridgeFault/ blockFault[E]
/bridgeFault/ blockFault[C]
/bridgeFault/ blockFault[D]
/bridgeFault/ end

Figure 5.4 A RBD modelling technique example model structure

The RBD consist of 5 interconnected blocks. There exist 4 combinations that interrupt the system operation: AB, CD, AED, BEC. Instead of finding these combinations manually, RBD finds them automatically based on the model structure. The elements and connections of the example are defined in Table 5.13.

Table 5.13 A bridge circuit RBD example model

	A	B	C	COMMENTS
1	/bridgeFault	instance	RBD	Reliability block diagram of bridge fault
2	bridgeFault/blockFault[A-E]	instance	FaultRBD	Block faults A,B,C,D and E
3	start/target	connect	blockFault[A,B]	The start is connected to blocks A and B
4	blockFault[A,B]/target	connect	blockFault[E]	A and B are connected to the block E
5	blockFault[E]/target	connect	blockFault[C,D]	The block E is connected to C and D
6	blockFault[A]/target	connect	blockFault[C]	The block A is connected to the block C
7	blockFault[B]/target	connect	blockFault[D]	The block B is connected to the block D
8	blockFault[C,D]/target	connect	end	Blocks C and D are connected to the end

Each block can contain its own RBD. For example, Table 5.14 includes the RBD class for the blocks of the previous bridge circuit example, and presents similar inner model for each. The model is an example of the n out of m logic, where at least 3 out of 5 items are required for the block to operate.

Table 5.14 A 3 out of 5 RBD example model

	A	B	C	COMMENTS
1	blockFault[A-E]	include	RBD	A RBD fault contains inner model
2	blockFault/itemFault[1-5]	instance	FaultRBD	Five item faults
3	blockFault/logic	instance	FaultRBD	An element with vote rule
4	start/target	connect	itemFault[1-5]	Start connected to all item faults
5	itemFault[1-5]/target	connect	logic	Item faults connected to logic operator
6	logic/target	connect	end	Logic operator connected to end
7	logic/atLeast	=	3	At least 3 out of 5 required

Similar definition of inner models could be continued further. For example, each item fault of the model could be defined by a fault tree. This approach enables defining more detailed inner models for elements that are considered the most important. The less significant parts of the model could be defined directly by using a failure probability or distribution function.

5.4 Examples of including function modelling

This section presents two example models, which use the function modelling technique (see Section 4.4). The first model combines basic mathematical functions to update an attribute value. This allows including the calculation of proper input values in the AoT model, which can reduce manual work in attribute value assignment. The second model illustrates the calculation of production by using a special user-defined function. The free algorithm creation ensures the flexibility that is sometimes required for inclusion of domain-specific features, such as calculation of special Key Performance Indicator (KPI).

Figure 5.5 illustrates a simple function model structure. The elements and the connections are defined in Table 5.15. The model implements the equation:

$$\text{result} = \frac{\text{duration}[1] + \text{duration}[2]}{\text{divisor}} \tag{5.1}$$

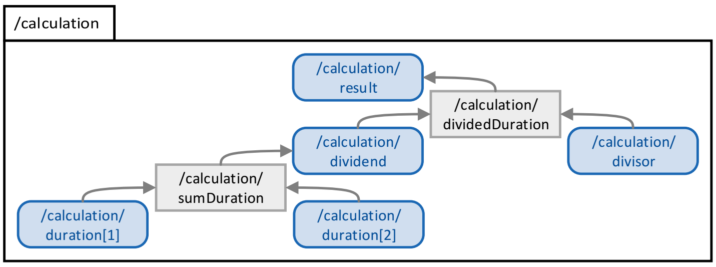
/calculation
/calculation/ result
/calculation/ dividedDuration
/calculation/ dividend
/calculation/ divisor
/calculation/ sumDuration
/calculation/ duration[1]
/calculation/ duration[2]

Figure 5.5 A function modelling technique example model structure

Table 5.15 A simple calculation example by using the function modelling technique

	A	B	C	COMMENTS
1	/calculation	instance	Folder	Simple calculation example
2	calculation/result	instance	Variable	Result of the calculation
3	calculation/dividedDuration	instance	Division	Quotient calculation
4	calculation/dividend	instance	Variable	Divident value
5	calculation/divisor	instance	Variable	Divisor value
6	calculation/sumDuration	instance	Addition	Sum calculation
7	calculation/duration[1,2]	instance	Variable	Duration values
8	dividedDuration/out	connect	result	Result value is dividend / divisor
9	dividedDuration/in[dividend]	connect	dividend	Connect the dividend
10	dividedDuration/in	connect	divisor	Connect the divisor
11	sumDuration/out	connect	dividend	The sum is the dividend of the division
12	sumDuration/in	connect	duration	The sum is duration[1] + duration[2]

Table 5.16 illustrates the definitions that connect the previously defined function model to an attribute of another model. The `VariableWrite` variables are activated automatically before simulation starts (see Listing 4.42 on page 120). The activation calculates the result value of the model and updates in to the defined attribute path, which in this case is *folder/element/attribute*.

