UNIVERSITÀ
DEGLI STUDI
FIRENZE

Università degli Studi di Firenze
Dipartimento di Ingegneria dell'Informazione (DINFO)
PhD Course in Information Engineering

Curriculum: Computer Engineering
Academic Discipline (SSD): ING-INF/05

Timed Failure Logic Analysis in a Model-Driven Engineering approach

Candidate
Dott. Samuele Sampietro

Supervisors
Prof. Enrico Vicario

Prof. Alessandro Fantechi

PhD Coordinator
Prof. Fabio Schoen

cycle XXXIII, 2017-2020

Università degli Studi di Firenze, Dipartimento di Ingegneria
dell'Informazione (DINFO).

Acknowledgments

I would like to acknowledge the efforts and input of my scientific supervisors, Prof. Enrico Vicario and Prof. Alessandro Fantechi, thanking also my sienese best friend as well as all my colleagues of the Software Technologies Lab (STLAB) who were of great help during the PhD. In particular my thanks goes to Dott. Jacopo Parri, a dear friend who has completely shared with me this academic experience, also collaborating in the main parts of this research.

Abstract

A complex System of Systems, integrating several hardware and software components in the holistic perspective of providing an emergent behaviour and operating within business-critical contexts, aims at affording contrasting requirements of reliability and complexity in delivered functions and quality of services by supporting system evolution and adaptation over time.

This dissertation contributes to the area of Model-Driven Engineering (MDE), proposing a model-driven approach supporting timed failure logic analysis of complex Cyber-Physical Systems (CPS) in business-critical scenarios.

The research defines a meta-model joining structural information about system architectures with their failure logic, decoupling representations of communication interfaces from those of failure propagation. The meta-model also supports runtime evolution (which can be very fast in the case of complex CPS) of concrete systems by enabling the configuration of *product lines*, capable of representing multiple variation points of a component, supporting continuous adaptation of offered products and services to business or customer needs.

The meta-model enables a *round-trip engineering* process through the definition of a set of *transformation rules*, supporting the automated and *correct-by-construction* initialisation of meta-model instances starting from SysML Block Definition Diagrams for system specification and stochastic Fault Trees for timed failure logic, thus activating co-evolution mechanisms propagating external manual modifications, applied on meta-model instances, directly to the adopted structural and reliability artefacts.

At the same time, a set of *transformation rules* has been defined so as to enable the automated generation of Stochastic Time Petri Nets (STPN) from meta-model instances, thus supporting quantitative evaluation of the timed failure logic.

The MDE approach is demonstrated on the case study of a CPS operating in a Smart City environment, evaluating at design time different configurations of the system with respect to the reliability of its cyber-side.
The research also addresses the design and the prototypical implementation of a tool offered both *as-a-service* and as a Java API.

Contents

Abbreviations and Acronyms

- API: Application Programming Interface
- BDD: Block Definition Diagram
- CDF: Cumulative Distribution Function
- CPS: Cyber-Physical Systems
- DET: Deterministic probability distribution
- EXP: Exponential probability distribution
- FT: Fault Tree
- GEN: General probability distribution
- IMM: Immediate
- IoT: Internet of Things
- JSON: JavaScript Object Notation
- MDE: Model-Driven Engineering
- PDF: probability Density Function
- REST: REpresentational State Transfer
- SoS: System of Systems
- STPN: Stochastic Time Petri Net
- SysML: Systems Modeling Language
- XML: eXtensible Markup Language
- UML: Unified Modeling Language

Chapter 1

Introduction

Scientific, technical and technological advancement have promoted the adoption of emerging paradigms (e.g., cloud computing, Internet of Things, Industry 4.0) contributing in the design and realisation of complex System of Systems (SoS), integrating several hardware and software components in the holistic perspective of providing additional functionalities and of manifesting an emergent behaviour, not natively resident in individual subsystems.

In this context, many Cyber-Physical Systems (CPS) [67] operating within business-critical contexts (for which any critical malfunction or failure could have a strong social, legal and economic impact) aim at affording contrasting requirements of reliability and complexity in delivered functions and quality of services, by supporting system evolution and adaptation over time.

Model-Driven Engineering (MDE) [18] helps designers, technicians, and non-technical actors in the specification, design, integration, validation and operation of a system, reducing the gap between Systems Engineers and Reliability Engineers through the adoption of models and meta-models (as partial or simplified approximations of the concrete counter-part) conciliating design abstractions with reliability artefacts.

Reliability models [57] (e.g., Reliability Block Diagrams, Component Fault Trees) usually represent the failure logic of a system, capturing internal failure propagations only through provided communication interfaces (i.e., *direct couplings*) directly interconnecting hardware components.
Complexity of emerging CPS is often handled on the cyber-side through

the adoption of software communication middlewares (e.g., message-oriented brokers) orchestrating data-level communications among components (i.e., *indirect couplings*), decoupling them from infrastructural interconnections, also enabling dynamic adaption of roles and responsibilities for components. Thus, failures occurring on the data-level are propagated through these indirect interfaces producing a kind of indirect failure propagation.

In this research, an executable software meta-model - *a meta-model software implementation producing runtime objects as executable instances and methods* - overcoming state-of-the-art reliability models limitations while capturing the hierarchical composition of a SoS and its related failure logic, considering salient characteristics of stochastically-timed failure propagation mechanisms (along the system hierarchy through direct communication interfaces or among distinct subsystems through indirect interconnections) is presented. The meta-model also supports runtime evolution (which can be very fast in the case of complex CPS) of concrete systems by enabling the configuration of *product lines*, capable of representing multiple variation points of a component, led by adaptations of offered products or services to business and customer needs.

The meta-model acts as the core of a MDE approach, interpreting the shift towards high-level and semi-formal models according to industrial contexts and engineers' mental attitude [87], for supporting reliability evaluations of a system: the approach initialises the meta-model by exploiting industrial artefacts of the common practice and provides a set of transformation rules (to be applied on the meta-model) for producing in output a set of stochastic timed models (i.e., Stochastic Time Petri Net) related to the modelled systems. In so doing, the approach also enables subsequent quantitative analysis and simulation of the failure logic, through the adoption of pre-existent software reliability tools (e.g., Möbius, Oris). The approach demands for a semi-automated configuration process exploiting two artefacts:

- SysML Block Definition Diagrams (BDDs) for capturing the components hierarchy of a system with a proper detail level describing direct communication interfaces as well as redundancy levels;
- stochastic Fault Trees (FTs) for capturing error modes, to be intended as internal processes of components (or subsystems) leading a set of faults in manifesting a specific failure, as well as failure propagations

among distinct components (or subsystems) which transforms a failure manifested as an output of a source component in an input fault of a destination component (in a direct or indirect relationship). The adopted class of FTs enables a stochastic characterisation of underlying failure processes.

Under the MDE perspective, the proposed meta-model guarantees: on the one hand, a *correct-by-construction* configuration of structural information about system compositions as well as their runtime verification; on the other hand, an adaptive configuration of system failure logic, mapping reliability artefacts (enhanced with stochastic features) to failure propagation mechanisms.

Furthermore, the co-evolution [49] of the proposed meta-model with generated stochastic timed models (i.e., STPN), as well as with architectural artefacts (i.e., BDDs) and reliability artefacts (i.e., FTs) is enabled by the definition of dedicated transformation rules. In so doing, each modification of the meta-model configuration can be automatically transferred into output models.

The research has also addressed the design and the prototypical implementation of a Java-based Application Programming Interface (API) which enables the programmatic definition of a meta-model executable instance, retrieving information from input artefacts. This API can be considered as the core of a tool (offered *as-a-service*) for configuring a meta-model instance and for automatically deriving STPN models. In a general and functional perspective, the output produced by the API can be integrated within a Model-Based Dependability Analysis tool-chain, through the adoption of other tools for analysis, risk assessment, dependability qualities evaluations, safety parameters estimation, and formal verification of systems (e.g., GreatSPN, Mercury, Möbius, Oris).

The MDE approach is demonstrated on a synthetic but realistic case study related to a SoS operating in a business-critical context of a Smart City, evaluating at design time different configurations of the system with respect to the reliability of its cyber-side.

The rest of this dissertation is organised as follows. In Chapter 2 the literature review about primary Model-Driven Engineering reliability approaches and models is presented together with a brief introduction about main fail-

ure logic tools. A presentation of the proposed MDE approach for Timed Failure Logic Analysis of business-critical and complex Systems is provided in Chapter 3, also describing how Cyber-Physical Systems may be affected by failures not propagating, conceptually, among components only through direct communication interfaces. The designed meta-model at the core of the MDE approach is presented in Chapter 4 including a description of the *round-trip engineering* process leading the transformation rules from artefacts to meta-model instances, and *vice versa*. In Chapter 5 a description of the transformation rules leading the generation of timed analysis models in the formalism of Stochastic Time Petri Net, starting from meta-model instances, is provided. A prototypical implementation of a tool implementing key concepts of the MDE approach is presented in Chapter 6. In Chapter 7 a significant case study, showing the application of the MDE approach in a business-critical context, with experimental results is reported. Finally, conclusions and future research plans are drawn in Chapter 8.

1.1 Contributions

The research described in this dissertation proposes a model-driven approach supporting timed failure logic analysis of complex Cyber-Physical Systems in business-critical scenarios, thus contributing to the area of Model-Driven Engineering (MDE) for reliability.

The main contributions are here summarised:

- a MDE approach bridging the gap between the perspectives of System Engineering, Software Engineering and Reliability Engineering;
- the definition of a *meta-model* that decouples system architecture specification from failure logic, supporting also the modelling of product families in a *product line* perspective (designed so as to cope with runtime modifications, favouring reuse of created configurations);
- a set of *rules of model-transformation* implementing a process of *round-trip engineering* from meta-model instances to adopted artefacts (i.e., SysML BDDs for components hierarchy specification and decorated stochastic FT for failure logic modelling and stochastic parameters specification) and back, so as to enable a forward engineering process

automatically initialising the meta-model instance, guaranteeing consistency by-construction, as well as a reverse engineering process enabling extraction of specification and modelling artefacts from a meta-model instance;

- a further set of *rules of model-transformation* from a model instance to STPN models for enabling quantitative evaluation of timed failure logic;
- a *co-evolution* between the meta-model and output artefacts with the primary advantage that modifications directly applied over executable configurations of the meta-model automatically propagate to output artefacts: SysML BDDs and FTs and STPN models, respectively exploiting transformation rules of timed failure logic and reverse *round-trip engineering*;
- an implementation of key components in a prototypical Java API, also at the core of a *tool-as-a-service*, featuring the key steps of a tool-chain that supports the MDE approach, opening the way to full automation. In so doing, the API enable an export process of associated STPN models, in a general format suitable in input to external analysis tools (e.g., Oris);
- an experimentation on a realistic case in the application context of a Pollution Monitor System within a Smart City environment.

Chapter 2

Literature review

This Chapter gives a brief survey of related works about main models and techniques for reliability evaluation of System of Systems and Cyber-Physical Systems adopting Model-Driven Engineering (MDE) approaches.

The theory of MDE and major works addressing the problem of co-evolution among systems and models are reported in Sect. 2.1. Structural and behavioural models, failure propagation models and (timed) analysis models are reviewed in Sect. 2.2. Primary tools for reliability and failure logic analysis are presented in Sect. 2.3. Finally, state-of-the-art MDE approaches for dependability are described in Sect. 2.4.

2.1 Model-Driven Engineering

In many design engineering approaches, the definition and the adoption of models as partial or simplified abstractions of concrete under design hardware and software components has become increasingly relevant. This consideration is exacerbated in the design and development of Information Technology (IT) platforms and systems based on digital software components, whose intrinsic behaviour and features strictly lay on intangible properties, modelling a kind of approximation of physical reality.

Models help designers, technicians, and non-technical actors in the specification, design, integration, validation and operation of a system; also sharing the evolution and scheduling of project activities and, at the same time, in consolidating a common vision about operative domain, fundamental requirements, superimposed constraints, and emerging functionalities.

A sufficiently broad and general categorisation of techniques and approaches based on models belongs to the *Model-Based Engineering* (MBE) area, also known as *Model-Based System Engineering* (MBSE), in the specific case of complex systems or System of Systems (SoS) [31].

In this broad research area, many theories have been issued and multiple definitions formulated: the survey in [24] and the work in [87] clarify and discuss relations between different taxonomies, defining an ontology about the concepts of system, model, meta-model, and modelling language.

First of all, *Model-Driven Engineering* (MDE) [18, 89] can be considered as a high-level general purpose software engineering approach which uses models as primary documentation artefacts, leading all the design, development, and verification stages without mandatorily demanding for a concrete tool support for the final implementation. Under this approach, design artefacts may also enable the automated or semi-automated derivation of further analysable models (e.g., a mathematical model or a Stochastic Time Petri Net), reflecting relevant and focused characteristics and behaviours coherently with the operative domain context.

Model-Driven Development (MDD), can be considered as another high-level approach which specialises MDE, focusing more on the adoption of models and artefacts to realise a specific development process. MDD is the common base in many Software Engineering methodologies, covering requirements, analysis, design and implementation stages with support to automated code generation of complete programs from models, and automated

verification of models (e.g., by executing reverse engineering processes). [91]

Model-Based Testing (MBT), can be considered as another specialisation of MDE, mainly oriented to automated testing through the definition of testing models, capturing the behaviour of a system under test. MBT approaches may depend on testing tools and frameworks. [4, 104]

Finally, *Model-Driven Architecture* (MDA) can also be considered as a MDE specialisation, proposed by OMG (Object Management Group), separating business and application logic from underlying platform technologies at different levels of abstraction (i.e., Computational Independent Models, Platform Independent Models, Platform Specific Models) so as to support automated model transformations between lifecycle phases. [96, 114]

A connection established on Model-Driven approaches between the perspectives of Software Engineering and Reliability Engineering may largely help in accommodating the contrasting needs of complexity and reliability, by making reliability models be structurally related to software design artefacts and their evolution through model-to-model transformations.

2.1.1 Co-evolution of systems and models

Model-Driven Engineering approaches have to consider the continuous evolution of system models and meta-models, in their concrete and designed configurations, as a major challenge for designers and developers. Indeed, during software system lifecycle, the technological progress and the operational domain may evolve imposing a change in requirements and system specification, thus increasing the need for a strategy to address the co-evolution problem, considering also the migration problem among existing models to updated versions of related meta-models. In literature, the co-evolution of systems/models/meta-models, also known as *coupled evolution* [50] or *co-adaptation* [110], aims at supporting migration processes enhancing productivity and reducing errors through the adoption of domain-specific modelling languages, frameworks and tools.

Many researches aiming at implementing tools for handling the co-evolution problem are based on the Eclipse Modeling Framework (EMF) [97], a collection of Eclipse plug-ins which supports data modelling, persistence, and automated code generation through the definition of meta-models. In EMF, in a reflexive perspective, a model consists in a concrete instance of a meta-

model which is built through Java annotations, UML, XMI or XML schemes.

In [23], a transformational approach is proposed for handling co-evolution between meta-models and models, specifically addressing the representation of software systems, by defining a higher-level meta-model capturing admissible changes over the lower-level meta-model (e.g., the modification, the deletion, or the addition of a class, an attribute, or a reference), thus leading the automated adaptation of the system model. The approach has been supported by an EMF-based tool, named *EMFMigrate* [26] which relies on a domain-specific language.

In [50], an integrated approach favouring reuse and expressiveness in specifying coupled and transactional transformations of meta-models and models is presented. The approach, named COPE, offers a set of primitives based on EMF with the aim of reducing migration efforts, thus complying with evolving syntax rules of different modelling language versions.

In [5, 99], the Henshin framework is described as a common foundation for a wide range of tool by offering a transformation language to manage models defined in the EMF [97] for supporting development processes while providing also model checking and analysis capabilities.

In [43], a solution to the problem of co-evolution of a system architecture with its quality models (e.g., Fault Trees) has been proposed by defining a set of transformation rules, designed so as to support the continuous adaptation and synchronisation of inter-related models, thus reducing the engagement of developers expertise. The approach, supported by a tool named *CoWolf* [42], has been experimented over a motivational scenario related to an automation engineering domain [44], where authors demonstrated that fully automated processes cannot resolve this specific co-evolution problem, which demands in some measure for the human expertise.

In [65], an approach for defining co-evolution of meta-models and models defined with EMF is presented, demonstrated through static and runtime consistency checks, and released as prototype tool based on Henshin. The approach is based on transformations and introspection rules applied over two EMF graph abstractions in mutual dependence, respectively a meta-model typed graph and a model instance graph: *i)* the meta-model graph includes nodes representing primitive typed attributes, while the model graph includes nodes related to conceptual entities (e.g., a place or a transition) living within the adopted model context (e.g., a Petri Net); *ii)* in both graphs, edges represent relationships among interconnected nodes.

2.2 MDE models

In industrial and engineering contexts, concrete development and runtime maintenance of systems, especially for the case of embedded systems [68,69], leverage the adoption of MDE approaches, continuously supporting domain experts, technicians and engineers in structural and behavioural design. The application of MDE techniques within complex SoS or CPS have not yet been largely documented to be considered as a consolidated practice: a shift from formal verification and fine-grained models (e.g., AADL) towards higher-level models (e.g., UML-based models) is required. [87]

To these purposes, many useful abstractions and artefacts have been introduced in the practice and in literature; also for modelling failure logic characteristics of a system or for enabling analysis processes (e.g., descriptive, predictive, prescriptive).

In this section, a review of significant abstractions is reported, with three different perspectives:

- architectural and behavioural models;
- failure propagation models;
- timed analysis models.

2.2.1 Architectural and behavioural models

MBE approaches entail systematic adoption of artifacts and graphic constructs in the design and development of Information Technology (IT) systems. In this area, the default modelling standard, defined by OMG (Object Management Group), is the *Unified Modeling Language* (UML) [17], which provides a set of useful and extensible diagrams, classified in structural and behavioural diagrams, respectively describing static structure (e.g., prototype) and dynamic behaviour (e.g., communications) of a system.

Structural diagrams notably include Class Diagrams (which model a set of classes, entities, and interfaces with their designed relationships), Object Diagrams (which model a set of static snapshots of object instances and their runtime relationships), Composite Structure Diagrams (which model the internal structure of multiple classes showing the interactions between them, providing a logical view of a part of a software system) and Package Diagrams (which model dependencies among packages of a layered system), Component Diagrams (which model the static implementation view

of a system, representing components organisation and dependencies), and Deployment Diagrams (which model a static configuration of a deployment hardware/software system, with details with run-time processing nodes and inner components).

Behavioural diagrams notably include Activity Diagrams (which model the flow of control among objects, highlighting the dynamic view of a system), Statechart Diagrams (which model state machines, consisting of states, transitions, events, and activities), Use Case Diagrams (which model available use cases for each designed actor of the operative context), and the isomorphic Collaboration and Sequence Diagrams (which model interactions among objects, emphasising the time-order of messages).

A major characteristic of the UML notation is the extensibility of its syntax and semantics, which guarantees the generation of *ad hoc* languages for specific application domains, overcoming limitations derived from the general purpose nature of UML. In so doing, the main extension mechanism consists in the definition of *UML Profiles* [37], grouping together a set of stereotypes, tagged values, and constraints for customising diagrams elements.

Among these profiles, the state-of-the-art *UML Profile* for supporting the specification, analysis, design, verification, and validation of a wide class of hardware/software systems (e.g., Systems of Systems, Cyber-Physical Systems), is the UML 2 Dialect named *Systems Modeling Language* (SysML), officially proposed and disciplined by OMG. [48] SysML includes diagrams that can be used to specify system requirements, behaviour, structure and parametric relationships, adopting *as-is* a subset of UML diagrams, modifying others, as well as defining new diagrams in response to address requirements and parameters formalisation.

Specifically, while SysML reuses Package Diagrams, Sequence Diagrams, Statechart Diagrams, and Use Case Diagrams, with slight or no syntax modifications, under a higher-level perspective about systems and subsystems functionalities, SysML defines: *i)* Block Definition Diagrams (BDD), static structural diagrams, extending the UML Class diagrams, with the purpose of representing system components in the form of blocks decorated with properties and interfaces. A BDD describes the hierarchy of a system and depicts system/components relationships; *ii)* Internal Block Diagrams (IBD), static structural diagrams, extending the UML Composite Structure Diagram, with the aim of modelling the internal structure of a BDD block, decomposing it

in Parts, Properties, Connectors, Ports, and Interfaces. An IBD can be summarised as a "white-box" perspective of an encapsulated block; *iii)* Parametric Diagrams (PAR), brand new structural diagrams which specialise IBDs to represent constraints on system properties through parameterized blocks (i.e., *ConstraintBlock*). A block models a constraint on a system parameter value (e.g., performance, reliability) and can contain mathematical equations or statistical values; *iv)* Activity Diagrams (ACT), behavioural diagrams extending the homonymous UML diagram, exploiting control and object data flows, to specify how the system dynamics satisfies functional requirements; *v)* Requirements Diagrams (REQ), brand new and static structural diagrams, qualifying requirements, highlighting how model elements satisfy them, and identifying which test case verify them.

In model-based analysis and specification of complex real-time embedded systems, *Architecture Analysis & Design Language* (AADL) [34], defined by Society of Automotive Engineers (SAE), represents a relevant industrial standard in modelling safety-critical domains. AADL includes different abstractions to represent application software components (i.e., thread, thread groups, process, static data, and subprogram), execution hardware platforms and resources (i.e., processor, memory, device, and bus), composite components (e.g., aggregations of sub-components), and error models (e.g., for specifying faults and theirs behaviours). The specification enables designers to express formal models about embedded systems with respect to performance-critical properties through textual artefacts, eXtensible Markup Language (XML) documents, graphical notation or through their combination. In so doing, information can be expressed interchangeably for convenience, with one or more alternative representations according to the expected usage (e.g., machine readable vs human readable formats). Among its significant semantic characteristics, AADL includes the definition of elements to express exchange and data control mechanisms (e.g., thread scheduling, synchronised access to shared components, remote procedure calls), also including operational modes and transitions to enable dynamic reconfiguration of the runtime architecture.

The *Modeling and Analysis of Real-Time and Embedded* systems (MARTE) is a standardized OMG *UML Profile* adding capabilities for model-driven development of real-time and embedded systems. [40] MARTE includes, at the same time, the definition of high-level system concepts about quantitative

features (e.g., delay, duration or clock time) or qualitative features (e.g., ordering of events in time) and the modelling of low-level structural concepts about hardware and software resources. Specifically, MARTE enables the design of AADL-driven systems, describing time properties, performance and scheduling features, enabling MDE approaches to system design and development. In this way, the designer would take benefit from verification and validation tools and techniques dedicated to both the abstractions. [32]

2.2.2 Failure propagation models

Widely adopted formalisms in system reliability, enabling failure propagation analysis are: *i)* Fault Trees (FT) and *ii)* Reliability Block Diagrams (RBD).

A standard FT, also known as static FT, is a graphical model which represents failure modes cause-effect relationships within a system through the definition of three types of nodes: events (e.g., basic, intermediate or top events), logic gates (i.e., *AND*, *OR*, *k-of-n*, and *INHIBIT* gates) and transfer symbols (i.e., transfer *In* or *Out* symbol). In so doing, FTs can be exploited to implement qualitative analysis techniques (e.g., minimal cut sets, minimal path sets, or common cause failures) so as to detect system vulnerabilities, starting from an ordered visit of the FT structure, or quantitative analysis techniques so as to detect components critical issue levels (e.g., through importance measures) or components failure probabilities (e.g., through probabilistic measures). [86]

A Reliability Block Diagram (RBD) is a graphical and mathematical model which decomposes a system into blocks of subsystems, specifying the conditions necessary for correct functioning of the system within a *success-oriented* flow logic. Similarly to FTs, RBDs enable quantitative analysis and qualitative analysis aimed at determining system reliability or component availability, as well as identifying (minimal) cut sets and importance measures. [112] The RBD formalism enables both the definition of a higher-order logic of common configurations (e.g., series, parallel, parallel-series and series-parallel) and the formal verification of their equivalent mathematical expressions. [1]

The subtended complementary nature of these two graphical models is highlighted and confirmed by the fact that, in most cases, an RBD can be converted to a FT and *vice versa* by interpreting a parallel connection as an AND gate and a series structure as an OR gate. [116]

The demand for capturing sequence-dependent failures (i.e., sequential relationships among failures manifested on distinct components within a system) leaded the extension of traditional formalisms with the introduction of their respective dynamic versions.

On the one hand, Dynamic Fault Trees (DFT) extend FTs by introducing new dynamic gates (i.e., Cold/Hot-Spare gate, Functional-Dependency gate, Priority-AND gate, Sequence-Enforcing gate) [30, 51], also enabling stochastic characterisation of basic events.

On the other hand, Dynamic Reliability Block Diagrams (DRBD) have been introduced to enhance traditional RBDs with innovative capabilities of DFTs, by introducing the possibility of representing redundant units (i.e., parallel connection of its simple instances) and load sharing policies (i.e., two or more units share the same workload) [28, 29].

A comparison between DFT and DRBD formalisms is reported in [27], also defining a mapping of DFT to the DRDB domain as well as a feasible reverse process.

Components Fault Trees (CFT), firstly proposed in [61], are directed acyclic graphs describing failure flows directly derived from system ports/interfaces among components in an architectural perspective. A FT, usually, is modelled complying with the hierarchy of failure influences rather than the system compositional hierarchy; in so doing, CFTs have been designed so as to cope with modularisation of sub-trees and for overcoming the independence assumption among events (e.g., a component failure may flow out to more than one external component, generating repeated events).

Finally, the Failure Propagation and Transformation Notation (FPTN) [35, 36] is a formal notation for graphically specifying failure behaviour of a system. FPTN is particularly suitable for modelling complex systems, enabling designers to model composite structures through the concept of *module*, the the atomic element related to a single component depicted as a box and decorated with incoming and outgoing failure modes as well as its SIL level. In so doing, the notation provide the ability of capturing the propagation of failures along the system architecture.

2.2.3 Timed Analysis models

Many MDE approaches adopt general purpose timed models so as to characterise the temporal behaviour of a system, also in reference to dependability analysis processes enabling continuous time failure propagation studies. The practical adoption of these models requires a great effort to designers and analysts, which must be experts, at the same time, about the operative domain and the underlying mathematical formalisms as well as about suitable quantitative analysis techniques (e.g., numerical or simulated).

