Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

A framework for the Dependability analysis of UML-based system designs with maintenance

Diplomaterv

Készítette
Hegedüs Ábel

Konzulens
Dr. Varró Dániel
Prof. Andrea Bondavalli
Gönczy László
Dr. Paolo Lollini

May 20, 2009

Budapesti Műszaki és Gazdaságtudományi Egyetem
Méréstechnika és Információs Rendszerek Tanszék

DIPLOMATERV FELADAT (ezt adják...)

Hegedüs Ábel
szigorló villamosmérnök hallgató részére
(nappali tagozat villamosmérnöki szak)

A framework for the Dependability analysis of UML-based system designs with maintenance
(a feladat szövege a mellékletben)

A tervfeladatot összeállította és a tervfeladat tanszéki konzulense:

Vég Béla
adjunktus

A záróvizsga tárgyai: Első tárgy
Második tárgy
Harmadik tárgy

A tervfeladat kiadásának napja:
A tervfeladat beadásának határideje:

dr. Diplomatervfelelős András
adjunktus, diplomaterv felelős

dr. Tanszékvezető Gábor
egyetemi tanár, tanszékvezető

A tervet bevette:
A terv beadásának dátuma:
A terv bírálója:

MELLÉKLET

A framework for the Dependability analysis of UML-based system designs with maintenance

A feladatkiírást a tanszéki adminisztrációban lehet átvenni, és a leadott munkába eredeti, tanszéki pecséttel ellátott és a tanszékvezető által aláírt lapot kell belefűzni (ezen oldal *helyett*, ez az oldal csak útmutatás. Fénymásolat nem jó, ezért mindenki igényeljen megfelelő számú eredeti iratot az adminisztrációban).

Dr. Varró Dániel
adjunktus

HALLGATÓI NYILATKOZAT

Alulírott *Hegedüs Ábel*, a Budapesti Műszaki és Gazdaságtudományi Egyetem hallgatója kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, és a diplomatervben csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Tudomásul veszem, hogy az elkészült diplomatervben található eredményeket a Budapesti Műszaki és Gazdaságtudományi Egyetem, a feladatot kiíró egyetemi intézmény saját céljaira felhasználhatja.

Budapest, May 20, 2009

Hegedüs Ábel
hallgató

Contents

Kivonat

Dolgozatomban egy olyan keretrendszert alakítottam ki, amely képes annotált UML-alapú rendszerek megbízhatósági analízis végrehajtásához használható modelljeinek automatikus létrehozására. A módszer figyelembe veszi a rendszerkomponensek egyéni megbízhatósági paramétereit továbbá a rendszerben definiált karbantartási stratégiákat és tevékenységeket.

Napjaink összetett rendszerei olyan komplexitással rendelkeznek, hogy a fejlesztőknek szükségük van olyan eszközökre, amelyek lehetővé teszik ezen rendszerek magas szintű funkcionális leírását és a megfelelő viselkedés specifikálását. Fontos tényező továbbá ezen rendszerek tervezésekor a végleges termék megbízhatósági tulajdonságainak ismerete. Azonban a megbízhatósági tulajdonságok számításához precíz matematikai (megbízhatósági) modellekre van szükség amelyeken mennyiségi analízis végezhető.

Tekintve, hogy ezeket a számításokat tervezési időben kell elvégezni, a megbízhatósági modelleket a rendszer specifikációja alapján kell létrehozni. A modellek létrehozásához azonban olyan szakemberre van szükség, akinek megfelelő ismeretei vannak mind a specifikációs, mind a matematikai modellek formalizmusából. Nagyméretű rendszerek esetén azonban még ezen ismeretek birtokában is fáradságos feladat a megbízhatósági modell létrehozása, továbbá sok a hibalehetőség. Ezért szükség van olyan módszerekre amelyek képesek automatikusan létrehozni a megbízhatósági modellt a specifikációs modellből, ezzel áthidalva a két modell közti szakadékot.

A dolgozatban definiált keretrendszer egy olyan új UML profil használatát javasolja, amely a beágyazott és valós-idejű rendszerek tervezéséhez segítséget adó, széleskörben használt MARTE profilt terjeszti ki a rendszer karbantartásának és monitorozásának leírására alkalmas elemekkel. Továbbá a dolgozatban definiálok olyan karakterisztikákat, amelyekkel a szolgáltatás-orientált rendszerek tervezéséhez használható UML4SOA profil alapján készített nem-funkcionális szolgáltatási szerződéseket kiegészítve lehetővé válik ezen rendszerek megbízhatósági analízise.

A keretrendszerben definiált automatikus modellgenerálás lépéseit modelltranszformációk segítségével lehet megvalósítani. A transzformációk a VIATRA Eclipse-alapú modelltranszformációs keretrendszerben kerülnek megvalósításra a VIATRA metamodellezésre és transzformáció definiálásra használható nyelveiben MDA alapon. Mind a VIATRA használata, mind az MDA alapú megvalósítás garanciát ad a módszer jövőbeli kiterjeszthetőségére és integrálhatóságára.

Abstract

In this document a novel framework is defined which can automatically generate models for dependability analysis of annotated UML-based systems. The method is capable of dealing with the dependability properties of the system component along with the maintenance policies and activities defined for the system. Developers of complex systems today use modeling languages like UML to specify, document and visualize the requirements, functionality and behavior of their product. Often extension or profiles are used to grasp the characteristics of domain-specific systems. Furthermore the non-functional properties such as availability or fault-tolerance are important especially in embedded and real-time systems hence the quantitative evaluation of these properties are required at design-time. However evaluation can only be carried out on precise mathematical models the creation of which is not trivial and needs a modeling expert with insight to both the developed system or its specification language and the mathematical formalism used for the dependability models.

In order to relieve the developer from the tiresome and error-prone task of model creation new methods have to be created to bridge the huge gap between the specification and dependability models. The method defined in this document provides automatic dependability model generation through the usage of a novel UML profile. This profile extends the industry standard MARTE profile which is widely used for the development of embedded and real-time systems with the concepts of maintenance and monitoring. Additionally the Service-Oriented Profile is extended by defining new characteristics for the non-functional service-contracts and thus the method provides support for the dependability evaluation of systems with service-oriented architecture.

The defined method is implemented in the Eclipse-based VIATRA model transformation framework which provides tools for creating the metamodels and transformation definitions required for the automatic model generation from the annotated UML models. The method was created according to the Model-Driven Architecture (MDA) paradigm and involves an intermediate model that acts as a transition between the specification and dependability models. Both the use of the VIATRA framework and the embracing of the MDA paradigm assures the possibility of future extensions.

Introduction

During the dawn of information technology the first applications created for the earliest computers were small and simple due to the limited resources such as memory, storage and processing power. As computers evolved, becoming more and more powerful, programming languages were created to overcome the challenges of developing larger applications. During the decades the computing power (or more specifically the number of transistors in a processor) increased according to the famous Moore's Law. Simultaneously new languages have appeared which strived to allow the description of applications on higher abstraction levels.

In today's information society applications reached a complexity level which can not be modeled using regular programming languages. In order to support developers in specifying, documenting and constructing the artifacts of a complex product modeling languages emerged. The *Unified Modeling Language (UML)* [51] is an industry standard modeling language of the Object Management Group (OMG). UML is widely used in software engineering for modeling systems in development partly because its visualization capabilities and support for creating abstract models. Additionally UML is created with the purpose to provide a standard way for extending the core language for domain-specific needs.

Although UML can be used to describe the many aspects of an application, the system models created using it only describe the structure and behavior of the system components. While this information is sufficient for developing the application itself but it excludes important aspects which are essential during the operation of the completed product. The maintenance and monitoring aspects are often left out of the system model and are dealt with once the product is ready. It is important to understand that these aspects have such impact on the properties of the system that their inclusion is necessary for the correct evaluation of the system model.

The developers of complex business applications quickly embraced the *Service-Oriented Architecture (SOA)* paradigm [19] that introduces an infrastructure of loosely coupled components which provide functionality through interoperable services. However to fully exploit the full potential of these concepts modelers need to be able to describe the characteristics of these systems in the UML language. Hence the *Service-Oriented Profile (UML4SOA)* [33] was created to extend the UML language with the notion of SOA.

Apart from the increasing complexity of applications the estimation of non-functional properties such as dependability and performance became a challenging task for the developers especially for embedded, fault-tolerant and high-availability systems. Low abstraction level mathematical languages were invented to provide precise modeling capabilities and to be the basis of qualitative and quantitative analysis.

Real-time and embedded systems represent a special domain where reliability, fault-tolerance and timing properties are of utmost importance and these aspects have to be considered during the specification and development. The *Modeling and Analysis of Real-time and Embedded systems (MARTE)* profile [38] extends the core UML language with the elements required to express these systems.

It is important to note that the non-functional properties required from safety-critical services and high-availability embedded or real-time systems are very similar hence MARTE is often used for modeling service-oriented systems as well along with the Service-Oriented Profile when applied. Furthermore the designing of embedded and mission-critical systems (e.g. aerospace applications) are converging towards a service-based paradigm as well.

Motivation

Non-functional properties of critical systems carry such importance that they have to be calculated in design-time in order to assure the proper characteristics for the developed system. Mathematical models can be used for evaluating these properties but only a modeling expert familiar with both the developed system model and the mathematical language is able to create the required models by hand.

In order to free system modelers from the need to have a deep knowledge of mathematical languages and to provide support for the quantitative dependability analysis of UML models these mathematical models should be created automatically instead of by hand.

However it is not sufficient to generate a dependability model from the UML models without assuring that the created model is a sound approximation of the concrete system. The results of the dependability evaluation performed on models created from known systems must resemble the dependability properties of these systems and also be close to the results acquired from the analysis of hand-made models created by modeling experts.

Furthermore to ensure the correct evaluation of a system designed in UML the state-of-the-art profiles (MARTE and UML4SOA) have to be extended with the ability to model maintenance and monitoring aspects of these systems.

Problem

The UML models are high-level abstract models using numerous diagrams with explicit and implicit, semantic and syntactic connections between elements while dependability models are low-level mathematical models with a limited building blocks (i.e. usable elements). Since the creation of dependability models is a difficult task even for a given system and hand-made by modeling experts novel methods are required to bridge the gap between the system and dependability models. Automatic methods should be used to create dependability models from a more abstract, high-level description without the participation of the system designer. Although the creation of the dependability model requires additional information about the system, this information should be represented in the system modeling language that the designer is already familiar with.

Innovation

The main topic of this document is the definition of a framework which can be used for the automated generation of dependability models from annotated UML diagrams. The method possesses the following important features:

- **UML diagrams which are annotated with the stereotypes defined in a new profile are used.** This profile extends the MARTE profile with maintenance and monitoring capabilities. By defining these elements the information required for the dependability evaluation can be represented in UML.

- **Non-functional service contracts are extended with dependability and maintenance characteristics.** This extension provides the designers of SOA applications with the ability to evaluate their system with the inclusion of the properties of external or internal services.

- **Intermediate model is defined as a transition between the UML and dependability models.** The intermediate model includes elements to represent the maintenance and failure characteristics of the system. The fault-tolerance is represented using fault-trees.

- **The transformation steps are defined both from UML to the intermediate model and from that to the dependability model.** The steps are described in great detail to include every construct of the models.

- **A novel algorithm is defined for creating a perfectly staggered maintenance schedule.** The algorithm is able to synchronize the many maintenance policies defined for the system components.

- **The framework for automatic execution is described along with the implementation steps.** The metamodels and transformations required for the method are specified.

Approach

The novel method defined in this document is created using the Model-driven Architecture paradigm. The UML profile defining the stereotypes used for annotating the system models is created in compliance with the *Eclipse Modeling Framework (EMF)* [12] and thus it is modeling tool independent as long as the tool complies with EMF. The UML models created in an appropriate modeling tool are imported in the modelspace of the VIATRA2 model transformation framework. The VIATRA2 framework provides metamodeling and graph transformation capabilities which are used to create the metamodels for the intermediate and dependability models and the transformations responsible for creating these models from the UML models.

The imported UML model is first transformed to an intermediate model which works as general model representing every information about the system that is required for the dependability evaluation. Furthermore a reference model is created to store the connection between the elements in the UML model and the intermediate model.

Next the intermediate model is transformed into the chosen dependability model (Multiple-Phased Systems are used in the document) and an other reference model is created to store the relation between the intermediate model elements and the dependability model elements. Finally the dependability model is exported in the input format of the analysis tool using a code generation transformation.

Structure of the thesis

The document is structured as follows:

- The related technologies and theory are introduced in Chapter 1 along with the description of one of the case studies used as a running example throughout the document.
- In Chapter 2 the related literature, methods and techniques are introduced.
- Next the new UML profile defined to include maintenance and monitoring is described in Chapter 3 along with the extension to the non-functional service-contracts of the UML4SOA profile.
- The definition of the intermediate model and the transformation steps generating the intermediate model from the UML models are defined in Chapter 4
- The transformation steps generating the dependability model from the intermediate model and the algorithm for creating the maintenance schedule are defined in Chapter 5.
- The application of the method is illustrated on the two case studies together with possible dependability properties defined for evaluation in Chapter 7.
- Finally the conclusion of the document and some possible future extensions are discussed in Chapter 8.

Chapter 1

Preliminaries

The method defined in this document is built on numerous standards, methods and theoretical topics which are introduced in this chapter to ease the understanding of the details of the method and the basic ontology used throughout the document. First the topic of system lifecycle is introduced briefly in Section 1.1 along with the possible malicious events (i.e. faults and failures) and the techniques used to detect and correct them. Next the paradigm of modeling is described along with the introduction of the Unified Modeling Language which is a modeling standard in Section 1.2. After that the properties used for describing the dependability of a system are introduced in Section 1.3 which is followed by Section 1.4 containing the introduction to the mathematical models capable of capturing the characteristics of systems and evaluating their dependability properties. Finally a Case Study is described in Section 1.5 which will be used as a running example throughout the document.

1.1 System lifecycle

The lifecycle of a system includes all phases of its existence from design and development, through production, operation and maintenance to phase-out or disposal. When engineering a system it is important to plan for the phases of the lifecycle after implementation and production is finished and the system starts its operation. In order to prepare for the support and maintenance of the system the possible faults, errors and failures have to be considered, the strategies dealing with their occurrence have to be defined and dedicated monitoring systems have to be planned for detecting the events indicating the state changes in the system.

1.1.1 Faults, errors and failures

A *failure* of a system or its part is a state or condition of not meeting a specified objective. An element that failed can no longer operate correctly or generate correct responses to requests. An *error* is the part of the state or condition of the system that may cause a subsequent failure, the failure occurs when an error reaches the service interface and alters the service. An error is *detected* if its presence in the system is indicated by an error message or error signal that originates within the system. Errors that are present but not detected are *latent* errors. A *fault* is the supposed cause of an error and is called *active* when it produces the error and *dormant* otherwise. The fault can be either random or intentional, permanent or transient and it may originate from a human or physical source. The origin of the fault can also be internal or external and introduced in the system during design or operation. It is also important to note that this classification is a function of the system hierarchy level. A failure in the lower level appears as a fault on the level above [50].

1.1.2 Maintenance

The practice of preventing malfunctioning of a system, performing routine actions which keep the system functioning and correcting any failure or error as it occurs is called *maintenance*. The maintenance *policy* defines when the maintenance is carried out or how often and how long the system remains during maintenance. A *corrective* policy indicates that maintenance on the system or its components is performed after a failure is detected but preferably before it reaches the boundary of the system (i.e. before the system fails). A *preventive* policy is aimed to uncover and remove faults before they might cause errors during normal operation. These latter faults may have occurred since the last preventive maintenance actions, or can be design faults that have led to errors in other similar systems [50]. A maintenance policy is also often referred to as a *strategy*.

A maintenance policy defined for a system contains an arbitrary number of *activities* which specify the actual tasks that are executed on the system or a component when it is under maintenance. An activity may consist of finding and eliminating the fault or error in a component (*repair*), exchanging the failed component to a new one without searching for the error (*replace*) or executing a series of tests and checks to determine the existence of an error and correcting it if found (*overhaul*).

1.1.3 Monitoring

While maintenance can be executed to keep the system running correctly, corrective maintenance can only operate with efficiency if the state of the system and its components are known at any time with high accuracy. The process of dynamic collection, interpretation and representation of information concerning the functional state of the system and its components is called *monitoring* [30]. In order to describe the monitoring of a system, both the dedicated data collector, parser, inspector systems and the mode of the monitoring have to be defined. The data can be gathered on a time or event-driven base and it can be done automatically or with human intervention. The monitoring of a given information or component can be either continuous (for example a measured value is constantly watched) or intermittent (for example a heartbeat signal is sent periodically to a component). The monitoring and maintenance of a system is highly interconnected and both have to be considered when designing the system.

1.2 System modeling

The engineers of large enterprise applications and complex, embedded or safety-critical systems required a process that enabled them to express these systems in a way that enables scalability, security, and robust execution under stressful conditions. Furthermore their structure - frequently referred to as their architecture - must be defined clearly enough that maintenance programmers can find and fix a bug that shows up long after the original authors have moved on to other projects [51]. *Modeling* is the designing of the system or application before production or implementation. In this section UML, the OMG modeling standard of software engineering is introduced with further focus on its extension possibilities called *profiles*.

1.2.1 Unified Modeling Language (UML)

The *Unified Modeling Language* is modeling standard that provides engineers with the expressive power to specify, visualize and documents models of systems including their structure and design. The UML 1.3 standard has been released by OMG in 2001 and saw several updates in the following years [52]. The UML 2.0 standard has been released in 2005 [53] and extended the earlier versions, for the sake of simplicity the abbreviation UML will always refer to the 2.x version from now on. The flexibility of UML allows the modeling of applications including any type of software and hardware elements operating on any operating system, implemented in any programming language and communicating on any network. UML also supports the Model-Driven Architecture (MDA) software engineering paradigm which enables engineers to create composable, cross-platform and middleware independent applications [40].

The UML standard specify a thirteen types of diagrams that can be divided into three categories. However it is important to remember that there is a difference between an UML model and the set of diagrams it contains. While the diagram is only an abstracted graphical representation of the system the model contains deeper semantic information as well such as documentation attached and the implicit connections between the diagrams [6].

The three categories are the following:

- **Structure diagrams** that represent the static application structure including the various software and hardware elements of the system, their attributes and relationships.

- **Behavior diagrams** can be used to model the dynamic behavior of the system and its elements, also representing the different requirements toward the system, the overall flow of control in the system and the internal changes of state of components.

- **Interaction diagrams** which describe the data and control flow between the different elements of the system as well as specifying the timing constraints and the sequence of messages during the lifespan of the collaborating elements.

1.2.2 UML profiles

The UML standard includes a generic extension mechanism called *profile* to customize models for specific platforms or domains such as financial applications or systems in aerospace or healthcare environments. The definition of a profile can contain stereotypes, tagged values and constraints that restrict the model elements they are applied to. It can also identify a subset of the UML metamodel, specify additional semantics expressed in natural language and common model elements expressed in the terms of the profile.

Some standard profile examples include profiles for CORBA, Enterprise Application Integration, Quality of Service and Fault Tolerance Characteristics, Schedulability, Performance, Time and System Engineering [42].

1.2.3 Modeling and Analysis of Real-time and Embedded systems

The *Modeling and Analysis of Real-time and Embedded systems* (MARTE) profile adds capabilities to UML for model-driven development of *real-time and embedded systems* (RTES) and provides support for specification, design, and verification/validation stages. The modeling parts provide support required from specification to detailed design of real-time and embedded characteristics of systems. The model-based analysis is also a concern and MARTE provides facilities to annotate models with information required to perform specific analysis [38].

The advantages of using MARTE are the following:

- It introduces a common modeling style for both the hardware and software aspects of a RTES for improving the communication between developers.
- It enables interoperability between development tools used for specification, design, verification, etc.
- It provides means to construct models that may be used to perform quantitative analysis of systems. Both the hardware and software characteristics and the real-time and embedded features of the system can be taken into account.

Figure 1.1 shows the architecture of the MARTE profile. The two concerns of the profile, the *design* model that provides support for modeling the real-time and embedded systems and the *analysis* model which is used to annotate the models to support verification and validation, both share commons concerns. These are included in the shared MARTE *foundations* package which contains the profiles for describing time and the use of concurrent resources. The *annexes* package contains extension profiles and a predefined library that can be used by engineers to denote their real-time and embedded applications.

1.2.4 Service-Oriented Profile (UML4SOA)

The Service-Oriented Architecture provides methods for the development and integration of software systems whose functionality is provided by *interoperable services*. The use of services introduces an infrastructure of *loosely coupled* components that interact with each

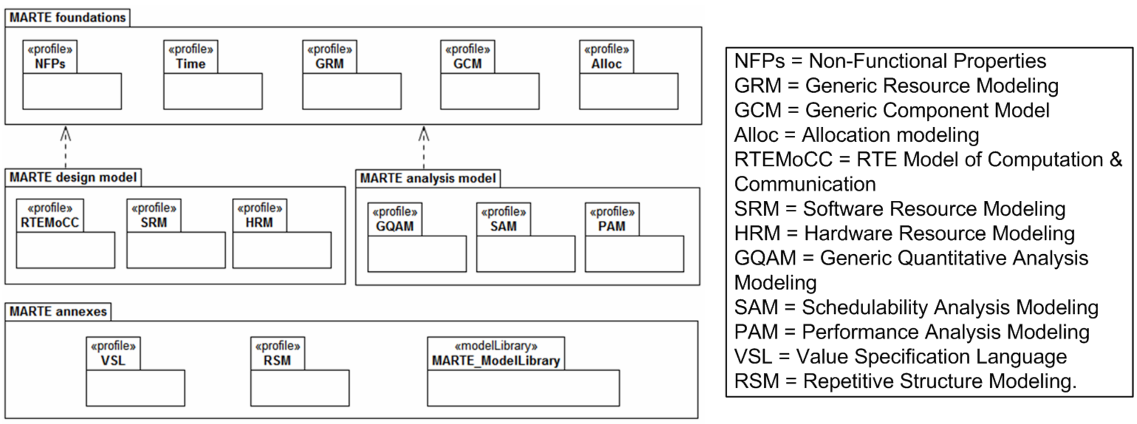

Figure 1.1: *The architecture of the MARTE profile [38]*

other through service operations. In order to provide engineers and modelers with means to exploit the full potential of these concepts, the UML standard had to be extended with specific elements which are geared towards expressing the new concepts on the right level of abstraction [33]. The SENSORIA research project is an IST project funded by the EU with the aim to develop a novel comprehensive approach to the engineering of software systems for service-oriented architectures where foundational theories, techniques and methods are fully integrated in a pragmatic software engineering approach [48].

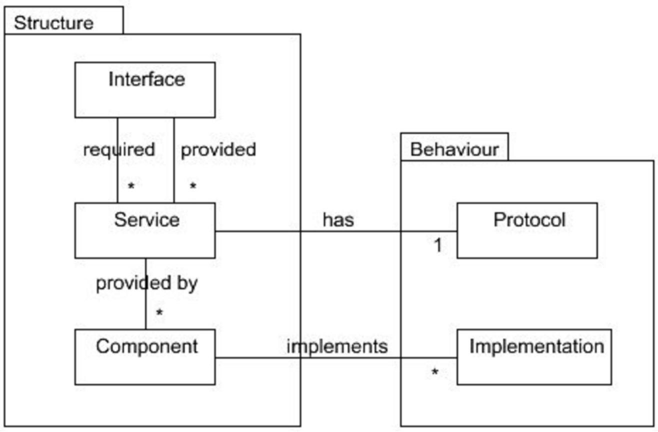

Figure 1.2: *Main concepts in the Service-Oriented Architecture Profile [33]*

The *Service-Oriented Profile* defines a domain-specific language for service-oriented systems. The main concepts introduced by SOAs are shown in Figure 1.2, which illustrates both the overview of the structural and behavioral aspects. The following concepts are introduced for the structural and behavioral view of such systems:

- **Services** are a defined set of functionality which is implemented by the component that provides the service and which is used by components that require the service.

- **Components** are software elements of the system that provide services by implementing them as ports and require services by using their functionality.

- **Interfaces** contain the operations of the services, a *provided* interface contains operations implemented by the service itself while *required* interfaces contain operations that the service uses and which must be implemented by other services.
- **Protocols** which define the behavior of their associated service.
- **Implementations** that define the internal behavior of components they implement.