Table 5.16 Connecting the function model to an attribute of another model

	A	B	C	COMMENTS
13	calculation/result	include	VariableWrite	Result of the calculation
14	calculation/result/path	=	folder/element/attribute	Store the calculated value

Sometimes the modelling of a complex value calculation is simpler by defining a user-defined function when compared to creating a structure that connects various basic built-in functions. For example, the equation

$$L_{int}(t_f) = \begin{cases} L_p t_f, & : t_f \leq t_{lv} \\ L_p t_l v + L_p \tau_L(\exp(-t_{lv}/\tau_L) - \exp(-t_f/\tau_L)) & : t_f \geq t_{lv} \end{cases} \tag{5.2}$$

defines how much integrated luminosity the Large Hadron Collider (LHC) in CERN can produce. This system KPI is formed based on the time length of a fill t_f, the peak value of instantaneous luminosity L_p, luminosity lifetime τ_L, and luminosity levelling time t_{lv} [85].

Equation 5.2 can be modelled with a user-defined `IntLumi` function, which is declared in Table 5.17. By using the "url" attribute, the code could be read from a separate file (see Table 3.14 on page 54) instead of a table cell. The procedure code is presented in Listing 5.1. The key of an array definition is used to distinguish the connected "in" variables (see Table 3.6 on page 47). The fill time is calculated from a difference of the current simulation time and the stored fill start time.

Table 5.17 Declaration of a user-defined function

	A	B	C	COMMENTS
1	IntLumi	class	Function	A class for calculation of integrated luminosity
2	IntLumi/function[getValue]	=	(see Listing 5.1)	The function algorithm

```
double IntLumi.getValue() {
  double fillTime = CALCULATION.currentTime - startTime;
  if (fillTime <= levelTime) {
    return THIS.out.getValue() + peakLumi * fillTime;
  }
  double levelled = peakLumi * levelTime;
  double decay = peakLumi * lifetime *
      (Math.exp(-levelTime / lifetime) - Math.exp(-fillTime / lifetime));
  return THIS.out.getValue() + levelled + decay;
}
```

Listing 5.1 The user-defined code for the integrated luminosity function

In this example the variable values are referred directly by using the key of the connection attribute. For example, instead of `THIS.in[''peakLumi''].getValue()` just `peakLumi` is used. A pre-processing of user-defined function codes needs to be included in the calculation engine for enabling this simplification. The principles of the procedure code pre-processing, which is made in the calculation engine, are not included in this thesis.

Table 5.18 shows the element creation and the attribute value assignment of the example function. The `Variable` instances are created directly to the connection attributes of the `IntLumi` function. It would be also possible to first create the elements separately and then connect them to the function.

Table 5.18 An example of a function with user defined code

	A	B	C	COMMENTS
1	/luminosity	instance	Folder	Luminosity calculation example
2	luminosity/intLumi	instance	IntLumi	Luminosity production function of a fill
3	intLumi/in[startTime]	instance	StoreTime	The stored start time of a fill
4	intLumi/in[levelTime]	instance	Variable	Constant levelling time
5	intLumi/in[levelTime]/value	=	6.5h	Value for the parameter
6	intLumi/in[peakLumi]	instance	Variable	Constant peak luminosity
7	intLumi/in[peakLumi]/value	=	5e34	Value for the parameter
8	intLumi/in[lifetime]	instance	Variable	Constant luminosity lifetime
9	intLumi/in[lifetime]/value	=	9.3h	Value for the parameter
10	intLumi/out	instance	VariableResult	Result luminosity production

The messages and listeners (see Table 3.22 on page 59) are used for connecting the function model to phase change model. The interface is used for avoiding the need of knowing the exact element UIDs of the phase change model. Any model that uses the correct message sending interface can be used with the defined function model. The start time is stored when a *prodStart* message is received, and the calculation of the luminosity production is made after the *prodEnd* message. Table 5.19 presents the assignment of the listeners.

Table 5.19 An example of a function with the user-defined code

	A	B	C	COMMENTS
1	intLumi/in[startTime]/listener	=	prodStart	Store the time when a production starts
2	intLumi/out/listener	=	prodEnd	Calculate the production of a fill

Table 5.11 on page 135 presents an example of a phase change model with a proper interface. The assignment of a *prod* mode was made for the "produce" state. The use

of default mode suffixes ”Start” and ”End” (see Rule 3.6 on page 58) allows using only one ”mode” assignment to send the needed messages. Similar definitions should be made to the model that defines the phase changes of the luminosity production.

5.5 A Petri net example model

This section presents an example model, which uses Petri net modelling technique (see Section 4.5). Figure 5.6 illustrates an order management process, where an order is closed when it is paid and delivered. A goal of ten completed orders per day is set to demonstrate the weight modelling.

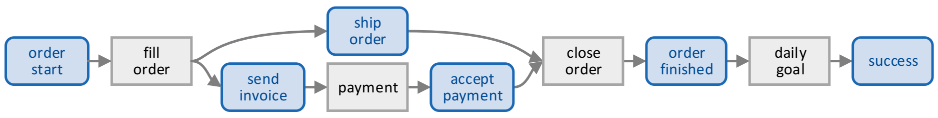

Figure 5.6 A Petri net example model

Table 5.20 presents the creation of the model elements. Each place and transition of a Petri net is translated directly to an element of the corresponding AoT model.