For overcoming this problem, many tools and methodologies (as respectively described in Sect. 2.3 and in Sect. 2.4), leverage on higher-level abstractions, describing systems architectures and failure propagations mechanisms, so as to (semi-)automatically derive lower-level timed analysis models, sharply separating the tasks of designers and analysts.

In the area of timed analysis model, Petri Nets (PNs) [81] are a significant abstraction. While classical PNs constitute a powerful formalism to model, to analyse, and to simulate behaviours of systems subject to discrete events or discrete actions, several extensions had been proposed in literature so as to address also continuous time events. A PN is a directed graph whose primary elements are places and transitions among places. Tokens contained in places move within the net in compliance with oriented edges directing their flow. Usually, transition firings represent modelled system events, while tokens and places represent the status of system in a custom/interpreted semantics.

Timed Petri Nets (TPNs), also known as Timed Transitions Petri Nets (TTPN), introduce the concept of time, classifying the nature of the model basing on typology of adopted transitions:

- with only immediate (IMM) transitions (i.e., associated to a probability density function of Dirac's delta function with parameter t so that $t = 0$), a TPN is equivalent to a classical PN;
- with only deterministic (DET) transitions (i.e., associated to a probability density function of Dirac's delta function with parameter t so that $t \geq 0$ and $t \in \mathbb{R}^+$), a TPN is named Deterministic Timed Petri Net (DTPN) [45];
- with only stochastic exponential (EXP) transitions, a TPN is named

Stochastic Petri Net (SPN) [83] and it is isomorphic with a Continuous-Time Markov Chain (CTMC), enabling the representation of negative exponential distributions in $\mathbb{R}^+$ for modelled events.

SPN formalisms have been extended in the Generalized Stochastic Petri Net (GSPN) [2, 72] so as to include inibithor arcs, priorities among transitions, and immediate transitions together with exponential transitions. In turn, GSPN have been enhanced in Stochastic Time Petri Net (STPN) [108] so as to cope also with general distributions on custom temporal supports equal to or contained in $\mathbb{R}^+$ (e.g., deterministic distributions, uniform distributions).

2.3 Reliability and failure logic Tools

Many lines of research addressing risk assessment, dependability qualities evaluations, safety parameters estimation, and formal verification of systems have prolifically contributed in the design and definition of tools for analysis and evaluation of system reliability and failure propagation. In this Section, a review of main research tools adopting fundamental modelling abstractions is reported.

ADAPT (from AADL Architectural models to stochastic Petri nets through model Transformation) [85] is a research tool for evaluating system dependability measures (e.g., reliability and availability) starting from AADL models. The tool interfaces OSATE (see below), enabling automated transformations of AADL architectural models to GSPN models.

ASTRO [94], a dependability modelling tool which exploits RBDs, Stochastic Petri Nets, and Continuous-Time Markov Chains; as input abstractions so as to enable dependability evaluations and simulations from different perspectives (i.e., specialised or non-specialised users) experimented within the specific context of data centers infrastructures.

CHESS [38, 75], a tool-set part of a framework for design, development, and analysis of safety-critical component-based systems in embedded and real-time contexts. Specifically, the tool-set defines a custom meta-language, named CHESS-ML, defined as a collection-extension of standard OMG languages (i.e., UML, MARTE and SysML), exploited by two safety analysis techniques: CHESS-FLA, performing Failure Logic Analysis through FI^{4}FA (Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures Analysis) [39], and CHESS-SBA, performing quantitative State-Based Analysis for dependability.

DEEM (DEpendability Evaluation of Multiple-phased systems) [14, 15] is a tool enabling performance and dependability analyses for system which must perform a series of sequentially ordered tasks for components subject to a particularly stressing environments, which may change configuration over time. DEEM adopts DSPN as the modelling abstraction and Markov Regenerative Processes(MRGP) for numerical solutions.

GreatSPN (GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets) [3] is a long-lived tool for modelling, validation, and performance evaluation of distributed systems based on Generalized Stochastic Petri Nets and Stochastic Well-formed Nets (i.e., a coloured extension of GSPN, defined as a syntactic restriction of Stochastic High-Level Nets with the possibility of exploiting Symbolic Reachability Graphs). Among its purposes, the tool provides a framework to experiment with Markovian solvers (i.e., steady state and transient performance evaluation) and interactive simulations.

Mercury [95] is a tool which provides an environment to model systems through the adoption of different kind of formalisms (i.e., CTMC, RBD, Energy Flow Models, and Stochastic Petri Nets). In particular, it includes *ad hoc* graphical editors for each typology and a scripting language for programmatic models construction and evaluation of dependability and performance.

Möbius [25] provides an integrated multi-formalism multi-solution approach exploiting Stochastic Activity Networks, Stochastic Petri Nets, and Stochastic Process Algebras, as well as Fault Trees and RBDs, with different analytical and numerical solvers through the use of a custom functional interface, enabling interaction among models and solvers.

Oris [78], a toolbox for quantitative evaluation of concurrent models with (non-)Markovian timers, including a fluent Java Application Programming Interface and a Graphical User Interface for modelling Stochastic Time Petri Nets. Oris provides several *built-in* engines for numerical solutions and simulative methods for steady-state and transient analysis of underlying stochastic processes.

OSATE (Open Source AADL Tool Environment) [33] offers an open source workbench developed through eclipse so as to provide a graphical editor, exploiting various models (e.g., AADL, RBD, Error Model Version 2), for modelling architectural and behavioural features of a system, providing also flow latency analysis, safety analysis (e.g., FMEA, FTA), and automated verification of functional and non-functional system properties.

SHARPE (Symbolic Hierarchical Automated Reliability and Performance

Evaluator) [103], a mature tool, adopted in the last decades by students, practising engineers, and researchers, offering analysis solutions for stochastic models of reliability, availability, performance, and performability through the implementation of algorithms based on various models (e.g., Fault Trees, RBDs, acyclic series-parallel graphs, acyclic and cyclic Markov and semi-Markov models, Generalized Stochastic Petri Nets).

SMART [22] applies numerical solution and simulation techniques over Petri Nets, discrete-time and continuous-time Markov chains (i.e., CTMC and DTMC), through the adoption of statements written in the SMART language (i.e., a strongly-typed computation-on-demand language), which permits the programmatic definition of constituent elements of a model (e.g., transitions with a lambda rate).

TimeNET [118] is a graphical tool for modelling and executing performability evaluations on Stochastic Petri Nets and its coloured variants, supporting rare-event simulation algorithms for these models as well as transient and steady state analyses over exponentially and non-exponentially distributed firing times.

2.4 MDE for Dependability

The adoption of Model-Driven Engineering approaches to the area of dependability analysis takes the name of Model-Based Dependability Analysis (MBDA) [60]. Specifically, for handling the increasing system complexity in several safety-critical or business-critical contexts[1], MBDA aims at reducing erroneous scenarios[2] by (semi-)automatically synthesising, from underlying design models, dependability information representing structural characteristics[3] and behavioural aspects[4]. As a logical consequence, also the co-evolution of dependability models and meta-models (as described in Sect. 2.1.1) becomes a key factor for designers and technicians in order to guarantee safety, reliability, and maintainability of the assets under monitoring/analysis.

MBDA approaches primarily rely on two leading paradigms [92]:

[1] e.g., industrial plants, automotive, healthcare, energy

[2] e.g, system configurations where data inconsistencies or data incompleteness may produce unexpected system behaviour

[3] e.g., the system hierarchical composition

[4] e.g., the system failure logic

- Failure Logic Synthesis and Analysis (FLSA) consists in the analysis of (semi-)automated generated system failure models (e.g., FTs, CFTs), often exploiting hierarchical and component-driven abstractions describing the system topology;

- Behavioural Fault Simulation (BFS) consists in the application of formal verification methods on system behavioural models (e.g., Finite State Automata, Stochastic Petri Nets) adopting fault injection techniques so as to simulate erroneous scenarios.

This dissertation addresses FLSA techniques, focusing on the *Fault Tree Analysis* (FTA) [106] family, exploiting different FT abstractions (e.g., Standard Fault Trees, Dynamic Fault Trees) so as to cope with different investigation intents. In general, while the analysis process can be easily automated, the designing stage of the FT artefacts is usually handcrafted by domain experts (e.g., a reliability engineer) after a stage of Failure Modes Effects Analysis (FMEA) [16, 19] or Failure Modes Effects and Criticality Analysis (FMECA) [20]. Nevertheless, several approaches have been proposed so as to automatically derive FTs from structural or behavioural artefacts, as described in [59, 70, 74], respectively adopting Finite State Automata, AADL notations, and SysML IBDs. The survey in [11] synthesises and classifies efforts done in this research topic specifically for UML-based models.

FTA can be performed on two distinct levels: *i)* on a *qualitative* level for identifying minimal cut-sets or root-causes analysis by applying an ordered visit along the structure of the FT; *ii)* on a *quantitative* level for detecting components criticalities through the computation of importance measures (e.g., Birnbaum) or stochastic characterisation of components failure probabilities. A complete survey about main qualitative and quantitative FTA algorithms is addressed in [86].

In [79], the Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) method is described, which executes FTA and FMEA analyses on FTs, automatically generated by visiting handcrafted compositional system models decorated with failure annotations. A technique based on HiP-HOPS, exploiting a temporal extension of FTs, is presented in [111] so as to enable analysis of systems where the order of occurred faults and manifested events is fundamental to model the failure behaviour.

In [47], a MBDA approach based on CFTs is presented; starting from system components identification, a FTPN module describing the internal

failure logic is annotated for each component. FTPN modules lead a subsequent automated derivation of CFTs, which in turn enable quantitative analysis of the system-level failure by hierarchically composing CFTs to the whole system.

The Open Modelling approach for Availability and Reliability of Systems (OpenMARS), an approach for risk assessment of complex industrial systems, is proposed in [80]. OpenMARS exploits at the same time failure logic models (e.g., FTs, RBDs) and behavioural abstractions (e.g., Markov models, Petri Nets) also combining different analysis techniques (e.g. FTA, Petri Net analysis).

A framework providing a Reliability Configuration Model (RCM) and a Static Fault Tree Model (SFTM) so as to embed system configurations and and error mechanism for enabling reliability analysis and automatic static FT generation is described in [115].

In [90], a custom SysML diagram (i.e., FMEA) for supporting design and documentation of FMEA information (e.g., faults, failure modes, propagation effects) for critical systems, thus enabling qualitative and quantitative analyses stages, is proposed.

Finally, some works address the proposal of intermediate models bridging the gap between high-level formalisms and quantitative models, in the same research direction followed by this dissertation.

A MDE approach exploiting a model-transformation from design artefacts (e.g., UML-based) to quantitative analysis models (e.g., Queueing Networks, Petri Nets, Markov Processes) based on a intermediate meta-model named KLAPER (Kernel LAnguage for PErformance and Reliability analysis) is described in [46]. This kernel language aims at reducing the gap between design-oriented and analysis-oriented notations, supporting in the generation of performance and reliability models from design models (in different notations).

In [76], a discussion about main dependability concerns to be considered during dependability analysis is addressed with the aim of defining a conceptual model, proposing an intermediate dependability model between high-level engineering languages (e.g., UML) and low-level dependability analysis formalisms (e.g., GSPN), actually at the core of the CHESS tool.

Chapter 3

The MDE approach for business-critical Systems

In this Chapter, a reasoned overview about System of Systems and their classification is addressed with the aim of contextualising the problem of failure propagation analysis with reference to systems salient characteristics (i.e., structural and behavioural aspects).

In particular, complex IT/OT systems operating within business-critical contexts are described in Sect. 3.1, where fundamental keywords (i.e., direct/indirect couplings, fault-to-failure, failure-to-fault) are introduced and defined. Sect. 3.2 introduces the Model-Drive Engineering approach, leveraging on the proposed meta-model of Chapter 4 for reducing the gap between design artefacts (i.e., structural models and reliability models) and analysis stages, where executable software models are adopted for enabling quantitative reliability evaluations.

3.1 Business-critical Systems

Scientific, technical and technological advancement, sustained by an increasing qualification of personnel and a greater inclination for innovation, has promoted the adoption of emerging paradigms (e.g., cloud computing, Internet of Things, Industry 4.0) in multiple public authorities, enterprises, and organisations operating in various fields (e.g., smart city, industrial manufacturing, environmental monitoring, smart healthcare, automotive).

In so doing, Information Technology (IT) and Operational Technology (OT) sectors are contributing in creating sophisticated software systems capable of satisfying companies needs, affording requirements of growing complexity in delivered functions and quality of services by supporting system evolution and adaptation over time. This concept is emphasised for software intensive systems [58]; this broad class of systems supports *business-critical* operations, for which any critical malfunction or failure could have a strong social, legal and economic impact.

To better contextualise the problem of failure propagation analysis within complex *business-critical* systems, it is necessary to briefly review salient features of main system families of interest.

According to the widely adopted definition in [13]: "a *system* is a collection of entities and their interrelationships gathered together to form a whole greater than the sum of the parts".
The above definition enlightens that a system must be considered as a hierarchical aggregation of different interacting components, designed as an ensemble so as to provide common and predefined functionalities.

The continuous evolution in terms of design and development with an increasing integration among system entities led to the conceptualisation of a new terminology which considers the case of distinct interconnected systems, namely *subsystems*, often distributed on a large geographic scale (as in the case of IT systems where modules may be deployed on the cloud as well as at-the-edge of Internet), manifesting an emergent behaviour. [71]

Specifically, *System of Systems* (SoS) have been defined in [66]: "a system of systems is a set of different systems so connected or related as to produce results unachievable by the individual systems alone". Under this perspective, a SoS can be seen as a complex entity equipped with high adaptability, able to provide additional functionalities not natively resident in individual subsystems, thus concretising an holistic vision which supports operational

and managerial independence.

An inner classification of SoS, based on their characterising aspects in terms of coordination, runtime behaviour, autonomy, and complexity, has emerged from literature.

A SoS commanded and controlled under many authorities, with a limited central power, is named *Federation of Systems* [88]. In this category, a SoS is managed by a federated coalition of partners through an active collaboration and coordination based on uniform agreements, standardised communication protocols, and shared behaviours while guaranteeing separation of powers and subsidiary interdependence.

Ultra-large-scale SoS highly adaptable at runtime in response to architectural, parametric, as well as functional changes, not expected at design-time, are named *Interwoven Systems* [102]. In this category, subsystems are characterised so as to face challenges in maintaining desired functional performances by autonomously self-organising and self-optimising their own behaviour, while achieving the overall system specification requirements reducing conflicts which may occur during operational lifecycle, being aware of existent mutual influences.

In general, systems must be considered as a hierarchy of entities (i.e., inner components or distinct subsystems) interacting through dedicated hardware and software communication interfaces complying with specific protocols and exchange data-contracts. As a consequence, the number of communication interfaces directly influences the complexity of system architectures, requiring an increasing effort in design, conduct, and control operations, further exacerbated within SoS and even more in *Federations of Systems* and *Interwoven Systems*.

Along interfaces the information, in its various kinds (e.g., data packets, action commands, electric signals), is exchanged between system entities contributing to the overall system control and functioning, thus generating a kind of communication flow. In so doing, also incorrect information, which may generate failure scenarios, converges through the same communication flow crossing all the provided interfaces in the designed order.

In the perspective of distributed systems, as in the case of a SoS, the communication flow follows two different modes:

1. *intra-subsystem* mode, when the information traverses local interfaces interconnecting inner components within a single subsystem. In this

case, the information mainly follows the physical and concrete structure specification;

2. *inter-subsystems* mode, where information traverses interfaces interconnecting remote components belonging to different subsystems. In this case, the information mainly respects functional requirements and integration policies following the separation of concerns paradigm.

Note that, the communication flow intrinsically generates dependencies in *failure-to-fault* couplings among interconnected components (or among distinct subsystems) in the sense that a failure manifested by a component may be propagated through interfaces in dependent components, generating a fault and thus entering in an error mode.

Logically, communication interfaces create low-level propagation couplings in terms of overall system reliability (e.g., a power supply failure or a Wi-Fi repeater failure may cause consequential faults in each interconnected device by, respectively, turning off or disconnecting dependent components).

At the same time, also couplings at an higher level of abstraction may be produced among components not directly interconnected by interfaces. These couplings are conceptually bound to failures originated within exchanged information data/meta-data (e.g., header, payload), which are propagated only to dependent components which should consume, interpret and manipulate them. Indeed, the components which exclusively route these information along the communication flow are not aware about the presence of errors, corruptions or missing data. These components cannot be considered as in an error state, while components dependent on exchanged data may become faulty.

The presence of couplings among components not directly interconnected is exacerbated within the class of SoS, widespread in the industrial context, named Cyber-Physical Systems (CPS). CPS are characterised by a software side that coordinates and controls components distributed across the overall system, assuming the responsibility of affording the functional and behavioural complexity. In this scenario, the cyber-side assumes a central role, giving intelligence to the system in supporting operation and integration of controlled assets, enforcing policies and governing enterprise components for which resilience comprises a core requirement.

As a consequence, from the point of view of the overall SoS, failures propagation is led by existing couplings among components of each subsystem. In

this dissertation:

- low-level propagation couplings among components (or subsystems) interconnected through *intra-subsystem* and *inter-subsystems* interfaces are named *direct couplings*; these couplings comply with the structural specification of the system;
- high-level propagation couplings among components (or subsystems) not directly interconnected by interfaces are named *indirect couplings*; these couplings comply with use case scenarios and business logic modelled within the orchestration logic embedded into the software controllers.

A SoS, to all effects, is much more than a container exploiting a collection of gathered subsystems acting as autonomous entities with different aims so that the integration logic realises a masterplan to offer value-added services, according to an Enterprise Application Integration vision. Indeed, SoS adopting Enterprise Integration Patterns for modelling inter-connections among subsystems, as in the case of Enterprise Service Bus (ESB), message-oriented Internet of Things brokers, and micro-service architectures, are not trivial to be managed by system designers, reliability engineers and maintenance technicians.

In the specific case, reliability analyses should not focus only on the study of *fault-to-failure* propagation within a single component but should also stress *failure-to-fault* propagation among different subsystems, basing on modelled *direct* and *indirect couplings* (see Fig. 3.1).

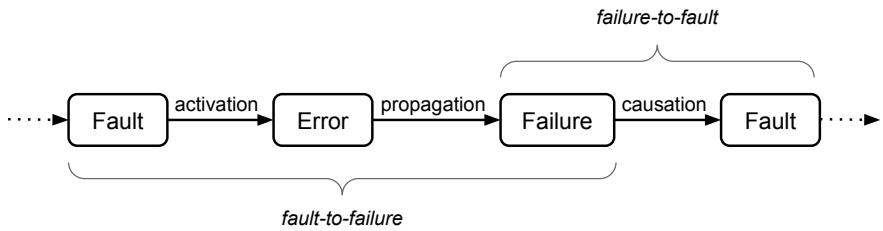

Figure 3.1: Figure enlightening the interpretation of fault-to-failure and failure-to-fault concepts within the chain of threats [7] (aka fault-error-failure chain), expressing a causality relationship between faults, errors, and failures.

In particular, analysts should account for two different failure propagation processes:

- an *intra-component* analysis to evaluate the *fault-to-failure* propagation, capturing how internal faults lead component to manifest failures;
- an *inter-component* analysis to evaluate *failure-to-fault* propagation, capturing how an output failure of a component affects other components acting as inner faults.

3.2 The MDE approach

The adoption of models and abstractions for the specification, the design, the integration, and the validation stages is a key factor for the continuous maintenance and operation of SoS. In so doing, Model-Driven Engineering (MDE) may support designers, technicians, and non-technical actors in the completion of these tasks, aiming also at reducing the negative impact of failures within business-critical contexts.

Specifically, this dissertation presents a MDE approach for SoS, addressing *direct* and *indirect couplings* in failure propagation mechanisms among components/subsystems, which reduces the gap between design artefacts (i.e., structural models and dependability models) and analysis stages where executable software models are adopted for enabling quantitative reliability evaluations.

The approach leverages on an executable *meta-model* representation, described in Chapter 4, capturing hierarchical composition of a SoS and related failure propagation mechanisms, considering salient characteristics of stochastically-timed failure propagation modes across *direct* and *indirect couplings*.

The approach demands for a fully automated configuration process exploiting two artefacts: *i) SysML Block Definition Diagrams* (BDDs) for capturing the components hierarchy of a system with a proper detail level describing direct communication interfaces as well as redundancy levels; and *ii) Fault Trees* (FTs) for capturing error modes, to be intended as internal processes of components (or subsystems) leading a set of faults in manifesting a specific failure; as well as, failure propagations among distinct components (or subsystems) transforming a failure manifested as an output of a source

component in an input fault of a destination component (in a direct or indirect relationship). Specifically, FT artefacts act as *stochastic Fault Trees* with additional capabilities of modelling repeated events (both on basic and intermediate events) leading to the generation of a Directed Acyclic Graph thus producing dependencies among events (i.e., a same event can be set in input to more than one gate), as well as introducing delays characterised by probability density functions and routing probabilities over fault propagations.

The MDE approach also realises a *round-trip engineering* mechanism, providing, on the one hand, a *forward engineering* process, which enables automation of meta-model instances initialisation by exploiting design and reliability artefacts, and, on the other hand, a *reverse engineering* process which activates a co-evolution mechanism between the meta-model instances and input artefacts guaranteeing that modifications applied over meta-model instances automatically update design and reliability artefacts. At the same time, the meta-model enables the automated generation of analysable quantitative models, in the shape of Stochastic Time Petri Nets (STPNs).

Expected advantages of the meta-model adoption, within the proposed MDE approach, are listed below:

- *correct-by-construction* configuration of structural information about system compositions, through BDDs artefacts as initialisation input of the proposed meta-model instances;
- the meta-model enables runtime verification of configuration conformance/compliance with respect to designed artefacts, especially for the CPS class exploiting self-representation mechanisms (e.g., to achieve self-adaptation capabilities);
- configuration of system failure logic, mapping reliability artefacts, produced by reliability engineers, so as to reflect the underlying *fault-to-failure* and *failure-to-fault* propagations within the meta-model;
- the meta-model is prepared for a fully automated generation of analysable quantitative models in the shape of STPN;
- in turn, analysable models enable runtime and design-time quantitative evaluations, also exploiting existent reliability tools (e.g., *Möbius*, *Oris*

Tool) as well as simulations of failure logic;

- the meta-model, enriched with a set of transformation rules, supports the co-evolution of its concrete configurations with respect to generated output structural models (i.e., BDD), failure logic models (i.e., FT), and analysable quantitative models (i.e., STPN).

Finally, a brief discussion about chosen artefacts for leading the MDE approach is here addressed.

On the one hand, structural information contained within BDDs can be considered sufficient to handle high-level features about involved components (or subsystems) and communication interfaces interconnecting them, while neglecting functional and failure logic, which in the approach are delegated to dedicated reliability artefacts (i.e., stochastic FTs). For these reasons, any other richer model subsuming structural information contained within BDDs, as intended in this dissertation, may be easily integrated in their substitution. Some alternative abstractions may be: SysML Internal Block Diagrams (IBDs), Architecture Analysis & Design Language (AADL), and Reliability Block Diagrams (RBDs).

On the other hand, stochastic FTs have been chosen for their purely functional perspective, fully decoupled from structural aspects, and their extensibility which enables decorations with stochastic features (i.e., to model failure propagation timings). Thus, RBDs and "pure" Component Fault Trees have been discarded inasmuch because of the strict pairing between structural information and behavioural or failure logic information, limiting expressiveness for high-level propagation couplings among not directly interconnected components (i.e., representability for *indirect couplings* is not provided).

The MDE approach has been included within a tool-chain perspective, through a newborn and prototype tool, presented in Sect. 6, which implements the proposed meta-model, offering a Java Application Programming Interface or a as-a-service mode for the specification of the system hierarchy and its failure logic, as well as for the export of derived STPN models.

Chapter 4

Meta-model specification

In this Chapter, a complete description and technical characterisation of the proposed meta-model, introduced only as a concept in Sect. 3.2, is reported.

In particular, in Sect. 4.1, the meta-model fragment providing a system structural specification, capable of abstracting System of Systems architectures is proposed. Sect. 4.2 focuses on: ***i)*** *the adopted reference taxonomy about failure logic;* ***ii)*** *the meta-model fragment describing fault-to-failure processes, while denominating failures, faults, and errors;* ***iii)*** *the meta-model fragment describing failure-to-fault processes, modelling direct and indirect couplings among components;* ***iv)*** *an overview of the whole meta-model. Sect. 4.3 proposes a Model-Driven Engineering approach for initialising a whole meta-model instance from Block Definition Diagrams, for providing an architectural description of system composition, and stochastic Fault Trees artefacts, for characterising the failure logic, with a practical example over a basic scenario including a system hierarchy of three components. In Sect. 4.4 a meta-model refinement addressing an adaptation towards Product Line Engineering is illustrated, also demonstrating advantages in terms of co-evolution and configuration reuse, with reference to the previous basic scenario.*

4.1 System structural specification

In several contexts where reliability and safety represent core requirements, a well-structured software representation of the under-monitored system is essential for supporting maintenance tasks and fault-tolerance policies by providing software digital replicas with runtime/offline diagnostic and predictive capabilities, thus enabling conformance analysis methods, particularly relevant in (self-)adaptive and re-configurable systems.

In these contexts, digital twins [113] offer a strong mechanism for providing a software abstraction on hardware counterparts, capturing operational behaviours of physical assets while enabling agile control as well as facilitating diagnoses, process plannings, process optimisations, virtual prototyping, and simulations.

In Sect. 3.2, under a Model-Driven Engineering perspective, the concept of reference meta-model for representing structural and behavioural aspects of a system has been introduced.

In Fig. 4.1, an UML Class Diagram fragment of the meta-model is reported. It enables self-representation mechanisms for Cyber-Physical Systems (CPS), acting as a mean for designing software digital twins, aware of structure and capabilities of the modelled system.

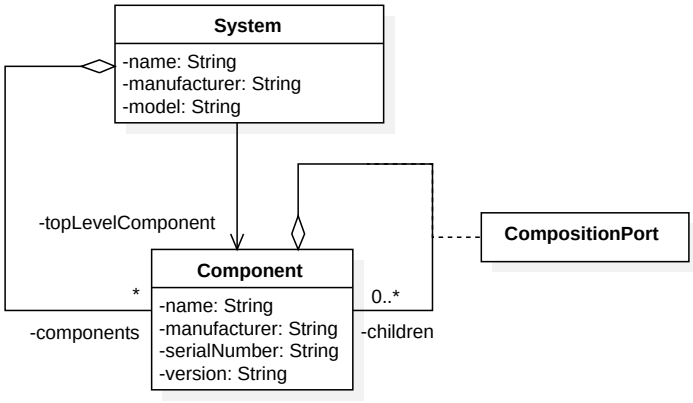

Figure 4.1: UML Class Diagram about a fragment of the meta-model of a component-based system, depicting a system abstract specification in terms of components hierarchies and communication interfaces.