However it is not enough to cover only the structural and behavioral view of the SOAs as they are used for modeling and implementing complex business applications. Therefore the *business goals*, *policies* and *non-functional properties* of services are also covered. Great emphasis is placed on the composition, or orchestration, of services and the UML4SOA profile provides elements for modeling the complete behavior specification of a composed service.

1.3 Dependability properties of systems

The *dependability* of a system is the collective term that describes the availability performance of a system and its influencing factors: reliability, maintainability and maintenance support performance [28]. These non-functional properties are highly important for both RTES and SOA systems as they are designed to operate in environments where failure to provide functionality or service can have enormous cost both from financial, influential or physical aspects. Therefore it is essential that these properties are calculated as precisely as possible during the design and operation of such systems. The most common properties used are *reliability* and *availability*.

Reliability is the ability of a system or component to provide its required functionality or services under given conditions for a specified period of time [29]. Reliability is often defined as a probability which can be expressed as (1.1), where $f(t)$ is the *failure density function* and t is the length of the period of time. The *cumulative distribution function* of T is represented by $F(t)$.

$$R(t) = Pr\{T > t\} = \int_t^\infty f(x)dx = 1 - F(t), \text{ where } F(t) = \int_{-\infty}^u f(u)du \tag{1.1}$$

It is important to note that reliability is restricted to given conditions and a period of time because no system can be designed for every condition and for infinite time. A system that is above a certain complexity level is certain to fail eventually in given conditions.

Availability is the ratio of total time the a system or a component is functional (ie. provides its services and capable of being used) during a specified period and the length of the period. Availability can represented by defining the status function $F_s(t)$ as (1.2) then availability $A(t)$ and *steady-state* availability A can be expressed by (1.3).

$$F_s(t) = \begin{Bmatrix} \text{1,functions at time of t} \\ \text{0,otherwise} \end{Bmatrix} \tag{1.2}$$

$$A(t) = Pr\{F_s(t) = 1\}, A = \lim_{t\to\infty} A(t) \tag{1.3}$$

The steady-state availability property is often used as the measure for the quality of a service or system. When partners agree on a service contract, the provider of the service agrees to a certain level of availability and a given fee that it will pay if it can not meet the requirement. On the other hand, the requester of the service agrees to pay for the service as long as it functions on the agreed level.

Other properties can be defined such as *maintainability*, *safety*, *integrity* or *survivability*. Maintainability can be specified as the probability that a component or system will be restored to a given condition within a period of time. Safety is described as the absence of serious consequences on the user or environment in case of failure. Integrity can be specified as the absence of improper alterations on the target system or component. Survivability can be defined as the ability of the system to remain functional after a natural or man-made disturbance (ie. disasters or physical damage).

1.4 Dependability modeling

As already mentioned knowing the dependability properties of a system is essential, in many cases before the system is completed. Furthermore the properties of the components used to construct the system are important in order to create a system with a given quality level. Many modeling techniques were developed for describing systems on an abstract, mathematical level where analysis and evaluation can be performed. In this section the paradigm of Petri Nets is introduced along with its different extensions.

1.4.1 Petri Nets (PN)

The *Petri Net* is a widely used mathematical modeling language for the description of discrete distributed systems. A Petri Net consists of *places*, *transitions* and *arcs* that connect places and transitions, have a direction and never run between places or between transitions. The places from which an arc leads to a transition are called *input places* of that transition and such arcs will be referred to as *inbound arcs*. The places to which an arc leads from a transition are called the *output places* of the transition and such arcs will be referred to as *outbound arcs*. Places may contain a non-negative number of tokens and the distribution of tokens over the places is called the *marking* of the net. A transition is *enabled* whenever there is a token at the end of each inbound arc. It may *fire* when it is enabled and by firing it consumes these tokens and places tokens on the end of each outbound arc. Firing is atomic and the order of firing multiple enabled transitions is non-deterministic. This characteristic makes the Petri Net well suited for modeling concurrent behavior. An example of a Petri Net using the traditional graphical notation is shown on Figure 1.3.

According to the formal definition the Petri Net [45] is the 5-tuple $PN = (P, T, E, w^*, M_0)$ where:

Figure 1.3: *An example Petri Net*

- $P = \{p_1, p_2, ..., p_\pi\}$ is the finite set of places.
- $T = \{t_1, t_2, ..., t_\tau\}$ is the finite set of transitions, and $P \cap T = \emptyset$.
- $E \subseteq (P \times T) \cup (T \times P)$ is the finite set of inbound and outbound arcs.
- $w^* : E \longrightarrow \mathbb{N}^+$ is the weight function used to specify multiple arcs between the same place and transition.
- $M_0 : P \longrightarrow \mathbb{N}^+$ is the initial marking of the places.

Priority and transition constraints in Petri Nets

The standard Petri Net definition can be extended with several additional features. The transitions can have a *priority* value that defines a partial ordering between multiple enabled transitions. A transition can only fire if it is enabled and no transition with higher probability is enabled. The weight of the arcs may have negative value with the meaning that the transition can not fire as long as there is at least as many tokens in the input place as the weight of the input arc. Arcs with negative weight are called *inhibitor arcs*. The previous Petri Net example with priority and inhibitor arcs added is shown on Figure 1.4.

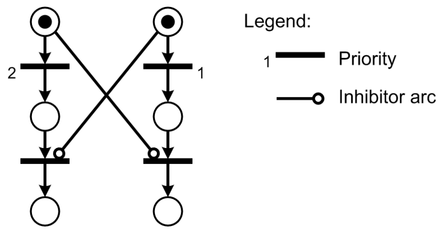

Figure 1.4: *An example Petri Net with priority and inhibitor arcs*

The formal definition of PN is extended as the following:

- $\Pi = T \longrightarrow \mathbb{N}$ is the finite set of priorities associated with the transitions.
- $w^- : (P \times T) \longrightarrow \mathbb{N}^-$ is the finite set of inhibitor arcs.

Deterministic and Stochastic Petri Nets

An other widely used extension of Petri Nets is the introduction of timing constraints to the firing semantics of the transitions. *Stochastic Petri Nets* allow the use of transitions with firing delay based on exponential distribution. This extension provides a powerful tool for modeling distributed systems where time plays a crucial part in performance and task scheduling. *Generalized Stochastic Petri Nets (GSPN)* may contain both immediate and exponential transitions [37].

In certain areas calculating the worst case scenario of system operation is in the center of attention. Hence the maximal time for executing a task is given and the maximal cycle time needs to be calculated. While the exponential distribution in the transition firing delays of GSPNs supplies timing properties it implies a random firing delay instead of a concrete delay. However *deterministic* transitions are defined with a concrete firing delay and are suited to model worst-case execution time. The Petri Nets that allow the use of immediate, exponential and deterministic transitions are called *Deterministic and Stochastic Petri Nets (DSPN).* The previously used Petri Net example extended with timed transitions is shown on Figure 1.5.

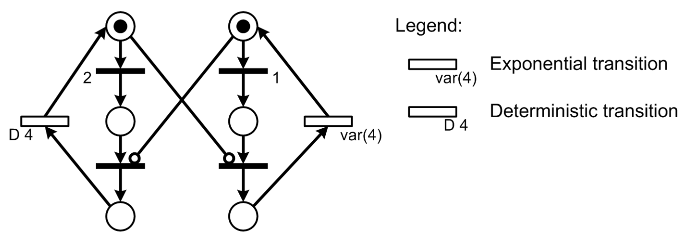

Figure 1.5: *An example Deterministic and Stochastic Petri Net*

1.4.2 Multiple-Phased Systems (MPS)

The operational life of embedded systems often consists of a sequence of non-overlapping periods called *phases*. Many of these systems are devoted to the control and management of critical activities and which require the execution of a series of tasks in sequence. Although the expressions *Phased Mission Systems* and *Scheduled Maintenance Systems* are often used to describe them the concept of phased execution is applicable to a wider variety of domains, systems and applications. Hence the name *Multiple-Phased Systems* is introduced to include all systems to which phased execution is applicable [9].

Phases can be characterized along many features for example the tasks performed within different phases, the performance or dependability requirements differ from phase to phase, the environment or the configuration may change between phases. Additionally the successful completion of a phase may have a different cost or benefit to the system with respect to other phases.

As the behavior of a system is often modeled with Petri Nets the execution of phases can be represented similarly. The phases themselves can be specified with places and the execution of a phase and the change to another can be illustrated with timed transitions. An example for the representation of phased execution is shown on Figure 1.6.

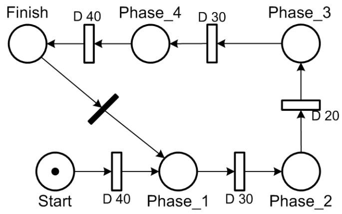

Figure 1.6: *An example of phased execution as a DSPN*

1.4.3 Dependability Evaluation of MPS (DEEM)

DEEM is a tool for the dependability evaluation of Multiple-Phased Systems. It uses DSPN as the modeling formalism and its solution technique uses efficient time-dependent analysis of Markov Regenerative Processes. DEEM provides the modelers with the possibility to model phase dependent behaviors using conditions captured with logic clauses. Modeling phase execution order and intra-phase behavior is also possible. Furthermore DEEM is capable of performing general dependability analysis and evaluation of generic performability measures with supporting the definition of *reward structures*. Reward structures allow the assigning of arbitrary reward values to states and events of the system. The total accumulated reward at a given point or averaged over a period can be used for most classical dependability and performance measures [9].

The modeling of an MPS is divided into two main parts, the system behavior and the phase execution. The behavior of the system is represented in DEEM with the System Net while the sequence of the phases can be defined in the Phase Net. While both are modeled in DSPN the following restrictions apply: (1) the duration of the phases is deterministic and (2) the System Net transitions can be only immediate or exponential. Figure 1.7 shows the interface of the DEEM tool with the Phase Net above and the System Net below.

DEEM also includes a set of modeling features to improve expressiveness of DSPN. Arbitrary functions of the model markings can be used when defining (1) firing times of timed transitions, (2) probabilities associated with immediate transitions, (3) enabling conditions of transitions, (4) arc multiplicities and (5) rewards.

1.5 Reactor Protection System case study

As a running example throughout the document the Westinghouse *Reactor Protection System* (RPS) will be used as it is a complex system including numerous software and hardware elements. The task of this system is to perform an automatic shutdown of the nuclear reaction when a potentially catastrophic event occurs in the nuclear plant [27]. An event is considered catastrophic if it can lead the plant to a state where the risks of damaging equipment, people and the environment is very high. The RPS performs a safety function by leading the plant to a safe state. In the following the description of the RPS is given along with the requirements defined toward it.

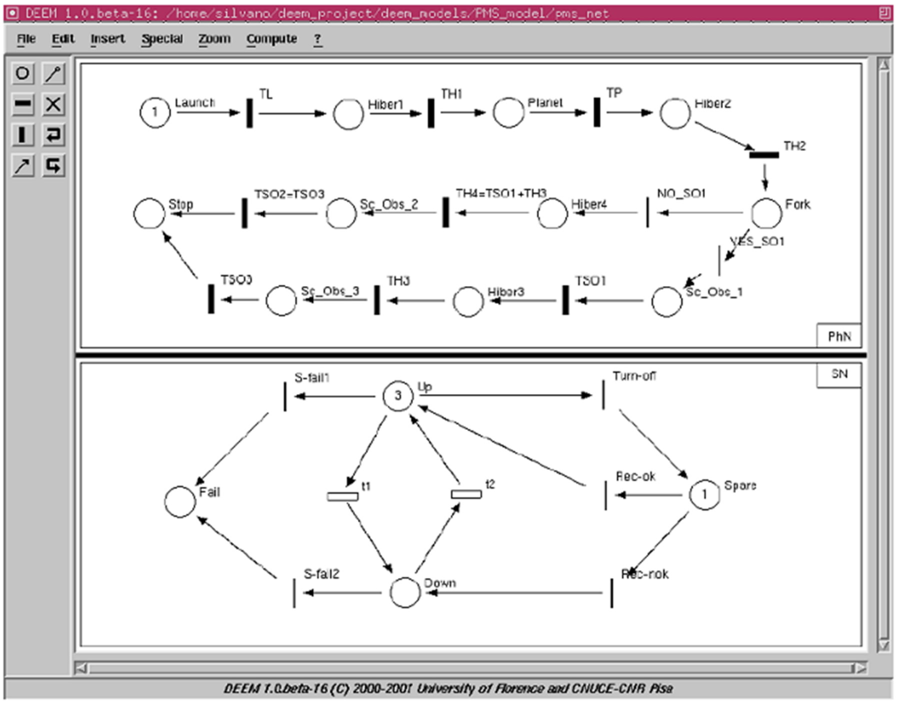

Figure 1.7: *The interface of DEEM and the DSPN models*

1.5.1 System description

The RPS consists of four segments connected in a series: the channels, the trains, the breakers and the rods. The channels monitor and process various physical quantities (pressure, temperature and others) continuously and generate a signal immediately if a single measure exceeds its set point value. The trains process the signals coming out from the four channels and generate a *trip signal* according to a two of four majority voter logic. The redundancy created by the four channels tolerates two simultaneous faults and allows the maintenance of fault tolerance capabilities. The trip signal starts the safety action that is completed by the breakers. The shutdown of the nuclear reaction is done by the descent of the rods into the reactor core. Figure 1.8 illustrates the functional structure of the RPS.

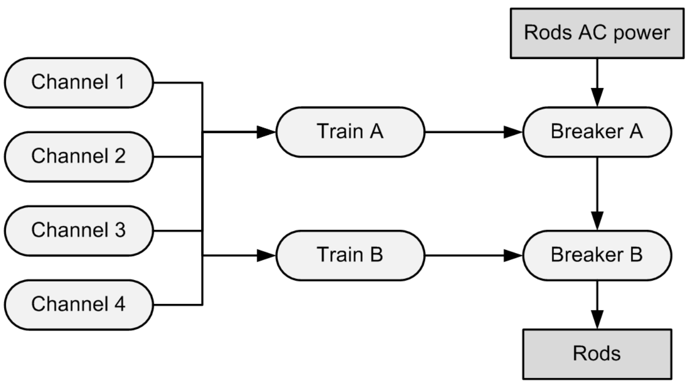

Figure 1.8: *The overview of the RPS architecture*

The four sections can be further extended by specifying their internal structure and contained components. In order to avoid overwhelming complexity only the structure of the trains segment is described here. The trains segment consists of two identical independent trains (A and B). Each train receives the signals from the channels and they are composed of n Solid State Logic (SSL) modules connected to the shutdown command generating modules. The SSL generates a trip signal according to a two of four voting logic. The trip signal is received by two devices the Auto Shunt trip (AS) and the Under Voltage (UV) which generate the same shutdown command. The state change of one of the devices is enough to start the shutdown command. Figure 1.9 shows the architecture of the trains.

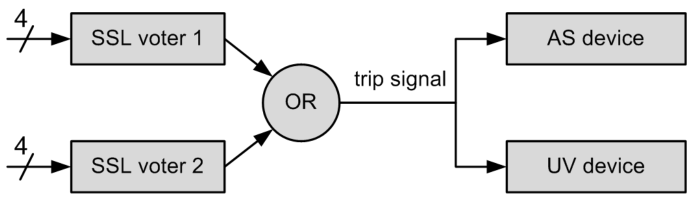

Figure 1.9: *The architecture of the train*

1.5.2 Failure possibilities of the RPS

The dependability modeling of the RPS needs the specification of the various failures that affect the components. For every failure the damaged component and the rate of the failure is given, the values are taken from [10] in which the values have been derived from [27]. The failures are listed in Table 1.1 where rates are given with failures per hours (f/h) unit.

Random event	**Rate(f/h)**
Breaker	2.5 E-7
AS device	4.7 E-6
SSL	2.6 E-7
UV device	4.1 E-6
Channel	7.0 E-6
CCF event	**Rate(f/h)**
3-of-4 channels	8.9 E-8
2-of-3 channels	3.0 E-7
SSL devices	1.5 E-8
UV devices	1.4 E-7
AS devices	1.6 E-7
Breakers	1.2 E-7

Table 1.1: *Failure rates of the RPS components*

1.5.3 Maintenance policy of the RPS

The RPS is maintained according to a test and maintenance (T&M) program that ensures that the system is always in a state that meets the necessary dependability requirement for the provided service and the reliability goal for each component. The T&M program defines a perfectly staggered maintenance policy which has proved to be less compromising to the system availability than the simultaneous T&M policy (where all the channels are tested at the same time). Table 1.2 shows the period and the duration of the T&M program for the channels and the train-breakers (which are maintained together). The rods are tested every 18 months but they are not included in the dependability model. It is important to note that the inner structure of the breakers contain bypass breakers parallel to the primary breakers. These bypass breakers work as spares during the maintenance of their respective primary breaker thus preserving fault tolerance capability.

Subject	**T&M period**	**Mean length**
Channels	3 months	4 hour (per trip signal)
Train-breaker	2 months	2 hour

Table 1.2: *T&M program parameters*

1.6 Financial case study

The Financial Case Study of the SENSORIA project [4] describes a credit portal application providing loan advice to the costumers of the bank. The case study document describes the main requirements for the model and implementation of the credit portal in a service-oriented environment. The main steps of the *credit request* scenario are defined as: the customer uploads the request which the bank employee reviews. After an internal verification the offer is sent to the costumer along with requests for additional data in case of high-risk credit requests. Finally the customer can accept the offer or upload the data while the request itself can be canceled.

In order to perform this steps the following services are necessary: *Authentication service* for user authentication, *Customer transaction service* for credit requesting, *Balance validation service* to evaluate requests and *Employee transaction* service to provide request review functionality. [21] introduces the non-functional service contract modeled for this case study. The non-functional service contract between the *Credit request service* and the external Balance validation service defined in that paper is extended with the characteristics defined here to show the feasibility of the method.

In this chapter the techniques and topics were introduced which are relevant for understanding the method defined in the document. The lifecycle of systems including errors, failures, maintenance and monitoring were described and the practice of modeling a system both for development and dependability evaluation purposes was introduced. The Westinghouse Reactor Protection System and the Credit portal will be used as a running examples to illustrate the use of the specified method.

Chapter 2

Related methods and techniques

In the Introduction and Chapter 1 the main topic of this document was discussed along with the most important techniques and standards. The subject of dependability modeling and using UML models for representing additional information and performing various evaluations is widely researched. In order to show the context of this work a brief summary is given on the important methods and research accomplishments in the related literature. In Section 2.1 the works related to modeling additional information in UML are introduced. Many of these works can be used for dependability analysis of systems. Finally in Section 2.2 a few of the tools are listed which can be used for dependability evaluation.

2.1 Modeling and analyzing dependability properties with UML

In [36] Bondavalli et. al. propose a profile for annotating UML models with stereotypes that cover the basic concepts of dependability properties. This approach provides support for the reliability and availability analysis of UML specifications by also defining a model transformation process which derives timed Petri Nets from the UML models through an intermediate model. The technique of using an intermediate model in later approaches can be credited to this work. Error propagation between components is supported along with random and common cause failures.

In [44] Pataricza uses the General Resource Modeling package of the *Schedulability, Performance and Time Specification* (SPT) profile (which was replaced by the MARTE profile) as the basis and extends it with the notion of faults and errors to support the analysis of the effect of local faults to the system dependability. The extension includes transient and permanent faults of the resources and the notion of error propagation as well.

In [46] Pataricza et. al. extend UML with stereotypes to perform completeness and consistency analysis on the behavioral description of a module in a safety-critical system. The functional design process is enriched by the modeling of potential faults and their effects. Error propagation and testability analysis is supported by using the extension.

Jürgens et. al. usees standard UML extension mechanisms to create safety [31] and reliability [32] check lists to support the identification of components in the software design that are prone to failure.

Bernardi et. al. [7] propose a class diagram based framework for collecting dependability properties and requirements of embedded systems along with real-time requirements. The approach provides support for a semi-automatic derivation of dependability analysis models such as Stochastic Petri Nets. A set of diagrams structured in packages is defined supporting systems with commercial off-the-shelf fault-tolerance mechanisms. In [8] they propose a method for assessing the quality of service of fault-tolerant distributed systems. The method derives performability models from UML models annotated using the SPT profile.

In [2] Addouche et. al. extend the SPT profile to provide concepts that enable the specification of real-time systems with stochastic and probabilistic information allowing dependability analysis. An extension to state charts semantics is proposed which are converted to probabilistic timed automata that can be used for the verification of various temporal properties related to dependability of real-time systems.

Dal Cin [13] proposes a profile for UML which defines a language for specifying and analyzing dependability mechanisms. It supports quantitative evaluation of fault-tolerance strategies and provides specifications for various dependability constructs to help in designing fault-tolerant systems. The drawback of this profile is that it lacks support for connecting the dependability mechanisms and the system components.

In [43] define a method for synthesizing dynamic fault-trees (DFTs) from the UML models. The information required for the analysis is added to the UML models in order to enable the generation of DFTs. The approach supports the modeling of error propagation sequences leading to failures along with redundancy and reconfiguration activities.

D'Ambrogio also uses fault-trees for predicting the reliability of component-based software in [17]. The method uses UML-based system specification which is mapped to a failure model that can used for reliability prediction of the final product. Cortollessa and Pompei [15] introduce an annotation for UML models using the SPT and QoS&FT profiles (both part of MARTE) for the reliability analysis of component-based systems. Grassi et. al. propose a model-driven transformation framework in [23] which uses the annotations in [15]. This approach also uses an intermediate model as a transition between the UML models and the analysis-oriented models.

Goseva et. al. [22] propose a risk assessment methodology using UML models at architectural level. These models are used to generate a Markov model which is then used for obtaining risk factors of scenarios. It can also identify critical components and connectors of the system requiring further analysis. In [24] Hassan et. al. define a methodology for severity analysis of software systems. Hazard analysis techniques including Functional Failure Analysis, Failure Modes and Effects Analysis and Fault Tree Analysis are integrated to identify hazards on the system and component level. The hazard analysis results and cost of failure information are represented on annotated UML models and are used for the analysis.

Mustafiz et. al. [41] propose a model-based approach for analyzing the dependability of use cases. A probabilistic extension to statecharts is used to formally model interaction requirements. The evaluation of the formal model is based on the success and failure probabilities of events and may lead to further refinement of the use cases. A visual modeling environment supporting the probability analysis of extended statecharts is implemented as well.

2.2 Analysis tools

DEEM is a tool for the dependability modeling and evaluation of Multiple-Phased Systems [9]. It is developed by the Dependable Computing Research Lab of the Institute of Information Science and Technologies (Pisa, Italy). The tool provides a wide range of features for modeling and analyzing systems using DSPN models as the modeling formalism and Markov Regenerative Processes for the model solution. The tool was introduced in more detail in Section 1.4.3.

Bondavalli et. al. use hand-made DSPN models in [10] to model and analyze the Westinghouse Reactor Protection System. This mission-critical system is modeled as a MPS and thorough dependability evaluation is carried out both for availability and performability properties along with sensitivity analysis of various parameters such as the maintenance schedule frequency and the reliability of the maintenance actions. The DEEM tool is used for performing the evaluation.

The Möbius Modeling Tool [14] is a modeling framework and *abstract functional interface* which can be used to eliminate the limitations arising when new and existing formalisms and solvers need to be compared. The infrastructure of Möbius provides support for multiple interacting formalisms and solvers. It can also be extended by new formalisms and solvers. The tool includes several modeling and compositional formalisms implemented by the creators of the tool.

PEPA or Performance Evaluation Process Algebra [25] is is an algebraic language which can be used to model performance properties of computer systems. The performance-related information may be used to predict the performance of the system while the behavioral information can be used for behavior analysis (e.g. for finding deadlocks or exhibiting equivalences between subcomponents). The PEPA workbench [20] is a prototype tool which supports the well-formedness checking of the PEPA model through state transition diagrams and the calculation of performance measures based on the infinitesimal generator matrix.

SPIN [26] is a tool that provides efficient software verification using a high level language to specify system descriptions called *PROMELA* (PROcess MEta LAnguage). The tool can be used for checking the logical consistency of specifications, deadlocks, unspecified receptions, race conditions and many more. The tool works on-the-fly without preconstructing a global state-graph or Kripke structure and can be used as a full linear temporal logic (LTL) model checking system. The main usage modes of the tool are as a simulator, an exhaustive verifier, a proof approximation system and as a driver for swarm-verification.