Table 5.20 The definition of the Petri net example model elements

	A	B	C	COMMENTS
1	/orderStart	instance	Place	Token at this place starts the process
2	/fillOrder	instance	Transition	Order process is stated
3	/ship	instance	Place	The product is sent to customer
4	/invoice	instance	Place	The invoice is sent to customer
5	/payment	instance	Transition	The customer pays the invoice
6	/accept	instance	Place	Made payment is accepted
7	/closeOrder	instance	Transition	Close the order when all ready
8	/orderFinished	instance	Place	The order process has ended
9	/dailyGoal	instance	Transition	Check when daily goal is achieved
10	/success	instance	Place	Success after daily goal

Table 5.21 adds the connections of the Petri net model structure. Each arc is translated directly to a connection of the corresponding AoT model. A weight is defined

for an arc from ”closeOrder” to ”dailyGoal”. The key of an array definition is used for indicating the connection that has the weight. The other arcs of the order process have the default weight 1, so special attribute value assignments are not needed.

Table 5.21 The definition of the Petri net example model connections

	A	B	C	COMMENTS
11	fillOrder/input	connect	orderStart	Start the order process
12	fillOrder/output	connect	ship,invoice	Start product and invoice sending
13	payment/input	connect	invoice	Wait for payment after sent invoice
14	payment/output	connect	accept	Accept after customer has paid
15	closeOrder/input	connect	ship,accept	Ship and payment before close
16	closeOrder/output	connect	orderFinished	End of an order process
17	dailyGoal/input[goal]	connect	orderFinished	Connect to daily goal
18	dailyGoal/inputWeight[goal]	connect	10	Weight of the daily goal
19	dailyGoal/output	connect	success	Success after daily goal

Like AoT function models, a Petri net can be connected to advanced FTA or other techniques by using messages and listeners (see Table 3.22 from page 59). At least the first *orderStart* place should be activated by an external event. Stochastic delays can be defined for transitions of the model by including, for example, `TransitionExp` and `TransitionWeibull` classes. The result of the model could be the number of daily goals reached.

Also direct connection of Petri net and advanced FTA models is possible. For example, the situation that no tokens exist in a place element can represent a fault, or vice versa. This requires a customisation of the `Gate` operators to allow also `Place` elements as ”child” connections (see Table 3.19 from page 57), and configuration of the `isTrue()` template method (see Listing 4.10 from page 100) for `Place` class.

6 DISCUSSION AND CONCLUSIONS

This thesis introduced the AoT framework for probabilistic risk and performance assessment of complex systems. A tabular Triplets data format is used for defining object-oriented models. Uniquely, the framework and the data format enables declaring the applied modelling technique before model creation, including customised features for handling of special needs, and combining various modelling techniques. This thesis also described the principles of a calculation engine, which can be configured for any modelling technique. When compared to traditional static simulation algorithms, the dynamic compilation of stochastic DES algorithms enables higher flexibility and more freedom to optimise the calculation of versatile analysis results.

Chapter 4 presented the declaration of traditional risk assessment techniques and various modelling features, which have been selected based on a large variety of concrete analysis cases from different industry sectors [18, 19, 20, 21, 22, 23]. With similar procedure any advanced or domain-specific features can be included in AoT models. Chapter 5 illustrated the model creation with the declared modelling techniques. Based on a review, other MBDA approaches do not similarly separate modelling technique declaration from model creation. With this feature AoT enables declaring an optimal modelling technique for any special situation, which makes the model creation as simple and efficient as possible. A future target is to build a generic library that provides predefined modelling techniques for various analysis needs.

The Triplets data format enables storing the declared modelling technique and the created model into a single table. When compared to high-level programming or markup languages that are used by other MBDA approaches, tables were found more familiar for basic system engineers. The tabular format and its array definition are especially suitable for modelling large systems with repetitive structures, such as the CERN's PS booster RF system availability model [86]. The applicability of tabular model definition has been verified in practise by modelling the dynamic operation phases of luminosity production [24] and by identifying the critical systems of a future circular collider [27]. Together the possibility of defining models by editing a table and the flexibility of declaring custom-made modelling techniques form an environment that simplifies the development of new analysis tools. This is useful, for example, in special situations where existing tools are not flexible or efficient enough. Based on a brief review, an in-house software was required at least in reliability studies [87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97]. In such situations, AoT can reduce the time required for developing domain-specific analysis software.

Future trends are likely to yield more robust integration between existing techniques and paradigms [56]. Safety Analysis Modelling Language (SAML) [60] and Guarded Transitions Systems (GTS) [69] are examples of integrative approaches. Their specifications can be written with high-level tools, transformed into the integrative model, and verified with a selected verification tool. Open-Probabilistic Safety Assessment (PSA) [98] and Petri Net Markup Language (PNML) [99] are similar model exchange formats for traditional techniques. Unlike the state-of-the-art model exchange and integration formats, the format presented in this thesis is tabular. As the Triplets data format is highly flexible, platform-independent, open and non-proprietary, it is suitable for integrating various modelling techniques and analysis tools. The clear data format also helps the development of graphical and automatised model creation interfaces, which enable more user-friendly application of the framework.

Instead of trying to replace existing tools, exchange formats and integration approaches, a future target is to make the AoT framework compatible with them. AoT modelling techniques have not been declared to support SAML, GTS, Open-PSA nor PNML, but no issues are known that would prevent it. Translating models can require certain restrictions, but even a limited compatibility to one direction would be beneficial for both approaches. A cabability to translate models from a format to another increases the number of suitable model definition interfaces and analysis tools for both formats. Similar benefits can be achieved by declaring modelling techniques to support, for example, coloured Petri net [41], Functional-Failure Identification and Propagation (FFIP) [51], Dynamic Flowgraph Methodology (DFM) [52], Software Reliability Growth Model (SRGM) [53] and Bayesian networks [54]. Project Evaluation and Review Technique (PERT) [100] for project management and Design Structure Matrix (DSM) [101] for requirement engineering are examples of integration possibilities from outside the field of reliability modelling.