In particular, the meta-model depicts a System of Systems (i.e., the class named *System*, for conciseness) as a hierarchical aggregation of distinct hardware/software components (i.e., the *Component* class), interconnected through communication interfaces (i.e., the *CompositionPort* class).

The proposed meta-model is agnostic on the typology of the modelled system, fitting scenarios characterised by isolated systems as well as complex SoS. An instance of *System* represents the monitored operative context as convenient for reliability experts, thus it can be "a part of" or a whole system, enabling fine-grained as well as coarse-grained digital configurations. Under this scheme, an instance of *Component* does not own information about the nature of its physical and concrete counterpart, thus it is not relevant if a child element is an atomic component or a system itself (i.e. a subsystem). The main *Component* which defines the *higher-level* system specification is represented by the *topLevelComponent* attribute, referenced by the *System* instance.

The software representation of the system comprises an abstract specification, which can be applied over many concrete installations of the same physical configuration. This reflects the dualism between the concept of an ideal product, identified by a model number, and the concrete product itself, identified by its serial number.

4.2 System failure logic specification

The meta-model, introduced in Sect. 4.1, comprises the ground for the specification of *faults*, *errors*, and *failures*; in so doing, the abstract specification of a system is able to capture both the system compositional structure and its failure logic.

In order to clarify concepts surrounding the failure logic design, a brief introduction about the reference taxonomy [7] for this meta-model is reported. Relevant terms are here reported:

- an **error** is a deviation from the expected system state;
- a **failure** is a manifestation of an error, deviating at least one of the system external states, thus producing tangible effects;
- a **fault** is one of the causes of an error and can be internal (i.e., a fault activated by an *endogenous* process, internally originated by the

component/subsystem itself, usually due to aging or manufacturing defects) or external (i.e., a fault activated by an *exogenous* process, originated from outside the affected component/subsystem).

Failure propagation insights can be captured within the meta-model, enhancing the abstract specification with *error*, *failure*, and *fault* concepts; as represented in the fragment of Fig. 4.2, each *Component* instance maintains a reference to its manifestable output failures (i.e., the *FailureMode* class) which are strictly related to errors and their causes (i.e., the *FaultMode* class), distinguishing also between internal and external faults (i.e., respectively the *EndogenousFaultMode* and the *ExogenousFaultMode* classes, as specialisations of the *FaultMode* super-class). The *fault-to-failure* propaga-

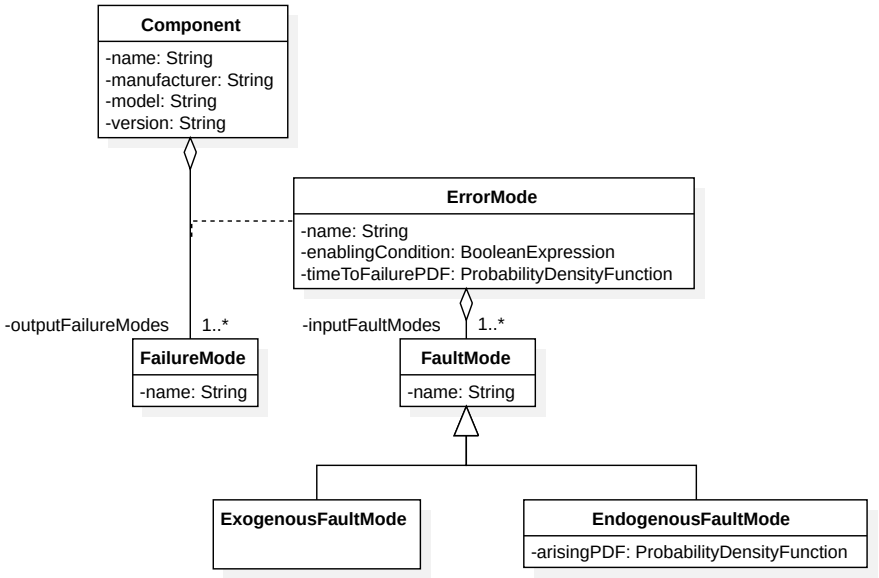

Figure 4.2: UML Class Diagram about a fragment of the meta-model, depicting error modes characterisation for each system component.

tion process, describing the internal failure logic of a component is modelled as an association class (i.e., the *ErrorMode* class) between the component itself and the manifested failure. An error mode is here designed so as to represent the Boolean logic expression (which can be easily mapped-in or mapped-from a Fault Tree) of faults leading to the manifested failure.

Note that classes mapping these dependability concepts are decorated by the *Mode* suffix specification, which enlightens the role of abstract and reusable specification of the meta-model, which can be applied over many concrete installations of the same specification (i.e., the same meta-model *System* instance). In so doing, the meta-model represents these concepts in a general perspective of knowledge, while real-time occurrences of these entities (e.g., during simulations) must be intended as their concretisations (i.e., failures, faults, and errors).

ErrorMode and *EndogenousFaultMode* classes have been stochastically characterised introducing probability density functions which respectively model the time to failure (i.e., *timeToFailurePDF*) and the fault arising (i.e., *arisingPDF*) probability distributions so as to support quantitative analysis over the meta-model. Exogenous faults, modelling external causes of errors, are not stochastically characterised and should be derived at analysis time extracting them from *direct* and *indirect couplings* information.

The *failure-to-fault* propagation processes can be captured within the meta-model only considering existent couplings which describe propagation flows of manifested failures towards external (i.e., exogenous) faults. This kind of faults strictly depends on SoS nature: indeed, CPS, operating in contexts such as Smart Cities, Industrial Internet of Things and Industry 4.0 [93, 100,117], promotes the integration of high-level communication mechanisms, adopting software middlewares for brokering messages among subsystems and high scale enterprise software agents.

In so doing, a CPS is designed so as to produce an *emergent behaviour* [54, 84], a collection of actions and patterns that cannot be predicted from isolated behaviours of constituent systems but resulting from local cooperation among subsystems and their environments. Modern systems intrinsically own *emergence* [64], implementing in many cases, big data ingestion processes, horizontal and vertical integration stages, and microservices orchestration patterns; thus enabling *self-organising* logic and emphasising the need for studies about *indirect* failure propagation couplings, in addition to *direct* ones.

In this architecture, role and responsibilities of the middleware becomes crucial: in a classical IoT architecture [63], a message broker is usually adopted for ingesting wide *in-motion* data streams, generated by a plethora of perception devices. Therefore, the broker is responsible for routing and

forwarding raw data to collector components, according to some policies. Manifested failures within the perception layer, "alterating" messages contents (e.g., reporting a wrong temperature), are inevitably propagated by the broker. Altered messages may produce inner faults inside final recipients. Clearly, in this scenario, the broker can't be considered as a failed or faulty component, despite having contributed to the generation of *indirect* couplings.

This kind of failure propagations cannot be effectively represented in the previous fragments of the meta-model (i.e., Fig. 4.1 and Fig. 4.2), which only express *direct couplings* occurring along the compositional architecture defined by *intra-subsystem* and *inter-subsystems* interfaces established by the system specification (i.e., the *CompositionPort* class).

The meta-model has been thus enriched as shown in Fig. 4.3, so as to permit specification of failure propagations occurring across *indirect couplings* determined by use case scenarios and middlewares business logic, involving components not directly interconnected through interfaces. In so doing, the

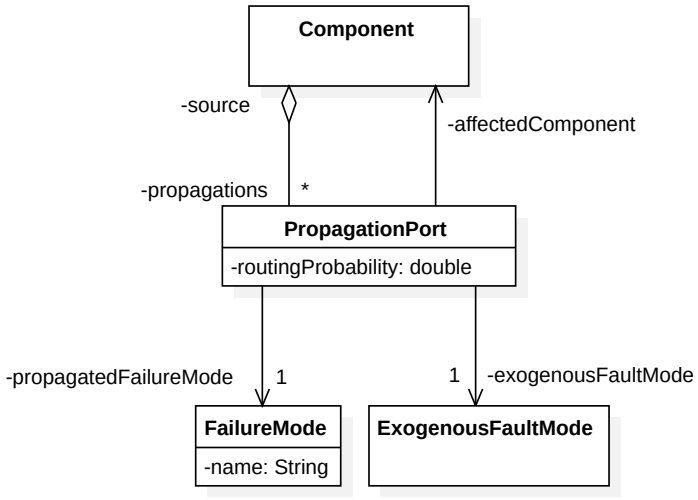

Figure 4.3: UML Class Diagram about a fragment of the meta-model, depicting failure propagation logics of each system component.

direct and *indirect couplings* are now modelled through the introduction of the *PropagationPort* class, in addition to the *CompositionPort* class pre-

sented in Fig. 4.1, separating the compositional hierarchy from the failures propagation logic. Each *PropagationPort* instance specifies how a *FailureMode* manifested by a source *Component* affects another *Component* (i.e., *affectedComponent*), raising an exogenous fault (i.e., *exogenousFaultMode*). *Failure-to-fault* processes may be decorated with a *routingProbability* determining the probability under which the failure mode instance of the source component becomes an exogenous fault instance for the affected one.

The whole meta-model is reported in Fig. 4.4, where previous fragments are combined all together. So, the meta-model constitutes the basis for an executable software representation, opening the way to perform both *fault-to-failure* and *failure-to-fault* propagation processes, deriving information from useful artefacts (as described in Sect. 4.3).

In summary, the meta-model defines three primary entities: *i)* the *CompositionPort* class represents the system hierarchy specification in terms of hardware/software communications among components, *ii)* the *ErrorMode* class stochastically characterises the *fault-to-failure* mechanism, and *iii)* the *PropagationPort* class describes the *failure-to-fault* propagation mechanism. As a positive consequence, the entire software meta-model reflects the abstract system specification, thus constituting a framework for designing and testing *product lines* by enabling "offline" analyses. At the same time, these abstract configurations can support "runtime" reliability analyses over many concrete installations of the same system. Moreover, runtime event observations and anomalies detection can lead the refinement of stochastic characterisation of endogenous faults.

4.3 A Round-Trip Engineering process

The meta-model enables a *round-trip engineering* process [73], leading, on the one hand, the instantiation of the meta-model itself and, on the other hand, the co-evolution of meta-model instances with adopted design and reliability artefacts.

Bidirectional transformations rules are given so as to generate meta-model instances, retrieving information from input artefacts (i.e., SysML BDDs and stochastic FTs), as well as to obtain synchronised versions of the structural and failure logic artefacts, starting from modifications on the executable meta-model instances (as summarised in Fig. 4.5).

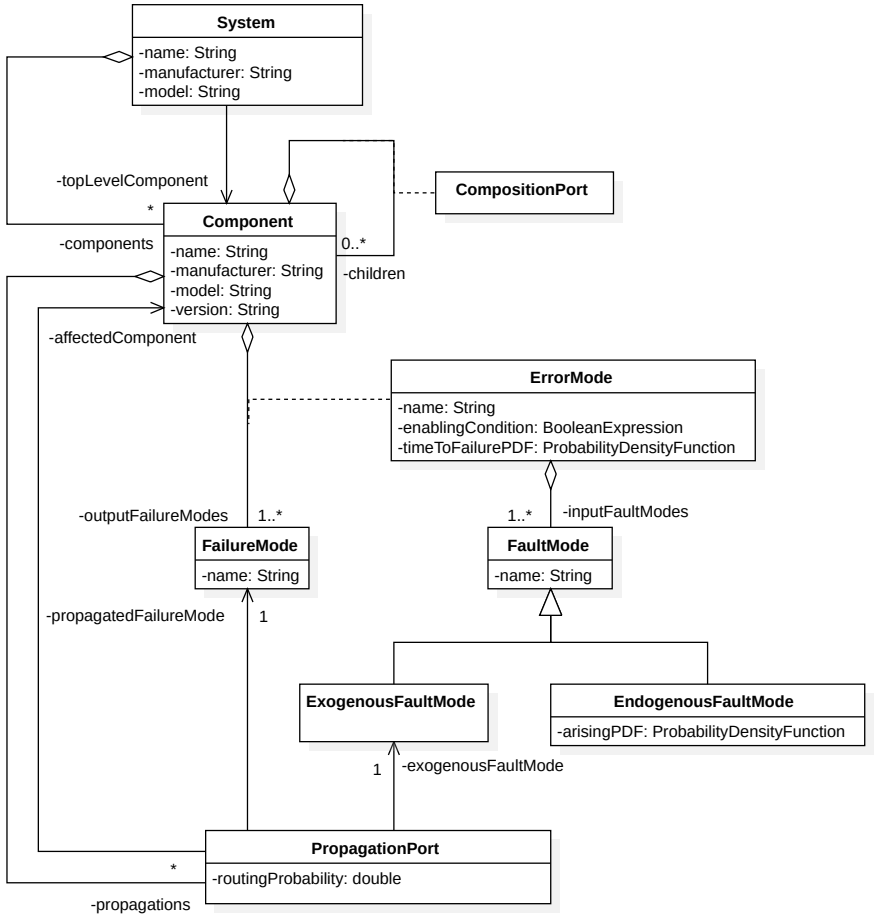

Figure 4.4: UML Class Diagram of the meta-model of a component-based system, refined with failure propagation mechanisms.

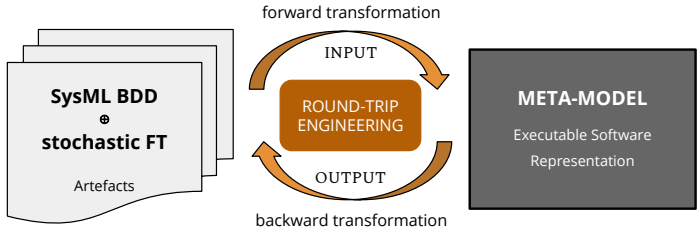

Figure 4.5: Round-trip engineering process.

The instantiation of the meta-model requires an initialisation effort by the domain experts; the complexity of managed information is based on a structural perspective, referred to system composition and system architecture, but also on a dependability perspective, referred to reliability design choices about system failure propagation processes and their stochastic characterisation.

The overall effort is mitigated by the adoption of MDE practices considering that the initialisation of executable object instances of the meta-model may be fully automated.

In particular, a concretisation of the presented meta-model fragment of Fig. 4.3 may be automatically derived from SysML BDD diagrams[1] by considering each block as a *Component* instance and each relationship between blocks as a *CompositionPort* instance, connecting two distinct components (see Alg. 1 in Sect. A.1 for details). A unique system is instantiated by referring to the top level block of the BDD.

The reverse engineering process, extracting BDD artefacts from the executable representation of the meta-model can be defined as a process visiting the hierarchical configuration made by drawing a basic block for each *Component* and interconnecting it with UML compositional relationship for each child identified through its *CompositionPort* instances (see Alg. 3 in Sect. A.2 for details).

Practically, within a basic exemplary scenario of a system built over three components as depicted in Fig. 4.6, the concrete domain of the meta-model

[1]Note that, conceptually, the same information may be similarly derived from more detailed abstractions such as SysML IBDs, AADL artefacts.

related to the system compositional hierarchy would be instantiated as in the UML Object Diagram of Fig. 4.7.

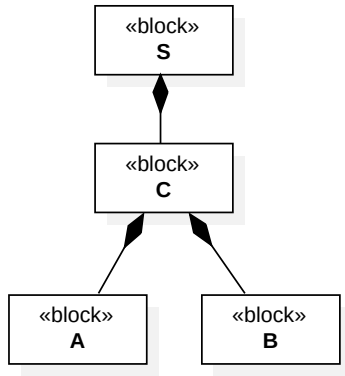

Figure 4.6: SysML Block Definition Diagram of a system S, composed by a top-level component, named C, with two children components, named A and B.

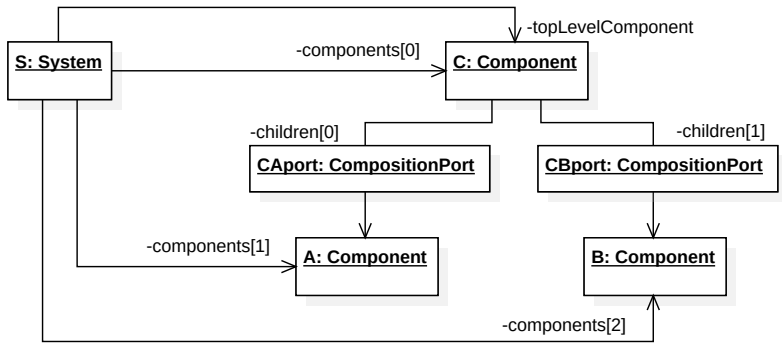

Figure 4.7: UML Object Diagram built on the meta-model, reading input data provided by BDD of Fig. 4.6.

The failure logic described within the fragment of Fig. 4.4 may be automatically derived from FTs artefacts, adopting a correct denomination of faults and failures of each modelled component within the system hierarchy.

In order to properly configure the propagation mechanisms within the

executable representation, extracting useful information from stochastic FTs, it is relevant to remember the fault classification, distinguishing between *internal* and *external* faults:

- internal faults, named *endogenous* faults, arise autonomously within components and are mainly caused by factory defects, natural obsolescence, or degradation processes;
- external faults, named *exogenous* faults, arise as input faults propagated from external components, depending not only on *direct* couplings, derived from the compositional hierarchy, but also on *indirect* couplings due to data-level dependencies, use case scenarios and business logic controlled by the cyber-side.

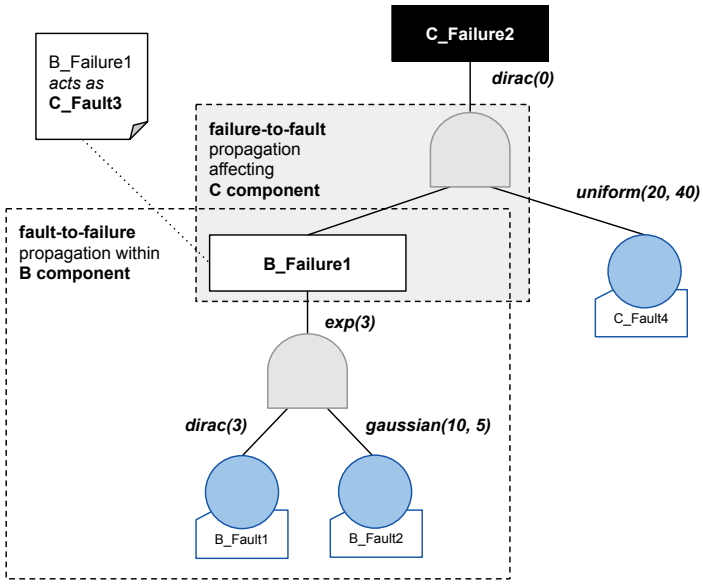

Figure 4.8: Stochastic FT artefact of failure logics for the system in the running example.

With reference to the previous exemplary scenario, the stochastic FT[2] de-

[2]As stated in Sect. 3.2, the adopted FT artefacts can be considered as stochastic FTs able to model repeated events (both basic and intermediate events) as well as to provide a stochastic characterisation of propagation delays through probability density functions, decorating edges with distribution labels, and routing probabilities, reported within notes.

picted in Fig. 4.8 can be automatically translated into the UML Object Diagrams of Figs. 4.9, 4.10, and 4.11, built over the meta-model, respecting the following transformation rules.

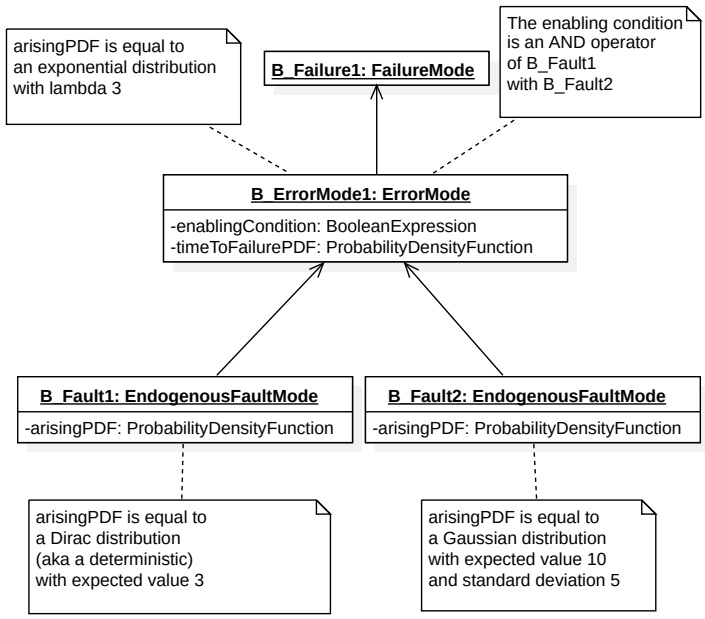

Figure 4.9: UML Object Diagram built on the meta-model, reading input data provided by FT of Fig. 4.8.

On the one hand, the *fault-to-failure* propagation has been interpreted through an *intra-component* mode, describing how faults may lead a component to manifest failures over time. The executable software representations depicted in Figs. 4.9 and 4.10 embed *fault-to-failure* propagation mechanisms within the *ErrorMode* class, which characterises stochastically the internal behaviour of B and C components, exploiting their denominated fault modes and failure modes. In particular, in the case of B component (see Fig. 4.9), the output failure, related to a specific error mode (i.e., *B_ErrorMode1*), is modelled through Boolean logic expressions of occurred internal faults (i.e., *B_Fault1*, *B_Fault2*), enriched with an exponential probability density function with lambda equal to 3 (i.e., *timeToFailurePDF*). At the same time, in Fig. 4.10, the error mode (i.e., *C_ErrorMode2*) describing the *fault-to-failure*

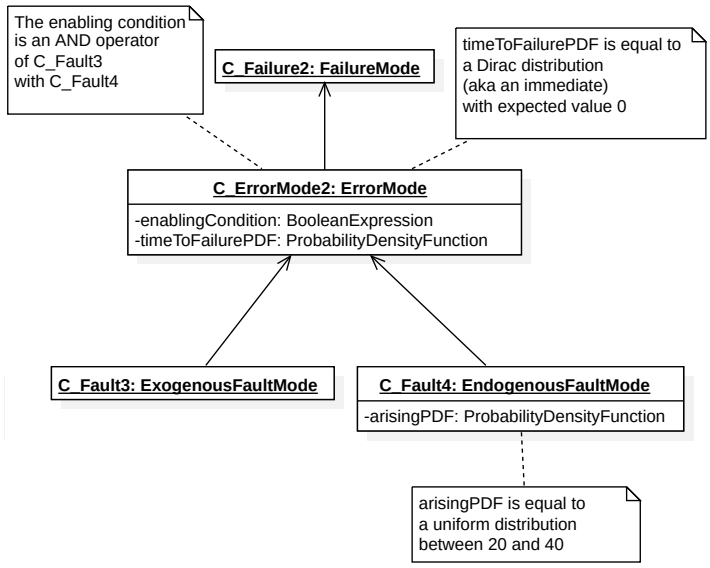

Figure 4.10: UML Object Diagram built on the meta-model, reading input data provided by FT of Fig. 4.8.

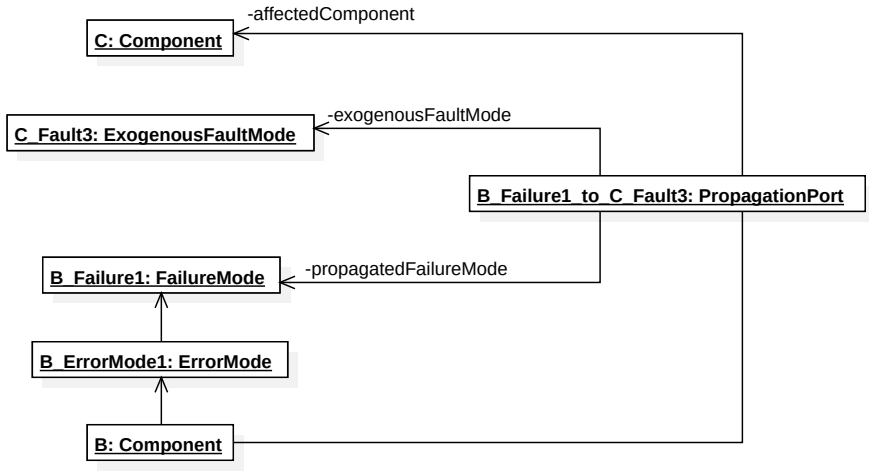

Figure 4.11: UML Object Diagram built on the meta-model, reading input data provided by FT of Fig. 4.8.

of C component leading to *C_Failure2* shows the case of an exogenous fault mode (i.e., *C_Fault3*), where the *B_Failure1* failure mode of an external component (i.e., B) acts as an input fault for C (i.e., subtending a *failure-to-fault* propagation).

On the other hand, the *failure-to-fault* propagation has been interpreted through an *inter-components* mode over the system hierarchy, describing how an output failure of a lower level component may affect other higher level components. The executable software representation depicted in Fig. 4.11 embeds *failure-to-fault* propagation mechanisms within the *PropagationPort* class, whose instances relate output failures of a component acting as inner faults for other distinct components. In particular, the failure mode *B_Failure1* (i.e., the *propagatedFailureMode*), acting as the exogenous fault mode *C_Fault3* (i.e., instantiated as an *ExogenousFaultMode* object) of C component (i.e., the *affectedComponent*), is mapped in a *PropagationPort* instance with B component as its source.

A formalised procedure for automatically mapping system FTs (decorated with temporal probability distributions over basic events or gate nodes) in the meta-model failure logic (see Alg. 5 in Sect. A.3 for details) is here summarised through a textual procedure composed of 3 phases:

1. for each error mode, the first phase aims at identifying confined sub-trees, confining internal behaviours (i.e., error modes) of each component[3].
   (a) failure modes of the FT act as boundaries separating the upper sub-tree from the lower sub-tree;
   (b) in so doing, each sub-tree owns a root node as top-event failure mode, and a set of leaves which represents internal faults as basic-events or external faults as propagated failure modes;
   (c) and a set of intermediate events, defined as a structure of logical gates combining basic events and lower level failure modes.
2. for each identified sub-tree, the second phase aims at instantiating an *ErrorMode*, reflecting *fault-to-failure* propagation mechanisms of each component.