SAL [49] is the Symbolic Analysis Laboratory model checking framework which can be used for the model checking of systems modeled in the SAL language using transition systems formalism. The tools included in the framework are able to check the well-formedness of the model, find deadlocks in the model and check any LTL formula. The SAL language provides support for user-defined types, subtype definitions, variables with limited or unlimited value range and composition of modules in asynchronous and synchronous way. The behavior of the system can be modeled with labeled transitions that have guards and variable assignments while functions can be declared as well.

Many other Petri Nets tools have been created which use different description languages and provide different features. A long list of Petri Nets tools are maintained at [47]. Different tools may provide a graphical interface or support for the evaluation of instantaneous or cumulative measures. The tools also differ on which family of Petri Nets (regular, timed, stochastic or deterministic) can they model.

In this Chapter the related literature for the modeling and analysis of systems modeled in UML are given by briefly introducing some of the important researches and methods. Furthermore a short summary is provided for some of the analysis tools that are available for the evaluation of dependability and non-functional properties. Although the method is described using the DEEM tool as the target platform, most of the introduced tools and languages can be used as target platforms either by creating a code generator or a transformation from the intermediate model along the lines of the method defined here.

Chapter 3

Modeling Systems with Maintenance in UML

As discussed in Chapter 1 the maintenance and monitoring of complex systems are important aspects which have to be considered design-time in order to be able to perform correct evaluation on the dependability properties of the developed product. In this Chapter the UML profile defined to support designers with the means to describe the maintenance and monitoring aspects of systems is introduced. The profile which extends MARTE is the main topic of [3] which concentrates on the reliability aspects of systems and the different aspects of maintenance and monitoring proposing a complex framework for including these aspects along with analysis results with the functional and behavioral models of the system. First the overview of the profile is described in Section 3.1 followed by the detailing of the extensions to MARTE in Section 3.2. Next the core elements and their connections are described in Section 3.3 followed by the specification of the System and Maintenance packages in Sections 3.4 and 3.5. Finally the model library containing additional data type definition is described in Section 3.6. The non-functional service contract definition of the Service-Oriented Profile introduced in Section 1.2.2 is also extended to provide support for the dependability evaluation of service-oriented architectures.

3.1 Overview of the complete profile

The modular architecture of UML designs are one of the greatest advantages over other modeling paradigms. This property allows the designer to separate the different aspects of the developed system while still retaining the ability for a connected overview. As mentioned in the introduction of this section, the structural and behavior design of the system itself can be extended to include maintenance, monitoring and optional analysis as well. Figure 3.1. shows the different aspects of the modeling with the connections between them.

The **System** package contains the structure and behavior specifications that are usually given for a designed system. The components of the system, their internal and external communication and interactions, the software and hardware elements that the system is

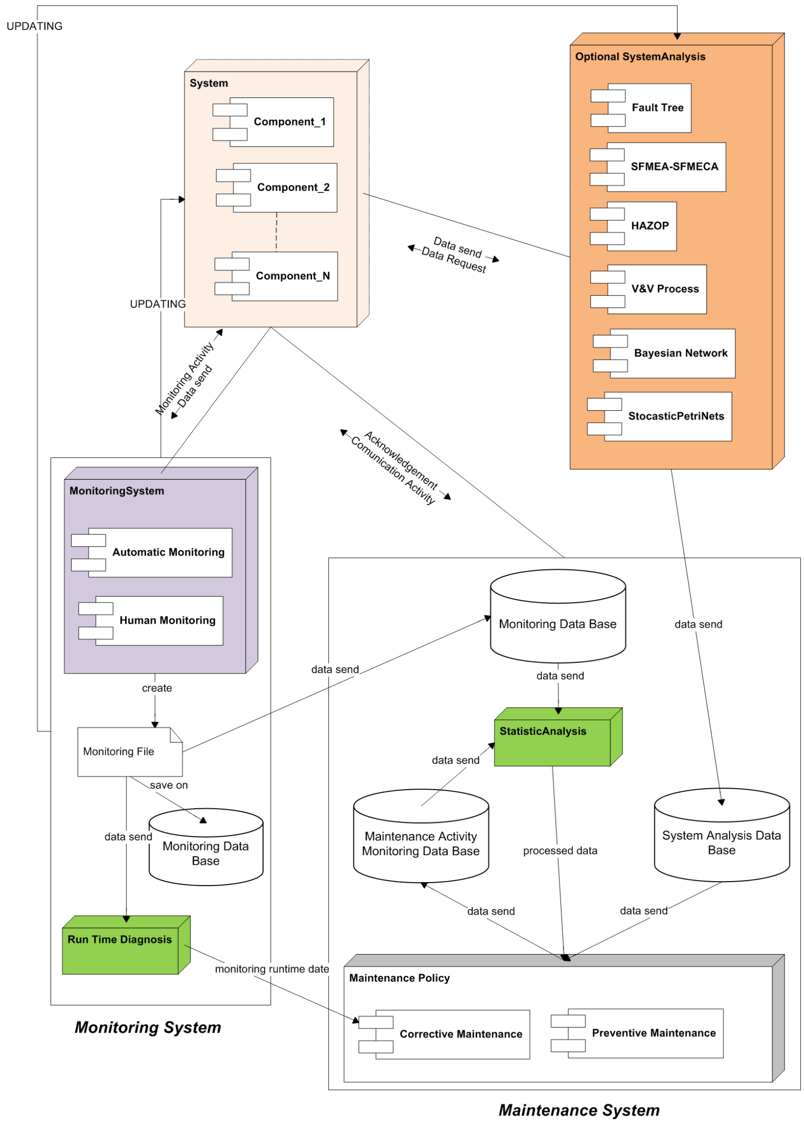

Figure 3.1: *The overview of the extended UML design*

built up from are all represented here.

The **Maintenance** package defines the various design decisions, policies and activities that specify how the system is managed during its operation and how the failures and errors are treated upon discovery. This package connects to the **System** package by associating the declared policies and activities to the elements in the system.

The **Monitoring** package describes the dedicated systems and decisions that define how the state of the elements in the system is monitored and what events or deadlines trigger the execution of the diagnosis on the target elements. This package connects with the **System** package by associating the monitoring components to the system elements and with the **Maintenance** package by associating the diagnosis to the appropriate maintenance policies and activities that are triggered for certain events.

Finally the **Analysis** package includes the different analysis types and their result on the system at design-time or during operation. The results can be used to refine the system design and to extend the maintenance policies and activities for optimal operation.

This package connects to the **System** package by using the properties and relations of the elements to feed the analysis with input and update these elements with the results. It also connects with the **Maintenance** package by sending the results of the analysis to optimize the policies. Furthermore the **Monitoring** package updates the analysis parameters with the data acquired from monitoring the system elements.

3.2 Extension model for MARTE

As discussed in Section 1.2.2 UML profiles can be used to extend the basic UML2 standard and thus providing system designers with state-of-the-art design patterns. The MARTE profile is widely used for the design of complex, embedded and fault-tolerant system which are the target of the method defined in this document as well. Hence it is a sound decision to use MARTE as a core for the profile created to provide support for the modeling of maintenance and monitoring. Figure 3.2 illustrates how the various packages of the profile are imported to create the model that extends MARTE.

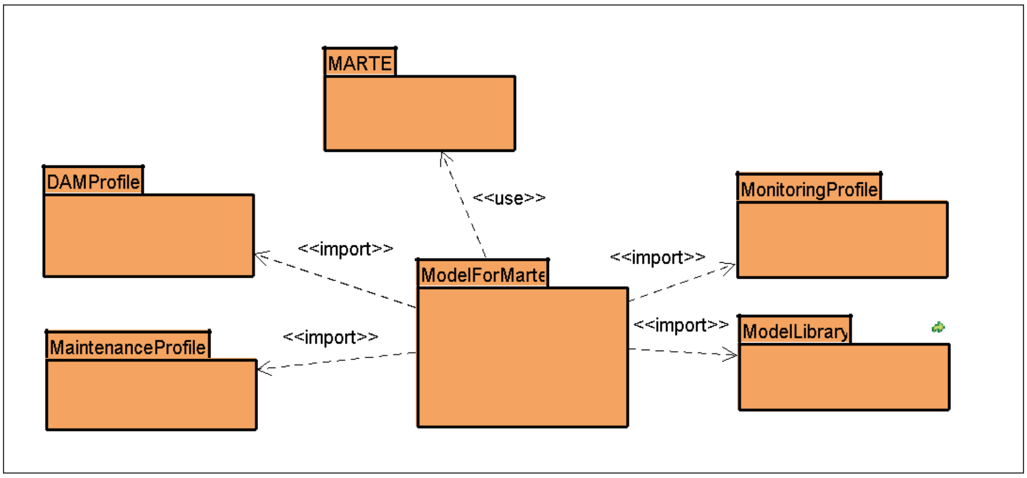

Figure 3.2: *The profiles of the extended UML design*

The System package defines several stereotypes which can be used to identify components in the system, the relations between them and the different failure types that damage parts of the system. The stereotypes specify attributes for the additional data needed to correctly model the dependability and maintenance behavior of the elements. The profile also includes support for explicit definition of the component hierarchy and composite structures.

The Maintenance package declares stereotypes which are used to represent the maintenance policies and their contained activities. There are also activity subtypes defined which represent different maintenance procedures. The attributes of the stereotypes can be used to parameterize the maintenance strategy of the system.

The Monitoring package includes stereotype definitions for modeling the different monitoring activities whether they are checking the occurrence of various events or executed on a time-driven base. These activities can be used by runtime diagnoses to determine the state of the system or its components.

The Model library contains the data and enumeration type definitions that are used as the attribute types of the stereotypes defined in the profiles. These types include the different maintenance policy types, monitoring actor types and several other.

3.3 The core metamodel

The profile diagrams of the UML contain only generalization and extension relations though there are several other relations between the elements defined. Figure 3.3 shows the core elements of the profile and the additional relations between them. These relations have to be created in the UML model which created by applying the profile. These relations are later referred to in the description of the profile and the transformation steps as well.

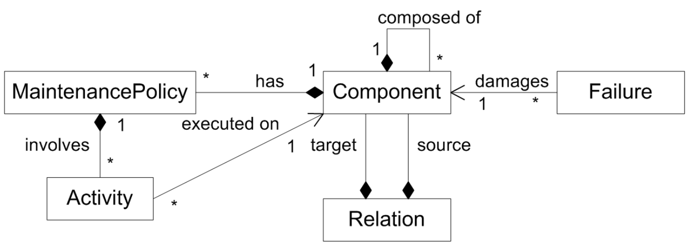

Figure 3.3: *The metamodel of the core UML profile elements*

The components can contain each other to create the composite structure of system, they can be connected by directed relations which can represent any kind of association or dependency. The component may be damaged by failures which have different characteristics. Furthermore components can have maintenance policies defined for them which may involve an arbitrary number of maintenance activities. Activities are executed on an selected component that is not necessarily the same with the component that has the policy involving the activity.

3.4 System package

The elements of the System package are shown on Figure 3.4 and their details are explained in the following:

- **Component:** this stereotype is used to mark the elements of the system that have to be considered in maintenance and monitoring. System elements without this stereotype are either not software or hardware elements or they are at a low abstraction level and are excluded from the dependability modeling. Components can have maintenance policies and monitoring activities attached to them. The attributes of the stereotype define the role the component plays in a redundancy structure

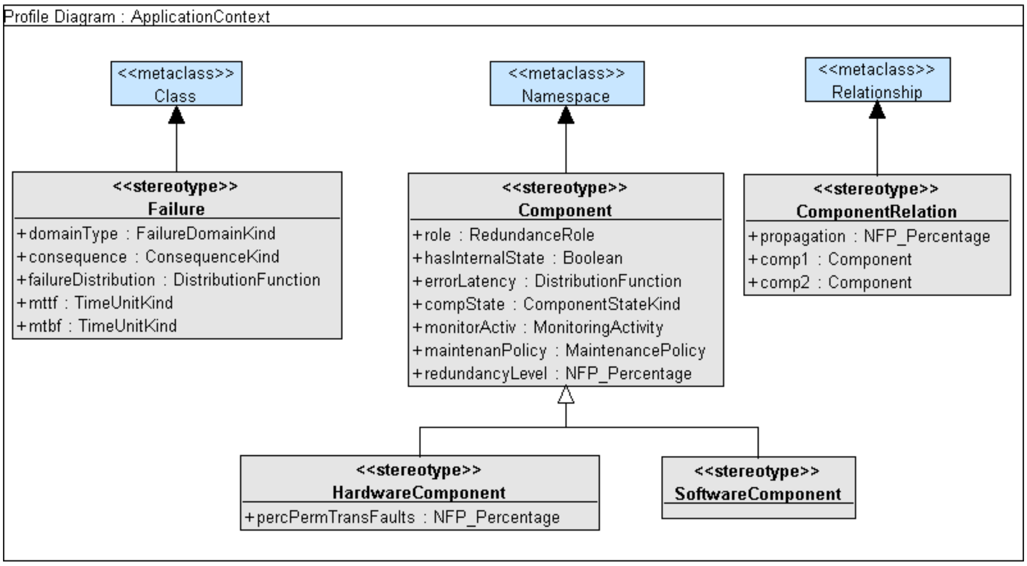

Figure 3.4: *The System package*

(**redundancyRole**), whether the component is stateful (**hasInternalState**), the latency that specifies how quickly error causes failures (**errorLatency**) and the level of redundancy the component has as a fault-tolerance structure (**redundancyLevel**).

- **HardwareComponent:** this stereotype is specialized from the **Component** type and it is used to identify system elements that represent hardware elements. It defines the additional **percPermTransFaults** attribute which specifies the rate of permanent (and transient implicitly) faults occurring in the component.

- **SoftwareComponent:** this stereotype is specialized from the **Component** type and it is used to identify system elements that represent software elements.

- **ComponentRelation:** this stereotype represents connections between components which are important for the dependability modeling. These relations indicate possible error propagation routes and constraints on the operation of the associated components. The probability of error propagation is defined by the **propagation** attribute.

- **Failure:** this stereotype can be used to indicate the possible failures that can damage the components. A *common cause failure* is associated with several components while *random* failures only damage one component. The attributes of the stereotype specify the occurrence rate of the failure and its consequence.

The **Component** stereotype can be used only on the following UML elements:

- **UseCase** and **Package** which are represented as software elements and are only stereotyped if their further refinement is not relevant for the dependability model

- **Object** is the most usual marked element and is represented as a software element.

- **Class** and **Component** are used as a default instantiation and represented as software elements if there is no **Object** of the class and further refinement for the component.
- **Node** is represented as a hardware element.

3.5 Maintenance package

The elements of the Maintenance package are shown on Figure 3.5 and their details are explained in the following:

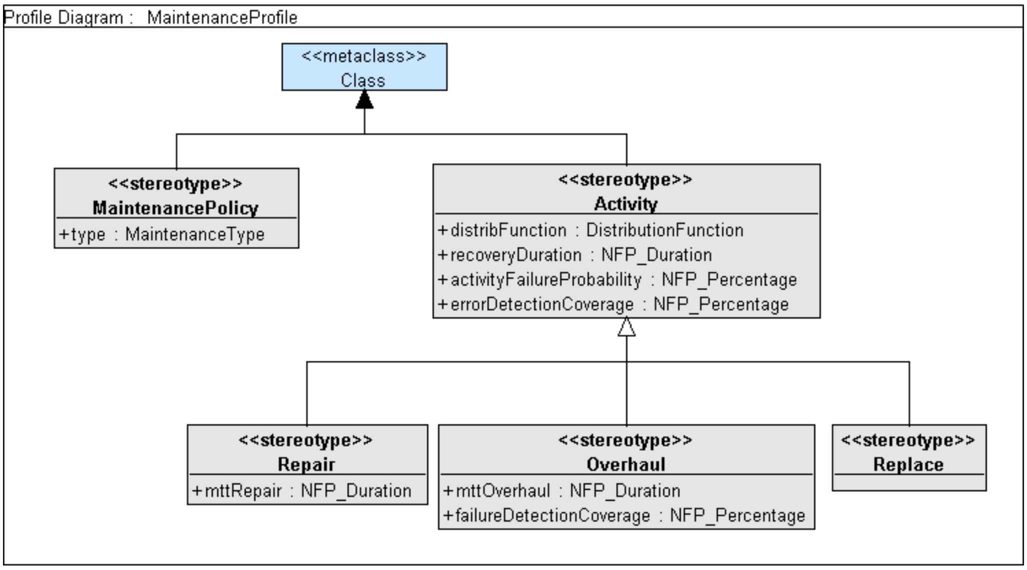

Figure 3.5: *The Maintenance package*

- **Maintenance Policy:** this stereotype represents a maintenance strategy for a given component. The type of the policy is specified with an attribute and the it is associated with several activities.
- **Maintenance Activity:** this stereotype can be used to mark elements that represent an activity that has a duration and is executed to correct a failed or erroneous component. The attribute **distribFunction** can be used to define the distribution for the time durations of the activities. The **recoveryDuration** attribute specifies the time required for the error recovery mechanisms when maintaining a stateful component. The **activityFailureProbability** attribute defines the probability that the maintained component fails during the activity execution. The **errorDetectionCoverage** attribute specifies the probability that an error is detected before a failure. The stereotype has subtypes which further specify the nature of the activity.
- **Repair:** this stereotype is used when the activity consists of finding and eliminating the cause of the failure in the associated component. The duration of the repair can be specified with the attribute **mttRepair**.

- **Replace:** this stereotype is used when the activity consists of replacing the associated component instead of correcting the failure.

- **Overhaul:** this stereotype is used when the activity consists of a series of tests, inspections and acceptability tests for deciding whether there are any errors and correcting them if there are. The duration of the repair can be specified with the attribute **mttOverhaul** while the coverage of failure detection can be defined by **failureDetectionCoverage**

3.6 Model Library

The model library contains additional enumeration and data types that are used as the type of the attributes in the profiles. The elements of the library are shown on Figure 3.6 and are further detailed in the following:

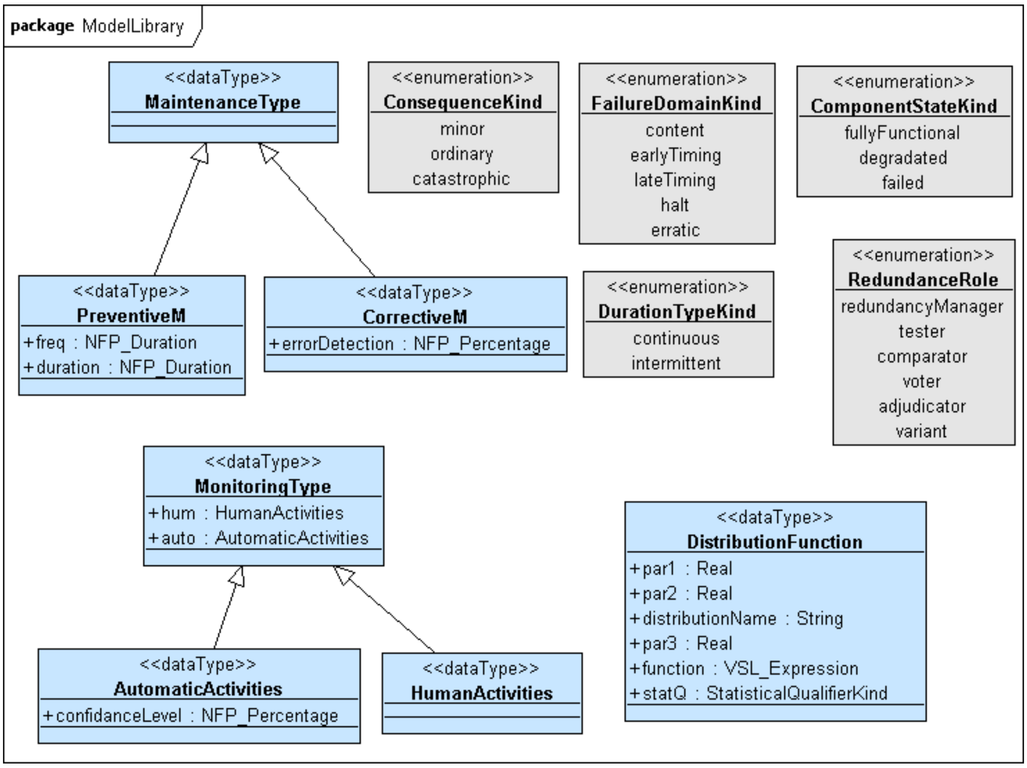

Figure 3.6: *The Model Library*

- **RedundancyRole:** this enumeration type defines the role the component plays in the redundancy structure.

- **MaintenanceType:** this data type specifies the overall strategy type of the maintenance and is the generalization of the specific maintenance policy types.

- **PreventiveM:** this data type is used for maintenance policies that strive to prevent the failure of the structure by executing activities before the various failures of the components could cause the system to fail. The attributes of the type define the time period that elapses between two maintenance executions and the time duration of the maintenance.

- **CorrectiveM:** this data type defines a maintenance policy where activities are executed when the associated component for the policy has failed.

- **Domain:** this enumeration type specifies the appearance of the fault that occurs defined by the **Failure** element. The appearance can be an erroneous value, a timing error or the complete stopping of the component.

- **Consequence:** this enumeration type can be used to define the effect of the fault when it causes the component to fail. The effect can range from minor to catastrophic.

- **ComponentState:** this enumeration type can be used to specify the actual state of a component. This type is useful if the model is updated with runtime diagnosis results and analysis is carried out with the actual information. The state can be fully functional, degraded or failed.

- **DistributionFunction:** this data type is defined as a customizable variable distribution which can be specified using the attributes. The distribution models the Weibull continuous probability distribution which is capable of mimicking various distributions using the *shape* and *scale* parameters [54].

3.7 Extending the UML4SOA profile with dependability and maintenance

One of the core concepts of Service-Oriented Architectures is the use of existing services instead of implementing the same functionality repeatedly. These services are often provided by a different party and their quality is bound by the service contract between the requester and the provider [19]. The UML4SOA profile extends the MARTE profile in order to provide the means for modeling and developing SOAs. The profile includes a metamodel for the definition of non-functional properties of a service. In the following this metamodel is described in Section 3.7.1 and the extensions defined to include dependability and maintenance characteristics in these models are introduced in Sections 3.7.2 and 3.7.3.

3.7.1 Non-functional service contracts

The metamodel defined in the UML4SOA profile for non-functional properties supports the creation of contracts between provider and requester parties. External third-party services can also be used for monitoring the runtime characteristics of the service. The non-functional aspects of the service (for example security or performance) are defined as *characteristics* while the set of attributes they contain are defined as *dimensions* [33]. The metamodel is illustrated on Figure 3.7.

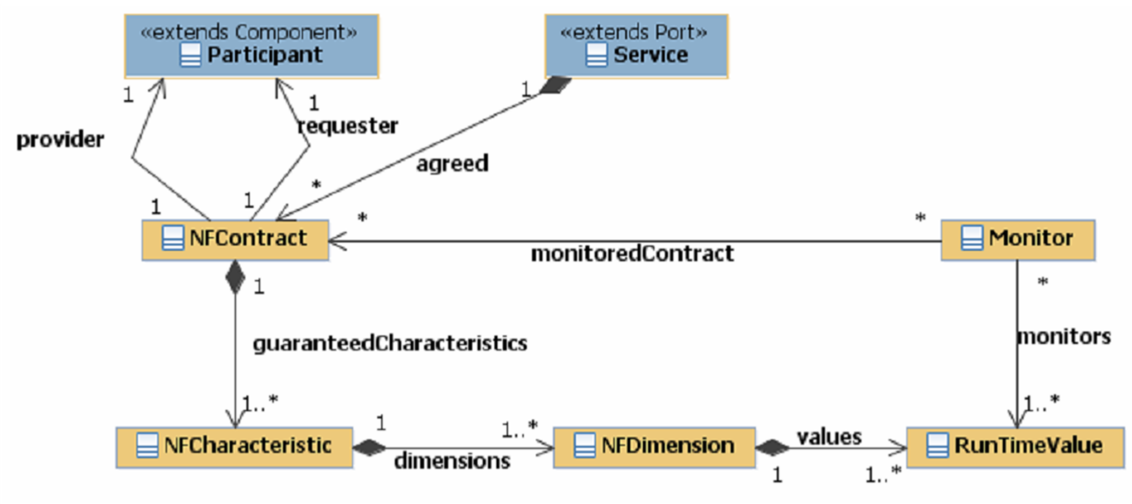

Figure 3.7: *The UML4SOA metamodel for non-functional properties [21]*

Non-functional characteristics can be represented by **NFCharacteristic** elements that are associated with the service contract representation **NFContract**. These characteristics can be defined for various properties of the service. The specifications of both the requester and provider party are bound with the contracted service.