The AoT framework aims to unify the declaration of modelling techniques and to enable their customisation. A common methodological foundation simplifies the combination of different techniques. In addition to risk assessment, analysis of the system performance is enabled by including a calculation of any KPI [9] in the model. The needed flexibility is achieved by using object-oriented paradigm as a basis of the AoT framework. A modelling technique is defined by declaring a catalogue of model object classes and their attributes. This approach is especially suitable to situations where complex, dynamic or domain-specific features need to be included in the analysis. With these properties the AoT framework proves the research hypothesis: *By applying the object-oriented paradigm it is possible to create a single framework that can utilise, combine and customise various risk and performance assessment techniques to answer the challenging analysis needs of today's complex and dynamic systems.*

The freedom and flexibility in modelling technique declaration enables a large variety of different approaches and solutions for each situation. This thesis presented fundamental classes, which form a common basis for all modelling techniques. The model elements are divided to vertices and edges, which makes the model structures similar to graphs in graph theory [77, 78]. The common base classes help understanding the similarities of different techniques. As illustrated by figures of this thesis, using special symbols only for vertices and edges enables clear model structure visualisation of an arbitrary technique. Also the combination of different techniques is more intuitive when the elements share common super classes that denote the connection possibilities. This approach of using fundamental base classes for various techniques is an answer to the first research question: *What kind of class declaration is suitable for efficient use of individual modelling techniques and enables a clear way to add connections between them?*

The Triplets data format improves the tabular model definition that was developed for the OpenMARS [25, 26] approach. A single table enables declaring the modelling techniques, creating the model structures, and assigning the model parameter values. Each definition is a triplet, which can be stored as a row of a three-column table. This thesis introduced nine keywords that divide the definitions to nine different cases. This strict and clear format simplifies the manual model definition and enables straightforward reading of tabular models. With this approach the Triplets data format is an answer to the second research question: *How the object-oriented models can be defined clearly and efficiently by using a tabular format?*

This theses presented a calculation engine to prove that the models of the AoT framework can be solved. The engine compiles dynamically a Java simulation program by combining a generic algorithm core and the configuration of a modelling technique. The IoC design principle [82] is applied for increasing the modularity of the simulation program. Only the procedures that are needed by the applied modelling techniques are included, which increases the efficiency of the analysis process. The possibility to freely configure the statistics data collection enables optimising the memory consumption for each technique or case. For large analyses the approach supports parallel simulation in a computing cluster. With these features the presented calculation engine is an answer to the third research question: *What stochastic DES procedure is suitable for generic and versatile analyses of object-oriented models?*

A higher-level motivation of this research was to improve the decision-making process. A made decision can be justified to be correct only if available information is fully taken into account and used in a sophisticated way. There exists usually some uncertainty related to the studied situations, which causes that a correctly justified decision made at some time point can be found later to be either good or bad. To make more good decisions, the available information should be improved, or it must be used more wisely. The flexibility of the AoT framework enables including systematically all essential details in the system model, which forms a basis for sophisticated risk and performance assessment. Considering these analysis results is integral when making justified decisions in complex system design and management.

REFERENCES

[1] E. Zio. Reliability engineering: Old problems and new challenges. *Reliability Engineering & System Safety* 94.2 (2009), 125–141. DOI: 10.1016/j.ress.2008.06.002.

[2] *Fault tree analysis (FTA)*. Standard IEC 61025. Geneva: International Electrotechnical Commission, 2006.

[3] E. Ruijters and M. Stoelinga. Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools. *Computer Science Review* 15–16 (2015), 29–62. DOI: 10.1016/j.cosrev.2015.03.001.

[4] *Risk management – Risk assessment techniques*. Standard IEC/ISO 31010. Geneva: International Organization for Standardization, 2009.

[5] *Reliability block diagrams*. Standard IEC 61078. Geneva: International Electrotechnical Commission, 2016.

[6] *Application of Markov techniques*. Standard IEC 61165. Geneva: International Electrotechnical Commission, 2006.

[7] *Analysis techniques for system reliability – Procedure for failure mode and effects analysis (FMEA)*. Standard IEC 60812. Geneva: International Electrotechnical Commission, 2006.

[8] *Analysis techniques for dependability – Petri net techniques*. Standard IEC 62551. Geneva: International Electrotechnical Commission, 2012.

[9] D. A. Bishop. Key Performance Indicators: Ideation to Creation. *IEEE Engineering Management Review* 46.1 (2018), 13–15. ISSN: 0360-8581. DOI: 10.1109/EMR.2018.2810104.

[10] D. Stamatis. *The OEE Primer: Understanding Overall Equipment Effectiveness, Reliability, and Maintainability*. Productivity Press, 2010.

[11] K. Holmberg. *Competitive Reliability 1996-2000*. Technology Programme Report. Helsinki: National Technology Agency, 2001.

[12] P. Hagmark and S. Virtanen. Specification and Allocation of Reliability and Availability Requirements. *RAMS '06. Annual Reliability and Maintainability Symposium, 2006*. IEEE, 2006, 304–309. DOI: 10.1109/RAMS.2006.1677391.