[3]In general, the top-event of a sub-tree must be a failure mode related to a single component, while the leaves of a sub-tree are lower-level failure modes of some components or basic events (acting as fault mode).

   (a) the top-event of the sub-tree is mapped in an *outputFailureMode* instance of the *FailureMode* class;

   (b) each basic event of the sub-tree is mapped in an *inputFaultMode* instance of the *EndogenousFaultMode* class;

   (c) each propagated failure mode leaf of the sub-tree is mapped in a *inputFaultMode* instance of the *ExogenousFaultMode* class.

3. the third phase aims at identifying the *failure-to-fault* propagations among components.

   (a) each boundary failure mode, acting as a leaf in the upper sub-tree and as a top-event in the lower one, is mapped in a *propagation* instance of the *PropagationPort* class, interconnecting the *propagatedFailureMode* from the source component with the *exogenousFaultMode* instance of *ExogenousFaultMode* of the affected component.

The reverse engineering process (see Alg. 9 in Sect. A.4 for details) leading to the generation of FT artefacts, starting from the meta-model instances, may be defined as a Top-Down visit of each highest-level failure mode (acting as the top-event of a FT) of the failure logic specification, by applying recursively these transformations rules: each *ErrorMode* instance is associated to a sub-tree for which the *enablingCondition* defines logical gates combination of sub-tree events. Each sub-tree is delimited by a failure mode of a *Component* (acting as the top-event) and *EndogenousFaultMode* instances (acting as basic events) or *ExogenousFaultMode* instances. Finally, the procedure must be applied recursively on each *FailureMode* instance related to each *ExogenousFaultMode* instance within a *PropagationPort* instance, identifying its sub-tree and connecting it to the exogenous fault event.

Note that, the MDE approach requires that domain experts provide a stochastic characterisation of the failure logic in the shape of stochastic FTs, so as to complete the configuration of the executable meta-model. This topic is not covered within the dissertation, but in principle probability density distributions may be based on information derived from classical FMEA/FMECA analyses and technical data-sheets, including Mean Time To Failure (MTTF) estimations provided by industrial manufacturers, as well as from custom conditions for *what-if* analysis processes.

Furthermore, the proposed MDE approach must not be regarded as a constrained process, as the idea is to define a universal ontology able to model relevant information for failure logic propagations and their timed analysis. In so doing, the approach leverages a core meta-model, whose concretisation is here illustrated by automated processes adopting BDDs and FTs.

In a wider perspective, the selected input abstractions can be obtained also starting from alternative *common practice* and/or *domain-specific* artefacts: indeed, BDDs can be substituted with AADL [34] documents for embedded systems scenarios, while a basic failure logic indication can be derived through mapping techniques applied over different abstractions (e.g., RBDs can be converted in FTs, as described in [116], while through FPTN modules is possible to built CFTs, which approximate FTs, as described in [47]) though requiring additional efforts in modelling stochastic characterisation, repeated events, as well as eventual *indirect* couplings occurring on the data-level.

4.4 Meta-model refinement for Product Lines

Complex systems including several hardware/software components may be subject to evolutive variations in their concrete installations, which may be led by adaptations of the offered products or services to business or customer needs as well as by tailoring stages of their variable constituent parts within domain contexts. In many other cases, continuous system maintenance processes, adding, removing, or substituting some components in operative installations, unintentionally, produce kinds of *system families*, exacerbating the resulting complexity.

These systems families, known as Product Lines, take the name of Multi Product Lines [53] for large-scale or ultra-large-scale software-intensive systems managed by different organisational units, as in the case of *Interwoven Systems* or *Federations of Systems*.

Product Line Engineering [6,82] may support complexity management within design and maintenance stages, providing useful approaches and strategies for reducing development costs and deployment times, enhancing system quality, and coping with an organised system evolution. The survey in [10] describes how the values of *variation modelling* are perceived in the industrial contexts, highlighting its importance in the management of existing

variability, in the product configuration, in requirements specification, in the derivation of products, as well as in the design and architecture planning of variability.

The modelling of *product lines* requires a deep domain expertise for providing a sufficient system representation capturing product families variations. For these reasons, the proposed meta-model should be enhanced, within its structural parts, in order to act as an ontology with the capability of representing variability. Among variability units [10] (e.g., features, configuration options, calibration parameters, decisions), the meta-model reported within the fragment of Fig. 4.12, extending the previous fragment of Fig. 4.1, leverages on the concept of variation point (i.e. the *VariationPoint* class).

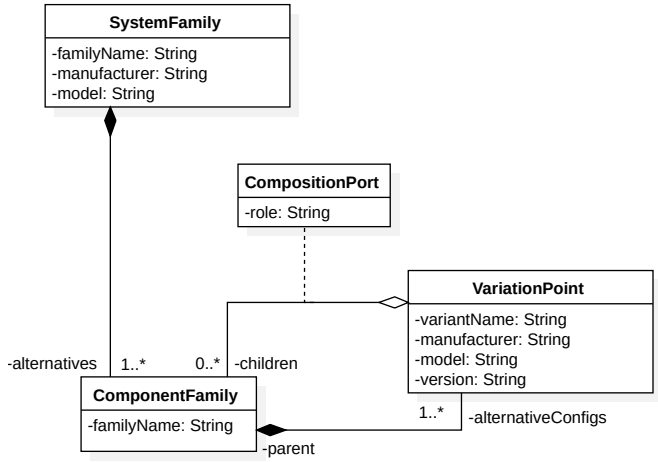

Figure 4.12: UML Class Diagram about a fragment of the meta-model, depicting in a Product Line Engineering perspective the structural, depicting a system family abstract specification in terms of components hierarchies and communication interfaces subject to variation points.

Specifically, the *System* class has been renamed in *SystemFamily* with the further addition of *alternatives* attribute, which models a collection of top-level components as available options. De facto, each component interprets the role of component family; thus the previous *Component* class has been renamed in *ComponentFamily*.

The *VariationPoint* class has been introduced so as to model variants of system component, listing and denominating alternative configurations for each *parent* component family. Note that, in terms of numerousness, a component family should have at least one alternative concrete configuration. Besides, the association class named *CompositionPort*, previously modelled on the self-relationship between parent and children instances of the *Component* class, while preserving its original meaning, has now been moved on the aggregation between *VariationPoint* and *ComponentFamily* classes.

In so doing, the concept of *product line* has been primarily declined in a structural perspective: alternative variations of a single component have to be intended as different children configurations and architectural specifications. Trivially, atomic component variations without children are recognisable by the absence of *children* instances of type *ComponentFamily*.

The refinement led the *ComponentFamily* class to act as the high-level concept of a component (i.e., the structural and functional role interpreted for the whole system), while the *VariationPoint* class effectively models concrete instances of the components, as alternative configurations.

Consequently, the failure logic has been adapted so as to be referred to concrete instances: *ErrorMode*, *FailureMode*, *FaultMode*, and *PropagationPort* classes have been detached from the *Component* class and attached to the *VariationPoint* class, maintaining their original semantics, as depicted in Fig. 4.13. Indeed, it's quite clear that the failure logic of each system component does not reside within the component family but in its effective alternative configurations, which may operate differently even providing different functionalities (e.g., in an IoT system, the simple addition of a new sensor type may affect the failure logic by producing data-level *indirect couplings* over ingested samples and active filtering policies).

The presented *round-trip engineering* process applied to the proposed metamodel remains valid also for this refined version, which maintains the capability of enabling automated initialisation as well as *co-evolution* mechanisms with respect to system architectural models (i.e., SysML BDDs) and reliability artefacts (i.e., stochastic FTs). Indeed, the *VariationPoint* of the refined model interprets the same roles and responsibilities of the previous *Component* class; furthermore, the introduction of a variation point at some level of the system hierarchy imposes only the instatiation of the component (or

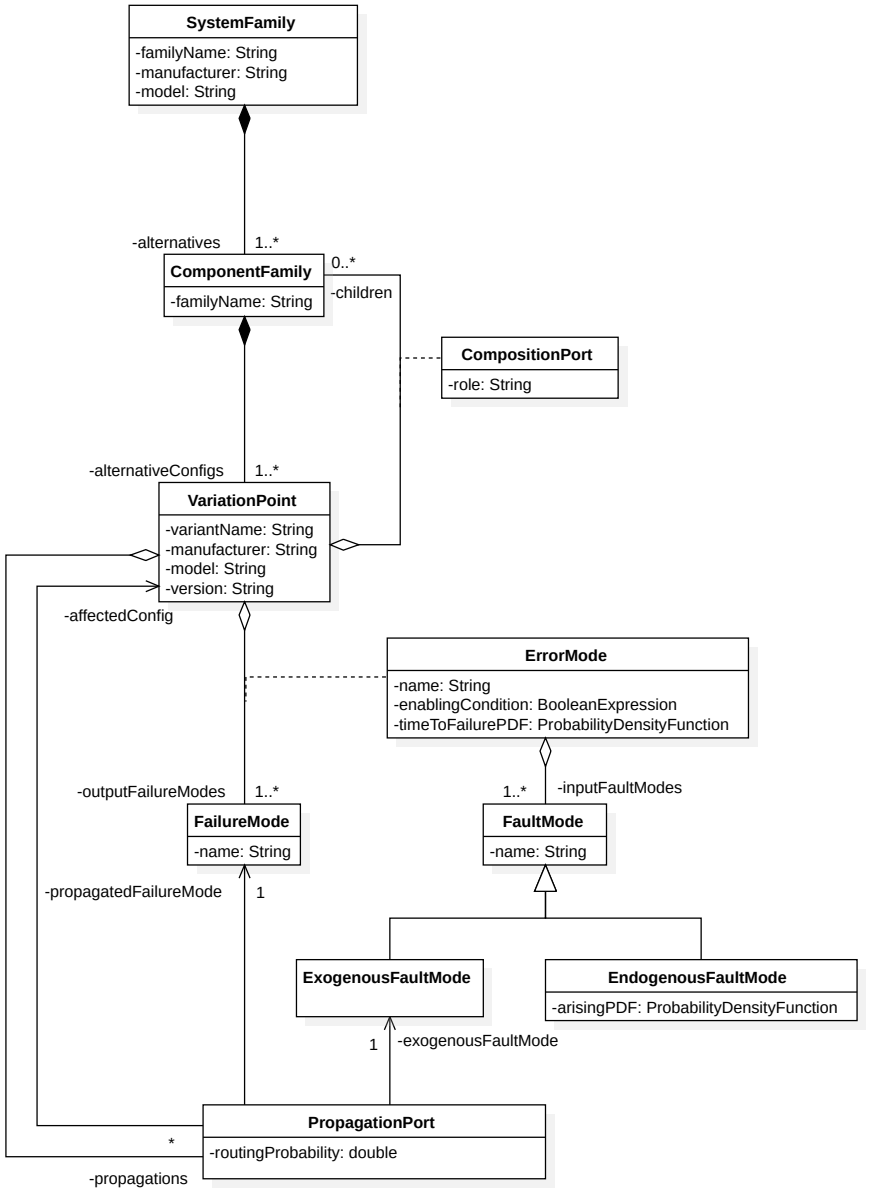

Figure 4.13: UML Class Diagram of the whole meta-model of a component-based system, refined with failure propagation mechanisms and variation points in a *product line* perspective.

subsystem) specification which may be derived from BDDs as described in Sect. 4.3: previous system configurations are preserved and the reuse of both structural and failure logic instances is guaranteed.

It is required only the update of newly introduced components in terms of error modes, failure modes and fault modes, characterising *fault-to-failure*, and induced *direct* and *indirect couplings* over ancestor components, characterising *failure-to-fault*. Within the MDE perspective, also the initialisation of the failure logic for *product lines* can be easily driven by reliability artefacts (e.g., FTs) in a modular approach combining information over changed components (or subsystems).

In practice, with reference to the exemplary basic system scenario of Fig. 4.6, a variant of C, introducing a new D component, is depicted in Fig. 4.14. The resulting executable representation of the structure of the variant is reported in the UML Object Diagram of Fig. 4.15.

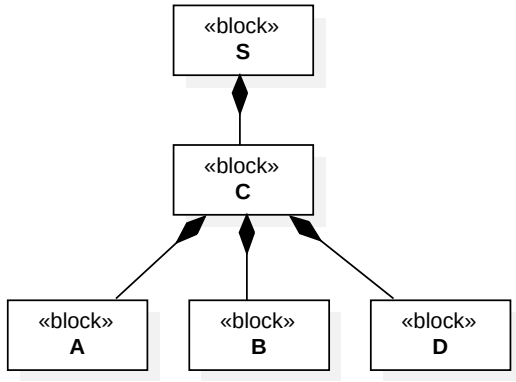

Figure 4.14: SysML Block Definition Diagram of the system S as product variant of the system in Fig. 4.6, adding a new component named D.

Also the failure logic has been updated (see Fig. 4.16) so as to represent the impact of the D component in system failure propagation mechanisms, affecting the C component with the influence of *D_Failure1* (i.e., the *failure-to-fault* coupling C and D components is mapped with *D_Failure1* acting as *C_Fault5*).

The failure logic concretisation of the variant is reported in the UML

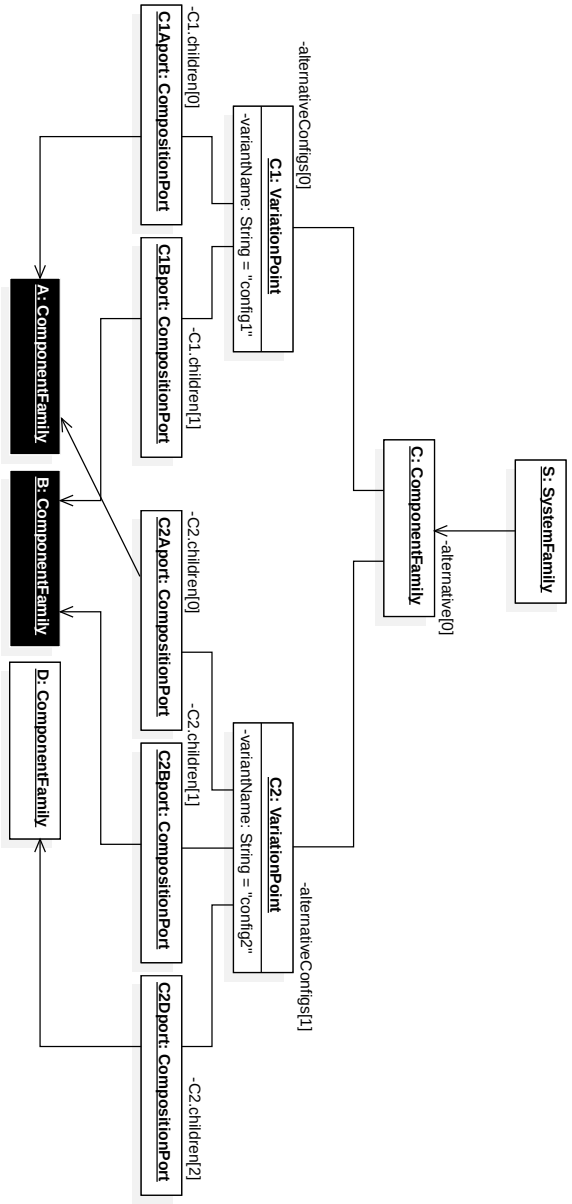

Figure 4.15: UML Object Diagram built on the meta-model, reading input data provided by BDD of Fig. 4.14.

Object Diagram of Fig. 4.17.

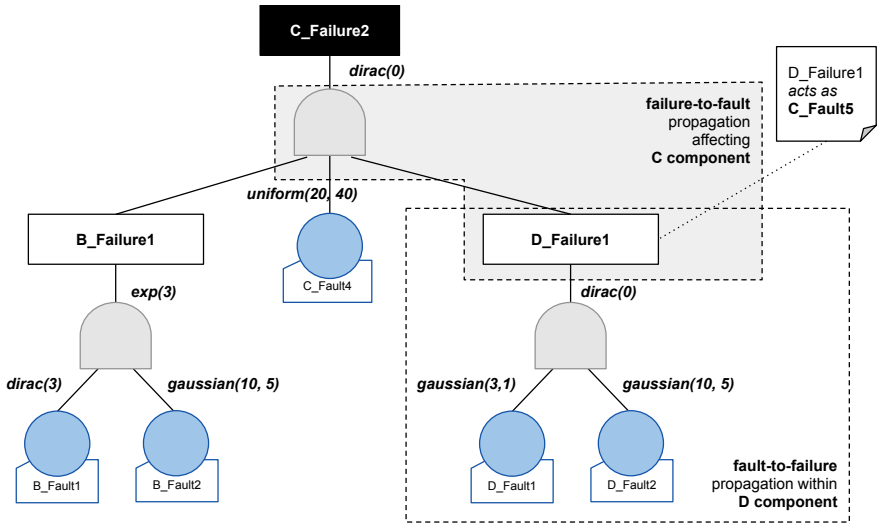

Figure 4.16: Stochastic FT artefact of failure logic for the system in the running example, where C component has a variation point.

UML Object Diagrams of Figs. 4.15 and 4.17 highlight the preservation of previous configurations, enabling the evolution over time of configurations with respect to changes on installed systems; furthermore, the instances reuse is demonstrated by the objects of black colour. Specifically, in the UML Object Diagram of Fig. 4.15, two alternative configurations of the C component family are depicted, the first (i.e., C1) refers to the system represented in the SysML BDD of Fig. 4.6 while the second (i.e., C2) the one of Fig. 4.14.

The example also enlightens how the meta-model refined for *product lines* reduces the effort to be spent by human technicians in response to configuration changes through the standardised adoption of modelling artefacts related to isolated component variations during the initialisation process.

In so doing, the modular composition of isolated sub-components information enables the extraction of the overall system failure logic in a FT artefact, summarising all behaviours, as well as the generation of executable timed analysis models for reliability purposes.

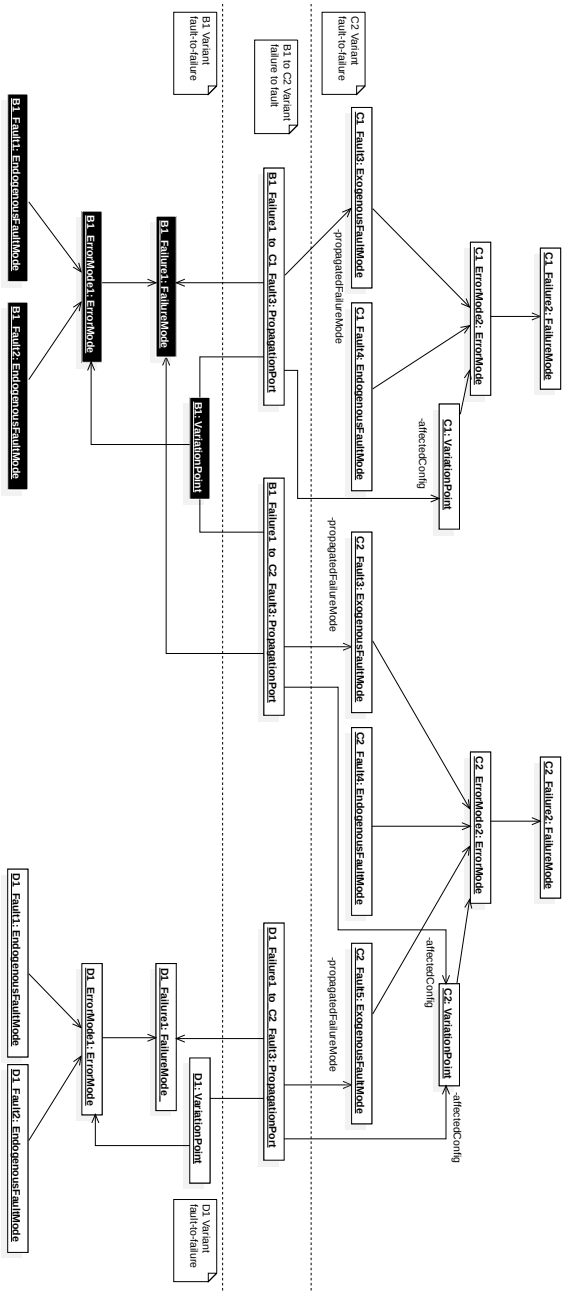

Figure 4.17: UML Object Diagram built on the meta-model, reading input data provided by FT of Fig. 4.16.

Chapter 5

Meta-model to Timed Analysis model transformation

In this Chapter a description of the transformation rules leading the generation of timed analysis models in the formalism of Stochastic Time Petri Net (STPN), starting from meta-model instances, is provided.

In particular, in Sect. 5.1 syntax and semantics of the adopted STPN model are described; in Sect. 5.2 formal transformation rules from a meta-model instance to an STPN are provided; while, in Sect. 5.3 an overview of main quantitative analysis techniques is reported within a tool-chain perspective.

5.1 Stochastic Time Petri Nets as Timed Analysis models

The meta-model, described in Chapter 4, has been integrated within a MDE approach with the aim of facilitating the initialisation of executable software representations of system hierarchies and their failure logic. The approach also enables a further generation of general purpose timed models, able of characterising temporal behaviours of systems under modelling, so as to apply quantitative analysis techniques (e.g., numerical or simulated).

The mathematical formalism selected at these purposes is a stochastic model, specified as a Stochastic Time Petri Net (STPN) [109]. The approach is accompanied by a procedure for the automated generation of STPN models from the meta-model, as described in Sect. 5.2.

STPNs are a class of Stochastic Petri Nets designed to specify concurrent timed systems where *transitions* (depicted as vertical bars) represent activities, *places* (depicted as circles) represent discrete components of the logical state with values encoded by a number of *tokens* (depicted as dots), and *directed arcs* from input places to transitions and from transitions to output places represent token moves occurring at the execution of activities.

A transition is enabled when all its input places contain at least one token, and its firing will remove a token from each input place and add one to each output place. The time elapsing from the enabling to the firing of a transition is a random variable (possibly imposing minimum and maximum duration). Besides, the choice between transitions with equal time to fire is solved by a random switch determined by probabilistic weights.

Syntax An STPN is a tuple $\langle P, T, A^-, A^+, B, U, R,\ EFT, LFT, \mathcal{F}, \mathcal{W}, Z\rangle$ where: P is the set of places; T is the set of transitions; $A^- \subseteq P \times T$ and $A^+ \subseteq T \times P$ are the sets of precondition and postcondition, respectively; B, U, and R associate each transition $t \in T$ with an enabling function $B(t) : \mathcal{M} \to \{\text{TRUE,FALSE}\}$ to restrict the enabling of a transition with general constraints on token counts, an update function $U(t) : \mathcal{M} \to \mathcal{M}$ to specify additional updates of token counts after the firing of a transition, and a reset set $R(t) \subseteq T$ to force the restart of selected transitions, respectively, where M is the set of reachable markings $m : P \to \mathbb{N}$; $EFT : T \to \mathbb{Q}_0^+$ and $LFT : T \to \mathbb{Q}_0^+ \cup \{\infty\}$ associate each transition with an *earliest* and a *latest*

firing time, respectively, such that $EFT(t) \leq LFT(t)\ \forall\ t \in T$; $\mathcal{F} : T \to F_t^s$ associates each transition with a Cumulative Distribution Function (CDF) with support $[EFT(t), LFT(t)]$; $\mathcal{W} : T \to \mathbb{R}^+$ associates each transition with a weight; $\mathcal{Z} : T \to \mathbb{N}$ associates each transition with a priority. A place p is termed an *input* or an *output* place for a transition t if $\langle p, t\rangle \in A^-$ or $\langle t, p\rangle \in A^+$, respectively. A transition t is called *immediate* (IMM) if $[EFT(t), LFT(t)] = [0, 0]$ and *timed* otherwise; a timed transition t is termed *exponential* (EXP) if $F_t(x) = 1 - e^{-\lambda x}$ over $[0, \infty]$ for some rate $\lambda \in \mathbb{R}_0^+$ and *general* (GEN) otherwise; a GEN transition t is called *deterministic* (DET) if $EFT(t) = LFT(t)$ and *distributed* otherwise. For each distributed transition t, it is assumed that F_t is absolutely continuous over its support and thus that there exists a Probability Density Function (PDF) f_t such that $F_t(x) = \int_0^x f_t(y)dy$.

Semantics The *state* of an STPN is a pair $\langle m, \tau\rangle$, where $m \in \mathcal{M}$ is a marking and $\tau : T \to \mathbb{R}_0^+$ associates each transition with a time-to-fire. A transition t is *enabled* by m if m assigns at least one token to each of its input places and the enabling function $B(t)(m)$ evaluates to TRUE; an enabled transition is *firable* if its time-to-fire is not higher than that of any other enabled transition. When multiple transitions are firable, one of them is selected to fire with probability $Prob\{t \text{ is selected}\} = \mathcal{W}(t)/\sum_{t_i \in T^f(s)} \mathcal{W}(t_i)$, where $T^f(s)$ is the set of firable transitions in s. When t fires, $s = \langle m, \tau\rangle$ is replaced by $s' = \langle m', \tau'\rangle$, where m' is derived from m by: *i)* removing a token from each input place of t and assigning zero tokens to the places in $L(t) \subseteq P$, which yields an intermediate marking m_{tmp}, *ii)* adding a token to each output place of t, and *iii)* and applying the update function $U(t)$ to the resulting marking. Transitions enabled both by m_{tmp} and by m' are said *persistent*, while those enabled by m' but not by m_{tmp} or m are said *newly-enabled*; if t is still enabled after its own firing, it is regarded as newly enabled [12]. The time-to-fire of persistent transitions is reduced by the time elapsed in s, while the time-to-fire of newly-enabled transitions takes a random value sampled according to their CDF.

5.2 Deriving an STPN model of failure logic

The specification of the failure logic embedded within the meta-model of Figs. 4.4, 4.13 can be translated into a corresponding STPN model (see

Alg. 12 in Sect. A.5 for details), thus enabling co-evolution mechanisms between meta-model configurations and quantitative models for analysis and simulation of timed failure logic.

The transformation may be realised applying the following derivation rules based on three distinct sub-procedures for endogenous faults occurrences, *fault-to-failure* propagations, and *failure-to-fault* propagations (i.e., exogenous faults occurrences), enlightening and quantifying components reliability measures.