Non-functional dimensions are represented by **NFDimension** elements contained in the **NFCharacteristic** elements. The actual values of the attributes are represented by the **RunTimeValue** elements composing the dimensions. These values are monitored by the third-party represented with the **Monitor** elements.

3.7.2 Provider Dependability Characteristics

The framework for dependability evaluation of UML designs can be used to analyze systems developed using the SOA paradigm. However the services used by the developed application have their own dependability properties and these properties have to be represented in the models as well. The non-functional service contracts defined in the UML4SOA profile are particularly suited for representing these characteristics using the means provided by the profile. The *provider dependability* characteristic is defined to allow the modeler to include the dependability attributes of the service. Figure 3.8 shows the **ProviderDependability** element which is marked with the **nfCharacteristic** stereotype and its contained **nfDimension** elements.

The **FailureMode** dimension can be used to define whether the service is created to be *fail-silent* (i.e. it stops receiving and answering requests when failed), the length of time after which no answer means the failure of the service (*timeout*) and the error propagation probability (*propagation*). The **Availability** dimension allows the definition of the *mean availability* of the service, the mean time between failures (*mtbFailure*) which is used in the modeling of the failure rate of the service as a component.

The same characteristic definition can be used to specify the properties of a requested service and a provided service. After the provider of a service performs dependability

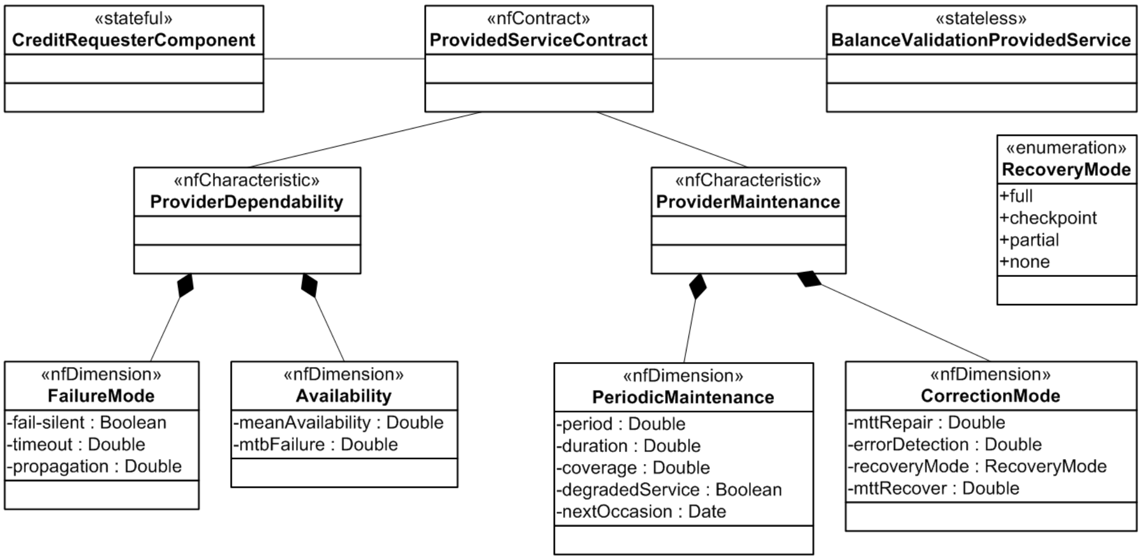

Figure 3.8: *The defined UML4SOA classes*

evaluation on its service, the results can be included in the specification of the provided service as part of the contract. On the other hand the requester of a service can use this characteristic to specify the properties of the used service as a component and perform dependability evaluation on the system using the service.

3.7.3 Provider Maintenance Characteristics

The defined dependability characteristic is useful to specify the properties of the system concerning its behavior during operation. However the characteristics regarding the maintenance of the service have to be provided as well to specify what measures are taken in the case of a failure event and how often is the service unavailable due to planned maintenance. The *provider maintenance* characteristic is defined for specifying these properties. The **ProviderMaintenance** element with **nfCharacteristic** stereotype is shown on Figure 3.8 together with its contained **nfDimensions**.

The **PeriodicMaintenance** dimension is used to specify how often planned maintenance is performed on the system (*period*), how long this maintenance takes (*duration*) and when it is scheduled to happen next (*nextOccasion*). This last is defined in order to give a starting point for the estimation of the maintenance events. Finally it is possible to define whether degraded service is available during the planned maintenance (*degraded-Service*) and the failure detection coverage of the maintenance activities (*coverage*).

The **CorrectionMaintenance** dimension defines the constraints promised by the provider of the service in the event of a failure. The mean time needed to repair the service is given (*mttRepair*) and the error recovery mechanisms in place are defined (*recoveryMode*). The recovery mode can be either *full* if the state before the failure is restored during the maintenance or *partial* if only certain data is restored. The recovery mechanism may use a *checkpoint* from a given time or it is possible that no recovery is performed (*none*) for example if the service is stateless. Additionally the mean time of error recovery can be given as well (*mttRecover*) along with the error detection coverage (*errorDetection*).

The practice of using the characteristic in two ways is present in the case of the maintenance characteristic as well. The provider of the service may generate the values used for the dimension by performing dependability evaluation of the service and include the results in a contract while the requester uses the values to include the service as a component in the evaluation of the system using the service.

If the service modeled with UML4SOA is external the attribute values of the dimensions for the dependability and maintenance characteristics in the non-functional service contracts are taken from the *Service Level Agreement* documents supplied by the business partners. On the other hand if the service is internal and its models are available then the values can be acquired by performing *dependability analysis* on the system responsible for providing the service.

In this chapter the packages of the UML profile created for providing means to modelers to include maintenance, monitoring and analysis aspects in their system designs were defined. The profile extends the MARTE profile which is a widely known industry standard created to support the modeling and analysis of real-time and embedded systems. Systems modeled using the defined profiles can be the target of dependability evaluation using the method defined in this document. Additionally the extensions to the UML4SOA non-functional service contracts are introduced which allow the engineers of service-oriented architectures to include the dependability and maintenance properties of external services in the evaluation of their system. The extension is also illustrated using the Financial case study from the SENSORIA project.

Chapter 4

From UML designs to Intermediate Dependability Models

The dependability evaluation of systems can be carried out on appropriate mathematical models such as introduced in Section 1.4. However the engineers and designers of complex, contemporary systems use UML and its extending profiles to model the systems in development. The method defined in this document provides the means to create a dependability model from the UML models automatically. It is important to note that there are numerous profiles that are used to model systems for different domains and a high number of tools that provide dependability evaluation capabilities. A key aspect of the method defined here is to separate the specific language used for system modeling and the specific tool (and its input format) used for analysis. In order to achieve this goal an *intermediate model* is defined for creating a transition between UML models and dependability models. The idea of creating an intermediate model between the source and target model is mostly credited to [36]. In this chapter the definition of the intermediate model is given in Section 4.1 and the numerous steps defined to create the intermediate model from the UML models are specified in Section 4.2.

4.1 Intermediate Model (IM)

The *intermediate model* is a general model describing a system, its nonfunctional properties and its maintenance strategy. The system is composed of multiple hardware and software elements, which can be organized in fault-tolerance structures and can be subject to several maintenance policies and activities. The structure of the IM is inspired by the approach presented in [11, 35]. In order to represent the system correctly, those models are modified and extended. The elements required to model maintenance are added while the fault-tree building blocks are elevated to element level from attribute level.

Note that in the context of this thesis the IM is indeed a transition or intermediate state between the UML and dependability model. However the expression *intermediate* is not entirely proper when describing the model itself. To be exact, the IM is a dependability *domain-specific model* which contains all the information required to generate a dependability model for a system in an arbitrary formalism.

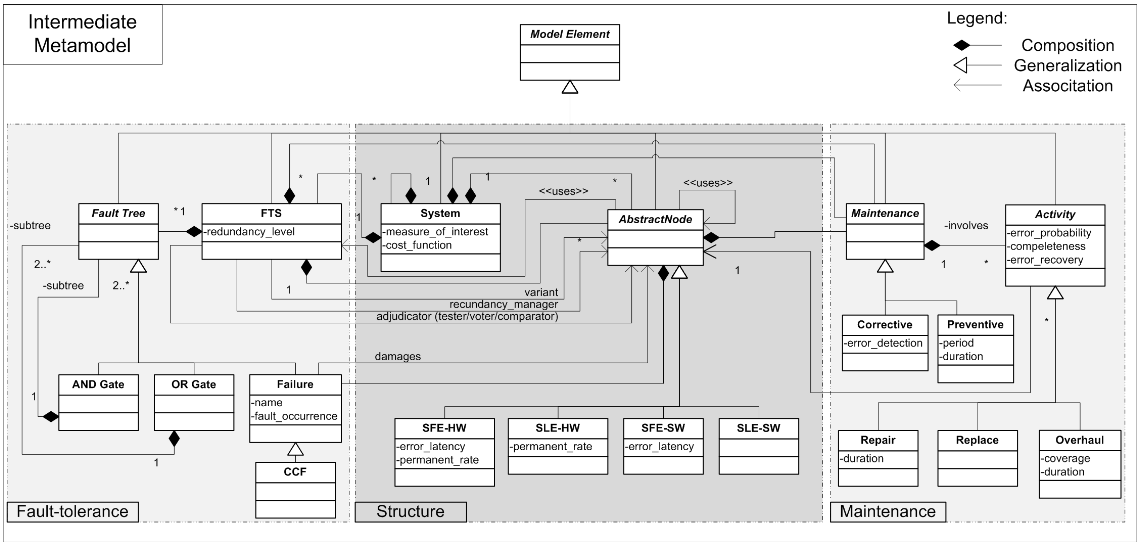

Figure 4.1: *The intermediate Metamodel*

Formally, the IM is a *hypergraph* $G = (N, A)$, where the elements in N represent entities or information derived from the set of UML diagrams, and each hyperarc in A represents a relation between elements. The relations are also projected from the UML diagrams and are part of the system structure itself. The elements and hyperarcs are labeled and have a set of attributes attached to complete their description. These attributes are obtained from the UML diagrams as well. In the following the semantic of the IM is given, by specifying the elements and relations shown in Figure 4.1 as well. The **ModelElement** is the most general element that is the (immediate or transitive) parent of each element in the IM. It is used as a common base for every element and is abstract therefore it isn't instantiated in any concrete model.

4.1.1 Nodes

The elements representing components whose further refinement is not relevant for the dependability model are called *nodes*. The nodes can be either *hardware* or *software* components both can be either *stateless* (purely functional) or *stateful* (having internal state). The **AbstractNode** element is the general node that is the source or target of the relations that any node can have. The four distinct type of nodes are tho following:

Stateless Hardware (type SLE-HW) nodes represent purely functional hardware components in the system. The only attribute for this type of node is the following:

<permanent_rate>, which field specifies the relative ratio of permanent and transient faults. At the moment, both the permanent and transient faults affecting a hardware element share the same failure process. This could be refined during the development of the method. Moreover, the field may be left unspecified if a more detailed fault submodel is included in the dependability model.

Stateful Hardware (type SFE-HW) nodes represent hardware components which do have internal state. Having internal state means that the occurrence of faults does not immediately lead to the failure of the component. First it creates some erroneous internal state, which eventually causes the failure of the component. The attributes for the **SFE-HW** type of nodes are the following:

<error_latency>, which refers to the mean duration of the process that leads to the failure from an erroneous state. It specifies the mean value of the exponential distribution that provides the random values for the process instances.

<permanent_rate>, is defined as for the **SLE-HW** nodes.

Stateless Software (type SLE-SW) nodes represent purely functional software components in the system. At this point there are no attributes foreseen for this type of node. Note that since the component is stateless, error recovery is not needed. Moreover, faults affecting software components are always of transient type.

Stateful Software (type SFE-SW) nodes represent the software components of the system that have internal state (for example variables). Having an internal state has the same consequences as explained at the **SFE-HW**, fault-occurrence leads to an erroneous state, which causes the component to fail eventually. As for the **SLE-SW**, faults affecting software components are always of transient type. The **SFE-SW** nodes have the following attribute:

<error_latency>, which is defined as for the SFE-HW nodes.

4.1.2 Structures

The elements that are further refined in the dependability model by defining the components they are composed of are called *structures*. These can either represent the hierarchical composition of components or the redundancy architecture created from them. In this section these elements and their relations are detailed.

System elements represent both the components of the system whose dependability properties are the target of the evaluation and the composite components as well. A system element may represent the system as a whole which provides services (functionalities, use cases) to users. However, the system element does not have to correspond to a specific UML entity. It can be used to represent several elements in the system, which interact with other system elements as a whole. Moreover, it may correspond to any entity in an UML diagram, if the dependability attributes for that particular entity should be estimated. In this case the node has the following attributes:

<measure_of_interest>, which defines the specific dependability property the designer has chosen for the analysis. This can be instantaneous or mean value of reliability or availability among others.

<cost_function>, which specifies a more precise function to measure additional properties and perform sensitivity analysis on a certain scale.

Fault-tolerance structures (type FTS) are composite elements composed of nodes. An **FTS** element is not an actual entity in some UML diagram, it only corresponds to the redundancy structure implemented by a group of nodes. The nodes that make up an **FTS** fall into three different categories. These categories define the role of a node in the redundancy structure. The main categories are the following:

- **Redundancy manager** identifies the node through which the service provided by the whole redundancy structure is available. An **FTS** can only have one redundancy manager.
- **Variant** refers to members of a set of redundant elements that provide the service and are controlled by the redundancy manager.
- **Adjudicator** that defines the nodes responsible for validating the behavior of the variants and thus assure the correct functioning of the service. It can be further refined to various subtypes such as *tester*, *voter* or *comparator*.

The **FTS** elements may define the following attribute:
<redundancy_level>, which further refines the number of failures the structure can tolerate without failure. This attribute is used in the automatic generation of fault-trees as the percent of variants that may fail before the structure fails because of the number of failures. For example a 2-out-of-4 voter has a 0.5 (i.e. 50%) redundancy level as it can tolerant two failed variants but not more.

In addition, **FTS**s may contain any number of *fault-trees*, which describe how the components together provide redundancy and which combination of failures will eventually cause the **FTS** to fail when the redundancy structure is not able to tolerate them. The description of the fault-tree can be gathered from the following sources:

1. From the analysis of the UML diagrams, if the fault-tolerance scheme is defined explicitly. In Section 4.2.5 the analysis procedure is described in detail.
2. From the fault-tolerance library, if the fault-tolerance scheme was selected from the list of schemes already described in the library.

Note that there are various methods to describe fault-tolerance and fault-trees are only one of these. However, in the method defined in this document the fault-trees together with the roles described above are used to specify redundancy of structures. The Intermediate Model may be refined later by including others like event-trees or reliability block diagrams.

Fault-tree in the intermediate model are hierarchical in the sense that every fault-tree consists of either a *single failure* or a *gate* (logical operation) that is composed of subtrees which are fault-trees themselves. The **FTS** elements can contain several fault-trees, all of which are composed from the following elements:

- The **Failure** elements are introduced to the intermediate model to represent one type of failure of a given component. The element is connected to one `Node` element with the *damages* association. The attributes of the `Failure` element is the following:

<name>, which identifies the failure so that it can be used in several fault-trees or more than one time in the same fault-tree. <fault_occurrence>, which describes the time needed for a fault to occur on the component itself. It specifies the mean value of the exponential distribution that provides the random values for the fault appearance instances.

- The **Common Cause Failure** (type CCF) element is a specialized `Failure` that represents the event when more than one component of the same type fails in the same time (because of a common cause). Therefore the **CCF** element is associated with more than one `Node` element.
- The **AND Gate** element represents the logical conjunction operation which is true only if all the subtrees it is composed of are true. The element has composition relations to at least two other subtrees and has no attributes.
- The **OR Gate** element represents the logical disjunction operation which is true if at least one of the subtrees it is composed of is true. The element has composition relations to at least two other subtrees and has no attributes.

4.1.3 Maintenance

The maintenance elements of the IM are used to represent the *policies* declared for the system in the UML diagrams. A policy describes the strategy and the schedule of maintenance executed on the components of the system. Maintenance policies may correspond to a single component, a fault-tolerance structure, a high-level service or the whole system. Moreover several policies may be defined for a single component or structure. Each policy involves several *activities* which specify the properties of the maintenance performed on given component. The policies and activities are detailed in the following.

Policies

Maintenance policies define the strategy followed by the operators of the system. Two policy types are defined in the IM, corrective and preventive maintenance, both are specialized from the abstract **Maintenance** element. *Corrective maintenance* is applied when components are maintained only in case of a failure. Whereas *preventive maintenance* aims to avoid failures by periodically renewing the condition of components.

Corrective Maintenance (type Corrective) represents a maintenance policy that executes the involved activities on components in the event of a failure. The element has the following attribute:
<error_detection>, which defines the probability whether the erroneous state of a component is detected by the maintenance (for example for part of a FTS).

Preventive Maintenance (type Preventive) represents a strategy that includes periodical execution of the contained activities thus increasing the chance of avoiding the failure of a service or system. The Preventive element has one attribute attached:

`<period>`, which specifies the time between consecutive maintenance executions. Finding the optimal period for a given fault-tolerant system is a difficult task, though it is possible through sensitivity analysis of the dependability model.

`<duration>`, which specifies how long the maintenance is performed in every period. The activities included in the policy may take more or less time to finish, but can not be started after the system returns to normal operation mode when this time elapsed.

Activities

Apart from policies, the other important aspect of a maintenance strategy is the activities involved. Every activity has a target node for which it specifies the properties needed to carry out the dependability analysis. Several activity types are defined in the IM, all of them are specialized from the abstract **Activity** element. This element includes the following common attributes of the different activity types:

`<error_probability>`, which defines the probability that the target component of the activity fails during the maintenance regardless of it's earlier state. It can also be defined as the *coverage of error correction* mechanism.

`<completeness>`, which defines the probability that a successful maintenance activity restores the component to a correct state if it did not fail but is in an erroneous state. It can also be defined as the *coverage of error detection* mechanism.

`<error_recovery>`, which represents the time needed for recovery from a erroneous or failed state. This attribute is considered only if the target node is stateful. It specifies the mean value of the exponential distribution that provides the random values for the recovery instances.

The following activity types are defined in the intermediate model:

Repair element represents the activity used when the target component is maintained by eliminating the causes of its failure without replacing the whole component. Repair elements have the following attribute:

`<duration>`, which defines the time needed for explicit repair of a failure. It specifies the mean value of the exponential distribution that provides the random values for the repair instances.

Replace element represents the maintenance task of exchanging the failed component with a new one. The replacement of a component is instantaneous for a stateless component, while error recovery is needed in case of stateful components. No attributes are foreseen for this element.

Overhaul element represents the task of inspecting the targeted component. An overhaul usually consists of acceptability tests and adjustments. Overhaul elements have the following attributes:

`<duration>`, which is similar to the definition in Repair activities, but in this case it defines the time needed to perform the inspection.

`<coverage>`, which defines the probability that the inspection reveals the failure of the target component and is also referred to as *coverage of failure detection*.

4.1.4 Relations

The hyperarcs in the set A represent the relations between the elements of the intermediate model. Several association relations have already been defined, such as the *involves* relation between the **Maintenence** and **Activity** elements or the *targets* relation between the **Activity** and **AbstractNode** elements. However, there are two additional hyperarcs, which are detailed in the following:

Uses the service of (abbr. uses) relation exists between two elements in the IM if during their operation they communicate in a client-server pattern. In the IM, software nodes may use the services of other software, hardware or FTS elements. Hardware nodes may use the services of other hardware or FTS elements. Actors use the top-level System elements to interact with the system under investigation. The *uses* relation is unidirectional and there is a possible fault-propagation path between the connected nodes. Whenever the server node fails, the client also fails (or reaches an erroneous state) with a non-zero probability as a result. Moreover, the failure of the client may also cause the error or failure of the server following a faulty request, but this type of relation is modeled with another *uses* relation in the opposite direction. Apart from fault-propagation, the *uses* relation also describes a constraint for the maintenance of the element. The element can not function properly after a failure as long as its external environment (the elements it uses) contains failed elements. Therefore, even if the maintenance of the element is finished, it becomes fully functional only after the used elements are behaving correctly as well. The *uses* relation has the following attribute:

`<propagation_probability>`, which defines the probability that a failure of the target element causes an error or failure in the originating element.

Is composed of (or composition) relation represents the hierarchical composition between the various components of the system. The composition hyperarc links the **FTS** elements to the set of nodes that they are composed of. Moreover, this relation is used to link the **System** element with the elements it consists of. The composition relation thus indicates the non-trivial dependencies between the connected elements. There are no attributes defined for this relation.

4.2 From the UML models to the Intermediate Model

The system designs created according to the UML profiles introduced in Section 3 are modeled with the Intermediate Model specified in 4.1 as part of the method to generate a model for dependability evaluation. In this section the analysis of the UML design is described and the creation of the IM elements. First the creation of various structure elements are defined in Section 4.2.1 then the fault-tolerance elements are created according to Section 4.2.2. Next the maintenance related elements are created in Section 4.2.3 and the **uses** relations are created in Section 4.2.4. Finally the algorithm for the automatic generation of the fault-trees is defined in Section 4.2.5 and the generation of the IM elements for non-functional service contracts is described in Section 4.3.

4.2.1 Creating Structure Elements of the IM

The **Component** stereotype and its subtypes are used to mark elements in the UML design which have to be included in the dependability evaluation. In the IM the **System**, **FTS** and the various **Node** subtypes are used to represent the structure elements. The elements in the UML design that are marked with the **Component** stereotype are represented in the IM according to the following guidelines:

- **Component elements which are not contained in an other Component element** are the top-level components of the system, these elements are represented with **System** elements in the IM.
- **Component elements which contain additional Component element** are components in the system that are below the top-level and also contain additional elements themselves. These elements are represented with **FTS** and **Node** elements in the IM if they are marked as **redundancy managers** and **System** elements otherwise.
- **Component elements which do not contain additional Components** are components whose further refinement is not included in the dependability model. These elements are represented with subtype elements of **Node** in the IM. The actual subtype depends on the additional properties of the element.

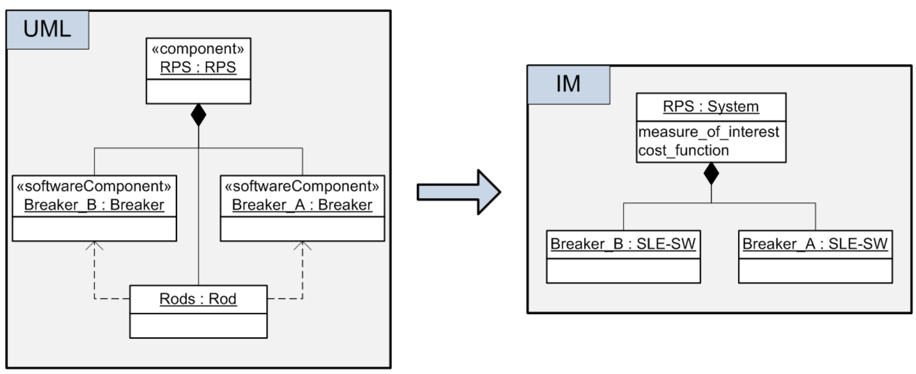

Figure 4.2: *Components and the created IM elements*

The **hasInternalState** attribute of the **Component** stereotype defines whether the component has an internal state. The **Node** element created for the actual **Component** is **SLE-SW/HW** type if the **hasInternalState** attribute is false and **SFE-SW/HW** if the attribute is true. Figure 4.2 shows how the structural architecture of the RPS in UML is represented in the IM. Note that the composite components of the RPS are modeled as software elements for the sake of the example. The attributes of the created elements receive the value already given in the attributes of the original UML elements. The composition relation between **Component** elements are represented with **composition** relations in the IM.

Most of the UML elements that are marked as leaf components are represented by software **Nodes**. However if the element is a **Node** UML element, it represents a hardware component and it is represented with a hardware **Node** in the IM. Figure 4.3 illustrates how the different UML elements are represented in the IM.