[13] P. Hagmark and H. Pernu. Risk evaluation of a spare part stock by stochastic simulation. *Proceedings of the European Safety and Reliability Conference 2006, ESREL 2006 – Safety and Reliability for Managing Risk*. CRC Press, 2006, 525–529. DOI: 10.13140/2.1.2636.6248.

[14] P. Hagmark and S. Virtanen. Simulation and calculation of reliability performance and maintenance costs. *RAMS '07. Annual Reliability and Maintainability Symposium, 2007*. IEEE, 2007, 34–40. DOI: 10.1109/RAMS.2007.328102.

[15] Ramentor. *Event Logic Modeling and Analysis Software (ELMAS)*. Accessed: December 1, 2018. URL: http://www.ramentor.com/elmas.

[16] J.-P. Penttinen. Analysis of failure logic using simulation. MA thesis. Tampere University of Technology, 2005.

[17] S. Virtanen, P. Hagmark and J.-P. Penttinen. Modeling and analysis of causes and consequences of failures. *RAMS '06. Annual Reliability and Maintainability Symposium, 2006*. IEEE, 2006, 506–511. DOI: 10.1109/rams.2006.1677424.

[18] Ramentor. *Customer cases*. Accessed: December 1, 2018. URL: http://www.ramentor.com/cases.

[19] J.-P. Penttinen and T. Lehtinen. Advanced Fault Tree Analysis for Improved Quality and Risk Assessment. *Proceedings of the 10th World Congress on Engineering Asset Management (WCEAM 2015)*. Springer International Publishing, 2016, 471–478. DOI: 10.1007/978-3-319-27064-7_45.

[20] M. Tammi, V. Vuorela and T. Lehtinen. Advanced RCM Industry Case — Modeling and Advanced Analytics (ELMAS) for Improved Availability and Cost-Efficiency. *Proceedings of the 10th World Congress on Engineering Asset*

Management (WCEAM 2015). Springer International Publishing, 2016, 581–589. DOI: 10.1007/978-3-319-27064-7_58.

[21] D. Imran Khan, S. Virtanen, P. Bonnal and A. K. Verma. Functional failure modes cause-consequence logic suited for mobile robots used at scientific facilities. *Reliability Engineering & System Safety* 129 (2014), 10–18. ISSN: 0951-8320. DOI: 10.1016/j.ress.2014.03.012.

[22] S. Virtanen, J.-P. Penttinen, M. Kiiski and J. Jokinen. Application of Design Review to Probabilistic Risk Assessment in a Large Investment Project. *Proceedings of the Probabilistic Safety Assessment and Management (PSAM) 12 Conference – Volume 9*. 2016, 200–213. URL: https://www.researchgate.net/publication/270214849.

[23] A. Niemi, A. Apollonio, J. Gutleber, P. Sollander, J.-P. Penttinen and S. Virtanen. Availability modeling approach for future circular colliders based on the LHC operation experience. *Physical Review Accelerators and Beams* 19 (12 2016), 121003. DOI: 10.1103/PhysRevAccelBeams.19.121003.

[24] A. Niemi. *Modeling Future Hadron Colliders' Availability for Physics*. Tampere University Dissertations. Tampere University, 2019. ISBN: 978-952-03-1056-1. URL: http://urn.fi/URN:ISBN:978-952-03-1057-8.

[25] J.-P. Penttinen, A. Niemi and J. Gutleber. *An Open Modelling Approach for Availability and Reliability of Systems — OpenMARS*. Specification CERN-ACC-2018-0006. CERN, 2018. URL: https://cds.cern.ch/record/2302387.

[26] J.-P. Penttinen, A. Niemi, J. Gutleber, K. T. Koskinen, E. Coatanéa and J. Laitinen. An open modelling approach for availability and reliability of systems. *Reliability Engineering & System Safety* 183 (2019), 387–399. ISSN: 0951-8320. DOI: 10.1016/j.ress.2018.11.026.

[27] A. Niemi and J.-P. Penttinen. Availability and critical systems of the Future Circular Electron–Positron Collider. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 963 (2020), 163759. ISSN: 0168-9002. DOI: 10.1016/j.nima.2020.163759.

[28] *Risk management – Vocabulary*. Standard ISO GUIDE 73. Geneva: International Organization for Standardization, 2009.

[29] *NASA Systems Engineering Handbook*. Tech. rep. NASA/SP-2016-6105 Rev2. National Aeronautics and Space Administration, 2016. URL: `https://www.nasa.gov/connect/ebooks/nasa-systems-engineering-handbook`.

[30] *Risk management – Principles and guidelines*. Standard ISO 31000. Geneva: International Organization for Standardization, 2009.

[31] *International electrotechnical vocabulary – Part 192: Dependability*. Standard IEC 60050-192. Geneva: International Electrotechnical Commission, 2015. URL: `http://www.electropedia.org`.

[32] *Maintenance – Maintenance terminology*. Standard EN 13306. Brussels: European Committee for Standardization, 2017.

[33] *Safety aspects – Guidelines for their inclusion in standards*. Standard ISO GUIDE 51. Geneva: International Organization for Standardization, 2014.

[34] M. Malhotra and K. S. Trivedi. Power-hierarchy of dependability-model types. *IEEE Transactions on Reliability* 43.3 (1994), 493–502. ISSN: 0018-9529. DOI: `10.1109/24.326452`.

[35] M. Malhotra and K. S. Trivedi. Dependability modeling using Petri-nets. *IEEE Transactions on Reliability* 44.3 (1995), 428–440. ISSN: 0018-9529. DOI: `10.1109/24.406578`.