First, endogenous faults occurrences are mapped into an STPN submodel defined by two places and one timed transition, representing the activation, the occurrence, and the time at which the fault will occur, respectively. In so doing, see Fig. 5.1, for each *EndogenousFaultMode* of the meta-model three elements are mapped in the STPN:

- the place named *Endogenous Fault Process* represents the endogenous process leading the involved system component to a deviation from its expected status. This place is marked with one token, to enable the transition;
- the place named *Endogenous Fault Occurrence* represents a raised internal fault;
- a single general transition stochastically characterises the process leading to the raise of the internal fault. The transition models the probability density function, defined by the *arisingPDF* attribute within the *EndogenousFaultMode* instance.

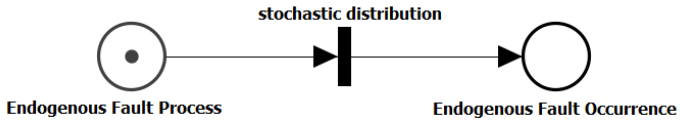

Figure 5.1: The STPN submodel of an endogenous fault process.

Secondly, each *fault-to-failure* propagation is mapped into a dedicated submodel composed by an input place containing one token, a timed transition decorated with an enabling condition (corresponding to the Boolean expression defined inside one of the error modes of the component), and an output

place where the token will be moved to, when the failure occurs. In so doing, see Fig. 5.2, for each *ErrorMode* of the meta-model three elements are mapped in the STPN:

- the place named *Fault-to-Failure Process* represents the active *fault-to-failure* process leading the involved system component to manifest a failure. This place is marked with one token, to enable the transition;
- the place named *Failure Occurrence* represents a manifested failure;
- a single general transition stochastically characterises the process leading to failure manifestation. The transition models the probability density function, defined by the *timeToFailurePDF* attribute within the *ErrorMode* instance. It is also decorated with an enabling condition defined in the *enablingCondition* attribute (within the same *ErrorMode*) of type *BooleanExpression*, referencing *Endogenous Fault Occurrence* and *Exogenous Fault Occurrence* places.

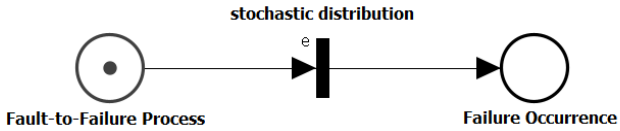

Figure 5.2: The STPN submodel of *fault-to-failure* mechanism.

Finally, each output place of a *fault-to-failure* submodel may act as an input place for a *failure-to-fault* submodel characterised by an immediate transition that moves the token to an output place representing the occurrence of an exogenous fault for some coupled component (see Fig. 5.3 for the case of a routing probability equal to 1, and Fig. 5.4 otherwise). In so doing, for each *PropagationPort* of the meta-model:

- the place named *Failure Occurrence* represents the *propagatedFailureMode* attribute, within the *PropagationPort* class, originated by a previous *fault-to-failure* process. This place is already built by a *fault-to-failure* sub-procedure;
- a place named *Exogenous Fault Occurrence* is instantiated for the current *ExogenousFaultMode* instance;

- if not already present, a single immediate transition outgoing from the *Failure Occurrence* place is instantiated.

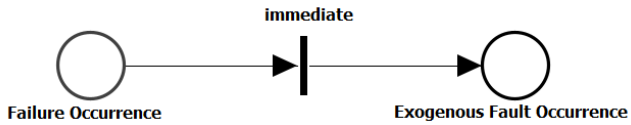

Figure 5.3: The STPN submodel of exogenous fault processes, representing the *failure-to-fault* propagation mechanism, in the case of a single exogenous fault and with a routing probability equal to 1.

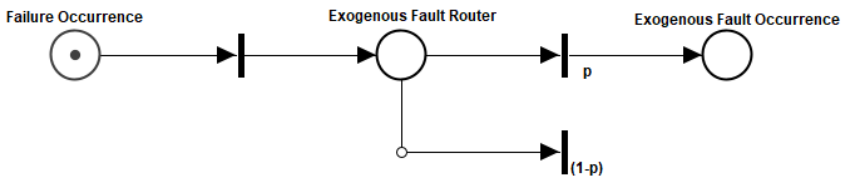

Figure 5.4: The STPN submodel of exogenous fault processes, representing the *failure-to-fault* propagation mechanism, in the case of a single exogenous fault and with a routing probability different from 1. Note that the propagation of the token from the *Failure Occurrence* place to the *Exogenous Fault Occurrence* place is now (with respect to Fig. 5.3) intermediated by a *Exogenous Fault Router* place, with two outgoing transitions weighted accordingly to the routing probability p.

Note that *failure-to-fault* propagations may affect several components involving many exogenous fault processes and a single failure may act as multiple external faults. In these cases, the *Failure Occurrence* place is not duplicated, nor is duplicated its outgoing transition, which is effectively connected with each *Exogenous Fault Occurrence* place. An example is reported in Fig. 5.5 where a single failure is propagated to three different exogenous faults, through three different *PropagationPort* instances, one of which is subject to a routing probability different from 1.

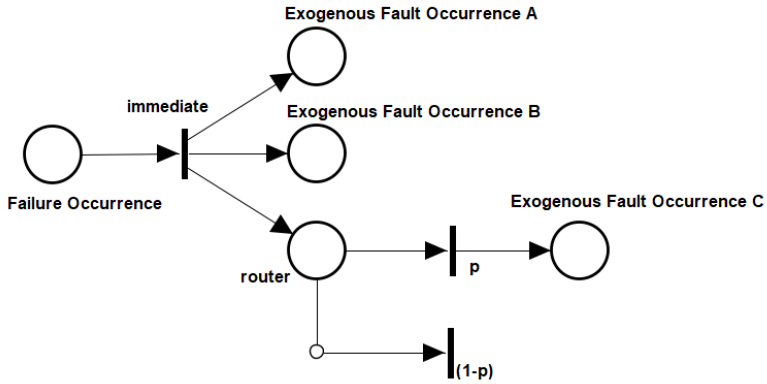

Figure 5.5: The STPN submodel of exogenous fault processes, representing multiple *failure-to-fault* propagation mechanisms. One of its propagation has a routing probability different from 1 (i.e., the propagation towards the *Exogenous Fault Occurrence C* place).

In conclusion, considering again the system depicted in Fig. 4.6 composed by 3 components (i.e., A, B, and C) with a failure logic depicted in the input FT artefacts of Fig. 5.6, leading the configuration of a meta-model instance for generating the subsequent STPN of Fig. 5.7 through the provided transformation rules.[1]

The most interesting aspect of the failure logic is the double propagation of the failure mode named *B_Failure1*, acting as an exogenous fault mode both for the A component (i.e., as *A_Fault2*), and for the C component (i.e, as *C_Fault3*), thus producing a directed acyclic graph in the reliability artefact. This double propagation is reflected by the generated STPN, in the outgoing transition of *B_Failure1* place, moving and duplicating the token in *A_Fault2* and in *C_Fault3* places through an intermediate router place for handling the routing probability.

[1]In Fig. 5.7, some labels of places or transitions have been suppressed for readability purposes.

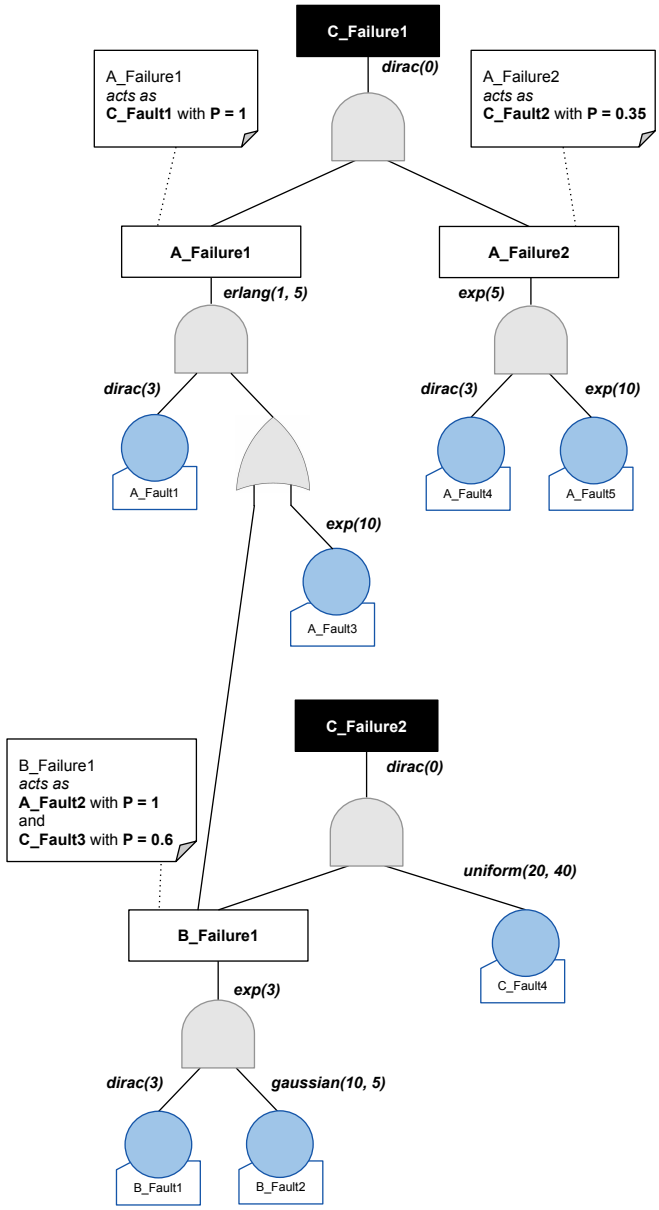

Figure 5.6: Stochastic FTs providing the failure logic configuration of the STPN in Fig. 5.7.

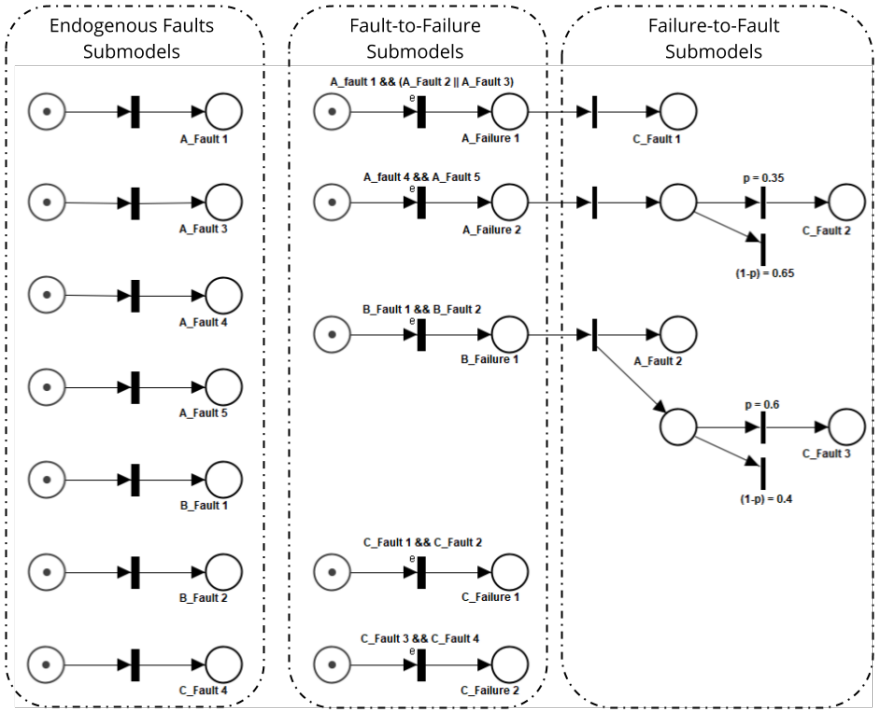

Figure 5.7: The STPN model of failure logic for the the system depicted in Fig. 4.6 composed by 3 components (i.e., A, B, and C) with a failure logic depicted in the input FT artefacts of Fig. 5.6. Note that have been generated: 7 "Endogenous Faults Submodels" , one for each *EndogenousFault* instance (thus, one for each basic event of the FT); 5 "Fault-to-Failure Submodels", one for each *FailureMode* on top of a bounded sub-tree, each one representing an *ErrorMode* instance; finally, 4 "Failure-to-Fault Submodels", one for each *PropagationPort*, each one identifiable in the FT as a textual note connected to an intermediate *FailureMode*.

5.3 Configuring STPN models in a tool-chain perspective

STPN models have been selected as general models able of capturing and characterising the temporal behaviour of a system in terms of failure logic, but in order to enable concrete quantitative analysis over these mathematical abstractions a customisation is required considering that solution techniques strictly depend on the stochastic parametrisation of the models.

Conceptually, while general distributions of the STPN may model any possible probability density function, as done by the meta-model, concrete implementations of the STPN should also account for quantitative solutions, offered by state-of-the-art tools. Thus, in these cases, an approximation strategy may be required to perform feasible analysis.[2]

While, in this perspective, single run simulations of STPN behaviours as well as Monte Carlo [77] simulations may exploit any possible stochastic distribution (as long as a sample of that distribution may be automatically generated); quantitative analysis approaches require adaptation or approximation of stochastic distributions to supported ones:

- Markovian transient and steady-state analysis [98] may be applied over STPN models with only exponential (EXP) and immediate (IMM) transitions, known in literature as GSPN models, which subtend Continuous-Time Markov Chains processes. In practice, stochastic distributions of the meta-model should be approximated with markovian ones (e.g., a single exponential distribution, an erlang distribution, or a phase-type approximation [56]);
- Markov Regenerative Processes (MRPs) may be exploited so as to evaluate transient and steady-state probabilities with [41] or without [55] the "enabling restriction" (i.e., at most one expolynomial transition is enabled in each state) [21]. In practice, stochastic distributions of the meta-model should be approximated with expolynomial ones, paying attention to further limitations;
- Non-deterministic analysis [107] over compact representations of the dense set of timed states that can be reached by STPN models, sup-

[2]Expected values should be equal to MTTFs of related components described by technical sheets, while higher-order moments may drive further refinements.

porting verification of qualitative properties of a model (e.g, whether a marking can be reached). In practice, identifying or approximating boundaries of the stochastic distributions within the model.

In so doing, in a tool-chain perspective, STPN models may also be exploited in different analysis tools (e.g., GreatSPN [3], Mercury [95], TimeNET [119]) or exported in different output formats so as to exploit their representations as formal inputs to specific tools: for example, the adoption of Petri Net Markup Language (PNML), an XML-based interchange format for Petri Nets, enables analysis based on several tools (e.g., ITS Tools [101], PNML Framework [52]); while the adoption of XPN format, a proprietary XML markup, enables the integration in Oris Tool [78].

Chapter 6

Towards a Tool for Reliability Analysis

In this Chapter a description of how the MDE approach and its related round-trip engineering process can be adapted within a concrete software implementation is reported. The proposed meta-model constitutes the core element for the realisation of a newborn tool, enabling the modelling of system specification and failure logic, as well as, supporting the generation of STPNs.

Specifically, the MDE approach within a tool-chain perspective is detailed in Sect. 6.1, and an overview of prototypical Java API, designed and implemented for supporting a programmatic definition of meta-model instances is described in Sect. 6.2. Finally, a brief description of a REST service implementation of the forward engineering process, laying the foundations for the tool in a as-a-service mode, is described in Sect. 6.3.

6.1 The MDE approach in a tool-chain perspective

The proposed MDE approach, leveraging on the meta-model described in Chapter 4, enables a *round-trip engineering* process as depicted in the summary image of Fig. 6.1.

On the one hand, the System specification artefacts (i.e., SysML BDDs) and failure logic artefacts (i.e., stochastic FTs) are given in input to the meta-model so as to activate an initialisation process of an executable instance, thus realising the so called *forward engineering* process. On the other hand, a *reverse engineering* process is supported by the meta-model, able to perceive external manual modifications which can be reflected at runtime so as to update all input artefacts.

At the same time, the meta-model enables the automated generation of STPNs, adopting a set of transformation rules, as described in Chapter 5.

For the sake of concreteness, in order to reap the benefits of the theorised MDE approach on concrete case studies, a tool implementation is needed for automatising all the underlying processes; also identifying convenient input/output data formats, target tools and frameworks, as well as exposing an Application Programming Interface (API) for runtime modifications of executable meta-model instances.

Specifically, the Oris Tool has been identified as the target analysis framework, requiring in input the XPN format for STPN representation; through the adoption of the Sirio Java library is possible to programmatically build

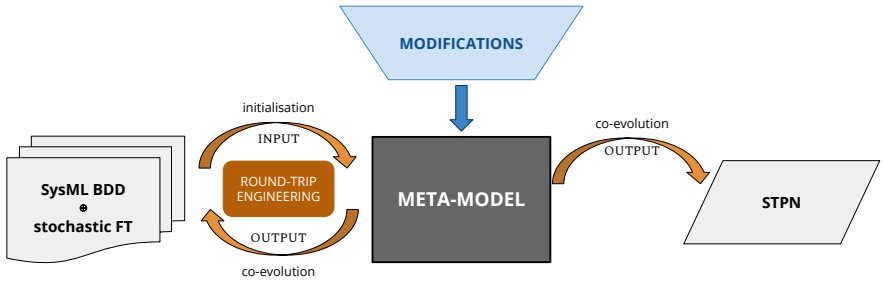

Figure 6.1: Summary of the *round-trip engineering* process based on the meta-model.

an instance of a STPN and export it in the XPN format. Besides, a custom Java API supports the initialisation of the meta-model and subsequent runtime modifications, as described in Sect. 6.2, also laying the foundations for the design and development of a prototypical tool *as-a-service*, within a cloud environment, as described in Sect. 6.3.

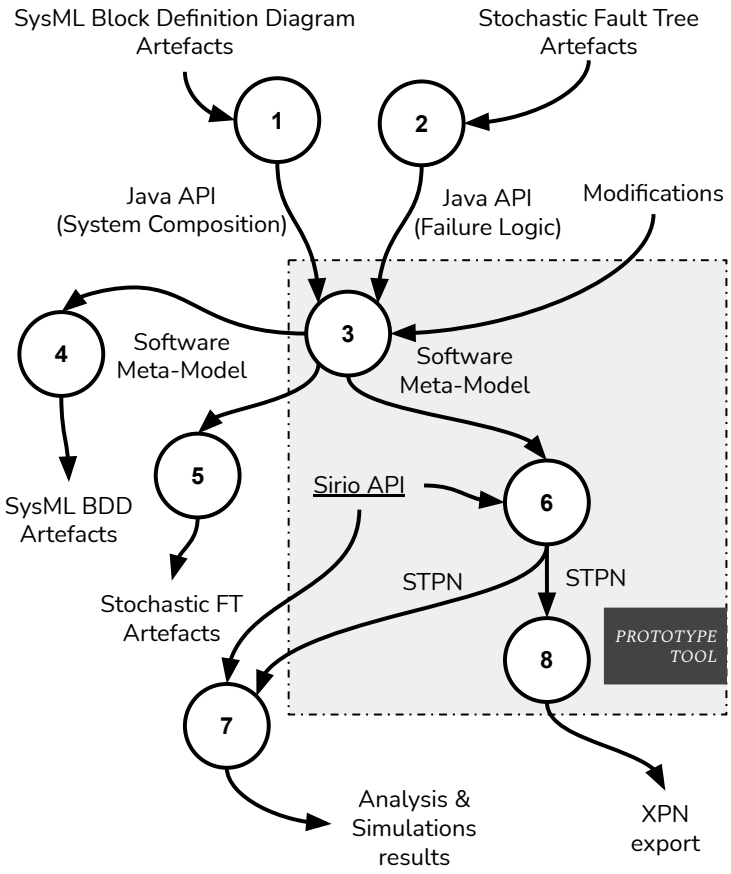

Figure 6.2: The designed workflow of the prototype tool, in the tool-chain perspective.

Fig. 6.2 reports the designed workflow of the prototype tool, within a toolchain perspective, highlighting internal processes actually implemented by the tool (i.e., processes bounded within the grey box) as well as input/output artefacts. Specifically:

- the *Process 1*, actually manually done by Java programmers but in practice automatable through the identification of specific data formats, transforms SysML BDDs into the specific Java code, exposed by the tool API;
- the *Process 2*, actually manually done by Java programmers but in practice automatable through the identification of specific data formats, transforms stochastic FTs into the specific Java code, exposed by the tool API;
- the *Process 3*, manually done by Java programmers over the Java API, implements runtime transformations over the initialised meta-model instance;
- the *Process 4* is responsible for the automated export of SysML Block Definition Diagram artefacts;
- the *Process 5* is responsible for the automated export of stochastic FT artefacts;
- the *Process 6* exploits the Sirio API so as to generate STPN executable models;
- the *Process 7* exposes a set of *built-in* analysis techniques, offered by the Sirio API;
- the *Process 8* provides functionalities to export an STPN model in the XPN format, analysable *via* Oris Tool.

While the actual implementation of the tool[1] provides only Processes 3, 6, and 8, as soon as adequate data format will be identified for SysML BDDs and for stochastic FTs, a concrete implementation also for remaining processes will be provided.

[1] Available for public access at `https://faultflow.dinfo.unifi.it`

6.2 The Java API

In this section, the specification of the system hierarchy and its failure logic through the prototypical Java API is described.

Considering again the system depicted in Fig. 4.6 composed by 3 components (i.e., A, B, and C) combined in a hierarchy where C represents the (root) parent of A and B, and with a failure logic defined by FTs of Fig. 5.6 (which, practically, are a Directed Acyclic Graph), listing 6.1 illustrates the usage of the API to programmatically build the hierarchy of the system S with its *CompositionPort* and *PropagationPort* instances, also characterising its failure logic in terms of *failure-to-fault* and *fault-to-failure* mechanisms.

Finally, API also provides a static method (i.e., *XPNExporter.export(system, "file-name.xpn")*) implementing the automated transformation from the meta-model instance of the system to a STPN representation, realised accordingly to the format of Oris Tool.

```
1  HashMap<String, FaultMode> faultModes = new HashMap<>();
2
3  // System hierarchy definition
4  System system = new System("S");
5  Component a = new Component("A");
6  Component b = new Component("B");
7  Component c = new Component("C");
8  system.addComponent(a, b, c);
9  system.setTopLevelComponent(c);
10 CompositionPort ac = new CompositionPort(a, c);
11 CompositionPort bc = new CompositionPort(b, c);
12 c.addChildren(ac, bc);
13
14 // Endogenous Fault Modes definition
15 EndogenousFaultMode enFM_A1 = new EndogenousFaultMode("A_Fault1");
16 enFM_A1.setArisingPDF("dirac(3)");
17 EndogenousFaultMode enFM_A3 = new EndogenousFaultMode("A_Fault3");
18 enFM_A3.setArisingPDF("exp(10)");
19 EndogenousFaultMode enFM_A4 = new EndogenousFaultMode("A_Fault4");
20 enFM_A4.setArisingPDF("dirac(3)");
21 EndogenousFaultMode enFM_A5 = new EndogenousFaultMode("A_Fault5");
22 enFM_A5.setArisingPDF("exp(10)");
23 EndogenousFaultMode enFM_B1 = new EndogenousFaultMode("B_Fault1");
24 enFM_B1.setArisingPDF("dirac(3)");
25 EndogenousFaultMode enFM_B2 = new EndogenousFaultMode("B_Fault2");
26 enFM_B2.setArisingPDF("erlang(10,5)");
27 EndogenousFaultMode enFM_C4 = new EndogenousFaultMode("C_Fault4");
28 enFM_C4.setArisingPDF("uniform(20,40)");
```

```
29
30 faultModes.put(enFM_A1.getName(), enFM_A1);
31 faultModes.put(enFM_A3.getName(), enFM_A3);
32 faultModes.put(enFM_A4.getName(), enFM_A4);
33 faultModes.put(enFM_A5.getName(), enFM_A5);
34 faultModes.put(enFM_B1.getName(), enFM_B1);
35 faultModes.put(enFM_B2.getName(), enFM_B2);
36 faultModes.put(enFM_C4.getName(), enFM_C4);
37
38 // Exogenous Fault Modes definition
39 ExogenousFaultMode exFM_A2 = new ExogenousFaultMode("A_Fault2");
40 ExogenousFaultMode exFM_C1 = new ExogenousFaultMode("C_Fault1");
41 ExogenousFaultMode exFM_C2 = new ExogenousFaultMode("C_Fault2");
42 ExogenousFaultMode exFM_C3 = new ExogenousFaultMode("C_Fault3");
43
44 faultModes.put(exFM_A2.getName(), exFM_A2);
45 faultModes.put(exFM_C1.getName(), exFM_C1);
46 faultModes.put(exFM_C2.getName(), exFM_C2);
47 faultModes.put(exFM_C3.getName(), exFM_C3);
48
49 // Failure modes of A and B Components definition
50 FailureMode fM_A1 = new FailureMode("A_Failure1");
51 ErrorMode eM_A1 = new ErrorMode("A_ToFailure1");
52 eM_A1.addInputFaultMode(enFM_A1, exFM_A2, enFM_A3);
53 eM_A1.addOutputFailureMode(fM_A1);
54 eM_A1.setEnablingCondition("A_Fault1 && (A_Fault2 || A_Fault3)",
       faultModes);
55 eM_A1.setPDF("erlang(1,5)");
56
57 FailureMode fM_A2 = new FailureMode("A_Failure2");
58 ErrorMode eM_A2 = new ErrorMode("A_ToFailure2");
59 eM_A2.addInputFaultMode(enFM_A4, enFM_A5);
60 eM_A2.addOutputFailureMode(fM_A2);
61 eM_A2.setEnablingCondition("A_Fault4 && A_Fault5", faultModes);
62 eM_A2.setPDF("exp(5)");
63
64 a.addErrorMode(eM_A1, eM_A2);
65
66 FailureMode fM_B1 = new FailureMode("B_Failure1");
67 ErrorMode eM_B1 = new ErrorMode("B_ToFailure1");
68 eM_B1.addInputFaultMode(enFM_B1, enFM_B2);
69 eM_B1.addOutputFailureMode(fM_B1);
70 eM_B1.setEnablingCondition("B_Fault1 && B_Fault2", faultModes);
71 eM_B1.setPDF("exp(3)");
72
73 b.addErrorMode(eM_B1);
74
75 // Propagation Ports definition
76 a.addPropagationPorts(
77     new PropagationPort(fM_A1, exFM_C1, c),
78     new PropagationPort(fM_A2, exFM_C2, c, 0.35));
79 b.addPropagationPorts(
80     new PropagationPort(fM_B1, exFM_A2, a, 0.6),
81     new PropagationPort(fM_B1, exFM_C3, c));
82
83 // Failure Modes of C Component definition
```

```
84 FailureMode fM_C1 = new FailureMode("C_Failure1");
85 ErrorMode eM_C1 = new ErrorMode("C_ToFailure1");
86 eM_C1.addInputFaultMode(exFM_C1, exFM_C2);
87 eM_C1.addOutputFailureMode(fM_C1);
88 eM_C1.setEnablingCondition("C_Fault1 && C_Fault2", faultModes);
89 eM_C1.setPDF("dirac(0)");
90
91 FailureMode fM_C2 = new FailureMode("C_Failure2");
92 ErrorMode eM_C2 = new ErrorMode("C_ToFailure2");
93 eM_C2.addInputFaultMode(exFM_C3, enFM_C4);
94 eM_C2.addOutputFailureMode(fM_C2);
95 eM_C2.setEnablingCondition("C_Fault3 && C_Fault4", faultModes);
96 eM_C2.setPDF("dirac(0)");
97
98 c.addErrorMode(eM_C1, eM_C2);
99
100 // Exports the STPN in XPN format
101 XPNExporter.export(system, "file-name.xpn");
```

Listing 6.1: Java code snippet for system definition in the running example.