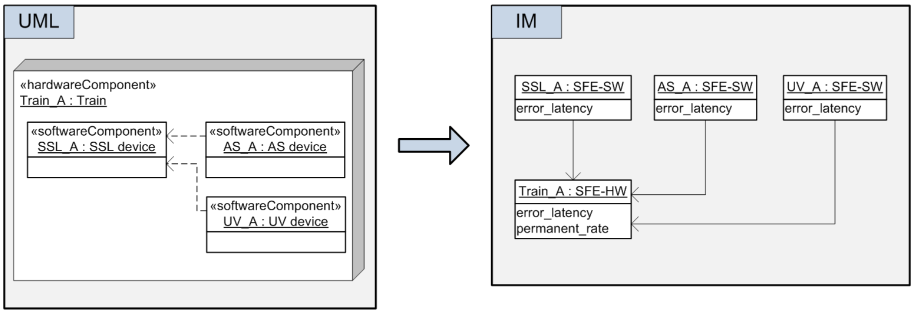

Figure 4.3: *Hardware Nodes and the created IM elements*

4.2.2 Creating Fault-tolerance Elements of the IM

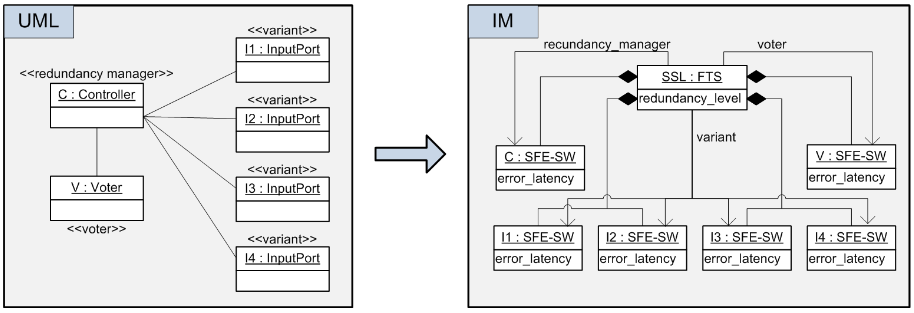

Figure 4.4: *Fault-tolerance stereotypes and the created IM elements*

The components of the fault-tolerance structure are created as specified in Section 4.2.1. However the fault-tolerance representation in the IM also contains the different roles attached to the components of the FTS, the fault-trees specified for the structure and the failures damaging the components. The roles are specified with an enumeration type in the UML model and they are represented with the appropriate relation between the **FTS** and the associated **Node** in the IM. These roles are the **redundancy manager**, the **adjudicator** and the **variant**. Note that the adjudicator can be further specialized as **tester**, **voter** or **comparator**. The created role relations for the fault-tolerance structure of the trains segment of the RPS are shown on Figure 4.4.

As mentioned in Section 4.1.2 the fault-trees of a fault-tolerance structure can be either selected from a fault-tolerance library if the modeled structure defines a known redundancy

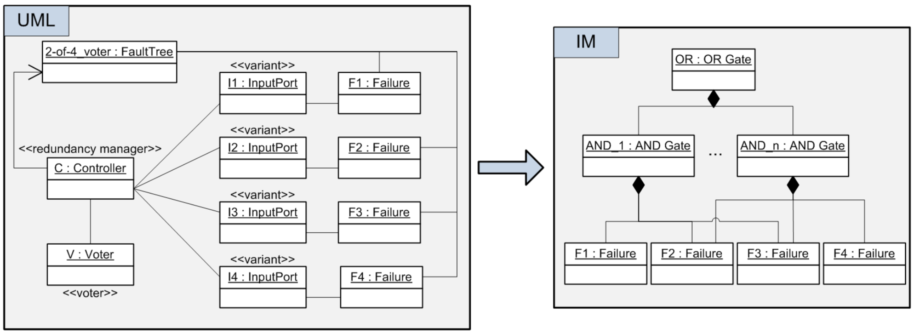

Figure 4.5: *Fault-tree from library and the created IM elements*

or it can be automatically generated from the UML model. The automatic generation is described in Section 4.2.5. A fault-tolerance library may contain a collection of fault-trees with an arbitrary complexity. These fault-trees are created using the elements defined in the dependability profile. The fault-trees in the IM are constructed using **AND gates**, **OR gates** and **Failures** composed according to the UML design. Figure 4.5 shows how the fault-tree definition of the example is represented in the IM.

Finally the random and common cause failures specified as the leaf elements of the fault-trees are examined. These elements are marked with the **Failure** stereotype and either damage one **Component** (random failures) or more (common cause failures). For each element an appropriate **Failure** or **CCF** element is created in the Intermediate Model. The attributes of the created elements receive their values by copying the attributes of the original UML element. The failures defined in Section 1.5.2 are defined in the UML design on Figure 4.6 along with the created elements in the IM. The **damages** association relations are also created between the failure and the associated **Node**.

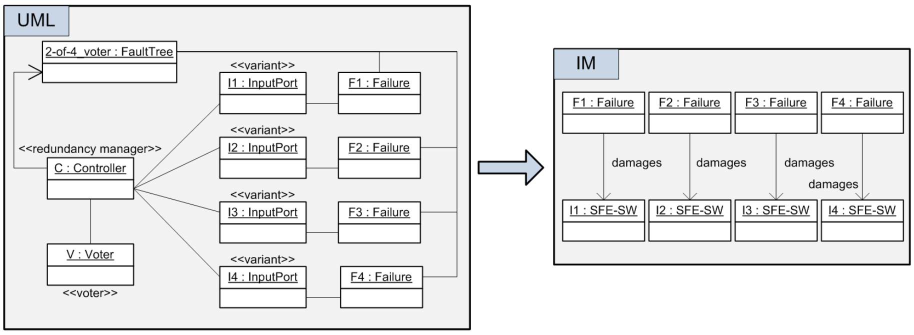

Figure 4.6: *Failure stereotypes and the created IM elements*

4.2.3 Creating Maintenance Elements of the IM

The maintenance part of the Intermediate Model is created by examining the elements with **Component** stereotype. If the component is connected to a **MaintenancePolicy** through the **mainPolicy** attribute then a **Maintenance** element is created in the IM. The type of the element is **Corrective** if the **type** attribute of the originating UML element contains a **CorrectiveM** element and **Preventive** if it contains a **PreventiveM** element. The **composition** relation is created between the associated IM element and the **Maintenance** element. The attribute values are copied from the attributes of the **type** element. Figure 4.7 shows how the policy defined for the channels segment is represented in the IM.

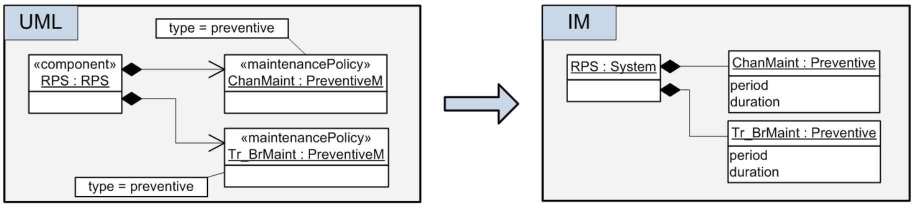

Figure 4.7: *Maintenance stereotypes and the created IM elements*

Each maintenance policy contains a set of activities describing the various maintenance tasks performed on the components associated with them. For each **Activity** element belonging to a given **MaintenancePolicy** an **Activity** element is created in the IM. The type of the activity can be **Repair**, **Replace** or **Overhaul** in the UML design and the same element types are present in the Intermediate Model as well. The attributes of the created activity elements are copied from the originating UML element. A **composition** relation is created between the maintenance policy and each activity it contains. Finally the association relation is created between the **Activity** element and the **Node** representing the associated component in the IM. The IM representation of the maintenance activities defined for the train-breakers segment of the RPS is illustrated on Figure 4.8.

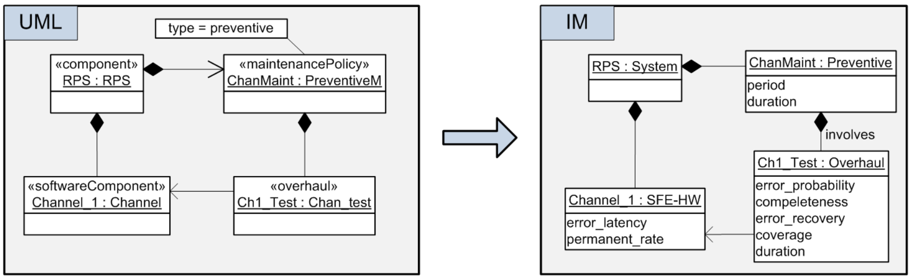

Figure 4.8: *Maintenance activities and the created IM elements*

4.2.4 Creating Relations between Structural Elements

The elements in the UML design can be connected by many relations which all represent some kind of connection between the elements they link. A large portion of these relationships may imply an error propagation paths between the end elements. Those relations that are considered as a propagation path by the designer are marked with the **Relation** stereotype. Certain UML relation elements imply a bidirectional propagation paths when the errors may propagate from both ends of the relation while others represent a propagation path in only one direction. The elements that can be marked with the **Relation** stereotype are listed in Table 4.1.

The UML model elements with **Relation** stereotype are examined and the **uses** relation is created in the IM for every marking where the value of the <propagation_probability>

Metamodel element	**Direction)**
Generalization with stereotype «extends» or «uses»	Direction from *supertype* to *subtype*
Dependency with stereotype «calls» or «uses»	Direction from *client* to *supplier*
Association with an AssociationEnd having *aggregate* attribute	Direction from the aggregate end to the normal one
Association with an AssociationEnd having *composite* attribute	Direction from the composite end to the normal one
Association (otherwise)	Bidirectional
Link having LinkEnd as instance of an AssociationEnd with *aggregate* attribute	Direction from the aggregate end to the normal one
Link having LinkEnd as instance of an AssociationEnd with *composite* attribute	Direction from the composite end to the normal one
Link (otherwise)	Bidirectional
Message	Direction from *sender* to *receiver* if the receiver is stateless, otherwise bidirectional
Relation stereotype from System profile	Bidirectional

Table 4.1: *UML elements implying error propagation paths*

attribute of the **uses** relation is copied from the attribute of the originating stereotype. The relations marked as error propagation paths in the RPS are illustrated on Figure 4.9 with the **uses** relations created in the IM.

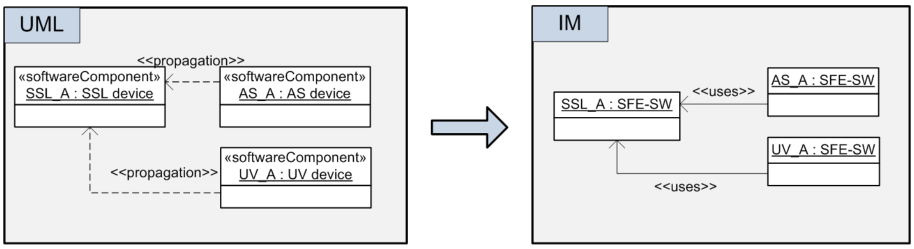

Figure 4.9: *Uses relations and the created IM elements*

4.2.5 Automatic Fault-tree creation

The fault-trees defined for the fault-tolerance structures specify the behavior of the structure as a whole when its contained components fail due to errors. These fault-trees describe which components can fail at the same time before the service provided by the structure becomes unavailable. As already discussed these fault-trees can be selected from a library if a known redundancy scheme is used in the structure. However the fault-trees can be created also automatically when the library is not used. In this case the fault-trees are constructed using the failures and fault-tolerance roles defined for the components and the additional relations between the composing elements.

Similar algorithms are defined in [11] by analyzing the statechart diagram of the redundancy manager and [39] by analyzing the state machines for reliability modeling.

The construction of the fault-trees is performed by examining the UML design and approximating the redundancy scheme of the fault-tolerance structure. First the role of the components in the structure is identified then the failures associated with the components are collected which will be represented by the leaf elements of the fault-trees. The elements marked with the **Relation** stereotype are used to identify additional connections between the components with different roles.The fault-trees created based on the UML elements and their relations are illustrated with an example on Figure 4.10. The `<redundance_level>` attribute of the **FTS** element is used for the cases when the fault-tree is generated based on the failure of several components.

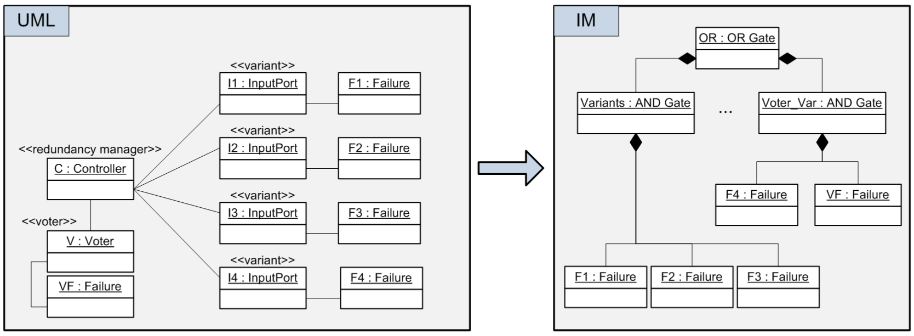

Figure 4.10: *Automatic Fault-tree generation*

The automatically created fault-trees are constructed according to the following guidelines:

- **Failure of redundancy manager** causes the structure to fail because it provides the service toward the rest of the system. If there is a **Failure** defined that damages the **Component** with **redundancy manager** stereotype, the created fault-tree includes an **OR gate** that contains the failure of the manager component and the rest of the fault-tree.

- **Failure of variant without adjudicator** causes the structure to fail as there is no mechanism defined to detect and suppress the failure of the variant. The created fault-tree is an **OR gate** containing the failure of the variant and the rest of the fault-tree. Note that this **OR-gate** can be the same as the one mentioned above.

- **Failure of a variant and its tester** causes the structure to fail if there is no further adjudicators connected to the variant. The created fault-tree is an **AND gate** containing the separate failures of the variant and the tester. If a common cause failure is defined for the variant and the tester an **OR gate** is created that contains the **AND gate** defined for the separate failures and the **CCF** defined for the common cause failure.

- **Failure of variants that are compared** can cause the structure to fail if the simultaneous failure of the variants renders the comparator unable to suppress the error. The created fault-tree is an **AND gate** containing the separate failures of the variants. If a common cause failure is defined for the variants an **OR gate** is created that contains the **AND gate** defined for the separate failures and the **CCF** defined for the common cause failure.

- **Failure of variant and its comparator** can cause the structure to fail because the failure of the comparator implies that the failure of the variant escapes notice. The created fault-tree is an **AND gate** containing separate failure of the variant and the comparator. If a common cause failure is defined for the variant and the comparator an **OR gate** is created that contains the **AND gate** defined for the separate failures and the **CCF** defined for the common cause failure.

- **Failure of the majority of the variants with a voter** can cause the structure to fail because the voter is unable to decide following the majority of the variants. The created fault-tree is an **OR gate** containing the **AND gates** for the different variations of which variants fail. These **AND gates** contain the separate failures of the variants. If common cause failures are defined for the variants then those are included in the **AND gates** as well by creating more variations.

- **Failure of a variant and its voter** causes the structure to fail because the voter is unable to perform a correct voting. The created fault-tree is an **AND gate** containing the separate failures of the variant and the voter. If common cause failures are defined for the variant and the voter an additional **OR gate** is created that contains the **AND gate** defined for the separate failures and the **CCF** defined for the common cause failure.

4.3 Intermediate model from non-functional service contract

In order to include the dependability and maintenance properties of a used service in the IM the non-functional service contract of the target service has to be examined. First the **ProviderDependability** characteristic is used to create the fault process and error propagation of the service then the **ProviderMaintenance** characteristic is used to create the maintenance policies and activities associated with the service.

4.3.1 IM elements for the Provider dependability characteristic

The service itself is modeled as a software node, if it is stateless then as an **SLE-SW** element otherwise as a **SFE-SW** element. The **FailureMode** dimension represents how the failure of the service affects its operation. If the `<fail-silent>` attribute is true then a **uses** association is created between the requester component and the service while the value of the `<propagation>` attribute is used for the probability of error propagation. Additionally the **Failure** element representing the possible failure of the service is created based on the **Availability** dimension. The fault occurrence rate is equal to the reciprocal of the *mtbFailure* attribute as the time between failures has to be transformed into the number of failures in a given time. Figure 4.11 illustrates how the UML elements are represented in the IM.

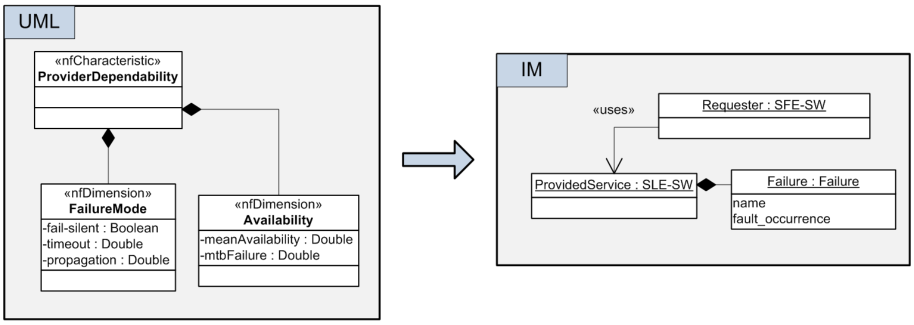

Figure 4.11: *Provider dependability and the created IM elements*

4.3.2 IM elements for the Provider maintenance characteristic

Correction mode dimension If the **CorrectiveMode** dimension is present in the non-functional service contract then the service has a corrective maintenance policy defined. This policy is represented by a **Corrective** element in the IM with the value `<error_detection>` attribute can be gathered from the UML element. Furthermore a **Repair** element is created to represent the maintenance activity for the service. The attributes of the **Repair** element are filled with the appropriate attribute values from the UML element. The IM elements created from the UML elements are illustrated on Figure 4.12.

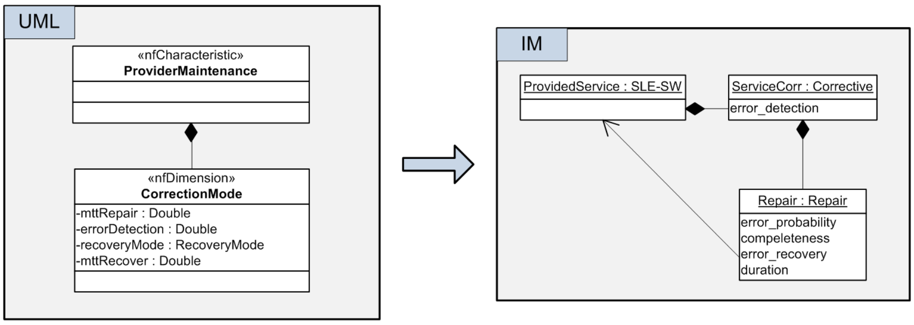

Figure 4.12: *Provider correction mode and the created IM elements*

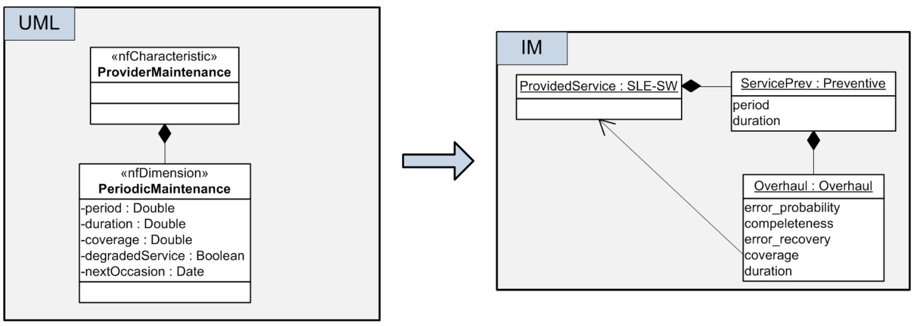

Figure 4.13: *Provider periodic maintenance and the created IM elements*

Periodic maintenance dimension Similarly to the corrective mode the **PreventiveMaintenance** dimension is represented by a **Preventive** and an **Overhaul** element, with the attributes getting the proper values from the dimension. Both maintenance policies are contained in the **SLE-SW** node representing the service and the defined activities are associated with it as well. Figure 4.13 illustrates the generated IM elements from the UML model.

In this chapter the intermediate model used as a transition point between the source UML models and the target dependability models were defined along with the steps of creating this intermediate model from UML models. The created intermediate model is used to create the dependability model and its creation is described in the next chapter. It is important to note, that the steps defined here are capable of creating the intermediate model only from UML models using the profile defined in Chapter 3.

Chapter 5

From the Intermediate Model to the Dependability Model

As discussed in Section 1.4.2 Multiple-phased systems consist of two main parts. The Phase Net is used to represent the various scheduled phases that are executed as part of the maintenance of the system. While corrective maintenance is executed at the time of error or failure detection, preventive maintenance is carried out on a regular basis using a predefined schedule. The algorithm for creating the Phase Net is described in Section 5.1. The System Net on the other hand describes the behavior of the system and its components which are subject to malfunctions, errors, failures and maintenance activities. The construction of the System Net based on the IM is defined in Section 5.2.

5.1 Deriving the Phase Net from the IM

In the UML diagrams of the designed system, the preventive maintenance policies are defined for different system parts. While this is useful for describing the maintenance in a distributed and thus less complex way, it can not be used as is when modeling the system as a whole. In order to create the Phase Net of the modeled system an algorithm is defined for the integration of preventive maintenance policies in the system design. The algorithm uses a table to gather the phases required to plan a perfectly staggered schedule for the maintenance of the components. This table will referred to as the Maintenance Schedule Table (MST). A perfectly staggered schedule (i.e. the components of the same fault-tolerant structure are tested at separate times) is in most cases less compromising to system availability then simultaneous maintenance (i.e. all the components are tested at the same time, one after the other).

5.1.1 Definition of the Maintenance Schedule Table(MST)

The phases of the system operation can be separated in two types, phases of full redundancy (when no component is under maintenance) are divided by phases of maintenance (when a given component or components are under maintenance). The MST contains the phases of the second type, identifying the components under maintenance, the period, duration and starting time of their maintenance. The entries on the table are sorted based on the starting time and entries with overlapping maintenance phases are shifted (their starting time changed) to avoid the loss of more redundancy than required. Figure 5.1 shows the MST for the components in the case study.

Component(index)	Period (h)	Duration (h)	Starting time (h)
Channel(1)	2160	4	540
Train_breaker(1)	1440	2	720
Channel(2)	2160	4	1080
Train_breaker(2)	1440	2	1440
Channel(3)	2160	4	1620
Train_breaker(1)	1440	2	2158
Channel(4)	2160	4	2160
Channel(1)	2160	4	2700
Train_breaker(2)	1440	2	2880
Channel(2)	2160	4	3240
Train_breaker(1)	1440	2	3600
Channel(3)	2160	4	3780
Train_breaker(2)	1440	2	4318
Channel(4)	2160	4	4320

Figure 5.1: *An example of the Maintenance Schedule Table*

5.1.2 The MST creating algorithm

In order to create the MST, first every **Preventive** maintenance element is gathered from the IM. The length of a full maintenance cycle (M) will be the *least common multiple* of all the elements found this way (5.1). This results from the notion that after that time the maintenance of the system will be carried out in exactly the same schedule as before. Trivially the starting time of the last entry in the MST will be M and overlapping entries will appear in the end of the table.

$$M = LCM(p_1, p_2, ...p_n),\ p_k = maintenance\ period\ of\ element\ k \tag{5.1}$$

Entry creation step The next step is to populate the table with the entries representing the maintenance phases of the components. Suppose there are n variant **Nodes** handled by a given **Preventive** maintenance policy. Given that M is divisible by p_k, $\frac{M}{p_k}$ is a positive integer and it is exactly the number of times a component is maintained during the complete cycle as part of the target policy. In this case exactly $\frac{M}{p_k}n$ new entries are added to the table with the following values: the name of the element and a running index (i) together identifies the component, the period p_k is copied from the **Preventive** element and the duration (d_k) can be obtained from the **Activity** defined for the element. The starting time ($start_k$) is calculated with the formula (5.2).

$$start_k(i) = \frac{p_k}{n}j,\ j = 1, \ldots, \frac{M}{p_k}n,\ i = j \pmod{n} \tag{5.2}$$

By repeating the *entry creation step* for every **Preventive** policy the entries of the MST are created. As already stated, overlapping entries will appear in the table, if there is at least two policy defined. In order to eliminate overlapping, the entries in question are shifted backwards until the starting time of each entry is greater than the starting time of the entry before by at least the duration of the maintenance of the earlier entry (5.3).

$$start_k(n) \geq start_j(n) + d_j,\ \forall j \neq k \tag{5.3}$$

Overlap elimination step To achieve this, take the entry among the overlapping ones with the second maximum duration (d^2_{max}). Shift every overlapping entry except the one with the maximum duration backwards with d^2_{max}. This step assures that the entry left in place does not overlap with any entry. By repeating the procedure with the still overlapping entries, the number of entries decreases with every step except if shifting the entries result in a new entry to overlap the shifted ones. If the sum of all the durations is less than M, then ultimately no entries will overlap and the procedure stops. Otherwise the procedure stops with the fewest possible overlapping entries. The algorithm can be refined to deal with non-solvable overlapping by making sure that entries for the same component and the same fault-tolerant structure don't overlap. This solution is not specified at this time.