[36] V. G. Kulkarni. Continuous-Time Markov Models. *Introduction to Modeling and Analysis of Stochastic Systems*. New York: Springer, 2011. Chap. 4, 85–145. ISBN: 978-1-4419-1772-0. DOI: `10.1007/978-1-4419-1772-0_4`.

[37] R. Pyke. Markov Renewal Processes: Definitions and Preliminary Properties. *The Annals of Mathematical Statistics* 32.4 (1961), 1231–1242. ISSN: 00034851. URL: `http://www.jstor.org/stable/2237923`.

[38] M. K. Molloy. Performance Analysis Using Stochastic Petri Nets. *IEEE Transactions on Computers* C-31.9 (1982), 913–917. ISSN: 0018-9340. DOI: `10.1109/TC.1982.1676110`.

[39] M. Ajmone Marsan, G. Conte and G. Balbo. A Class of Generalized Stochastic Petri Nets for the Performance Evaluation of Multiprocessor Systems. *ACM Transactions on Computer Systems* 2.2 (1984), 93–122. ISSN: 0734-2071. DOI: `10.1145/190.191`.

[40] G. Chiola, S. Donatelli and G. Franceschinis. GSPNs versus SPNs: what is the actual role of immediate transitions?: *Proceedings of the Fourth International Workshop on Petri Nets and Performance Models PNPM91*. 1991, 20–31. DOI: `10.1109/PNPM.1991.238785`.

[41] K. Jensen and L. M. Kristensen. *Coloured Petri Nets*. Springer Berlin Heidelberg, 2009. DOI: `10.1007/b95112`.

[42] *Software and System Engineering – High-level Petri nets - Part 1: Concepts, Definitions and Graphical notation*. Standard ISO/IEC 15909-1. Geneva: International Organization for Standardization, 2004.

[43] J. B. Dugan, K. S. Trivedi, R. M. Geist and V. F. Nicola. *Extended Stochastic Petri Nets: Applications and Analysis*. Tech. rep. Durham: Duke University, 1984.

[44] J.-P. Signoret, Y. Dutuit, P.-J. Cacheux, C. Folleau, S. Collas and P. Thomas. Make your Petri nets understandable: Reliability block diagrams driven Petri nets. *Reliability Engineering & System Safety* 113 (2013), 61–75. ISSN: 0951-8320. DOI: `10.1016/j.ress.2012.12.008`.

[45] P. Pozsgai and B. Bertsche. Conjoint Modelling with Extended Coloured Stochastic Petri Net and Reliability Block Diagram for System Analysis. *Probabilistic Safety Assessment and Management: PSAM 7 – ESREL '04 June 14–18, 2004, Berlin, Germany, Volume 6*. London: Springer, 2004, 1382–1387. DOI: `10.1007/978-0-85729-410-4_223`.

[46] M. Westergaard and H. (Verbeek. *CPN Tools*. Accessed: December 1, 2018. URL: `http://cpntools.org/`.

[47] SATODEV. *GRIF-Workshop BStoK module*. Accessed: December 1, 2018. URL: `http://grif-workshop.com/grif/bstok-module/`.

[48] F. Long, P. Zeiler and B. Bertsche. Modelling the production systems in industry 4.0 and their availability with high-level Petri nets. *IFAC-PapersOnLine* 49.12 (2016). 8th IFAC Conference on Manufacturing Modelling, Management and Control MIM 2016, 145–150. ISSN: 2405-8963. DOI: `10.1016/j.ifacol.2016.07.565`.

[49] S. Kabir. An overview of fault tree analysis and its application in model based dependability analysis. *Expert Systems With Applications* 77 (2017), 114–135. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2017.01.058.

[50] *Criticality Classification of Equipment in Industry*. Standard PSK 6800. Helsinki: PSK Standards Association, 2008.

[51] S. Sierla, I. Tumer, N. Papakonstantinou, K. Koskinen and D. Jensen. Early integration of safety to the mechatronic system design process by the functional failure identification and propagation framework. *Mechatronics* 22.2 (2012), 137–151. ISSN: 0957-4158. DOI: 10.1016/j.mechatronics.2012.01.003.

[52] C. J. Garrett, S. B. Guarro and G. E. Apostolakis. The dynamic flowgraph methodology for assessing the dependability of embedded software systems. *IEEE Transactions on Systems, Man, and Cybernetics* 25.5 (1995), 824–840. ISSN: 0018-9472. DOI: 10.1109/21.376495.

[53] S. Yamada. Elementary software reliability growth modeling. *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*. 2017, 2–10. DOI: 10.1109/ICRITO.2017.8342389.

[54] H. Langseth and L. Portinale. Bayesian networks in reliability. *Reliability Engineering & System Safety* 92.1 (2007), 92–108. DOI: 10.1016/j.ress.2005.11.037.

[55] O. Lisagor, T. Kelly and R. Niu. Model-based safety assessment: Review of the discipline and its challenges. *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*. IEEE, 2011, 625–632. DOI: 10.1109/icrms.2011.5979344.

[56] S. Sharvia, S. Kabir, M. Walker and Y. Papadopoulos. Model-based dependability analysis: State-of-the-art, challenges, and future outlook. *Software Quality Assurance*. Ed. by I. Mistrik, R. Soley, N. Ali, J. Grundy and B. Tekinerdogan. Boston: Morgan Kaufmann, 2016, 251–278. ISBN: 978-0-12-802301-3. DOI: 10.1016/B978-0-12-802301-3.00012-0.