Note that some utility methods have been defined for simplifying the API notation. Specifically, the *ErrorMode* class exposes two methods: the first (i.e., *setEnablingCondition()*) accepting in input a Boolean expression, written as a textual parameter in the format of a logic algebra (e.g., A_Fault1 && A_Fault2), as presented in the Backus-Naur Form (BNF) within Listing 6.2, where E is a valid expression and $Fault$ is a fault name; the second (i.e., *setPDF()*) accepting in input a string indicating the probability distribution type with its specified parameters (e.g., "$exp(5)$" represents an exponential distribution with the parameter λ equal to 5, while "$dirac(3)$" a deterministic distribution in the time instant t equal to 3).

```
1     E ::= E "&&" E | E "||" E | K"/"N"("Fault","*")" | "("E")" | Fault
2
3     Fault ::= "A_Fault1" | "A_Fault2" | "A_Fault3"
4
5     K ::= ? the K positive integer parameter in the voting OR ?
6     N ::= ? the N positive integer parameter in the voting OR ?
```

Listing 6.2: Backus-Naur Form related to expressions in input to *setEnablingCondition()* method. Considering that the valid syntax depends on fault names of the components (e.g., "*A_Fault1*", "*A_Fault2*", "*A_Fault3*"), the BNF form is here exemplified over the "A" *Component*, thus describing the syntax for its *ErrorMode* instances.

6.3 The Tool-as-a-Service

While a full implementation of the tool in a *as-a-service* mode is currently under development, a stateless REST service implementing the forward engineering processes from design artefacts directly to the output STPN[2] is provided. This service consumes in input a JSON file providing the architectural specification of a system with its failure logic in a custom interpretation of structural and reliability artefacts[3].

The designed JSON format (see Listing 6.3) is conceptually inspired to SysML BDD diagrams and stochastic FT artefacts, enabling a complete configuration of their main concepts. The implementation can be easily integrated with specific data mappers, for accepting in input further equivalent formats, with the aim of extending the tool compatibility, always in the tool-chain perspective.

An exemplary fragment of BDD configuration through the JSON *bdd* attribute is reported in Listing 6.4; while the exemplary fragment about stochastic FT configuration through the JSON *faultTree* attribute is detailed in Listing 6.5.

```
{
  "bdd": {
      "blocks": [...],
      "parentings": [...],
      "rootId":"..."
  },
  "faultTree": {
      "nodes": [...],
      "parentings": [...],
      "topEvents": [...]
  }
}
```

Listing 6.3: JSON input file containing both BDD and FT specifications.

[2]Actually, this service implements both the transformation from design artefacts to a meta-model instance and the one from the meta-model instance to the output STPN.

[3]In this way the *tool-as-a-service* is decoupled from specific data formats of external graphical tools, thus providing an intermediate representation for *Process 1* and *Process 2* in the tool-chain perspective workflow of Fig. 6.2.

```
"bdd": {
  "blocks": [
    {
      "externalId": "S",
      "description": "The whole System"
    },
    {
      "externalId": "A",
      "description": "The A Component"
    },
    {
      "externalId": "B",
      "description": "The B Component"
    },
    {
      "externalId": "C",
      "description": "The C Component"
    }
    ],
  "parentings": [
    {
      "parentId": "S",
      "childId":"C",
      "label":"c_label"
    },
    {
      "parentId": "C",
      "childId":"B",
      "label":"b_label"
    },
    {
      "parentId": "C",
      "childId":"A",
      "label":"a_label"
    }
  ],
  "rootId":"S"
}
```

Listing 6.4: JSON input file of the BDD in the example.

```
"faultTree": {
  "nodes": [
    {
      "externalId": "B_Fault1",
      "componentId": "B",
      "label": "Sensor1_MD",
      "nodeType": "BASIC_EVENT",
      "pdf": "dirac(3)"
    }, {
      "externalId": "B_Fault2",
      "componentId": "B",
      "label": "Sensor1_ED",
      "nodeType": "BASIC_EVENT",
      "pdf": "gaussian(10,5)"
    }, {
      "externalId": "b_prop1",
      "componentId": "B",
      "label": "b_prop1",
      "nodeType": "GATE",
      "gateType": "AND"
    }, {
      "externalId": "B_Failure1",
      "componentId": "B",
      "label": "Sensor1_PressureValueFailure",
      "nodeType": "FAILURE",
      "pdf": "exp(3)",
      "actsAs": [
        {
          "componentId": "A",
          "faultName":"A_Fault2",
          "routingProbability": 1
        }, {
          "componentId": "C",
          "faultName":"C_Fault3",
          "routingProbability": 0.6
        }
      ]
    },
    ...
  ],
  "parentings": [
    { "parentId": "b_prop1", "childId": "B_Fault1" },
    { "parentId": "b_prop1", "childId": "B_Fault2" },
    { "parentId": "B_Failure1", "childId": "b_prop1" },
    { "parentId": "a_prop", "childId": "B_Failure1" },
    { "parentId": "c_prop2", "childId": "B_Failure1" },
    ...
  ],
  "topEvents": ["C_Failure1", "C_Failure2"]
}
```

Listing 6.5: JSON input file of the FT in the example.

Chapter 7

Case Study

In this Chapter a synthetic case study related to a System of Systems operating in a business-critical context, where failure logic presents both direct and indirect couplings, is presented. The modelled scenario highlights the role of the Model-Driven Engineering approach in early design stages of a Cyber-Physical System, thus enabling offline analysis leading the identification of reliability requirements for the cyber-side of the system.

Specifically, in Sect. 7.1 the operative context of a "Pollution Monitor System", operating within the IoT scenario of a Smart City, impacting on urban traffic and accesses policies for the city, is addressed.
The system structural design is reported in Sect. 7.2, while its failure logic design is detailed in Sect. 7.3, introducing two interesting scenarios. In Sect. 7.4 analysis evaluating the impact of the software quality for the overall system reliability is evaluated in an early design perspective.
Finally, a brief discussion about expected benefits, deriving from the adoption of proposed MDE approach with respect to the presented case study is addressed in Sect. 7.5.

7.1 Operative context

The *Pollution Monitor System* is contextualised within a Smart City IoT scenario where sensors, actuators, and other physical devices cooperate to achieve common objectives inside a distributed environment.

In particular, the case study models a Cyber-Physical System (CPS) exploiting a sensor network able to perceive actual air pollution levels[1] so as to provide decision-making support to Smart City technicians with the aim of protecting air quality in urban areas through various actions, notably including traffic restriction, energy consumption reduction, building temperature monitoring.

As a common trait of IoT architectures, the system conveniently exploits a message broker software component in order to enhance system scalability by decoupling edge devices (responsible of raw data acquisition and preprocessing) from distributed cloud services (responsible for data refinement, storage and presentation).

Message brokers, usually implement Enterprise Integration Patterns, such as the Publish-Subscribe Channel for handling communications among subsystems, defining topics of interest, acting as routes from publishers to subscribers software components. Communication routes are dynamically determined by runtime configurations and, *a fortiori*, by self-organising strategies (e.g., a broker may adopt dynamic policies for dispatching queued messages, also in response external events), thus producing *indirect couplings* among software components not directly interconnected through hardware/software interfaces.

The system is characterised by a data flow rising from physical field devices that periodically send acquired telemetries to a central cloud storage: the broker intermediates every communication, also realising *map-reduce* refinement processes, such as calculating the moving average on a fixed number of samples.

Refined data are then analysed by an intelligent software agent, whose results populate a real-time dashboard, suggesting and supporting critical decisions for the Smart City Municipality.

[1] Air pollution indicators include: Sulfur dioxide (SO_2), Nitrogen oxides (NO and NO_2), Ozone (O_3), Carbon monoxide (CO), Benzene (C_6H_6 or BTEX), PM_{10} particulate matter, $PM_{2.5}$ particulate matter, Benzo(a)Pyrene (B(a)P), Arsenic (As), Cadmium (Cd) and Nickel (Ni).

In such a scenario, the cyber-side of the system plays a relevant role for the reliability of the entire system; software bugs may produce runtime failures in monitoring processing, thus leading to the adoption of inadequate policies, which in turn may produce huge damage for the economy of the Smart City.

7.2 Structural design

The system design is depicted in the SysML Block Definition Diagram of Fig. 7.1; where the overall SoS is represented by the block named *Pollution Monitor System*.

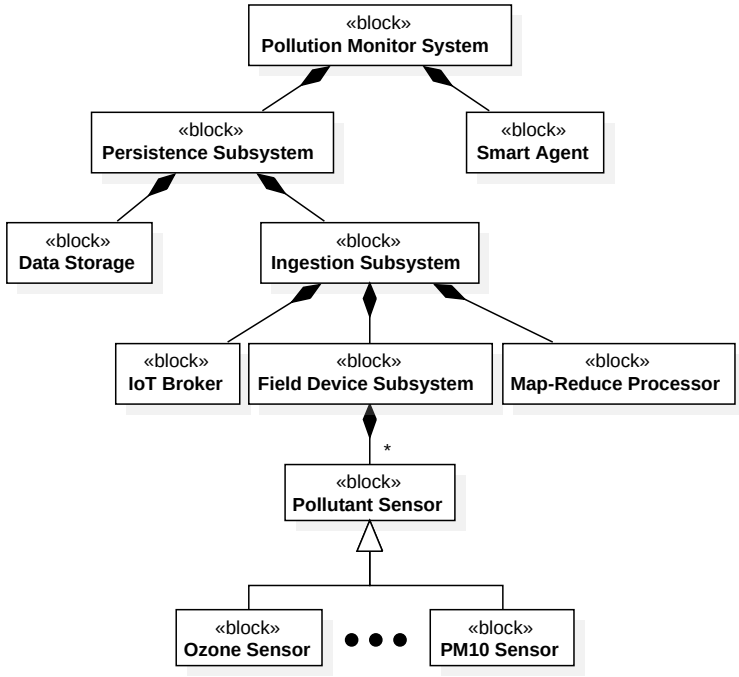

Figure 7.1: SysML BDD of the *Pollution Monitor System*.

Subsystems of the *Pollution Monitor System* have been designed so as to interpret specific roles with specific responsibilities:

- the *Smart Agent* is responsible for retrieving refined data from the *Persistence Subsystem* and for interpreting them so as to support decision-making processes to end-users through a dedicated dashboard;
- the *Persistence Subsystem* is responsible for storing within the *Data Storage* all the refined data, originated by the *Ingestion Subsystem*;
- the *Data Storage* is responsible for concretely handling storage and querying operations, through the adoption of a long-term Database Management System;
- the *Ingestion Subsystem* is responsible for managing lower level components so as to provide data-level/oriented communications, to acquire sampled data through subscription on topics within the message broker platform, as well as to refine and to synthesise data;
- the *IoT Broker* is responsible for providing data channels, each one associated to a data topic[2], where *Field Device Subsystems* act as publishers and Map-Reduce Processors act as subscribers (in the Publish-Subscribe pattern) about data streams;
- the *Map-Reduce Processor* is responsible for managing and intermediating data-level/oriented communications, acquiring sampled data through subscription on IoT Broker topics, and refining data (e.g., through moving average strategies);
- the *Field Device Subsystem* is responsible for instantiating a collection of software components acting as publisher on data topics of the *IoT Broker* (i.e., each publisher is responsible for sending raw data originated by a specific pollutant sensor of the perception layer);
- the *Pollutant Sensor* is responsible for perceiving environmental raw data, sampling pollutants. There are several types of sensors, each one dedicated to the sampling of a different pollutant (e.g., *Ozone Sensor*, *PM10 Sensor*).

[2]A topic is defined for each sampled parameter (e.g., the pollutant type).

7.3 Failure logic design

Notably, under the failure logic perspective, the concept of topic, provided by the *Ingestion Subsystem*, behaves as a pseudo-interface connecting software components through the intermediation of *IoT Broker* channels, generating several different data streams. Map and reduce processes, offered by the *Map-Reduce Processor*, have to be considered as software components acting both as subscribers, retrieving data flowing within channels, and publishers, when they republish synthesised data, for the broker.

Under these assumptions, the case study includes elements of complexity derived from *direct* and *indirect* failure *couplings* among components, as detailed in Sect. 3.1.

Indeed, many malfunctions may occur at each level of the SoS, eventually, deviating system components from their expected behaviour: damaged or defected IoT sensors, transmission and connectivity issues, wrong broker configurations (e.g., publishing or subscribing to wrong topics, data overflows over the ingestion buffer), defects in map and reduce processes (e.g., erroneous implementation of moving averages), excessive delays in delivery of information, or latent software bugs within the analytic agent business logic.

These malfunctions may lead to various failure manifestations, such as data precision errors, data corruption, data losses or unavailable subsystem which inevitably affect suggestions offered by the software agent inside the dashboard, in the worst case influencing human technicians decisions.

Two main failure scenarios are considered for the *Pollution Monitor System* so as to exemplify the MDE approach in leading early stages for the design of a CPS, showing how the proposed meta-model becomes an executable representation subsuming information from structural and reliability artefacts so as to automatically derive analysable quantitative models.

To facilitate the identification of the various processes within a FT with respect to the ontology defined by the meta-model, in this Section, the FTs have been decorated with:

- *dashed areas* labelled with the name of a component split FTs in fragments. The failure logic represented by a bounded fragment has to be intended as a single *fault-to-failure* propagation characterising the

associated component, which should be mapped in an *ErrorMode* instance within the meta-model (also instantiating all necessary faults and failure modes);

- *notes* with dashed links, each one interconnected with a failure mode, indicate failure modes acting as exogenous fault modes for the components of higher level in the FT, involved in a *failure-to-fault* propagation. Each single note must be mapped in a *PropagationPort* instance, coupling *ExogenousFaultMode* instances with *FailureMode* instances;

- probability distributions, decorating some edges, report the indication of the stochastically characterisation of basic events and *fault-to-failure* propagations.
  The *probability density function* attributes of *ErrorMode* and *EndogenousFaultMode* instances must be created according to these probability distributions. Rate parameters of probability distributions must be defined according to their specific syntax (e.g., the rate parameter *lambda* of an exponential distribution must be intended as the reciprocal of the Mean Time To Failure). The unit of measure adopted for the experimentation is the day, which means that an exponential with $lambda = 0.1$ has a Mean Time To Failure of 10 days.

7.3.1 Scenario 1

In a first scenario, the FT of Fig. 7.2 models the case of *no suggestion failure*, which is manifested when *Smart Agent* is not able to provide any suggestion supporting decision-making for the final user. This may be due either to the unavailability of the software agent itself or to the absence of perceived data at analysis time (i.e., the agent does not retrieve any data from the *Persistence Subsystem* or no samples arrived from the *Ingestion Subsystem*).

In this scenario, according to the data flow, the *Smart Agent* acts as a higher level component compared to the *Persistence Subsystem*, even though both components belong to the same level in the BDD. This is the representation of an *indirect coupling*, occurring between the *Persistence Subsystem* and the *Smart Agent*, due to dependencies subtended by the data retrieving modalities, inasmuch the *Smart Agent* may clearly use ingested data persisted within the *Data Storage*.

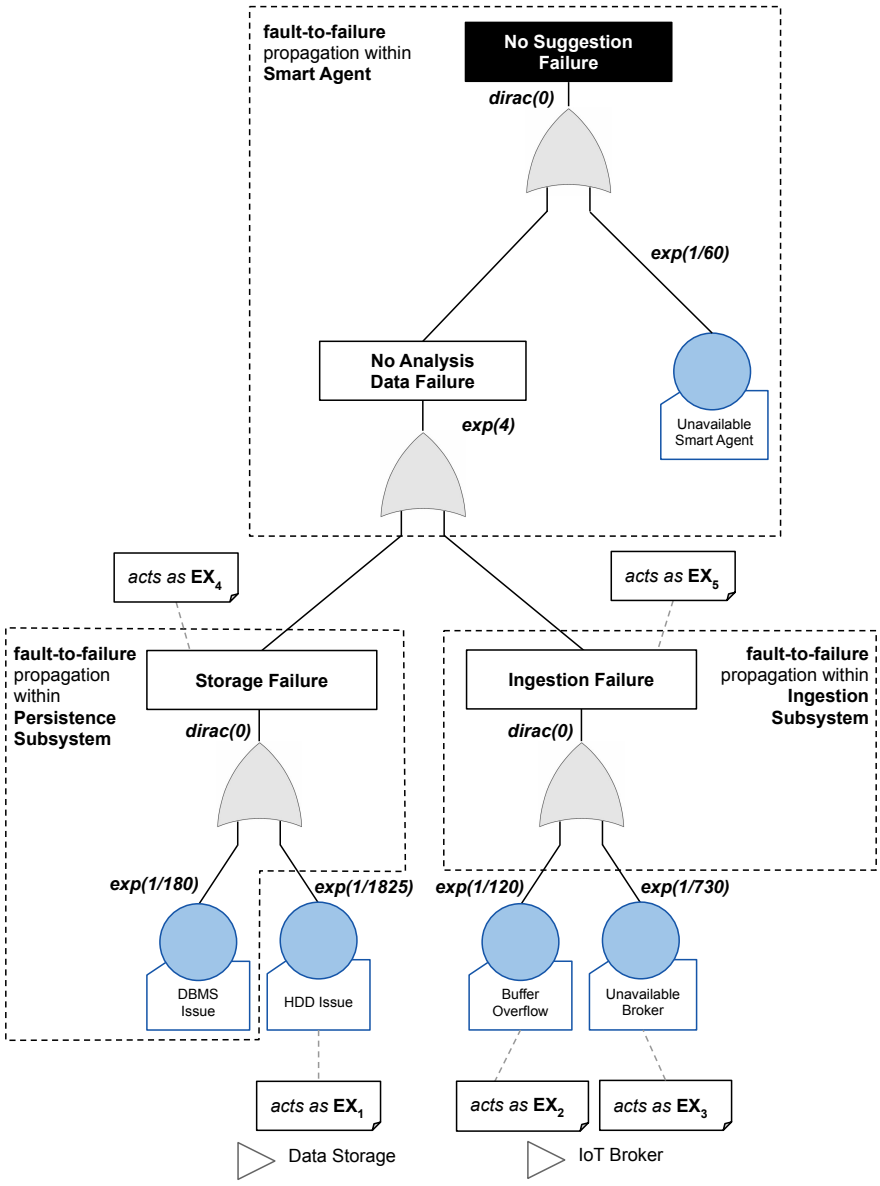

Figure 7.2: *Pollution Monitor System* FT of *No Suggestion Failure*.

In Tab. 7.1 a summary of the failure logic, derivable from the FT of Fig. 7.2, is reported, highlighting salient features leading the model-driven configuration of the executable meta-model, which cannot be conveniently represented within an *UML Object Diagram* because of the structural and failure logic complexity leading to a wide number of expected object instances.

Subsystem	**fault-to-failure**	**failure-to-fault**
Data Storage	· HDD issue	none
Persistence Subsystem	· Storage Failure [*DBMS issue* OR EX_1]	EX_1 := HDD issue
IoT Broker	· Buffer Overflow · Unavailable Broker	none
Data Storage	· Ingestion Failure [EX_2 OR EX_3]	EX_2 := Buffer Overflow EX_3 := Unavailable Broker
Smart Agent	· No Analysis Data Failure [EX_4 OR EX_5] · No Suggestion Failure [*No Analysis Data Failure* OR *Unavailable Smart Agent*]	EX_4 := Storage Failure EX_5 := Ingestion Failure

Table 7.1: Summary of the failure logic of the first scenario, derivable from the FT of Fig. 7.2, reporting for each subsystem its interpretations in terms of *fault-to-failure* and *failure-to-fault* processes of the meta-model. Note that the EX_j notation represents an exogenous fault, while Boolean expressions of each *fault-to-failure* process (i.e., an *ErrorMode* instance) are reported within square brackets, near the failure mode generated on top of it.

Note that the meta-model does not impose restrictions about the level of detail to adopt; indeed, in the example, a self-referred *failure-to-fault* propagation has been explicited for highlighting and for stochastically characterising the error mode leading to the *No Analysis Data Failure*.

The *Smart Agent* contains a first error mode producing the *No Analysis Data Failure*, which in turn acts as an exogenous fault for the error mode characterising the *No Suggestion Failure*. In so doing, several reasons may lead to the explication/explicitation of an intermediate failure, exploiting self-referred *failure-to-fault* propagations, such in the case of *No Analysis*

Data Failure: *i)* the failure mode is involved also in external processes (acting as an exogenous fault for external components); *ii)* for reliability purposes, the intermediate failure mode must be evaluated with quantitative approaches; *iii)* to better approximate stochastically the whole behaviour of the component owning the failure mode.

7.3.2 Scenario 2

In a second scenario, the FT of Fig. 7.3 models the case of *wrong suggestion failure*, which is manifested when *Smart Agent* provides inappropriate suggestions to final users, thus possibly inducing in wrong decisions of the Municipality. This may be caused either by a software bug or by analysis applied over corrupted data. This second FT captures the whole complexity about configurations of the *Ingestion Subsystem* and about data synthesis processes. In the specific case, the *Map-Reduce Processor* computes a moving average over static temporal windows.

The sub-tree of *PM10 Data Synthesis Failure* models significant disalignaments for a specific pollutant between the calculated moving average and the real world values: this failure is manifested only if a wide number of data acquired from sensors are defective for a continuous time period. In this scenario, a filtering strategy able to remove at most K'' outliers (over N'' values) at a time has been considered. A defective data can be caused by a defective sensor or by a wrong topic configuration of related publisher client processes (within the *Field Device Subsystem*), thus producing a sensible delay in the failure propagation mechanism. At the same time, it can be supposed that the *Smart Agent* is able to tolerate at most the failure of K' pollutants data refinement processes over N' monitored pollutants.

In Tab. 7.2 a summary of the failure logic, derivable from the FT of Fig. 7.3, is reported. Also in this case it is unfeasible to represent a complete *UML Object Diagram* about propagation processes.

7.4 Quantitative Analysis

The experimental case study emphasises the importance of evaluating system reliability, focusing primarily on the *Smart Agent*, whose failures affecting the

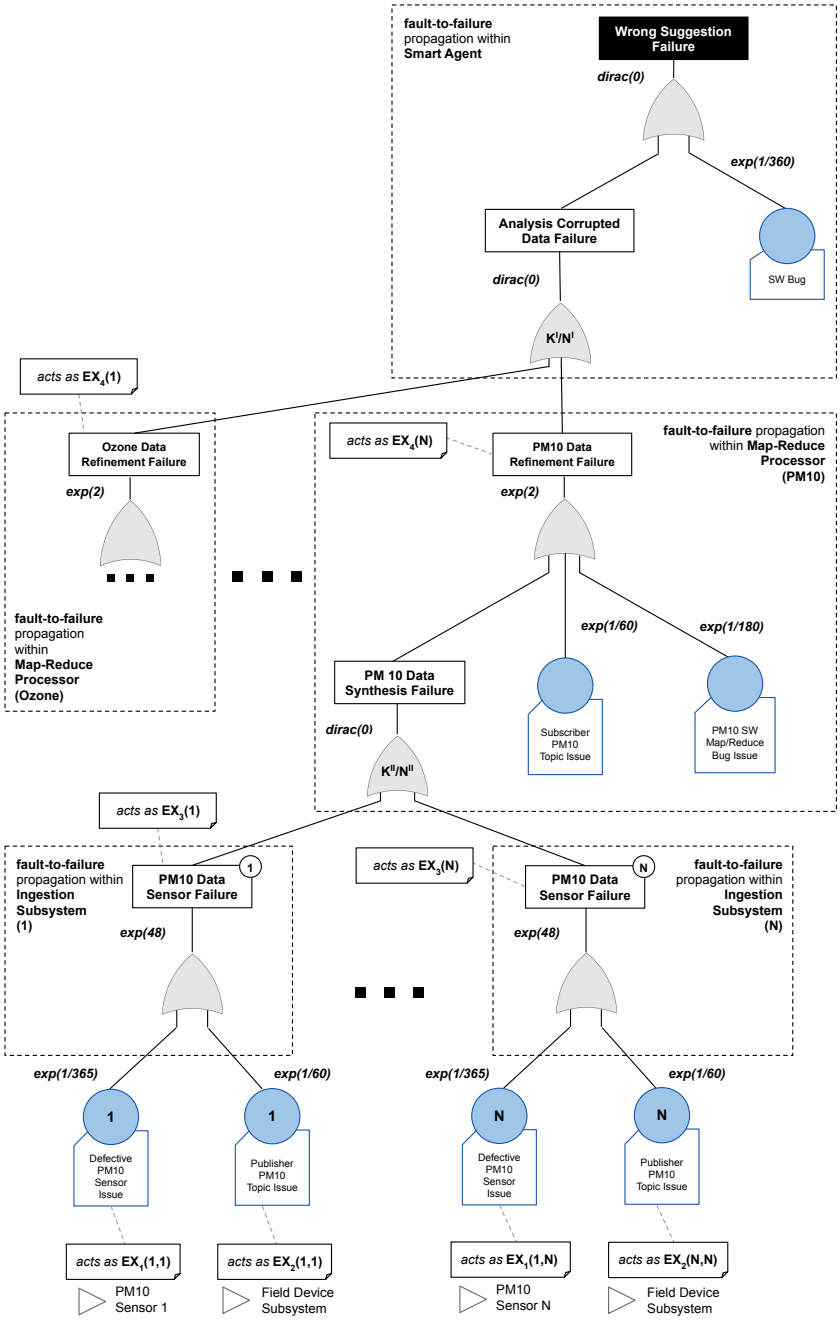

Figure 7.3: *Pollution Monitor System* FT of *Wrong Suggestion Failure*.