5.1.3 Production of the Phase Net from the MST

The creation of the Phase Net from the MST is done by defining the Petri Net parts for each entry in the table and connecting these parts with the necessary arcs. The Phase Net created from the table in Figure 5.1 is shown on Figure 5.2. The starting phase of system operation is not part of the full maintenance cycle. As the first step, a **Start** place is defined with one initial token. Next, a timed transition **t_Start** is placed with the deterministic firing time equal to $start_k(1)$ from the first entry of the MST, and an inbound arc from **Start**.

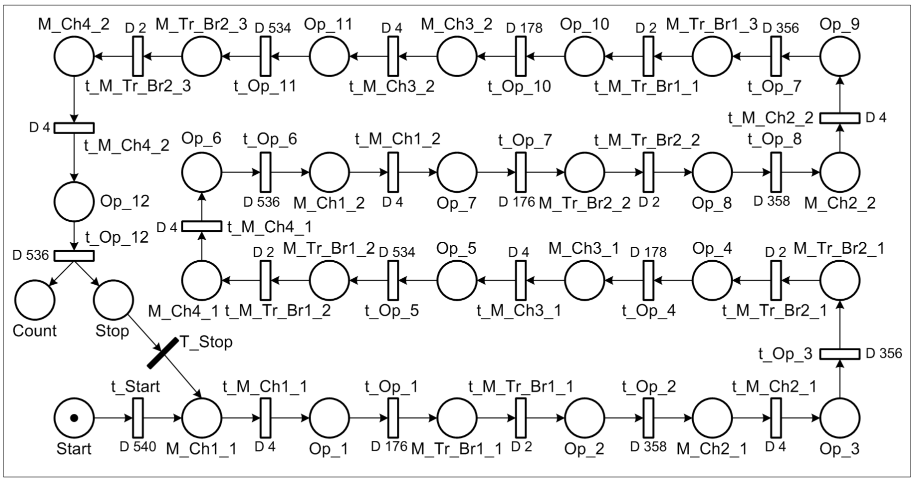

Figure 5.2: *The Phase Net created from the MST*

Phase Net construction step Next, the places **M_Comp_k_1** and **Op_1** representing respectively the maintenance of the component identified by the first entry and the normal operation after it are created. An outbound arc from **t_Start** to **M_Comp_k_1** is added. Then a timed transition **t_M_Comp_k_1** representing the end of maintenance is placed with the deterministic firing time equal to d_k from the first entry. After that an inbound arc from **M_Comp_k_1** and an outbound arc to **Op_1** are added for this transition. Finally a timed transition **t_Op_1** representing the end of normal operation is created, its deterministic firing time ($operate_k^1$) equal to the difference between the ending time of the actual maintenance activity and the starting time of the next entry ($Comp_j(1)$, if there is one) in the table (5.4), and an inbound arc is added from **Op_1**. If $operate_k^1$ equals 0, then the transition is not created because an other maintenance phase comes right after the current.

$$operate_k^1 = start_j(1) - (start_k(1) + d_k) \tag{5.4}$$

Repeat the *Phase Net construction step* for every entry in the MST, naming the places representing the maintenance periods with with the name found in the table and the operation periods with an increasing index. The first outbound arc always leads from the last transition (**t_Op_1**) to the first place (**M_Comp_j_1**).

For the last entry, the firing time of the last transition (**t_Op_N** is calculated with the same formula but the following parameters, $start_k(1) = 0$ and $start_j(1)$ equal to the starting time of the first entry. Finally the places **Count** and **Stop** are created representing respectively the number of full cycles completed by the system and the end of the cycle. Inbound arcs from the transition **t_Op_N** are added to these places. Then an immediate transition **T_Stop** representing the restarting of the cycle is placed with the enabling function $Mark(Count) < max\_count$, that stops the operation when the number of full cycles completed reaches a predefined limit. In the end an inbound arc from **Stop** and an outbound arc to **M_Comp_k_1** are added to the Phase Net, closing the cycle.

5.2 Creating the System Net subnets from the IM

The dependability model is generated by examining the Intermediate Model. Given that the IM always has at least one **System** element as root, the construction of the System Net starts from these root elements and the contained elements are dealt with iteratively. For every **Node**, **System** and **FTS** element in the IM, a basic subnet is created with the possible states the component or structure can have. Then the fault-tolerance structures are parsed to create the Petri Net match of the fault-trees and failures. Next the maintenance policies and their contained activities are handled and the subnets representing them are created. Finally, the propagation subnets for every **uses** hyperarc in the IM are generated.

5.2.1 Basic subnets

The places of the basic subnets represent the states an element can be in during normal operation (i.e. not under maintenance). Every element starts in the **Healthy** state representing correct operation and they may change to the **Failed** state due to a failure of the component or its contained components. The subnet containing these places is shown on Figure 5.3.

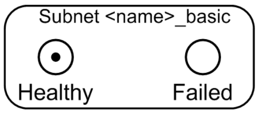

Figure 5.3: *The basic subnet of stateless nodes, System and FTS elements*

Additionally if the component is stateful then it has an **Erroneous** state that represents the presence of an internal error that may cause the element to fail. The transition **t_latency** is a timed transition with firing properties given by the `<error_latency>` attribute of the given node. This transition is enabled if there is a token in the **Erroneous** place and generates tokens in the **Failed** place by firing. This means that an internal error may cause multiple faults in the component. The basic subnet for stateful components is shown on Figure 5.4.

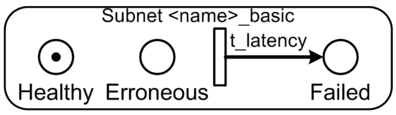

Figure 5.4: *The basic subnet of stateful nodes*

5.2.2 Fault-tolerance structure subnets

Every **FTS** element contains one or more fault-trees which in turn are constructed from logical operators (*gates*) and failures as described in Section 4.1.2. Also, an implicit fault-tree can be assigned to **System** elements. This fault-tree consists of a single **OR gate** with a **Failure** element for each contained component. Fault-trees in the IM are transformed to two Petri Net subnets, one represents the failure of the structure as a result of the failure of components, the other describes the effect that the repair of components allows the structure to function correctly again. The later one is referred to as the *repair tree* from now on. The repair tree is the dual of the fault-tree in the sense that it can be obtained by changing the **OR gates** to **AND gates** and vice versa. The inclusion of these subtrees is illustrated on Figure 5.5.

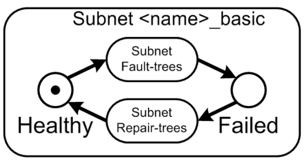

Figure 5.5: *The basic subnet with fault-trees and repair-trees included*

Failure subnets

The fault-trees of the **FTS** elements can contain several gates but the leaves of the tree at the lowest level are **Failure** or **CCF** elements. These elements are associated with one or more (in case of the CCF) **Nodes** and the basic subnets of these nodes are extended to include the representation of the failure mechanism. In addition for every CCF element a new subnet is created which controls the occurrence of the common cause failure and enables the transitions in the basic subnet of the associated nodes.

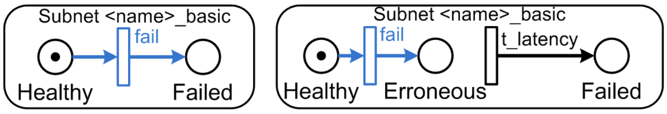

Figure 5.6: *The basic subnets with failure transition included*

First the **Failure** elements are handled by looking up the basic subnet of the associated node. If the basic subnet does not have a failure transition defined then a new timed transition **fail** is created with an inbound arc from the **Healthy** place and an outbound arc to the **Erroneous** or **Failed** place depending on whether the node is stateful or stateless. The firing rate of the transition is equal to the `<fault_occurrence>` attribute of the **Failure** element. On the other hand if the transition is already present in the subnet the `<name>` attribute of the **Failure** element is checked. If a **Failure** element with the same name and associated node has already been handled then the subnet is not changed. However if no corresponding **Failure** element is found then the firing rate of the **fail** transition is changed to include the occurrence of the actual failure. This is done by a simple addition operation between the current firing rate and the `<fault_occurrence>` attribute of the **Failure** element. The basic subnets with the included transition are shown on Figure 5.6.

Next the subnet for **CCF** elements is generated in the following way. A *common cause failure* means that several nodes fail at the same time due to the same cause. As with the **Failure** elements if a **CCF** element with the same `<name>` attribute and associated nodes has been handled already the System Net is not changed. In order to correctly model the simultaneous nature of the CCF a separate subnet is created to handle the possible stages of the failure and the subnets of the corresponding nodes are extended with a transition that represents the effect of the CCF. Three different stages are defined in the created subnet, the **CCF_No** place represents normal operation stage before the CCF happens, the **CCF_Yes** place represents that the CCF occurred while the **CCF_UM** place represent the stage when the associated nodes are under maintenance after the failure. The places are connected with three transitions which are the following:

- The timed transition **ccf_fail** represents the occurrence of the CCF, it has an inbound arc from **CCF_No** and an outbound arc to **CCF_Yes**. The firing rate is equal the the `<fault_occurrence>` attribute of the **CCF** element.
- The immediate transition **ccf_maintain** represents that maintenance is started for the associated nodes. It has an inbound arc from **CCF_Yes** and an outbound arc

to **CCF_UM**. The transition is enabled if at least one of the associated nodes is under maintenance.

- The immediate transition **ccf_repair** represents that maintenance is finished and the effect of the CCF is removed thus an other CCF can occur in the future. The transition has an inbound arc from **CCF_UM** and an outbound arc to **CCF_No**. The transition is enabled if every associated node is in the **Healthy** state.

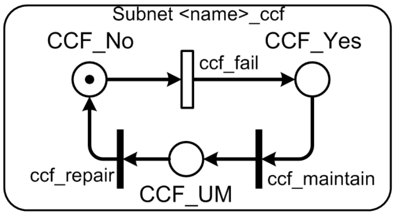

Figure 5.7: *The subnet for common cause failure*

Finally the basic subnet of the related nodes are extended with the immediate **ccf** transition. It has an inbound arc from the **Healthy** place and an outbound arc to the **Erroneous** or **Failed** place depending on whether the node is stateful or stateless. The transition is enabled if there is a token in the **CCF_Yes** place in the CCF subnet. If the **ccf** transition is already present in the basic subnet then its enabling function is extended so that it can fire if there is a token in the **CCF_Yes** place of the actual CCF subnet. The created CCF subnet is illustrated on Figure 5.7 while the extended basic subnets of the associated nodes are shown on Figure 5.8.

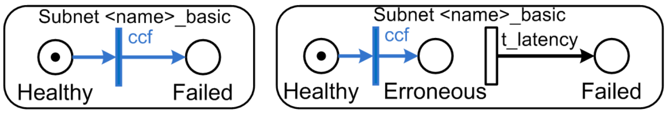

Figure 5.8: *The basic subnets with common cause failure transition included*

Fault-tree subnet

The construction of the **failure** subnet from the fault-tree is presented through an example. On Figure 5.9 the fault-tree and the generated subnet are illustrated. The **FT** structure fails if either components **A** and **B** have failed at the same time or if **C** fails. The corresponding subnet is created using the following rules:

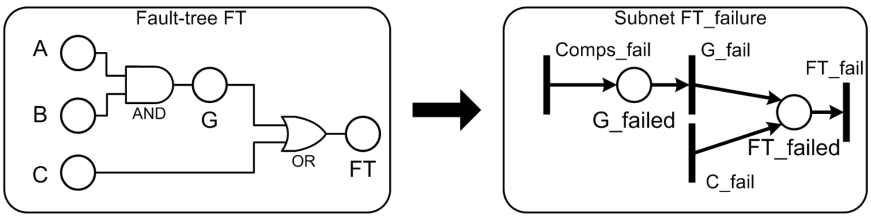

Figure 5.9: *Fault-tree transformed to failure propagation subnet*

- **Root transition:** Create an immediate transition that represents the failure of the structure caused by the failure of the components (**FT_fail**).

- **OR gate:** Create a place representing the event that at least one of the subtrees of the gate failed (**FT_failed** and create an immediate transition for each subtree the gate connects to (**G_fail**, **C_fail**). These transitions represent the failure of the subtree. Connect the transitions to the place and the place to the transition of its parent (**FT_fail** on the example). Furthermore, the place **FT_failed** has a capacity limit of 1.

- **AND gate:** Create a place representing the event that all of the subtrees of the gate failed (**G_failed** and create an immediate transition to which each subtree of the gate will connect to (**Comps_fail**). This transition represent the failure of every subtree. Connect the transition to the place and the place to the transition of its parent (**G_fail** on the example). Also, the place **G_failed** has a capacity limit of 1.

The resulting subnet is a fair representation of the fault-tree, but the basic subnets of the components and the structure has to be linked in as a final step. Each **Failure** element in the fault-tree is associated with one **Node** which it damages while **CCF** elements are associated with several nodes. The **Failed** place of each damaged node is connected in both directions to the appropriate transition created in the failure subnet when handling the gates. In the case of a CCF element, the **CCF_Yes** place is connected. Moreover, the **Healthy** and **Failed** place of the basic subnet of the structure is connected to the root transition, changing the state from healthy to failed. On Figure 5.10 the connections are illustrated.

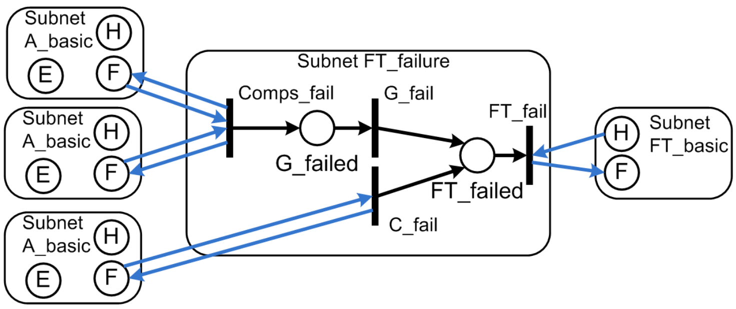

Figure 5.10: *Basic subnets connected to the failure propagation subnet*

Repair-tree subnet

The construction of the **repair** subnet is done by transforming the repair-tree in mostly the same way as it was done with the fault-tree. The differences are the following:

- The root transition is called **FT_repair**. And it changes the state of the structure from **Failed** to **Healthy** when firing.

- The places created for gates are called **X_repaired** while the transitions are named **X_repair**.
- The **Healthy** place from the basic subnets of the damaged nodes and the **CCF_No** place from the **CCF** subnet are connected in both directions to the transitions.

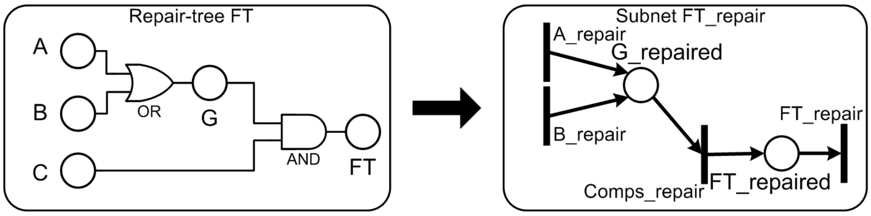

Figure 5.11: *Repair-tree transformed to repair propagation subnet*

The transformation of the repair tree is illustrated on Figure 5.11 while the connections between the basic subnets and the repair subnet are shown on Figure 5.12.

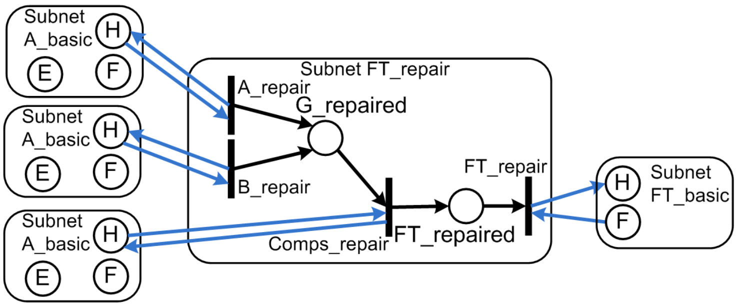

Figure 5.12: *Basic subnets connected to the repair propagation subnet*

Connecting the fault-tree subnets

In order to completely define the connection between the failure and repair subnets a final step is required. The current definition of the subnets is problematic for the following reason: although the places in the subnets have a limited capacity, the number of tokens in the subnet at any given time varies due to the non-deterministic firing of the concurrent immediate transitions. This means that although the nodes connected to a given gate have changed state, the token representing the active state of the gate remains in place. In order to absorb the unnecessary tokens in the subnets a set of inbound arcs are added between the corresponding failure and repair subnets. The following arcs are added to the subnets:

- For each transition that has an outbound arc to each place **X_failed** in the failure subnet **FT_failure**, add an inbound arc from the place **X_repaired** in the repair subnet **FT_repair**.
- For each transition that has an outbound arc to each place **X_repaired** in the repair subnet **FT_repair**, add an inbound arc from the place **X_failed** in the failure subnet **FT_failure**.

These arcs make sure that at any given time, the tokens in the subnets represent the active state of the corresponding gate in the fault or repair-tree.

5.2.3 Maintenance subnets

The Intermediate Model provides several elements which makes it possible to model complex maintenance policies and customized activities. The construction of the subnets representing these elements are described in this section. First the subnet controlling the preventive maintenance policy execution is described then the subnets for the various activities are explained.

Maintenance policy subnet

As already described in Section 4.1.3 the two policy types are handled differently. While the corrective maintenance is executed when the failure is detected, the preventive maintenance is carried out periodically. In Section 5.1 the generation of the Phase Net based on the **Preventive** maintenance policies in the IM is described. In addition to the phases, an other subnet is created to represent the activation of a given policy. It contains two places, the **No_M** place representing that no maintenance is carried out at the time and the **Yes_M** place representing that maintenance of elements is under way. The places are connected with two immediate transitions, the first is the **start_M** transition that represents the beginning of the maintenance. It is enabled when the Phase Net has a token in one of the appropriate **M_Comp_X_i** places (preventive policy) or if the associated element has failed hence has a token in the **Failed** place (corrective policy). Second is the **end_M** transition representing the end of the maintenance. It is enabled when there is no token in the aforementioned Phase Net places (preventive policy) or if the associated element is working correctly hence has a token in the **Healthy** place (corrective policy). The first transition has an inbound arc from **No_M** and an outbound arc to **Yes_M**. The second transition has an inbound arc from **Yes_M** and an outbound arc to **No_M**. The **policy** subnet is illustrated on Figure 5.13.

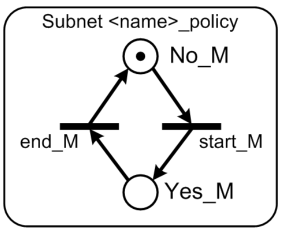

Figure 5.13: *Maintenance policy transformed to subnet*

Maintenance activity subnet

Although the **Activity** element of the IM is abstract and thus it is never instantiated, the generic subnet part is presented here because it is the same for every concrete activity, the subnets for which are described in detail afterwards. The **activity** subnet of a maintenance activity differs based on the type of its associated **Node**. For stateless nodes error recovery is not necessary thus now we only define the place **Pre_UM** which represents the state when the component is scheduled for maintenance but the actual activity is not started yet. There are two immediate transitions, **start** and **clear**. The first represents the activation of the maintenance process and has an outbound arc to **Pre_UM**. It is only enabled when there is no token in the **Pre_UM** place so that failures generated before the start of the actual maintenance are handled together. The second is enabled when there is a token in **Pre_UM** and represents gathering of failures before the execution of the activity. Additionally if the activity is part of a **Preventive** maintenance then the transitions are only enabled if the **policy** subnet of the policy is in the **Yes_M** state.

Stateful nodes require error recovery after maintenance to return to correct operation. The place **ER** represents the state when the activity is completed and error recovery commences. The timed transition **recover** represents the completion of error recovery, it has an inbound arc from **ER** and its firing rate is equal to the `<error_recovery>` attribute of the **Activity** element.

The subnets are connected to the basic subnets of the associated nodes with several arcs. The **start** and **clear** transitions have inbound arcs from the **Failed** place. The **recovery** transition has an inbound arc from the **Erroneous** place and an outbound arc to the **Healthy** place. These arcs are shown on Figure 5.14.

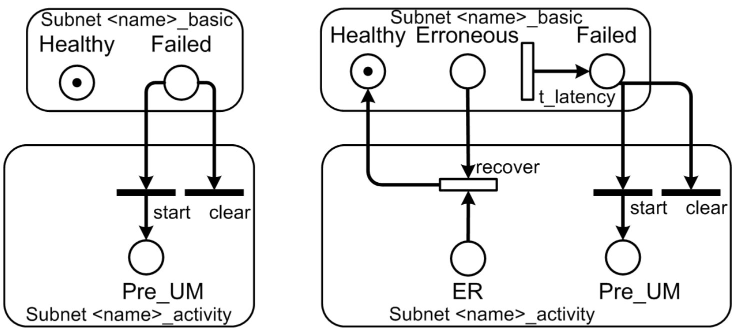

Figure 5.14: *Basic subnets connected to the activity subnet*

Error detection, completeness and error probability attributes

The `<error_detection>` attribute of the **Corrective** element indicates that the detection of internal errors of components is possible to a certain degree. If this attribute is given then the subnets of the activities contained within the given policy are extended with the **Not_D** place representing that the actual error in the component has not been detected by the activity and three transitions. The **detect** and **not_detect** immediate transitions represent the non-deterministic choice whether the error is detected or not. They are enabled concurrently and their firing probability is d and $1 - d$ respectively where d is equal to the aforementioned `<error_detection>` attribute. An outbound arc is added from **detect** to the **Pre_UM** place and from **not_detect** to the **Not_D** place. The immediate **clean** transition represents that another error can be detected again if the earlier one has been corrected. It has an inbound arc from **Not_D** and is enabled if the associated node is no longer in **Erroneous** state.

To connect the basic subnets of the nodes to the **actvity** subnet inbound and outbound arcs are added to both **detect** and **not_detect** transitions from and to the **Erroneous** place. The subnets with the required arcs are illustrated on Figure 5.15.

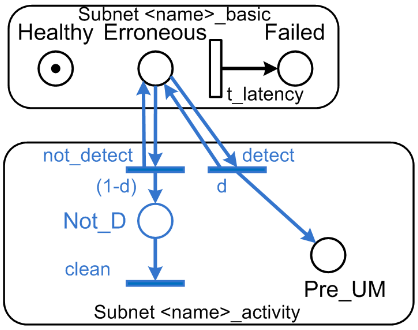

Figure 5.15: *Basic subnets connected to the error detection subnet*

The `<completeness>` attribute of the **Activity** is used to give the probability that the maintenance is carried out before the component fails. While error detection provides a probability that the error is detected, completeness indicates a probability that the error is successfully corrected before a failure occurs. The **activity** subnet that is shown also on Figure 5.16 is thus extended with the following:

- A new **No_F** place is created to represent that no failure has occurred before the maintenance began. The **s_3** immediate transition represents the beginning of maintenance to correct an error, it has outbound arcs to **No_F** and **Pre_UM**. It is enabled only when there is no token in the **Failed** place of the node.
- For every transition with outbound arcs to the **ER** place, two additional immediate transitions are created **found** and **not_found** which represent the non-deterministic choice whether the error is found during the maintenance or not. Their firing probability is *comp* and $1 - comp$ respectively where *comp* equals the `<completeness>` attribute of the **Activity** element. The inbound arcs of the original transition are copied for the new transitions and an inbound arc from **No_F** is added to both transitions. Moreover an outbound arc to **ER** is added to **found**. The original transition can only fire if the **No_F** place is empty.
- When connecting the basic subnets to the activity subnet, an inbound arc is added from the **Erroneous** place to **s_3** and an outbound arc to the same place is added to **not_found**.