[57] P. Fenelon and J. McDermid. An integrated tool set for software safety analysis. *Journal of Systems and Software* 21.3 (1993), 279–290. ISSN: 0164-1212. DOI: 10.1016/0164-1212(93)90029-W.

[58] M. Wallace. Modular Architectural Representation and Analysis of Fault Propagation and Transformation. *Electronic Notes in Theoretical Computer Science* 141.3 (2005), 53–71. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.02.051.

[59] Y. Papadopoulos and J. A. McDermid. Hierarchically Performed Hazard Origin and Propagation Studies. *Computer Safety, Reliability and Security*. Ed. by M. Felici and K. Kanoun. Springer Berlin Heidelberg, 1999, 139–152. ISBN: 978-3-540-48249-9. DOI: 10.1007/3-540-48249-0_13.

[60] M. Gudemann and F. Ortmeier. A Framework for Qualitative and Quantitative Formal Model-Based Safety Analysis. *2010 IEEE 12th International Symposium on High Assurance Systems Engineering*. 2010, 132–141. DOI: 10.1109/HASE.2010.24.

[61] P. Hönig and R. Lunde. A New Modeling Approach for Automated Safety Analysis Based on Information Flows. *25th International Workshop on Principles of Diagnosis (DX14)*. Graz, Austria, 2014.

[62] T. Prosvirnova. AltaRica 3.0: a Model-Based approach for Safety Analyses. PhD thesis. Ecole Polytechnique, 2014. URL: https://pastel.archives-ouvertes.fr/tel-01119730.

[63] A. Pfeffer. *Practical Probabilistic Programming*. Manning Publications Co., 2016. URL: https://www.manning.com/books/practical-probabilistic-programming.

[64] Y. Papadopoulos, M. Walker, D. Parker, E. Rüde, R. Hamann and Andreas Uhlig et al. Engineering failure analysis and design optimisation with HiP-HOPS. *Engineering Failure Analysis* 18.2 (2011). The Fourth International Conference on Engineering Failure Analysis Part 1, 590–608. ISSN: 1350-6307. DOI: 10.1016/j.engfailanal.2010.09.025.

[65] M. Lipaczewski, S. Struck and F. Ortmeier. SAML goes eclipse — Combining model-based safety analysis and high-level editor support. *2012 Second International Workshop on Developing Tools as Plug-Ins (TOPI)*. 2012, 67–72. DOI: 10.1109/TOPI.2012.6229813.

[66] P. Hönig, R. Lunde and F. Holzapfel. Model Based Safety Analysis with smartIflow. *Information* 8.1 (2017), 7. DOI: 10.3390/info8010007.

[67] M. Batteux, T. Prosvirnova and A. Rauzy. *AltaRica 3.0 Language Specification*. Tech. rep. AltaRica Association, 2015. URL: `https://www.openaltarica.fr/docs-downloads/`.

[68] J. Noble, A. Taivalsaari and I. Moore. *Prototype-Based Programming: Concepts, Languages and Applications*. London: Springer-Verlag, 1999.

[69] A. Rauzy. Guarded transition systems: A new states/events formalism for reliability studies. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability* 222.4 (2008), 495–505. DOI: `10.1243/1748006xjrr177`.

[70] A. Rauzy and C. Blériot-Fabre. Towards a sound semantics for dynamic fault trees. *Reliability Engineering & System Safety* 142 (2015), 184–191. ISSN: 0951-8320. DOI: `10.1016/j.ress.2015.04.017`.

[71] M. Lipaczewski, F. Ortmeier, T. Prosvirnova, A. Rauzy and S. Struck. Comparison of modeling formalisms for Safety Analyses: SAML and AltaRica. *Reliability Engineering & System Safety* 140 (2015), 191–199. ISSN: 0951-8320. DOI: `10.1016/j.ress.2015.03.038`.

[72] M. Batteux, T. Prosvirnova, A. Rauzy and L. Kloul. The AltaRica 3.0 project for model-based safety assessment. *11th IEEE International Conference on Industrial Informatics (INDIN)*. 2013, 741–746. DOI: `10.1109/INDIN.2013.6622976`.

[73] M. Bouissou. Automated Dependability Analysis of Complex Systems with the KB3 Workbench: the Experience of EDF R&D. *Proceedings of The International Conference on Energy and Environment, CIEM*. 2005. URL: `http://marc.bouissou.free.fr/KB3workbenchCIEM2005.pdf`.

[74] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler and F. Yergeau. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Specification. World Wide Web Consortium (W3C), 2008. URL: `http://www.w3.org/TR/xml/`.

[75] J. Gosling, B. Joy, G. L. Steele, G. Bracha and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. 1st. Addison-Wesley Professional, 2014.

[76] J. Waldo. Controversy: The Case for Multiple Inheritance in C++. *Computing Systems* 4 (1991), 157–171.

[77] J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms and Applications*. 2nd ed. Springer-Verlag, 2009. DOI: `10.1007/978-1-84800-998-1`.

[78] K. Ruohonen. *Lecture notes: Graph Theory*. Accessed: December 1, 2018. 2013. URL: `http://math.tut.fi/~ruohonen/GT_English.pdf`.

[79] T. Berners-Lee, R. Fielding and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. Specification RFC 3986. The Internet Society, 2005. URL: `https://tools.ietf.org/html/rfc3986`.