Subsystem	**fault-to-failure**	**failure-to-fault**
Pollutant Sensor	· Defective Pollutant Sensor Issue(p,i)	none
Field Device Subsystem	· Publisher Pollutant Topic Issue(p,i)	none
Ingestion Subsystem	· Pollutant Data Sensor Failure(p,i) [$EX_1(p,i)$ OR $EX_2(p,i)$]	$\mathrm{EX}_1(p,i)$:= Defective Pollutant Sensor Issue(p,i) $\mathrm{EX}_2(p,i)$:= Publisher Pollutant Topic(p,i)
Map-Reduce Processor	· Pollutant Data Synthesis Failure(p) [K'' out of $N''EX_3$(p,i)] · Pollutant Data Refinement Failure(p) [*Pollutant Data Synthesis Failure(p)* OR *Subscriber Pollutant Topic Issue(p)* OR *Pollutant SW Map/Reduce Bug Issue(p)*]	$\mathrm{EX}_3(p,i)$:= Pollutant Data Sensor Failure (p,i)
Smart Agent	· Analysis Corrupted Data Failure [K' out of $N'EX_4$(p)] · Wrong Suggestion Failure [*Analysis Corrupted Data Failure* OR *SW Bug*]	$\mathrm{EX}_4(p)$:= Pollutant Data Refinement Failure(p)

Table 7.2: Summary of the failure logic of the second scenario, derivable from the FT of Fig. 7.3, reporting for each subsystem its interpretations in terms of *fault-to-failure* and *failure-to-fault* processes of the meta-model. Note that the EX_j notation represents an exogenous fault, while Boolean expressions of each *fault-to-failure* process (i.e., an *ErrorMode* instance) are reported within square brackets, near the failure mode generated on top of it. Besides, in this scenario, the failure logic strictly depends on redundancy levels of installed sensors and of monitored pollutant types. To synthesise the table, some failures and processes have been parameterized with p and i, respectively representing the iteration over the monitored pollutant and the $i-th$ sensor within the $i-th$ field device.

monitoring dashboard may lead to improper solutions and decisions by Smart City executives (e.g., a severe traffic block, while still bringing environmental benefits, may also have huge impact societal and economical).

In this Section, advantages of adopting the presented MDE approach (see Sect. 3.2) in supporting early design of Cyber-Physical Systems are exemplified, by evaluating the reliability of the cyber-side with respect to the physical-side, comparing the *preliminary* failure model (depicted in the FTs of each scenario in Fig. 7.2 and Fig. 7.3), where hardware and software components may fail, with an *ideal* model where software components never fail.

In the case of unsatisfied reliability requirements, further compared analysis among different system configurations may be performed, integrating software fault-tolerance techniques (e.g., redundancy policies, voting algorithms, or caching strategies) in order to determine a *target* system design guaranteeing to satisfy both requirements specification and business criteria.

In so doing, the approach conciliates the contrasting needs of increasing the reliability of software components up to an acceptable level while maintaining acceptable design and development costs.

7.4.1 Scenario 1

Specifically, the Boolean logic modelled in Fig. 7.2 represents the limit case of a FT equivalent to a great disjunction on all the possible endogenous faults, everyone acting as single point-of-failures. Whereas, moreover, all the *fault-to-failure* logic propagate very quickly (either through IMM transitions or through EXP with high rate, i.e., $\lambda = 4$), the top event occurrence (i.e., *No Suggestion Failure*) can be stochastically approximated by an exponential distribution with λ equal to the summation of each endogenous fault rate (i.e., $\lambda = 3.25 * 10^{-2}$). As a consequence, in absence of redundancy policies, the resulting system availability is very short with a probability of 0.63 to be failed at the first month. This is due to the fact that, while hardware components behave reliably, system failures are mainly determined by the cyber-side. Specifically, just by introducing a dual modular redundancy operating mode for the *Smart Agent* which is the main responsible for the overall system failure (due to an $MTTF$ of 60 days), the probability of being failed at the first month can be decreased to 0.38.

7.4.2 Scenario 2

The second scenario of *Wrong Suggestion Failure*, depicted in Fig. 7.3, presents a business-critical failure, which may have a higher impact on the Smart City economy, as reflected by the internal propagation logic further exacerbated by the presence of some *voting OR gates*. Markovian analysis results, reported in Fig. 7.4, stress the role as runtime raw data manager played by the cyber side (in particular by IoT publisher and subscriber clients, and by map-reduce processes) and the way it affects once again the overall system reliability, making ineffective and superfluous the adoption of a broad set of professional perceiving hardware sensors and driving the system to a probability of failure of 0.84 at 100 days of operation (as outlined by the light grey curve of Fig. 7.4).

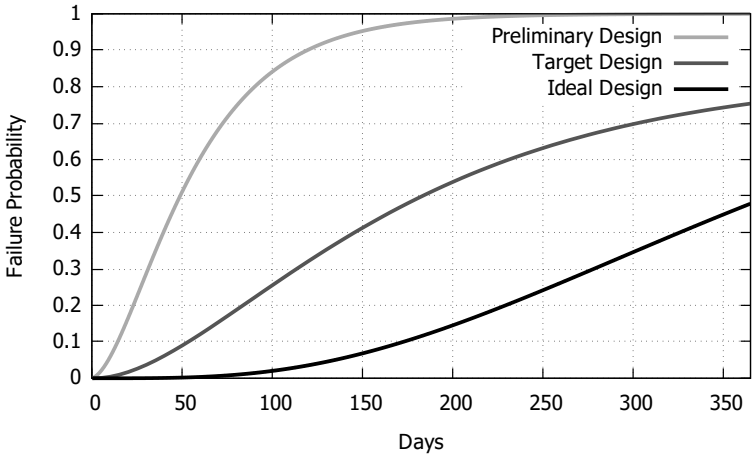

Figure 7.4: Transient probabilities of the *Wrong Suggestion Failure* top event, computed under 3 different configurations of the *Pollution Monitor System* (i.e., preliminary, ideal, and target failure models).

To face this reliability issue, risk analysts can leverage the MDE approach, within the tool-chain perspective, to derive an *ideal* lower bound for the system failure probability (i.e., the black curve of Fig. 7.4), obtained by neglecting the impact of software components in the propagation logic.

On the one hand, this ideal software design supports the identification of minimum reliability requirements for software components, driving the

adoption of tailored software fault-tolerance techniques (e.g., Multi-Version Software [62], software rejuvenation policies [105]) and demanding for quality improvements on the overall development process (e.g., through the concrete implementation of eXtreme Programming methodologies [8] or Test-Driven Development practices [9]). As a practical example, the grey curve of Fig. 7.4 represents a feasible *target* failure model for the software development stage and has been obtained reducing by one-third the rate related to endogenous software faults, thus decreasing the system failure probability to 0.26 (at $t = 100$ days).

On the other hand, this enables to pay further attention to the real impact of hardware components, possibly acting coarse- and fine- tuning configurations to achieve a trade-off between economical costs and benefits in terms of reliability.

7.5 Discussion

The presented case study emphasises the role of the MDE approach in driving early design stages of a CPS, exploiting results produced by offline reliability analysis, thus weighing the development costs for the cyber-side (whose reliability is a key factor).

In the same way, the MDE approach may also be useful: in runtime monitoring processes[3], for analysing the system behaviour during operation; in determining advanced strategies for just-in-time maintenance techniques or software rejuvenation policies by fault-forecasting. Indeed, assuring a constant alignment between meta-model instances and their respective physical counterparts may realise the digital twins paradigm, where runtime and monitored data over physical components may enable refinements over probability density functions distributions, associated to endogenous fault and error mode instances, as well as over routing probabilities

In so doing, runtime transformations in STPN format enable runtime reliability evaluations and comparisons over different SoS configurations, exploiting alternatives derived from Product Line Engineering design to realise adaptive systems, which require to change their configurations in response

[3]Note that execution times, observed in the experimentation of the case study on a consumer-level notebook (equipped with 16GB of DDR4 RAM and *Intel Core i7-9750H* CPU) oscillate from 10ms to 70ms for the consecutive transformations from the JSON input to the XPN export. These executions times can surely be considered acceptable for the application in runtime analysis, not subject to real-time constraints.

to dynamic fault occurrences, as in the case of Interwoven Systems for which changes frequently occur at architectural, at parametric, as well as at functional level.

Chapter 8

Conclusion

This Chapter summarises the contribution of the thesis and discusses avenues for future research.

8.1 Summary of contributions

This dissertation contributes to the area of Model-Driven Engineering (MDE), proposing a model-driven approach supporting timed failure logic analysis of complex Cyber-Physical Systems (CPS) in business-critical scenarios.

Specifically, the research defines a meta-model joining structural information about system architectures with their failure logic, decoupling representations of communication interfaces from those of failure propagation, also coping with data-level failures. Erroneous data, generated by the perception layer of an Internet of Things (IoT) architecture, may produce faults which do not affect intermediate software components (e.g., in Message-Oriented Middlewares acting as brokers) orchestrating the communication flow, altering instead the behaviour of smart systems exploiting data for supporting remote decision-making processes not directly interconnected to physical sensors.

The meta-model enables a *round-trip engineering* through the definition of a set of *transformation rules*, supporting the automated and *correct-by-construction* initialisation of meta-model instances starting from SysML Block Definition Diagrams (BDD) for the system specification and stochastic Fault Trees (FT) for its timed failure logic, thus activating co-evolution mech-

anisms, which propagate external manual modifications applied on meta-model instances directly to adopted structural and reliability artefacts.

At the same time, a set of *transformation rules* has been defined so as to enable the automated generation of Stochastic Time Petri Nets (STPN) from meta-model instances, thus supporting quantitative evaluation of timed failure logic.

In so doing, the MDE approach, leveraging on the proposed meta-model, bridges the gap between the perspectives of System Engineering, Software Engineering and Reliability Engineering: indeed, a software implementation of the meta-model, allowing runtime modifications over its executable representation, directly combines reliability analysis models with design artefacts through automated transformations in a single and coherent vision.

The meta-model has been designed also for supporting the modelling of product families in a *product line* perspective, favouring reuse of meta-model instances while maintaining up-to-date runtime configurations in response to frequent changes.

A prototypical implementation of the proposed MDE approach has been introduced through a plain Java API as well as in a *tool-as-a-service* mode, exposing functionalities for automatically instantiating meta-model instances, starting from input artefacts, and for exporting STPN models in the format of Oris Tool.

The MDE approach has been experimented on a realistic case study related to a CPS operating within a Smart City, demonstrating its usefulness in the stochastic evaluation of reliability features with respect to system and failure logic requirements and design artefacts.

8.2 Directions for future work

The research activities will address the enhancement of the proposed meta-model for representing:

- *transient faults*, a class of faults that are not persistent within a system, but rather are characterised by a temporary nature (e.g., a transient fault may disappear from the system by itself, after a certain period);

- *recovery mechanisms*, capable of activating a repairing process for a faulty component (e.g., software approaches may exploit fault detection

techniques with alternative implementations);

- *software rejuvenation*, a fault-tolerant approach for restoring initial levels of reliability, thus fighting software ageing processes.

On the one hand, the interpretation of transient faults and recovery mechanisms concepts is someway different and a final design for the meta-model abstraction must be yet defined.

Considering that the application of recovery mechanisms produces transient faults, both these concepts may be modelled within a STPN by introducing a general transition, which moves the token from an *Endogenous Fault Occurrence* place to an *Endogenous Fault Process* place, restarting the timed transition.

However, at the time when the fault is restored, a transient fault may have contributed in a failure manifestation; thus, further investigations about how to deal with the propagation of these failures should be addressed, answering to the question *"should exogenous faults, activated by failure-to-fault processes, be considered transient or persistent?"*.

On the other hand, the rejuvenation process can be modelled in a STPN as a transition which removes and replaces the token from the starting place of an endogenous fault process, thus restarting the outgoing transition and enabling evaluations about scheduled restarts for software components.

Besides, the research will consider the adoption of alternative input artefacts (e.g., AADL, Reliability Block Diagrams, Component FTs) for the *round-trip engineering* process and output artefacts (e.g., queuing networks) for the generation of quantitative analysis models.

In the tool perspective, the Java API, also at the core of the *tool-as-a-service* mode, will be integrated with software modules able to implement the transformation rules from meta-model instances to design and reliability artefacts, thus realising the *reverse* process of the *round-trip engineering*.

Moreover, the tool will be improved with the implementation of *built-in* analysis and simulations (e.g., root-cause analysis, fault injection practices for rare events studies) and of a User Interface for graphically modelling the system specification and its failure logic, as well as, for easing the input of external modifications about system architecture and failure propagation mechanisms.

Appendix A

Appendix

This appendix is related to the transformation algorithms at the core of the round-trip engineering process (see Sect. 4.3) between the meta-model and adopted structural and reliability artefacts, as well as to the transformation algorithm for generating an STPN model from a meta-model instance.

A.1 SysML Block Definition Diagram to Meta-model instance transformation algorithm

In this Section, a pseudo-algorithm related to the transformation rules, described in Sect. 4.3, leading the automated instantiation of an architectural fragment of the meta-model configuration starting from a SysML Block Definition Diagram (BDD) artefact is presented.

Essentially, the algorithm is based on a in-depth visit of the SysML BDD artefact, starting from the top-level item, recursively exploring each child item.

Algorithm 1: SysML BDD to Meta-model architectural specification transformation

```
   input  : bdd (SysML BDD artefact)
   output: MMSA (Meta-Model System Architecture instance)

   // Global variable initialisation
1  MMSA ← ∅

   // Extracts the System block within BDD
2  systemItem ← extractTopLevelSystem(bdd)
3  system ← new System(extractItemName(systemItem))
4  topLevelComponent ← new Component(extractItemName(systemItem))
5  recursiveBDDvisit(systemItem, topLevelComponent, system)
6  add system to MMSA
7  return MMSA
```

Algorithm 2: `recursiveBDDvisit`

```
   input  : item (the current BDD block)
   input  : parentComponent (the parent Component)
   input  : system (the overall System)

1  component ← new Component(extractItemName(item))
2  add component to system components
3  if parentComponent is not null then
4  |   compositionPort ← new CompositionPort(parentComponent, component)
5  |   add compositionPort to parentComponent
6  end
7  childrenItems ← children of item
8  foreach childItem ∈ childrenItems do
9  |   recursiveBDDvisit(childItem, component, system)
10 end
```

A.2 Meta-model instance to SysML Block Definition Diagrams transformation algorithm

In this Section, a pseudo-algorithm related to the transformation rules, described in Sect. 4.3, enabling the automated export of a SysML Block Definition Diagram (BDD) artefact from a runtime instance of the architectural fragment of the meta-model is presented.

Essentially, the algorithm is based on a in-depth visit of the meta-model instance, starting from the *System* instance, exploring recursively each child *Component*, identified through its *CompositionPort* instances.

Algorithm 3: Meta-model architectural specification to SysML BDD transformation

```
   input : mmsa (Meta-Model System Architecture instance)
   output: BDD (SysML BDD artefact)
   // Global variable initialisation
1  BDD ← ∅
2  system ← extractSystem(mmsa)
3  systemItem ← new Item(name of system)
4  add systemItem to BDD
5  topLevelComponentInstance ← topLevelComponent of system
6  topLevelComponentItem ← new Item(name of topLevelComponentInstance)
7  add topLevelComponentItem to children of systemItem
8  recursiveMetaModelVisit(topLevelComponentInstance, topLevelComponentItem)
9  return BDD
```

Algorithm 4: `recursiveMetaModelVisit`

```
   input : component (the current Component)
   input : parentItem (the parent BDD block)
1  compositionPorts ← compositionPorts of component
2  foreach compositionPort ∈ compositionPorts do
3  |  childComponent ← childComponentInstance of compositionPort
4  |  componentItem ← new Item(name of childComponent)
5  |  add componentItem to children of parentItem
6  |  recursiveMetaModelVisit(childComponent, componentItem)
7  end
```

A.3 Stochastic Fault Tree to Meta-model instance transformation algorithm

In this Section, a pseudo-algorithm related to the transformation rules, described in Sect. 4.3, leading the automated instantiation of a meta-model configuration starting from a stochastic Fault Tree artefact is presented.

Note that the algorithm is based on the recursive procedure *elaborateBoundedFT*, invoked over each top event, which firstly identifies the boundaries of a sub-tree corresponding to an error mode, and then for each lower level failure mode (i.e., a failure mode acting as a basic event in the bounded FT) invokes the recursion.

In turn, the *elaborateBoundedFT* exploits a recursive subprocedure *recursiveElaboration*, which effectively identifies endogenous and exogenous faults of the bounded sub-tree, at the same time building the enabling condition over the composition of logical gates.

Algorithm 5: Stochastic FT to Meta-model failure logic transformation

```
   input  : sft (Stochastic Fault Tree artefact)
   input  : mmsa (Meta-Model System Architecture instance)
   output: MMC (Meta-Model Configuration instance)

   // Global variable initialisation
 1 MMC ← mmsa

   // It identifies Stochastic FT top-events
 2 topEvents ← extractTopEvents(sft)
 3 foreach topEvent ∈ topEvents do
 4 |   fmLabel ← extractFailureModeLabel(topEvent)
 5 |   sourceC ← extractSourceComponent(topEvent)
 6 |   failureMode ← new FailureMode(fmLabel, sourceC)
 7 |   add failureMode to MMC
 8 |   elaborateBoundedFT(topEvent)
 9 end
10 return MMC
```

Algorithm 6: `elaborateBoundedFT`

```
   input  : bftTopEvent

   // Global variables
 1 INTERMEDIATE_FAILURE_MODES_QUEUE ← ∅
 2 EXOGENOUS_FAULTS ← ∅
 3 ENDOGENOUS_FAULTS ← ∅

 4 topEventGate ← child of bftTopEvent
 5 enablingCondition ← recursiveElaboration(topEventGate)
 6 timeToFailurePDF ← extractTimeToFailurePDF(topEventGate)
 7 errorMode ← new ErrorMode(EXOGENOUS_FAULTS, ENDOGENOUS_FAULTS,
     failureMode, enablingCondition, timeToFailurePDF)
 8 add errorMode to MMC

 9 while INTERMEDIATE_FAILURE_MODES_QUEUE is not empty do
10 |   intermediateFailureMode ← pull an item from
   |     INTERMEDIATE_FAILURE_MODES_QUEUE
11 |   elaborateBoundedFT(intermediateFailureMode)
12 end
```

Algorithm 7: `recursiveElaboration`

```
   input  : event
   output: booleanExpression
 1 if event is a Gate then
 2 |   booleanExpression ← gateElaboration(event)
 3 |   return booleanExpression
 4 end
 5 else if event is a BasicEvent then
 6 |   endFmLabel ← extractExogenousFaultModeLabel(event)
 7 |   endoFm ← findEndogenousFaultMode(endFmLabel)
 8 |   if endoFm is NULL then
 9 |   |   endoFmPDF ← extractEndogenousFaultModePDF(event)
10 |   |   endoFm ← new EndogenousFaultMode(endFmLabel, endoFmPDF)
11 |   |   add endoFm to ENDOGENOUS_FAULTS add endoFm to MMC
12 |   end
13 |   return endFmLabel
14 end
15 else if event is a FailureMode then
16 |   fmLabel ← extractFailureModeLabel(event)
17 |   failureMode ← findFailureMode(fmLabel)
18 |   if failureMode is NULL then
19 |   |   add event to INTERMEDIATE_FAILURE_MODES_QUEUE
20 |   |   sourceC ← extractOwnerComponent(event)
21 |   |   failureMode ← new FailureMode(fmLabel, sourceC)
22 |   |   add failureMode to MMC
23 |   end
24 |   exoFmLabel ← extractExogenousFaultModeLabel(event)
25 |   exoFm ← new ExogenousFaultMode(exoFmLabel)
26 |   add exoFm to EXOGENOUS_FAULTS add exoFm to MMC
27 |   routingProb ← extractRoutingProbability(event)
28 |   c ← extractAffectedComponent(event)
29 |   propagationPort ← new PropagationPort(c,failureMode,exoFm,routingProb)
30 |   add propagationPort to MMC
31 |   return exoFmLabel
32 end
```

Algorithm 8: `gateElaboration`

```
   input  : event
   output: booleanExpression
 1 booleanExpression ← NULL
 2 boolExprs ← ∅
 3 children ← children of event
 4 foreach child ∈ children do
 5 |   boolExpr ← recursiveElaboration(child)
 6 |   add boolExpr to boolExprs
 7 end
 8 if event is a OrGate then
 9 |   booleanExpression ← "("+boolExprs [0]+" || "+boolExprs [1]+")"
10 end
11 if event is a AndGate then
12 |   booleanExpression ← "("+boolExprs [0]+" && "+boolExprs [1]+")"
13 end
14 if event is a VotingOr then
15 |   booleanExpression ← " K/N( "
16 |   while i ← 0; i < N; i++ do
17 |   |   append boolExprs [i] to booleanExpression
18 |   |   if i ≠ N then
19 |   |   |   append "," to booleanExpression
20 |   |   end
21 |   end
22 |   append ")" to booleanExpression
23 end
24 return booleanExpression
```

A.4 Meta-model instance to stochastic Fault Tree transformation algorithm

In this Section, a pseudo-algorithm related to the transformation rules, described in Sect. 4.3, enabling the automated export of a Stochastic Fault Trees artefact from a runtime instance of the failure logic fragment of the meta-model is presented.

Essentially, the algorithm is based on a in-depth visit of the meta-model instance, starting from *FailureMode* istances acting as top-events, exploring associated *ErrorMode* and *FaultMode* instances, then recurring on lower-level *FailureMode* instances acting as *ExogenousFaultMode* instances.

Algorithm 9: Meta-model to stochastic FT transformation

input : mm (Meta-Model instance)
output: SFT (Stochastic Fault Tree artefact)

```
// Global variable initialisation
SFT ← ∅
system ← extractSystem(mm)
// a top-event failure mode in the meta-model instance is
// a FailureMode instance not acting as propagatedFailure
// in any PropagationPort instance
teFMs ← extractTopEvents(system)
foreach teFM ∈ teFMs do
    topEvent ← name of teFM
    add topEvent to SFT
    childEvent ← recursiveFailureModeElaboration(teFM)
    add childEvent to children of topEvent
end
return SFT
```

Algorithm 10: `recursiveFailureModeElaboration`

```
input : failureMode (FailureMode instance)
output: gateEvent (event gate producing the event associated to the
        FailureMode instance)

// the ErrorMode instance producing the FailureMode instance is
retrieved
1  errorMode ← extractErrorMode(failureMode)
2  enablingCondition ← extractEnablingCondition(errorMode)
3  timeToFailurePDF ← extractTimeToFailurePDF(errorMode)
4  endoFMs ← extractEndogenousFaultModes(errorMode)
5  exoFMs ← extractExogenousFaultModes(errorMode)
6  basicEvents ← ∅
7  foreach endoFM ∈ endoFMs do
8  |   basicEvent ← new FaultModeEvent(name of endoFM)
9  |   add basicEvent to basicEvents
10 end
11 fmEvents ← ∅
12 foreach exoFM ∈ exoFMs do
13 |   propagationPort ← extractSourcePropagationPort(exoFM)
14 |   sourceFailureMode ← extractSourceFailureMode(propagationPort)
15 |   fmEvent ← findEvent(name of sourceFailureMode)
16 |   if fmEvent is NULL then
17 |   |   fmEvent ← new FailureModeEvent(name of sourceFailureMode)
18 |   |   childEvent ← recursiveFailureModeElaboration(sourceFailureMode)
19 |   |   add childEvent to children of fmEvent
20 |   end
21 |   add routingProbability for exoFM to fmEvent
22 |   add fmEvent to fmEvents
23 end
24 gateEvent ← transformEnablingCondition(enablingCondition, basicEvents,
     fmEvents)
25 set timeToFailurePDF in the output edge of the gateEvent
26 return gateEvent
```

Algorithm 11: `transformEnablingCondition`

input : enablingCondition (full or partial BooleanExpression instance)
input : basicEvents (Fault Tree Basic Events)
input : fmEvents (Failure Mode Intermediate Events)
output: event (the event associated to the enabling condition)

```
booleanOperator ← extractLogicalOperator(enablingCondition)
if booleanOperator is not NULL then
    event ← ∅
    if booleanOperator is OrGate then
        event ← new OrGateEvent()
    end
    else if booleanOperator is AndGate then
        event ← new AndGateEvent()
    end
    else if booleanOperator is VotingOrGate then
        K ← extractK(VotingOrGate)
        N ← extractN(VotingOrGate)
        event ← new VotingOrGateEvent(K, N)
    end
    splittedExpressions ← splitBooleanExpression(enablingCondition,
      booleanOperator)
    foreach splittedExpression ∈ splittedExpressions do
        childEvent ← transformEnablingCondition(enablingCondition,
          basicEvents, fmEvents)
        add childEvent to children of event
    end
end
else
    event ← findByName(basicEvents, enablingCondition)
    if event is NULL then
        event ← findByName(fmEvents, enablingCondition)
    end
end
return event
```

A.5 Meta-model instance to Stochastic Time Petri Net transformation algorithm

In this Section, a pseudo-algorithm related to the transformation rules, as described in Sect. 5.3, leading the automated extraction of a Stochastic Time Petri Net from a meta-model instance is presented.

Note that the algorithm is splitted in three subprocedures, respectively mapping *EndogenousFaultMode* instances, *ErrorMode* instances, and *PropagationPort* instances into corresponding STPN submodels.