The `<error_probability>` attribute of the **Activity** element represents the probability that a maintenance activity creates an error or failure in the component instead of correcting any. The **activity** subnet is extended by duplicating every transition that: (1)

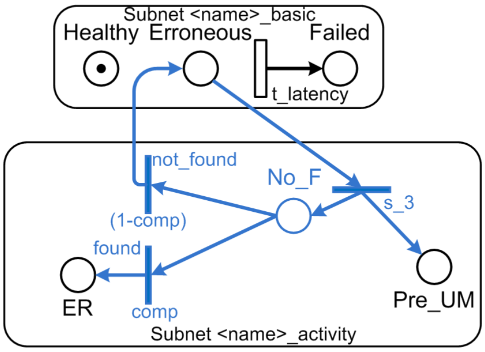

Figure 5.16: *Basic subnets connected to the completeness subnet*

has an outbound arc to the **Healthy** place of the associated stateless node, (2) has an outbound arc to the **ER** place. These transitions (**m_succ**) represent the end of maintenance execution and their created duplicates (**m_fail**) represent the failure of component during the maintenance. The firing probability of transitions **m_fail** and **m_succ** is ep and $1 - ep$ respectively where ep equals the aforementioned `<error_probability>` attribute. An outbound arc to the **Failed** or **Erroneous** place is added whether the node is stateless or stateful. The extended subnet and the connections are illustrated on Figure 5.17.

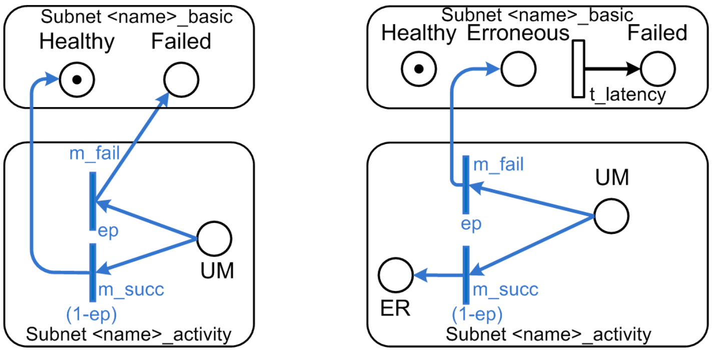

Figure 5.17: *Basic subnets connected to the error probability subnet*

Repair activity subnet

If the activity instance is a **Repair** element, the **activity** subnet is extended with the actual representation of the maintenance. The extended subnet (**repair**) includes two new places (**UM_p** and **UM_t**) and four transitions (**permanent**, **transient**, **maintain** and **reset**). The **UM_p** place represents the execution of explicit repair on the component while **UM_t** represents implicit repair. The concurrent **permanent** and **transient** immediate transitions represent the non-deterministic choice whether the actual failure is caused by a permanent or a transient fault. Their firing probability is $perm$ and $1 - perm$ respectively where $perm$ equals the `<permanent_rate>` attribute of the associated **Node** element. They both have inbound arcs from **Pre_UM** and outbound arcs to **UM_p** and **UM_t** respectively. The timed transition **maintain** represents the completion of explicit repair, its firing rate equal to the `<duration>` attribute of the **Repair** element and has an inbound arc from **UM_p**. The immediate transition **reset** represents the completion of the implicit repair and has an inbound arc from **UM_t**. If the associated node is stateful both transitions have an outbound arc to **ER**.

The basic subnets of the associated nodes are connected with a few outbound arcs in the case of stateless elements. Outbound arcs to the **Healthy** place are added to the **maintain** and **reset** transitions. The connected subnets are shown on Figure 5.18.

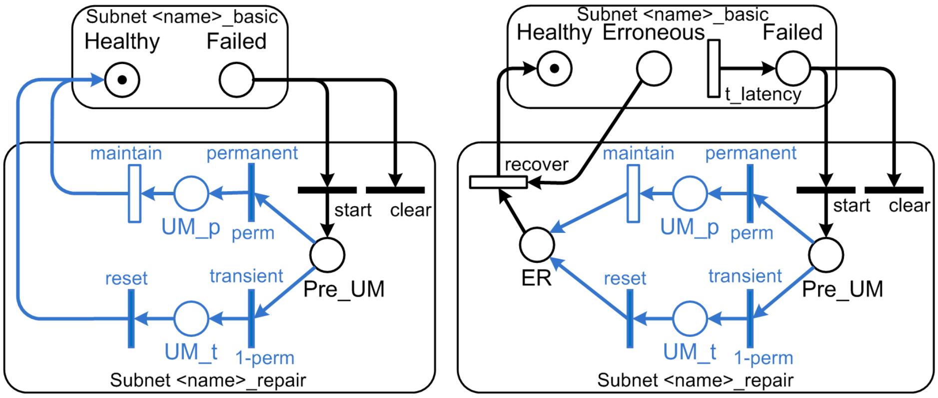

Figure 5.18: *Basic subnets connected to the repair subnet*

Replace activity subnet

The extended subnet for **Replace** elements contains an additional place and two more transitions compared to the **activity** subnet. The **UM** place represents the execution of the maintenance (i.e. the replacing of the component) while the immediate transitions **remove** and **replace** represent the removal of the failed and the replacement of the new component. The **remove** transition has an inbound arc from **Pre_UM** and and outbound arc to **UM** while the **replace** transition has an inbound arc from **UM** and an outbound arc to **ER** if the associated node is stateful or to the **Healthy** place of the node if it is stateless. The **replace** subnet and the connections with the basic subnets are shown on Figure 5.19.

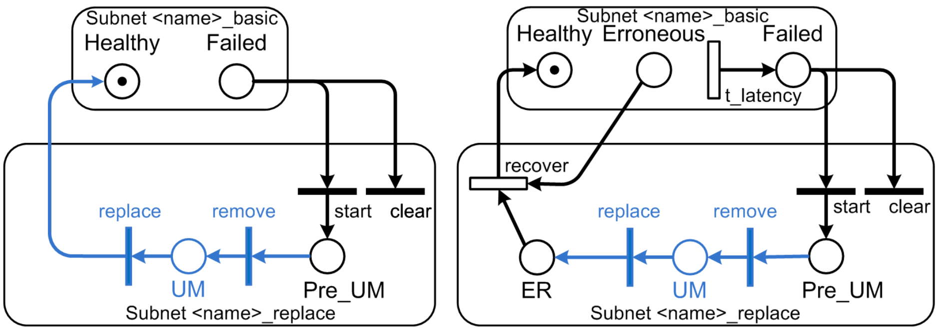

Figure 5.19: *Basic subnets connected to the replace subnet*

Overhaul activity subnet

While both the **Repair** and **Replace** activities indicate maintenance that is only carried out if the components failed the **Overhaul** activity is used for maintenance executed regardless of component state. It is mostly used in preventive policies as a regular test and maintenance for running components. However this characteristic also implies that the activity can miss a failure and finish the maintenance without correcting it. The **activity** subnet is extended with the **UM** place and several transitions. The **UM** place represents that the testing is finished and the failures are corrected if found. The timed transition **check** represents the execution of tests and the correction of failures. It has an inbound arc from **Pre_UM** and an outbound arc to **UM**, its firing rate is equal to the `<duration>` attribute of the **Overhaul** element. The immediate transitions **maintain** and **oversee** represent the non-deterministic choice whether the failures are found or not, their firing probability is cov and $1 - cov$ respectively where cov is equal to the `<coverage>` attribute of the **Overhaul** element. They both have an inbound arc from **UM** and **maintain** has an outbound arc to **ER** if the associated node is stateful.

Additional transitions are created to handle the fact that the activity can start when the node has not failed yet. The immediate **start_2** (**s_2** in short) and **s_3** transitions represents that the activity starts when the node is in the **Healthy** or **Erroneous** states. Both transitions have outbound arcs to **Pre_UM**. The **overhaul** subnets are shown on Figure 5.20.

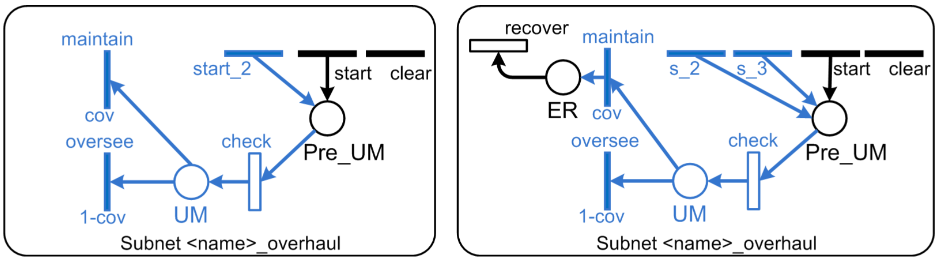

Figure 5.20: *Overhaul activity transformed to subnet*

The basic subnets of the associated nodes are connected to the **overhaul** subnet with several arcs. In addition to the arcs already mentioned for the **activity** subnet, outbound arcs are added from the **start** transition to the **Failed** place and from **maintain** to **Healthy** if the node is stateless. In and outbound arcs are added between the **Healthy** place **start_2** and the **Erroneous** place and **s_3** if the node is stateful. Finally an outbound arc is added from **Failed** to **maintain** and the enabling function of **clear** is changed that it only fires if there is more then two token in **Failed**. The connected subnets are illustrated on Figure 5.21.

5.2.4 Propagation subnets and repair constraints

As described in Section 4.1.4 the **Uses the service of** hyperarcs represent a client-server connection between two elements. This relation has two effects on the dependability model, the failure of the server element can cause an internal error or actual failure in the client element and the client element can return to normal operation after maintenance only if the server element is not under maintenance itself.

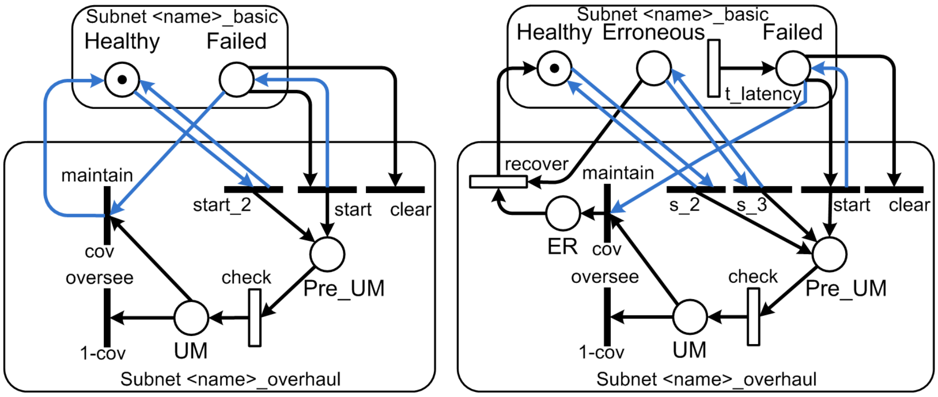

Figure 5.21: *Basic subnets connected to the overhaul subnet*

Error propagation

The error propagation is modeled with the generation of a subnet for every **uses** hyperarc. The subnet has three places defined for representing the process of propagation. The **New** place represents the beginning of the propagation process when the server element has not failed yet. The **Used** place is used for storing the information that the effect of the server failure has been handled and no further propagation happens until the server is not in normal operation. Finally the **Choice** place represents that the server indeed failed and there is a possibility that the failure propagates to the client. These places are connected with several immediate transitions which are described in the following:

- The **may_prop** transition represents the activation of the propagation process. It has an inbound arc from the **New** place and outbound arcs to the **Used** and **Choice** places.
- The **prop** and **no_prop** transitions represent the non-deterministic propagation of the error and are always enabled concurrently. They both have inbound arcs from the **Choice** place, their firing probability is p and $1-p$ respectively, where p equals the `<propagation_probability>` attribute of the **uses** hyperarc.
- The **restart** transition represents the reactivation of the propagation process when the server has returned to its healthy state. It has an inbound arc form the **Used** place and an outbound arc to the **New** place. The transition is only enabled when there is a token in the **Healthy** place of the basic subnet for server element.

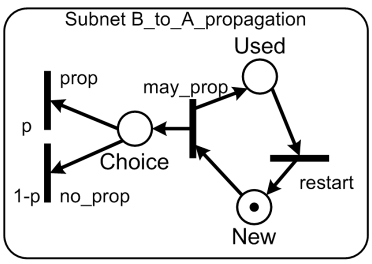

Figure 5.22: *Error propagation subnet for the uses hyperarc*

The created subnet is connected with the basic subnet of the client and server elements. An inbound and outbound arc is added to the **may_prop** transition from and to the **Failed** place of the basic subnet of the server element (**B**). The **prop** transition is connected with an inbound arc from the **Healthy** place and an outbound arc to the **Erroneous** or **Failed** place, all in the basic subnet of the client element (**A**), depending on whether the element is stateful or stateless. The propagation subnet created for a **uses** hyperarc is shown on Figure 5.22 and the connections with the basic subnet for the **A uses B** relation is illustrated on Figure 5.23.

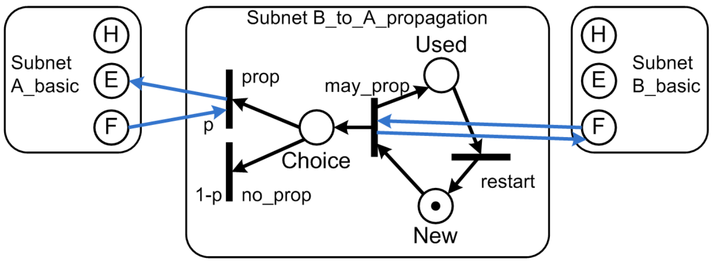

Figure 5.23: *Basic subnets connected to the error propagation subnet*

Repair constraints

Finally the repair constraints are created in the form of inhibitor arcs between the various subnets of the client and server element. These arcs disable the transitions in the **activity** and **FT_repair** subnets of the client which have an outbound arc to the **Healthy** place of their associated element. The arcs are added to these transitions from the place that connects with an inbound arc to the same type of transitions in the subnets of the server element, specifically the **UM, UM_t, UM_p, ER** named places for nodes and **Failed** basic subnet place for FTSs. These restrictions assure that the client element can restart normal operation if the server element is not under maintenance (though it may still be erroneous or failed). It is important to note that if a cycle exists in the IM formed by the **uses** hyperarcs, these constraints may lead to a potential deadlock of the System Net where every element waits for an other to finish their maintenance. However, the existence of such a situation can indicate a design flaw in the original system or it can be typical for a context where additional refinement of the maintenance processes is necessary. Figure 5.24 shows the repair constraints in the subnets of the client element.

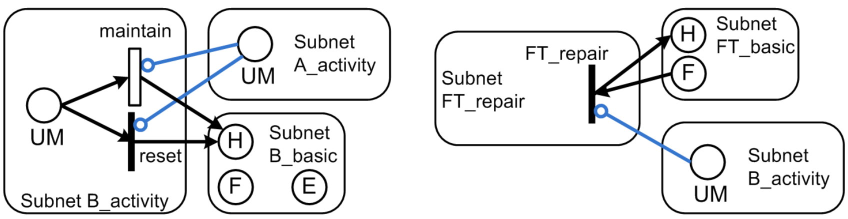

Figure 5.24: *Repair constraints included in the client subnets*

In this chapter the algorithm for creating the Phase Net from the intermediate model is defined along with the generation procedure of the subnets of the System Net. By executing the steps defined here the MPS dependability model can be created which can be used for the evaluation of the system whose UML models were examined to create the intermediate model.

Chapter 6

Implementation

In Section 4 and 5 the theory of dependability modeling of UML designs has been defined. This chapter describes how the presented models can be generated automatically using existing tools. First the different technologies used for the implementation are introduced then their role in creating the current method is described. It is important to note that the implementation follows the Model-Driven Architecture (MDA) paradigm [40]. According to the MDA, the development happens on three levels: the initial system design (platform-independent model, PIM) is transformed to a target dependability model (platform-specific model, PSM) which is chosen depending on the desired verifications to be executed. Finally a specific executable code (or program input) is generated from the dependability model. While the creation of the IM from the UML design is mostly a PIM-PIM transformation the generation of the DSPN from the IM is a PIM-PSM transformation. The DSPN model can be used to generate an input file for several tools created for DSPN verifications (MPS in the case of DEEM). The overview of the specified approach is illustrated on Figure 6.1.

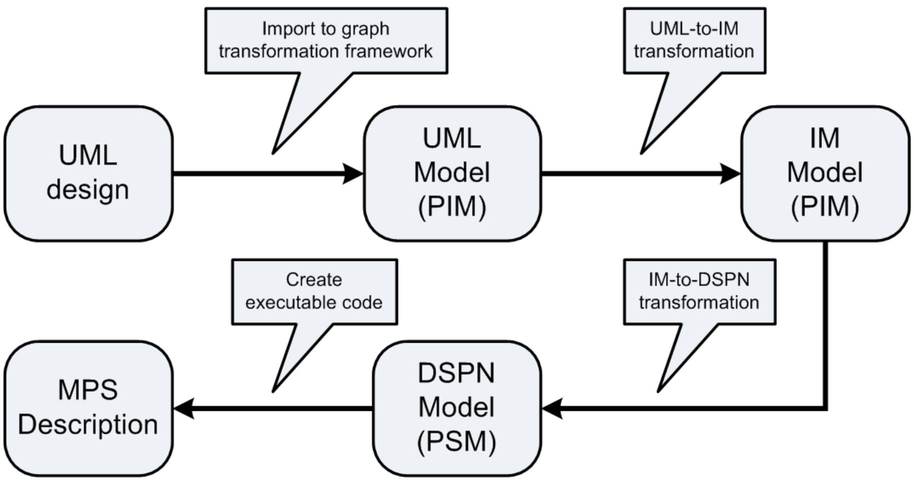

Figure 6.1: *The overview of the approach*

First an introduction to metamodeling which is the base of graph transformation can be read in Section 6.1. Next graph transformation itself is described in detail in Section 6.2. The transformations are implemented and executed with the VIATRA framework, the features of which are listed in Section 6.3. Finally in Section 6.4 and 6.5 the actual applications are detailed.

6.1 Metamodeling

The goal of metamodeling is to provide a frame for possible models by specifying an abstract description on what rules these models have to conform to. The constraints specified in the metamodel have to be satisfied by the concrete modeling language or its models. The UML static structure and object diagrams are used in most cases for both notation system and concrete syntax [51]. For easier understanding this is applied in the following.

The metamodel is constructed from the following elements:

- The *class* is an element that identifies an entity of the modeling language. Classes that can not be instantiated immediately are called abstract similar to those in UML. Subclasses can be descendants of classes and the original class is called a parent by convention in such cases (this is also called *generalization*). An instance of the descendant class is an instance of the parent class as well. Furthermore the descendant class keeps the structure of the parent and may naturally extend it.
- The *association* is a binary relation between the instances of classes. There are two types: *aggregation*, if the target element is contained in the source element and *reference* otherwise.
- The *attribute* represents a property which is the parameter of the class.

The model created based on a metamodel contains the following elements:

- An *object* is an instance of a non-abstract class of the metamodel and has a unique identifier.
- A *link* connects objects and is an instance of an association of the metamodel.
- A *slot* is a storage place in objects for the attributes defined in the metamodel.

The type of an element in the model is an element in the metamodel. Therefore type can be interpreted as a function which associates an element in the metamodel to an element of the model and has the following constraints:

- The type of an object has to be a class.
- The type of a link has to be an association and either aggregation or reference.
- The type of a slot has to be an attribute.

The connection between the classes and associations of the metamodel and the objects and links of the metamodel are illustrated on Figure 6.2. The two elements on the top part of the figure are classes of the metamodel with an association between while on the bottom part two objects of the model are connected with a link.

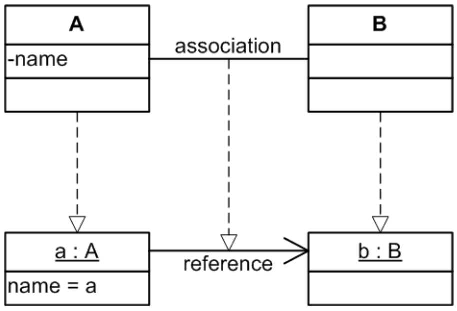

Figure 6.2: *The connection between metamodel and model*

The immediate type of an element is the element in the metamodel which it is an instance of. In accordance with the generalization relation, an object can have mediate types as well. These are the *supertypes* received with the descendance and include every element from which the immediate type is descended.

6.2 Graph transformation

Graph transformation is a technique that creates a new graph from a given graph using rules and patterns. Every rule has a left and a right hand side and the application of the rule replaces the matches of the pattern on the left hand side according to the pattern on the right side. A rule is applicable if there is a graph part that matches the left hand side pattern but does not match any of the defined negative application conditions which can be contained in each other recursively.

The rules of graph transformation are described with a 6-tuple. The parts of a given $r = (LHS, Neg, RHS, Cond, Assign, Par)$ rule are defined in the following:

- The left hand side (*LHS*) graph which is the positive condition of the applicability of the rule. Also referred to as *precondition* because elements in *LHS* have to be present in the model before application.
- The set of graphs representing the negative application condition (*Neg*) which prevents the rule execution.
- The right hand side (*RHS*) graph which specifies the result of the application. Also referred to as *postcondition* because elements in *RHS* have to be present in the model after application.
- The set of logical conditions (*Cond*) which can describe additional conditions for attributes of the objects in the *LHS* and *Neg* graphs.
- The set of value setting functions (*Assign*) which overwrite the attributes of the objects in the *RHS*.
- The descendance function (*Par*) associates to every element their immediate parent in the *LHS* and *Neg* graphs sorted as tree-structures. The root element in this structure is the *LHS* itself.

The left and right hand side graphs of the graph transformation rules does not have to be disjunct. The same element may appear on both sides of the rule. Furthermore if there is a parent-child relation between two graphs in the tree-structure created by the *Par* function then an object can be part of both.

Elements on the same level of the *Par* tree-structure represent conditions in logical conjunction with each other. The negative condition of a parent element is the actual condition of the child element. This relationship allows the embedding of negative conditions into each other at an arbitrary depth.

One of the most important steps of graph transformation is pattern-matching which looks for parts in the graph on which transformation rules can be applied. The applicability of a rule has positive and negative conditions. Consider a given graph G which is an element of the *LHS* or *Neg* set. A positive pattern exists if G has an isomorph or at least homomorphous image in M model therefore the rule can only be applied on a positive match if the following conditions are satisfied:

- Every element in G can be type-safely mapped to an element in M. Furthermore if the match is isomorph then every element in M has to be different.
- Every edge in G can be mapped to an edge in M.
- The conditions defined for the attributes of a given element in G are satisfied for the attributes of the mapped element in M.

For a negative pattern take an arbitrary G' graph in *Neg*. For every G' the following is true: no set of elements exist in the M model which G' can be mapped to and G' is the immediate child of G (i.e. $par(G') = G$). These matches have to be checked on every parent graph and the matches are stored and extended if needed.

The application of a graph transformation rule transforms an M model to an M' model by replacing the *LHS* of the rule to the *RHS*. The steps of rule application are the following [1]:

1. **Graph pattern matching:** Find a match for *LHS* in M that includes the negative conditions of *Neg* and the additional conditions of *Cond*. If more than one match is found the rule is applied on one of the matches non-deterministically.
2. **Deletion:** Remove every element from the model which matches *LHS* but does not match *RHS*.
3. **Creation:** By connecting *RHS* to M the M' model is created. Practically, new objects and links are created for every element that exists in *RHS* but not in *LHS* and the attribute settings defined in *Assign* are applied.

The introduced method uses graph transformation on two places. First the UML model is transformed to the Intermediate Model, second when this IM is transformed to the Dependability Model. From a technical viewpoint the creation of the MPS input file is a graph transformation as well, but it does not create a new model only an other format of the original model. The modelspace of the VIATRA framework introduced in Section 6.3 is the graph on which the transformation rules will be applied to create the models needed for dependability analysis.