[80] S. Robinson. *Simulation: The Practice of Model Development and Use*. John Wiley & Sons, 2004. ISBN: 0470847727.

[81] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading: Addison-Wesley, 1995.

[82] R. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming* 1.2 (1988), 22–35. URL: `http://www.laputan.org/drc.html`.

[83] G. E. P. Box and M. E. Muller. A Note on the Generation of Random Normal Deviates. *The Annals of Mathematical Statistics* 29.2 (1958), 610–611. DOI: `10.1214/aoms/1177706645`.

[84] W. Q. Meeker and L. A. Escobar. *Statistical Methods for Reliability Data (Wiley Series in Probability and Statistics)*. Wiley-Interscience, 2014.

[85] J. Wenninger. *Simple models for the integrated luminosity*. Tech. rep. CERN-ATS-Note-2013-033 PERF. Geneva: CERN, 2013. URL: `http://cds.cern.ch/record/1553471`.

[86] O. Rey Orozko, A. Apollonio, M. Jonker and M. Paoluzzi. Dependability Studies for CERN PS Booster RF System Upgrade. *Proc. of International Particle Accelerator Conference (IPAC'16), Busan, Korea, May 8-13, 2016*. International Particle Accelerator Conference 7. Geneva: JACoW, 2016, 4159–4162. DOI: `10.18429/JACoW-IPAC2016-THPOY030`.

[87] J. Penttilä and V. Ikonen. Dynamic Simulation of Dependability of Nickel Reduction Plant. *ASME 2013 International Mechanical Engineering Congress and Exposition*. Vol. 15. ASME, 2013, V015T12A012. DOI: `10.1115/imece2013-64308`.

[88] K. Mahlamäki, A. Niemi, J. Jokinen and J. Borgman. Importance of maintenance data quality in extended warranty simulation. *International Journal of COMADEM* 19.1 (2016), 3–10. URL: `https://www.researchgate.net/publication/296596963`.

[89] W. Hopp and M. Spearman. Throughput of a constant work in process manufacturing line subject to failures. *International Journal of Production Research* 29.3 (1991), 635–655. DOI: `10.1080/00207549108930093`.

[90] J. Laitinen and P.-E. Hagmark. Modeling of closed spare part circulation for better availability of aircraft. *Advanced Reliability Modeling IV – Beyond the Traditional Reliability and Maintainability Approaches*. Taipei: McGraw Hill, 2010, 409–416.

[91] A. Apollonio, M. Jonker, R. Schmidt, B. Todd, S. Wagner and D. Wollmann et al. HL-LHC: Integrated Luminosity and Availability. *IPAC2013: Proceedings of the 4th International Particle Accelerator Conference*. Geneva: JaCoW, 2013, 1352–1354. URL: `http://accelconf.web.cern.ch/AccelConf/IPAC2013/papers/tupfi012.pdf`.

[92] A. Apollonio. Machine Protection: Availability for Particle Accelerators. PhD thesis. Vienna: Vienna University of Technology, 2015. URL: `https://cds.cern.ch/record/2002820`.

[93] E. McCrory and P. Lucas. A Model of the Fermilab Collider for Optimization of Performance. *Proceedings Particle Accelerator Conference*. Vol. 1. IEEE, 1995, 449–451. DOI: `10.1109/PAC.1995.504687`.

[94] E. McCrory. Monte Carlo of Tevatron Operations, Including the Recycler. *Proceedings of the 2005 Particle Accelerator Conference*. IEEE, 2005, 2479–2481. DOI: `10.1109/PAC.2005.1591151`.

[95] T. Himel, J. Nelson, N. Phinney and M. Ross. Availability and Reliability Issues for ILC. *2007 IEEE Particle Accelerator Conference (PAC)*. IEEE, 2007, 1966–1969. DOI: `10.1109/PAC.2007.4441325`.

[96] E. Bargalló, P. Sureda, J. Arroyo, J. Abal, A. De Blas and J. Dies et al. Availability simulation software adaptation to the IFMIF accelerator facility RAMI analyses. *Fusion Eng. Des.* 89 (2014), 2425–2429. DOI: `10.1016/j.fusengdes.2013.12.004`.

[97] M. Motyka. Impact of Usability for Particle Accelerator Software Tools Analyzing Availability and Reliability. MA thesis. Blekinge: Blekinge Institute of Technology, 2017. URL: `http://bth.diva-portal.org/smash/record.jsf?pid=diva2%3A1104892&dswid=398`.

[98] S. Epstein and A. Rauzy. *Open-PSA Model Exchange Format*. Specification 2.0.d-120-g703be91. The Open-PSA Initiative, 2017. URL: `https://open-psa.github.io/mef/`.

[99] L. Hillah, E. Kindler, F. Kordon, L. Petrucci and N. Tréves. A primer on the Petri Net Markup Language and ISO/IEC 15909-2. *Tenth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*. Aarhus: Department of computer science University of Aarhus, 2009, 101–120. DOI: `10.7146/dpb.v38i590.7187`.

[100] J. Moder and C. Phillips. *Project management with CPM and PERT*. New York: Van Nostrand Reinhold Inc, 1964.

[101] S. Nonsiri, E. Coatanéa, M. Bakhouya and F. Mokammel. Model-based approach for change propagation analysis in requirements. *2013 IEEE International Systems Conference (SysCon)*. IEEE, 2013, 497–503. DOI: `10.1109/SysCon.2013.6549928`.