Algorithm 12: Meta-model to stochastic Time Petri Net transformation

```
   input : mm (Meta-Model instance)
   output: STPN (Stochastic Time Petri Net instance)
   // Global variable initialisation
 1 STPN ← ∅
 2 endogenousFaultModes ← extractAllEndogenousFaultModes(mm)
 3 foreach endogenousFaultMode ∈ endogenousFaultModes do
 4 |   buildEndogenousProcessesSubModel(endogenousFaultMode)
 5 end
 6 errorModes ← extractAllErrorModes(mm)
 7 foreach errorMode ∈ errorModes do
 8 |   buildFaultToFailureSubModel(errorMode)
 9 end
10 propagationPorts ← extractAllPropagationPorts(mm)
11 foreach propagationPort ∈ propagationPorts do
12 |   buildFailureToFaultSubModel(propagationPort)
13 end
14 return STPN
```

Algorithm 13: `buildEndogenousProcessesSubModel`

input : endogenousFaultMode (EndogenousFaultMode instance)

```
1  label ← extractName(endogenousFaultMode)
2  startPlaceName ← label + " Process"
3  startPlace ← new Place(startPlaceName, 1)
4  endPlaceName ← label + " Occurrence"
5  endPlace ← new Place(endPlaceName, 0)
6  pdf ← extractPDF(endogenousFaultMode)
7  transition ← new Transition(startPlace, endPlace, pdf)
8  add startPlace to STPN
9  add endPlace to STPN
10 add transition to STPN
```

Algorithm 14: `buildFaultToFailureSubModel`

input : errorMode (ErrorMode instance)

```
1  label ← extractName(errorMode)
2  startPlaceName ← label + " Fault-to-Failure Process"
3  startPlace ← new Place(startPlaceName, 1)
4  failureMode ← extractFailureMode(errorMode)
5  failureModeName ← extractName(failureMode)
6  endPlaceName ← failureModeName + " Occurrence"
7  endPlace ← new Place(endPlaceName, 0)
8  ttfPDF ← extractTTF(errorMode)
9  enablingCondition ← extractEnablingCondition(errorMode)
10 transition ← new Transition(startPlace, endPlace, ttfPDF, enablingCondition)
11 add startPlace to STPN
12 add endPlace to STPN
13 add transition to STPN
```

Algorithm 15: `buildFailureToFaultSubModel`

input : propagationPort (PropagationPort instance)

```
1  failureMode ← extractPropagatedFailureMode(propagationPort)
2  failureModeName ← extractName(failureMode)
3  label ← failureModeName + " Occurrence"
4  startPlace ← searchPlaceByName(STPN, label)
5  startTransition ← findOutgoingTransition(STPN, startPlace)
6  if startTransition is null then
7  |   startTransition ← new Transition(startPlace, null, "immediate")
8  |   add notPropagatingTransition to STPN
9  end
10 exogenousFaultMode ← extractExogenousFaultMode(propagationPort)
11 exoFmName ← extractName(exogenousFaultMode)
12 hookTransition ← null
13 routingProbability ← extractRoutingProbability(propagationPort)
14 if routingProbability is 1 then
15 |   hookTransition ← startTransition
16 end
17 else
18 |   routingPlaceName ← exoFmName + " Router"
19 |   routingPlace ← new Place(routingPlaceName, 0)
20 |   propagatingTransition ← new Transition(routingPlace, null, "immediate",
   |     routingProbability)
21 |   notPropagatingTransition ← new Transition(routingPlace, null,
   |     "immediate", (1 − routingProbability))
22 |   add routingPlace to STPN
23 |   add propagatingTransition to STPN
24 |   add notPropagatingTransition to STPN
25 |   hookTransition ← propagatingTransition
26 end
27 endPlaceName ← exoFmName + " Occurrence"
28 endPlace ← new Place(endPlaceName, 0)
29 finalTransition ← new Transition(hookPlace, endPlace, "immediate")
30 add endPlace to STPN
31 add endPlace to output places of hookTransition
```

Appendix B

Publications

Research activity, carried out during the PhD Course, exploited experimentation opportunities offered by several "Research & Development" projects, which I was involved in with research grants, leading to the following publications in international journals and conferences.

International Journals

1. J. Parri, **S. Sampietro**, E. Vicario. "Deploying digital twins in a lambda architecture for industry 4.0", *ERCIM News*, vol. 115, 2018. (Special Issue: Digital Twins)
2. J. Parri, F. Patara, **S. Sampietro**, E. Vicario,. "A framework for Model-Driven Engineering of resilient software-controlled systems", *Springer Computing Journal*, 2020. [DOI: 10.1007/s00607-020-00841-6]
3. L. Carnevali, A. Fantechi, G. Gori, J. Parri, M. Pieralli, **S. Sampietro**. "A heuristic approach for predictive diagnosis of wheels wear based on a low cost track-side equipment", *IF-Ingegneria Ferroviaria, Collegio Ingegneri Ferroviari Italiani*, 2021.

International Conferences and Workshops

1. J. Parri, F. Patara, **S. Sampietro**, E. Vicario. "JARVIS, A Hardware/Software Framework for Resilient Industry 4.0 Systems", in *Proc. of XI International Workshop on Software Engineering for Resilient Systems (SERENE)*, Naples (Italy), 2019. [DOI: 10.1007/978-3-030-30856-8_6]

Bibliography

[1] W. Ahmed, O. Hasan, and S. Tahar, "Formalization of reliability block diagrams in higher-order logic," *Journal of Applied Logic*, vol. 18, pp. 19–41, 2016.

[2] 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 (TOCS)*, vol. 2, no. 2, pp. 93–122, 1984.

[3] E. G. Amparore, G. Balbo, M. Beccuti, S. Donatelli, and G. Franceschinis, "30 years of greatspn," in *Principles of Performance and Reliability Modeling and Evaluation*. Springer, 2016, pp. 227–254.

[4] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. Mcminn, A. Bertolino *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.

[5] T. Arendt, E. Biermann, S. Jurack, C. Krause, and G. Taentzer, "Henshin: advanced concepts and tools for in-place emf model transformations," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2010, pp. 121–135.

[6] C. Atkinson, C. Bunse, and J. Bayer, *Component-based product line engineering with UML*. Pearson Education, 2002.

[7] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004.

[8] K. Beck, *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[9] ——, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[10] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski, "A survey of variability modeling in industrial practice," in *Proceedings of the Seventh International Workshop on Variability Modelling of Software-intensive Systems*, 2013, pp. 1–8.

[11] S. Bernardi, J. Merseguer, and D. C. Petriu, "Dependability modeling and analysis of software systems specified with uml," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, pp. 1–48, 2012.

[12] B. Berthomieu and M. Diaz, "Modeling and Verification of Time Dependent Systems Using Time Petri Nets," *IEEE Trans. on Soft. Eng.*, vol. 17, no. 3, pp. 259–273, March 1991.

[13] J. Boardman and B. Sauser, "System of systems-the meaning of of," in *2006 IEEE/SMC International Conference on System of Systems Engineering.* IEEE, 2006, pp. 6–pp.

[14] A. Bondavalli, S. Chiaradonna, F. Di Giandomenico, and I. Mura, "Dependability modeling and evaluation of multiple-phased systems using deem," *IEEE Transactions on Reliability*, vol. 53, no. 4, pp. 509–522, 2004.

[15] A. Bondavalli, I. Mura, S. Chiaradonna, R. Filippini, S. Poli, and F. Sandrini, "Deem: a tool for the dependability modeling and evaluation of multiple phased systems," in *Proceeding International Conference on Dependable Systems and Networks. DSN 2000.* IEEE, 2000, pp. 231–236.

[16] V. Bonfiglio, L. Montecchi, F. Rossi, P. Lollini, A. Pataricza, and A. Bondavalli, "Executable models to support automated software FMEA," in *2015 IEEE 16th International Symposium on High Assurance Systems Engineering.* IEEE, 2015, pp. 189–196.

[17] G. Booch, *The unified modeling language user guide.* Pearson Education India, 2005.

[18] M. Brambilla, J. Cabot, and M. Wimmer, "Model-driven software engineering in practice," *Synthesis lectures on software engineering*, vol. 3, no. 1, pp. 1–207, 2017.

[19] C. S. Carlson, "Understanding and applying the fundamentals of fmeas," in *Annual Reliability and Maintainability Symposium*, vol. 10, 2014, pp. 1–35.

[20] M. Catelani, L. Ciani, L. Cristaldi, M. Faifer, M. Lazzaroni, and M. Khalil, "Toward a new definition of FMECA approach," in *2015 IEEE International Instrumentation and Measurement Technology Conference (I2MTC) Proceedings.* IEEE, 2015, pp. 981–986.

[21] H. Choi, V. G. Kulkarni, and K. S. Trivedi, "Markov regenerative stochastic petri nets," *Performance evaluation*, vol. 20, no. 1-3, pp. 337–357, 1994.

[22] G. Ciardo and A. S. Miner, "Smart: The stochastic model checking analyzer for reliability and timing," in *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.* IEEE, 2004, pp. 338–339.

[23] A. Cicchetti, D. Di Ruscio, R. Eramo, and A. Pierantonio, "Automating co-evolution in model-driven engineering," in *2008 12th International IEEE Enterprise Distributed Object Computing Conference.* IEEE, 2008, pp. 222–231.

[24] A. R. Da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Computer Languages, Systems & Structures*, vol. 43, pp. 139–155, 2015.

[25] D. D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. G. Webster, "The mobius framework and its implementation," *IEEE Transactions on Software Engineering*, vol. 28, no. 10, pp. 956–969, 2002.

[26] D. Di Ruscio, L. Iovino, and A. Pierantonio, "What is needed for managing co-evolution in mde?" in *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, 2011, pp. 30–38.

[27] S. Distefano and A. Puliafito, "Dynamic reliability block diagrams vs dynamic fault trees," in *2007 Annual Reliability and Maintainability Symposium.* IEEE, 2007, pp. 71–76.

[28] ——, "Dependability evaluation with dynamic reliability block diagrams and dynamic fault trees," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 1, pp. 4–17, 2009.

[29] S. Distefano and L. Xing, "A new approach to modeling the system reliability: dynamic reliability block diagrams," in *RAMS'06. Annual Reliability and Maintainability Symposium, 2006.* IEEE, 2006, pp. 189–195.

[30] J. B. Dugan, S. J. Bavuso, and M. A. Boyd, "Dynamic fault-tree models for fault-tolerant computer systems," *IEEE Transactions on reliability*, vol. 41, no. 3, pp. 363–377, 1992.

[31] J. A. Estefan *et al.*, "Survey of model-based systems engineering (mbse) methodologies," *Incose MBSE Focus Group*, vol. 25, no. 8, pp. 1–12, 2007.

[32] M. Faugere, T. Bourbeau, R. De Simone, and S. Gerard, "Marte: Also an uml profile for modeling aadl applications," in *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007).* IEEE, 2007, pp. 359–364.

[33] P. Feiler, "The open source aadl tool environment (osate)," Carnegie Mellon University Software Engineering Institute Pittsburgh United, Tech. Rep., 2019.

[34] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The architecture analysis & design language (aadl): An introduction," Carnegie-Mellon Univ Pittsburgh PA Software Engineering Inst, Tech. Rep., 2006.

[35] P. Fenelon and J. A. McDermid, "An integrated tool set for software safety analysis," *Journal of Systems and Software*, vol. 21, no. 3, pp. 279–290, 1993.

[36] P. Fenelon, J. A. McDermid, M. Nicolson, and D. J. Pumfrey, "Towards integrated safety analysis and design," *ACM SIGAPP Applied Computing Review*, vol. 2, no. 1, pp. 21–32, 1994.

[37] L. Fuentes-Fernández and A. Vallecillo-Moreno, "An introduction to uml profiles," *UML and Model Engineering*, vol. 2, no. 6-13, p. 72, 2004.

[38] B. Gallina, M. A. Javed, F. U. Muram, and S. Punnekkat, "A model-driven dependability analysis method for component-based architectures," in *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. IEEE, 2012, pp. 233–240.

[39] B. Gallina and S. Punnekkat, "Fi4fa: A formalism for incompletion, inconsistency, interference and impermanence failures' analysis," in *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2011, pp. 493–500.

[40] S. Gérard and B. Selic, "The uml–marte standardized profile," *IFAC Proceedings Volumes*, vol. 41, no. 2, pp. 6909–6913, 2008.

[41] R. German, *Performance analysis of communication systems with non-Markovian stochastic Petri nets*. John Wiley & Sons, Inc., 2000.

[42] S. Getir, L. Grunske, C. K. Bernasko, V. Käfer, T. Sanwald, and M. Tichy, "Cowolf–a generic framework for multi-view co-evolution and evaluation of models," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2015, pp. 34–40.

[43] S. Getir, L. Grunske, A. van Hoorn, T. Kehrer, Y. Noller, and M. Tichy, "Supporting semi-automatic co-evolution of architecture and fault tree models," *Journal of Systems and Software*, vol. 142, pp. 115–135, 2018.

[44] S. Getir, A. Van Hoorn, L. Grunske, and M. Tichy, "Co-evolution of software architecture and fault tree models: An explorative case study on a pick and place factory automation system." *NiM-ALP@ MoDELS*, vol. 13, pp. 32–40, 2013.

[45] D. Giglio, "Definitions and applications of deterministic-timed petri nets (dtpn)," in *2006 IEEE International Conference on Systems, Man and Cybernetics*, vol. 4. IEEE, 2006, pp. 3060–3067.

[46] V. Grassi, R. Mirandola, and A. Sabetta, "Filling the gap between design and performance/reliability models of component-based systems: A model-driven approach," *Journal of Systems and Software*, vol. 80, no. 4, pp. 528–558, 2007.

[47] L. Grunske and B. Kaiser, "Automatic generation of analyzable failure propagation models from component-level failure annotations," in *Fifth International Conference on Quality Software (QSIC'05)*. IEEE, 2005, pp. 117–123.

[48] M. Hause *et al.*, "The sysml modelling language," in *Fifteenth European Systems Engineering Conference*, vol. 9, 2006, pp. 1–12.

[49] R. Hebig, D. E. Khelladi, and R. Bendraou, "Approaches to co-evolution of metamodels and models: A survey," *IEEE Transactions on Software Engineering*, vol. 43, no. 5, pp. 396–414, 2016.

[50] M. Herrmannsdoerfer, S. Benz, and E. Juergens, "Cope-automating coupled evolution of metamodels and models," in *European Conference on Object-Oriented Programming*. Springer, 2009, pp. 52–76.

[51] L. Herscheid and P. Tröger, "Specification of dynamic fault tree concepts with stochastic petri nets," in *2014 Eighth International Conference on Software Security and Reliability (SERE)*. IEEE, 2014, pp. 177–186.

[52] L.-M. Hillah, F. Kordon, L. Petrucci, and N. Treves, "Pnml framework: an extendable reference implementation of the petri net markup language," in *International Conference on Applications and Theory of Petri Nets*. Springer, 2010, pp. 318–327.

[53] G. Holl, P. Grünbacher, and R. Rabiser, "A systematic review and an expert survey on capabilities supporting multi product lines," *Information and Software Technology*, vol. 54, no. 8, pp. 828–852, 2012.

[54] J. H. Holland, *Emergence: From chaos to order*. OUP Oxford, 2000.

[55] A. Horváth, M. Paolieri, L. Ridi, and E. Vicario, "Transient analysis of non-markovian models using stochastic state classes," *Performance Evaluation*, vol. 69, no. 7-8, pp. 315–335, 2012.

[56] A. Horváth and M. Telek, "Phfit: A general phase-type fitting tool," in *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*. Springer, 2002, pp. 82–91.

[57] International Electrotechnical Commission (IEC), "Dependability standards and supporting standards," 2019. [Online]. Available: https://tc56.iec.ch/dependability-standards/

[58] ISO/IEC, "std-42010: Systems and software engineering, architectural description," July, 2007.

[59] A. Joshi, S. Vestal, and P. Binns, "Automatic generation of static fault trees from aadl models," in *DSN Workshop on Architecting Dependable Systems*, vol. 10. Springer Berlin, 2007.

[60] S. Kabir, "An overview of fault tree analysis and its application in model based dependability analysis," *Expert Systems with Applications*, vol. 77, pp. 114–135, 2017.

[61] B. Kaiser, P. Liggesmeyer, and O. Mäckel, "A new component concept for fault trees," in *Proceedings of the 8th Australian workshop on Safety critical systems and software-Volume 33*. Citeseer, 2003, pp. 37–46.

[62] J. P. Kelly, A. Avižienis, B. T. Ulery, B. Swain, R.-T. Lyu, A. Tai, and K.-S. Tso, "Multi-version software development," *IFAC Proceedings Volumes*, vol. 19, no. 11, pp. 43–49, 1986.

[63] R. Khan, S. U. Khan, R. Zaheer, and S. Khan, "Future internet: the internet of things architecture, possible applications and key challenges," in *2012 10th international conference on frontiers of information technology*. IEEE, 2012, pp. 257–260.

[64] H. Kopetz, O. Höftberger, B. Frömel, F. Brancati, and A. Bondavalli, "Towards an understanding of emergence in systems-of-systems," in *2015 10th System of Systems Engineering Conference (SoSE)*. IEEE, 2015, pp. 214–219.

[65] C. Krause, J. Dyck, and H. Giese, "Metamodel-specific coupled evolution based on dynamically typed graph transformations," in *International Conference on Theory and Practice of Model Transformations*. Springer, 2013, pp. 76–91.

[66] A. J. Krygiel, "Behind the wizard's curtain. an integration environment for a system of systems," 1999.

[67] E. A. Lee, "Cyber physical systems: Design challenges," in *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing (ISORC)*. IEEE, 2008, pp. 363–369.

[68] G. Liebel, N. Marko, M. Tichy, A. Leitner, and J. Hansson, "Assessing the state-of-practice of model-based engineering in the embedded systems domain," in *International conference on model driven engineering languages and systems*. Springer, 2014, pp. 166–182.

[69] ——, "Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice," *Software & Systems Modeling*, vol. 17, no. 1, pp. 91–113, 2018.

[70] P. Liggesmeyer and M. Rothfelder, "Improving system reliability with automatic fault tree generation," in *Digest of Papers. Twenty-Eighth Annual In-*

ternational Symposium on Fault-Tolerant Computing (Cat. No. 98CB36224). IEEE, 1998, pp. 90–99.

[71] M. W. Maier, "Architecting principles for systems-of-systems," *Systems Engineering: The Journal of the International Council on Systems Engineering*, vol. 1, no. 4, pp. 267–284, 1998.

[72] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis, "Modelling with generalized stochastic petri nets," *ACM SIGMETRICS performance evaluation review*, vol. 26, no. 2, p. 2, 1998.

[73] N. Medvidovic, A. Egyed, and D. S. Rosenblum, "Round-trip software engineering using uml: From architecture to design and back," in *Proceedings of the Second International Workshop on Object-Oriented Reengineering (WOORâ99), Toulouse, France*, 1999, pp. 1–8.

[74] F. Mhenni, N. Nguyen, and J.-Y. Choley, "Automatic fault tree generation from sysml system models," in *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*. IEEE, 2014, pp. 715–720.

[75] L. Montecchi and B. Gallina, "Safeconcert: A metamodel for a concerted safety modeling of socio-technical systems," in *International Symposium on Model-Based Safety and Assessment*. Springer, 2017, pp. 129–144.

[76] L. Montecchi, P. Lollini, and A. Bondavalli, "Dependability concerns in model-driven engineering," in *2011 14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*. IEEE, 2011, pp. 254–263.

[77] C. Z. Mooney, *Monte carlo simulation*. Sage publications, 1997, vol. 116.

[78] M. Paolieri, M. Biagi, L. Carnevali, and E. Vicario, "The oris tool: quantitative evaluation of non-markovian systems," *IEEE Transactions on Software Engineering*, 2019.

[79] Y. Papadopoulos and J. A. McDermid, "Hierarchically performed hazard origin and propagation studies," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 1999, pp. 139–152.

[80] J.-P. Penttinen, A. Niemi, J. Gutleber, K. T. Koskinen, E. Coatanea, and J. Laitinen, "An open modelling approach for availability and reliability of systems," *Reliability Engineering & System Safety*, vol. 183, pp. 387–399, 2019.

[81] J. L. Peterson, "Petri nets," *ACM Computing Surveys (CSUR)*, vol. 9, no. 3, pp. 223–252, 1977.

[82] K. Pohl, G. Böckle, and F. J. van Der Linden, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.

[83] A. Puliafito, M. Telek, and K. S. Trivedi, "The evolution of stochastic petri nets," 1997.

[84] D. Roca, D. Nemirovsky, M. Nemirovsky, R. Milito, and M. Valero, "Emergent behaviors in the internet of things: the ultimate ultra-large-scale system," *IEEE micro*, vol. 36, no. 6, pp. 36–44, 2016.

[85] A.-E. Rugina, K. Kanoun, and M. Kaâniche, "The adapt tool: From aadl architectural models to stochastic petri nets through model transformation," in *2008 Seventh European Dependable Computing Conference*. IEEE, 2008, pp. 85–90.

[86] E. Ruijters and M. Stoelinga, "Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools," *Computer science review*, vol. 15, pp. 29–62, 2015.

[87] S. Russo, "Finding a way in the model driven jungle: Invited keynote talk," in *Proceedings of the 9th India Software Engineering Conference*, 2016, pp. 13–15.

[88] A. P. Sage and C. D. Cuppan, "On the systems engineering and management of systems of systems and federations of systems," *Information knowledge systems management*, vol. 2, no. 4, pp. 325–345, 2001.

[89] D. C. Schmidt, "Model-driven engineering," *Computer-IEEE Computer Society*, vol. 39, no. 2, p. 25, 2006.

[90] F. Scippacercola, R. Pietrantuono, S. Russo, A. Esper, and N. Silva, "Integrating fmea in a model-driven methodology," in *International Space System Engineering Conference (DASIA)*, 2016.

[91] B. Selic, "The pragmatics of model-driven development," *IEEE software*, vol. 20, no. 5, pp. 19–25, 2003.

[92] S. Sharvia, S. Kabir, M. Walker, and Y. Papadopoulos, "Model-based dependability analysis: State-of-the-art, challenges, and future outlook," in *Software quality assurance*. Elsevier, 2016, pp. 251–278.

[93] F. Shrouf, J. Ordieres, and G. Miragliotta, "Smart factories in industry 4.0: A review of the concept and of energy management approached in production based on the internet of things paradigm," in *2014 IEEE international conference on industrial engineering and engineering management*. IEEE, 2014, pp. 697–701.

[94] B. Silva, G. Callou, E. Tavares, P. Maciel, J. Figueiredo, E. Sousa, C. Araujo, F. Magnani, and F. Neves, "Astro: An integrated environment for dependability and sustainability evaluation," *Sustainable computing: informatics and systems*, vol. 3, no. 1, pp. 1–17, 2013.

[95] B. Silva, R. Matos, G. Callou, J. Figueiredo, D. Oliveira, J. Ferreira, J. Dantas, A. Lobo, V. Alves, and P. Maciel, "Mercury: An integrated environment for performance and dependability evaluation of general systems," in *Proceedings of Industrial Track at 45th Dependable Systems and Networks Conference, DSN*, 2015.

[96] R. Soley *et al.*, "Model driven architecture," *OMG white paper*, vol. 308, no. 308, p. 5, 2000.

[97] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.

[98] W. J. Stewart, *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994.

[99] D. Strüber, K. Born, K. D. Gill, R. Groner, T. Kehrer, M. Ohrndorf, and M. Tichy, "Henshin: A usability-focused framework for emf model transformation development," in *International Conference on Graph Transformation*. Springer, 2017, pp. 196–208.

[100] G. Suciu, A. Vulpe, S. Halunga, O. Fratu, G. Todoran, and V. Suciu, "Smart cities built on resilient cloud computing and secure internet of things," in *2013 19th international conference on control systems and computer science*. IEEE, 2013, pp. 513–518.

[101] Y. Thierry-Mieg, "Symbolic model-checking using its-tools," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2015, pp. 231–237.

[102] S. Tomforde, J. Hähner, H. Seebach, W. Reif, B. Sick, A. Wacker, and I. Scholtes, "Engineering and mastering interwoven systems," in *ARCS 2014; 2014 Workshop Proceedings on Architecture of Computing Systems*. VDE, 2014, pp. 1–8.

[103] K. S. Trivedi and R. Sahner, "Sharpe at the age of twenty two," *ACM SIGMETRICS Performance Evaluation Review*, vol. 36, no. 4, pp. 52–57, 2009.

[104] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Elsevier, 2010.

[105] K. Vaidyanathan and K. S. Trivedi, "A comprehensive model for software rejuvenation," *IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 2, pp. 124–137, 2005.

[106] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, "Fault tree handbook," Nuclear Regulatory Commission Washington DC, Tech. Rep., 1981.

[107] E. Vicario, "Static analysis and dynamic steering of time-dependent systems," *IEEE transactions on software engineering*, vol. 27, no. 8, pp. 728–748, 2001.

[108] E. Vicario, L. Sassoli, and L. Carnevali, "Using stochastic state classes in quantitative evaluation of dense-time reactive systems," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 703–719, 2009.

[109] ——, "Using stochastic state classes in quantitative evaluation of dense-time reactive systems," *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 703–719, 2009.

[110] G. Wachsmuth, "Metamodel adaptation and model co-adaptation," in *European Conference on Object-Oriented Programming*. Springer, 2007, pp. 600–624.

[111] M. Walker, L. Bottaci, and Y. Papadopoulos, "Compositional temporal fault tree analysis," in *International Conference on Computer Safety, Reliability, and Security*. Springer, 2007, pp. 106–119.

[112] W. Wang, J. Loman, and P. Vassiliou, "Reliability importance of components in a complex system," in *Annual Symposium Reliability and Maintainability, 2004-RAMS*. IEEE, 2004, pp. 6–11.

[113] E. R. Weippl and B. Sanderse, "Digital twins - Introduction to the special theme." *ERCIM News*, vol. 2018, no. 115, 2018.

[114] J. Whittle, J. Hutchinson, and M. Rouncefield, "The state of practice in model-driven engineering," *IEEE software*, vol. 31, no. 3, pp. 79–85, 2013.

[115] J. Xiang, K. Yanoo, Y. Maeno, and K. Tadano, "Automatic synthesis of static fault trees from system models," in *2011 Fifth International Conference on Secure Software Integration and Reliability Improvement*. IEEE, 2011, pp. 127–136.

[116] L. Xing and S. V. Amari, "Fault tree analysis," in *Handbook of performability engineering*. Springer, 2008, pp. 595–620.

[117] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi, "Internet of things for smart cities," *IEEE Internet of Things journal*, vol. 1, no. 1, pp. 22–32, 2014.

[118] A. Zimmermann, "Modelling and performance evaluation with timenet 4.4," in *International Conference on Quantitative Evaluation of Systems*. Springer, 2017, pp. 300–303.

[119] A. Zimmermann, J. Freiheit, R. German, and G. Hommel, "Petri net modelling and performability evaluation with timenet 3.0," in *International conference on modelling techniques and tools for computer performance evaluation*. Springer, 2000, pp. 188–202.