It is important to note why graph transformation has been chosen among other model transformation techniques. The declarative definition of patterns is a huge advantage compared to imperative languages like *Java* as complex pattern matching algorithms are already implemented in the VIATRA framework. The other popular transformation tool, *Extensible Stylesheet Language Transformations* (XSLT) has performance and scalability problems when it comes to complex pattern matching challanges [16].

6.3 VIATRA2 model transformation framework

The implementation of the transformations is carried out in the *Visual Automated model Transformations* (VIATRA2 R3) framework [5]. VIATRA itself is an open source Eclipse-based tool and tool-integration framework that can be easily extended with additional components. It is an official Eclipse subproject developed at the Department of Measurement and Information Systems of the Budapest University of Technology and Economics [1].

6.3.1 Modelspace

VIATRA stores the elements which will be the subject of transformations in a hierarchic modelspace. Two type of elements can be specialized from the abstract **ModelElement** type in the modelspace: **Entity** and **Relation** [1]. Figure 6.3 shows the connections between the three elements.

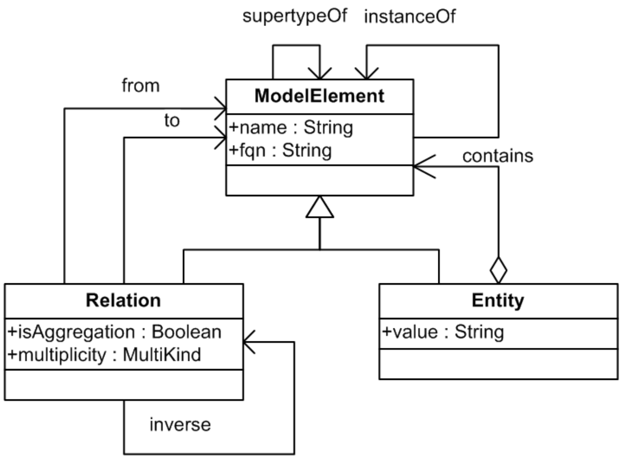

Figure 6.3: *Elements of the VIATRA modelspace*

A **ModelElement** can be the *instance* or a *supertype* of an other ModelElement. Every ModelElement has a String typed attribute which stores the *name* of the element and a derived attribute which is the unique identifier of the element (*fully qualified name* or *fqn*). The name of the elements on the same level in the strict hierarchy of the modelspace has to be unique as the *fqn* is obtained by taking the local names of the hierarchy levels and concatenate them, separated with dots. Hence two elements may have identical local names and still they can be uniquely identified by the fqn.

The modelspace has a tree-structure as **Entities** can contain other entities or relations as illustrated with the *contains* association. The *contains* associations must not create a cycle in the modelspace in order to retain the tree property. A String typed attribute *value* can be associated with entities.

Relations create a connection between two ModelElement and can be described as a directed edge where both ends of the edge is an element in the modelspace. The relations are automatically contained in the source element in the hierarchy. The attribute *isAggregation* specifies whether the target of the relation is contained within the source element. While the *multiplicity* attribute imposes a restriction on the model structure and can have the following values: one-to-one, one-to-many, many-to-one or many-to-many.

6.3.2 Navigation in the modelspace

In order to be able to create models in the VIATRA modelspace form the presented elements, two type of files have to be created. First the metamodels have to be defined whose importance has been described in Section 6.1. Second the transformation definitions are created to navigate in the models based on the metamodels.

Metamodels

The modeling language of the VIATRA2 framework is the *VIATRA Textual Metamodeling Language* (VTML) which uses the *Visual and Precise Metamodeling* (VPM) metamodeling approach [1]. The metamodel is described as a set of entities and relations. The entities represent the elements of the domain-specific modeling while relations represent the connections between these elements. The literals associated with the entities contain application-specific information.

The entities are declared with the `type(name)` form where the type has to be an existing entity and the name has to be unique. The relation are declared as `type(name, source, target)` where type has to be an existing relation, the source and target refer to the entities between which the relation exists. Every element must have a type when declared and the types can be organized in a hierarchy, on top of which are the *entity* and the *relation* types.

The VTML also provides support to define descendance and instantiation. The earlier can be specified with the `supertypeOf(parent type, child type)` form while the later can be done with the `instanceOf(instance, type)` form.

Transformation definitions

The pattern and rule-based model-manipulation language is the *VIATRA Textual Command Language* (VTCL) which defines patterns, graph transformation rules and the elements of the abstract state-machines (ASM) that define control structures such as *variables*, *rules* and *asm functions* [1]. Furthermore VTCL provides support for the use of *native functions* which can be written in Java by the user to extend the framework.

The VTCL files contain the implementation of the graph transformation by defining the patterns for which matches have to be found and how the model has to be changed as the result of the transformation. Two different strategy can be used when a pattern has several matches in the model. The `choose` construct applies the rule on one of the matches selected non-deterministically, this can be repeated iteratively as long as there is a match in the model. In contrast to this the `forall` construct gathers every match and applies the rule on all of them. It is important to note that the `iterate choose` construct in different because pattern matching is repeated after every application.

6.4 UML-IM transformation

In Section 4 the method of generating an Intermediate Model from the UML design is discussed in detail. The implementation of the method is described in the following section, including the UML and IM metamodels, the reference model and the structure of the transformation itself.

6.4.1 Metamodels

UML2 metamodel. The metamodel used is the official UML2 metamodel included in the VIATRA2 R3 framework. This metamodel is compliant with the EMF-based Eclipse implementation of the OMG standard [18, 51]. VIATRA also features an UML2 importer which can create models from the output of UML2 designer tools that use the EMF-based format to store the diagrams. Among many others the metamodel defines entities like `Class`, `Inteface`, `Parameter`, `Profile` or `Stereotype` just to mention a few examples.

IM metamodel. The elements of the Intermediate Model are defined in Section 4.1 with their attributes and the connections between them. These elements are described in a VTML metamodel in which a type is created for every element in the IM. The connections of the IM are modeled using relations except for the *uses* relation which has an attribute and has been modeled as a type itself. Listing 6.1 shows the definitions of the **SLE-HW** and **uses** type. The SLE-HW type has an attribute as specified in the IM and is the subtype of **Node**. The *uses* relation has a source and a target element (which can be either node or FTS) and an Double typed attribute.

```
entity(sleHW){
        relation(permanentRate, sleHW, String);}
supertypeOf(node, sleHW);
[...]
entity(usesRelation){
        relation(prop_prob, usesRelation, Double);
        relation(source, usesRelation, element);
        relation(target, usesRelation, element);}
```

Listing 6.1: *Excerpt from the IM metamodel*

UML-IM Reference metamodel

When creating the IM from the UML2 model a reference model is used to store the information concerning which UML elements have been transformed already and which elements in the IM correspond to a given UML element. This reference model can be used during the transformation for example when an association or dependency relation is handled. In most cases only the IM element created for one end of the relation is known and the element for the other end can be obtained by using the reference model. After the transformation the reference model is useful for identifying the UML2 elements that have to be changed because the results of the analysis demands it. Listing 6.2 shows the definition of the generic reference between UML elements and IM elements (`umlE2IME`) along with a more specific reference between UML Node elements (used for representing hardware components) and nodes in the IM model (`UMLNode2IMNode`).

```
entity(umlE2IME){
    relation(umlElement,umlE2IME,uml2.metamodel.Element);
    relation(IMElement,umlE2IME,imm.ModelElement);}
entity(UMLNode2IMNode){
    relation(umlElement,UMLNode2HWNode,uml2.metamodel.Node);
    supertypeOf(umlE2IME.umlElement, UMLNode2IMNode.umlElement);
    relation(IMElement,UMLNode2HWNode,imm.node);
    supertypeOf(umlE2IME.IMElement, UMLNode2IMNode.IMElement);}
supertypeOf(umlE2IME, UMLNode2IMNode);
```

Listing 6.2: *Excerpt from the UML-IM reference model*

6.4.2 UML-IM Transformation structure

The IM containing entities with the types defined in the metamodel is generated from the imported UML2 model with a transformation stored in a VTCL file. This transformation is built from ASM rules that parse the entire UML2 model using patterns and create the corresponding entities in the IM. Figure 6.4 shows the structure of the transformation. The execution starts with choosing the target UML2 model and the IM is generated automatically after that. The rules of the transformation are defined to create the elements as specified in Section 4.2.

First a *model* entity is created in the modelspace that will contain the whole entity structure of the IM. Then the transformation parses the `UseCase` entities in the model to create the root System elements. In the next steps the packages, objects, classes, components and nodes are all parsed to create the software and hardware elements. After that the elements with the fault-tolerance structure stereotypes are gathered and the FTS elements are created with the corresponding fault-trees, failures and composition relations. Next the maintenance policies and their contained activities are parsed and the appropriate elements are created in the IM. Finally the *uses* relations are created by examining every element that is included in the IM and finding the associations, dependencies and other kind of connections in the UML2 model that have the `relation` stereotype and connect the represented elements. If the tagged value indicates a non-zero probability, then the

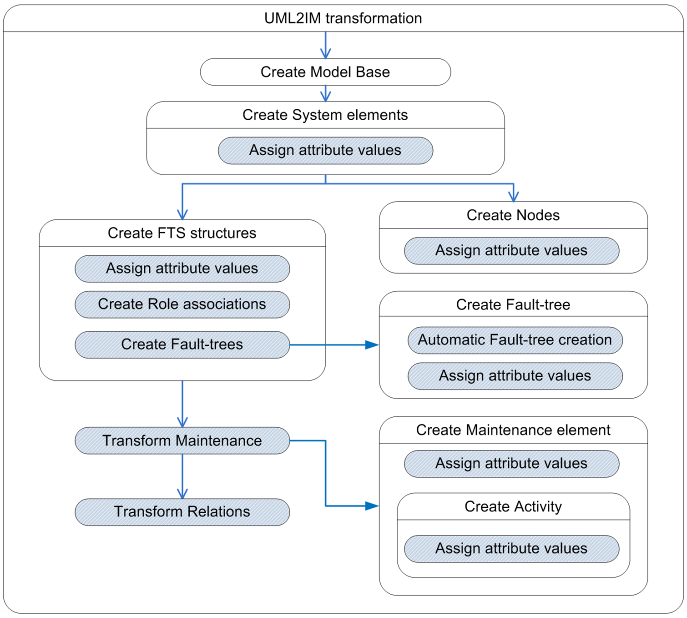

Figure 6.4: *The structure of the UML-IM transformation*

uses relation is created in the IM in the direction defined in Section 4.2.

6.5 IM-DSPN transformation

The method of creating the Dependability Model from the Intermediate Model is defined in Section 5. In this section the implementation of the DSPN metamodel, the IM-DSPN reference metamodel and the transformation itself is described.

6.5.1 DSPN metamodel

The elements of a DSPN are detailed in Section 1.4.1 and the metamodel is created according to that description. The DSPN model consists of the System and Phase Nets which contain places and transitions, the arcs connecting them are stored under the transition they connect to. There are several transition and arc types defined which have different attribute sets. When creating the metamodel the entities representing the different elements are defined. Listing 6.3 shows the definition of the root element containing the two nets and the System Net definition. The remaining elements are defined in a similar fashion,

```
entity(net){
        relation(name,net,String);
        relation(phn,net,phn);
        relation(sn,net,sn);}
[...]
//system net
entity(sn){
        relation(place,sn,place);
        relation(trans,sn,trans);}
```

Listing 6.3: *Excerpt from the DSPN metamodel*

the different attributes are stored as a relation to built-in datatypes while the connections between elements are modeled as relations.

6.5.2 IM-DSPN Reference metamodel

In order to be able to decide which DSPN model parts correspond to a given IM element a reference metamodel is created. This metamodel can also be used for back-annotation and traceability as for every entity in the DSPN model the original IM element can be found. In Section 5.2 we already used the subnet expression to describe a part of the System Net that represents a common characteristic. It is also important to note that in the DSPN context the subnet as an entity does not exist though it is particularly suited for encapsulating the places and transitions that are strongly related. Hence the `subnet` type is defined in the reference model to overcome this shortcoming. Furthermore the `element2subnet` is defined which connects a given element in the IM with the corresponding subnets in the DM. The definition of these entities are shown on Listing 6.4.

```
entity(subnet){
        relation(subnet.place, subnet, place);
        relation(subnet.trans, subnet, trans);}
[...]
entity(element2subnet){
        relation(element, element2subnet, modelElement);
        relation(subnet, element2subnet, subnet);}
```

Listing 6.4: *Excerpt from the reference metamodel*

However in some cases the type of the referred element needs to be restricted to increase performance for larger models and supertyping is a perfect solution to create specialized types. For example the `node2subnet` entity is created to store the association between a **Node** and the created subnets. The supertype of this entity will be the original `element2subnet` entity as also illustrated on Listing 6.5. The `element` relation is also restricted to only accept the appropriate typed entities.

```
entity(node2subnet){
        relation(element, node2subnet, node);
        supertypeOf(element2subnet.element, node2subnet.element);}
supertypeOf(element2subnet, node2subnet);
[...]
entity(healthy);
supertypeOf(place, healthy);
entity(failed);
supertypeOf(place, failed);
entity(erroneous);
supertypeOf(place, erroneous);
```

Listing 6.5: *Type restrictions and extensions*

There are several places defined in Section 5.2 which are unique for every node, system or FTS element. While the DSPN metamodel does not distinguish multiple place types the transformation is more understandable if we differentiate them. For this purpose the places of the basic subnets (**Healthy**, **Failed** and **Erroneous**) are represented with specialized types as shown in Listing 6.5.

6.5.3 IM-DSPN Transformation structure

The DSPN models conforming to the defined metamodel is generated from the models created based on the IM metamodel with a graph transformation implemented in a VTCL file. During the transformation ASM patterns and rules are used to find and process the elements in the IM model. The structure of the transformation is shown on Figure 6.5. At the beginning of the execution the appropriate IM model has to be chosen in the modelspace and the DSPN model is generated automatically by the transformation.

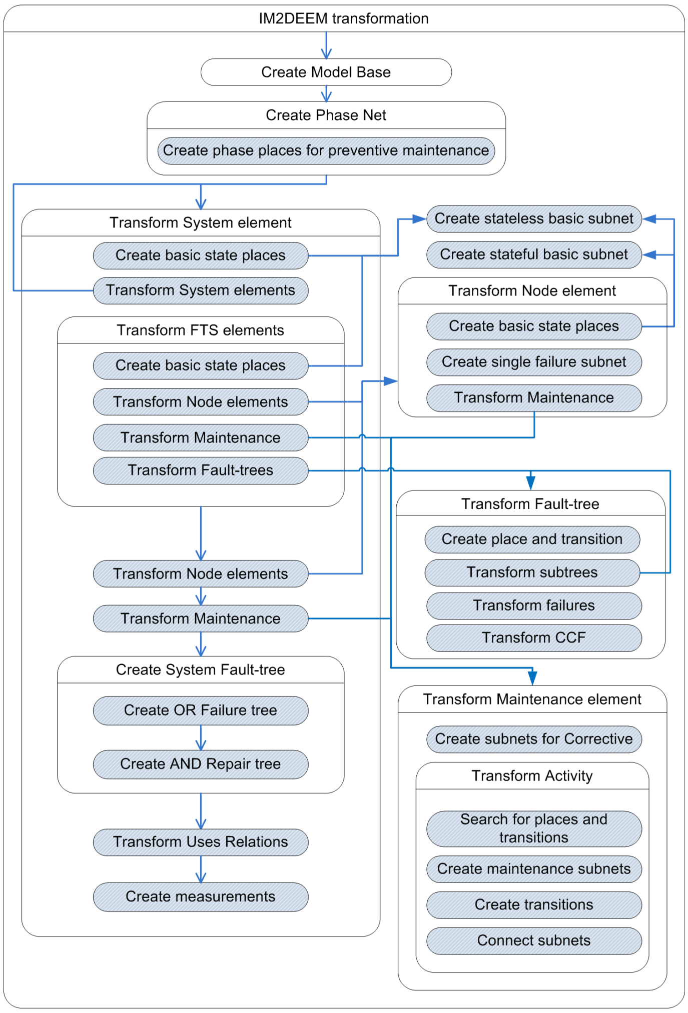

Figure 6.5: *The structure of the IM-DSPN transformation*

First a so called *model base* is created that includes the DSPN model root `net` element and both the `sn` and `phn` entities for the System and Phase Net respectively. Next the Phase Net places and transitions are created as described in Section 5.1. After that the IM model elements are recursively transformed first the System elements and then the System, FTS and Node elements they contain. The rules in the transformation are defined to match the different subnet creation definitions as discussed in Section 5.2. The Maintenance policies and activities associated with the elements are transformed when the element itself is transformed. Finally the *uses* relations are handled as also defined in Section 5.2 and if measures of interest are given for the System elements they are added to the model as well.

```
rule create_Arc(in From, in To, in Name, in Type) =
 let R = undef, Arc = undef in seq{
        if (find trans(From)) seq{
                new(entity(Arc) in From);
                new(instanceOf(Arc, Type));
                rename(Arc, Name);
                new(oarc.fromTrans(R, Arc, From));
                new(oarc.toPlace(R, Arc, To));}
        else seq{
                new(entity(Arc) in To);
                new(instanceOf(Arc, Type));
                rename(Arc, Name);
                if( Type == dspn("iarc")) seq{
                        new(iarc.fromPlace(R, Arc, From));
                        new(iarc.toTrans(R, Arc, To));}
                else seq{
                        new(inharc.fromPlace(R, Arc, From));
                        new(inharc.toTrans(R, Arc, To));}}}
```

Listing 6.6: *Rule for creating arc in the model*

There are several additional rules defined which help in making the transformation understandable. A key aspect of this transformation is creating places, transitions and arcs that make up the System and Phase Nets. In order to keep the subnet creation rules clear of the unnecessary code that is responsible for the low-level value assignment of the attributes of these elements the `create_Place`, `create_Trans` and `create_Arc` rules are created. Transitions have three additional subtypes for which different rules are created as well, these rules call the `create_Trans` rule to create the common elements of the transition. Listing 6.6 shows the `create_Arc` rule which is used to create a new arc in the model by specifying the elements the arc connects, the type of the arc and the name that will identify it.

```
call create_Place(FSubnet, dspn("place"), [...], FaultPlace);
call create_Arc(FaultPlace, ParentFaultTrans, [...], dspn("iarc"));
call create_Place(RSubnet, dspn("place"), [...], RepairPlace);
call create_Arc(RepairPlace, ParentRepairTrans, [...], dspn("iarc"));
call create_ImmTrans(FSubnet, [...], FaultTrans);
call create_Arc(FaultTrans, FaultPlace, [...], dspn("oarc"));
// parse subtree
forall SubTree in FaultTree with find faultTree(SubTree) do
 let RepairTrans = undef in seq{
        call create_ImmTrans(RSubnet, [...], RepairTrans);
        call create_Arc(RepairTrans, RepairPlace, [...], dspn("oarc"));
        call transform_FaultTree(FTS, SubTree, FSubnet, RSubnet,
                FaultTrans, RepairTrans);}
```

Listing 6.7: *Rule part for creating fault-tree in the model*

After defining these rules the assembling of complex subnets is straightforward as it mostly consists of calling the creator rules with the correct parameters. The example transformation code in Listing 6.7 shows how an **AND gate** of a fault-tree is transformed to the appropriate elements as defined in Section 5.2.2. First the places in the **failure** and **repair** subnets are created and arcs are placed to connect them to the parent transitions which are passed as parameters to the rule. Then the transitions to which the subtrees are connecting are created together with the connecting arcs. Finally the rule itself is called recursively for every subtree with the proper parameters.

MPS description generation. The dependability analysis of a DSPN model can be carried out on any particular tool that is able to handle DSPNs. However every tool has a given input language format that defines how the DSPN itself has to be described. The VTCL file describing the transformation that generates the MPS description for the DEEM tool from the DSPN model was already created for [34]. This transformation parses the DSPN model and for every entity it writes the appropriate lines to an output file. The result file can be used with the DEEM tool for dependability analysis.

In this chapter the theory and the framework behind the implementation of the defined method was described along with the definition of the implementation steps that are taken to provide an automatic tool for generating DSPN descriptions from UML models. The theory of metamodeling and graph transformation was described and the VIATRA2 model transformation framework was introduced. Finally the metamodels and transformations required for the method were detailed.

Chapter 7

Dependability model generation for the case study

In order to show the application of the method the Reactor Protection System and the Financial case studies are used. In this chapter the models for both case studies are presented first the RPS in Section 7.1 then the Financial case study in Section 7.2. Several dependability property evaluation possibilities are described as examples for the usability of the created dependability models.

7.1 Reactor Protection System Case Study

The RPS case study was introduced in Section 1.5 along with the maintenance policies and failure rates defined for the components. In the following the system is modeled first in UML using the profile defined in Chapter 3 then the intermediate model is created according to the steps defined in Section 4.2. This intermediate model is then used to create the dependability model according to the steps defined in Chapter 5. The attributes defined in the stereotypes are illustrated in a gray rectangle connected to the element to which the stereotype is applied.

7.1.1 UML models of the RPS

First the static structure of the RPS is created as shown on Figure 7.1. The stereotypes of the profile are applied on the classes to mark the structures and components of the system. The **RPS** element represents the system itself, which contains the **Channels**, **TrainBreaker** and **Rods** elements. The **Channels** element contains **Channel** components while the **TrainBreaker** contains **Train** and **Breaker** segments. The **Train** is composed of **SSL device**, **AS device** and **UV device** elements. The elements which are related in the system description are connected with dependency relations.

The concrete structure of the RPS is illustrated on Figures 7.2 and 7.3. The RPS contains a channels segment with four channels and two train-breaker segments that have the same structure. Only one of the train-breaker segments is illustrated as they are identical.

The channels have a common preventive maintenance policy defined for them (**ChannelsTM**)

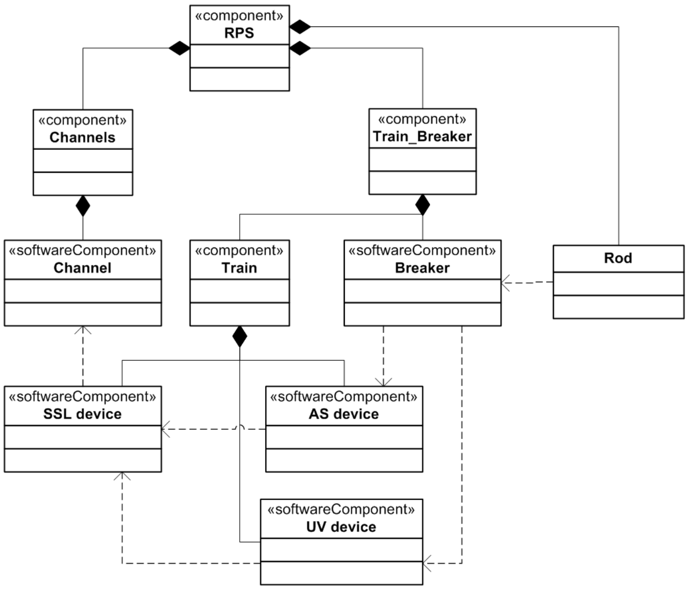

Figure 7.1: *The static structure diagram of the RPS components*

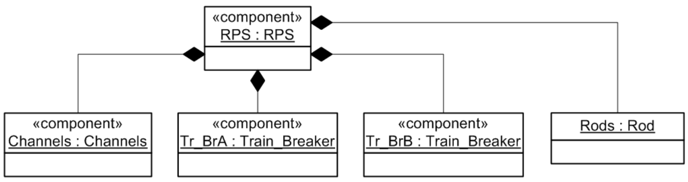

Figure 7.2: *The object diagram of the RPS components*

which defines repair activities for every channel. Additionally the random and common cause failures are created for the channels. The random failures are the same for every instance of the channels but the common cause failure rate is different for two and three channels. Hence these failures are defined for the instances of the channels. This part of the model is illustrated on Figure 7.4.

Similarly to the channels segment the train-breaker segments have a common preventive policy as well (**TrainBreaker_TM**) which defines overhaul activities for the breakers and repair activities for the devices contained in the trains. The random and common cause failures for the breakers are also defined on Figure 7.5.

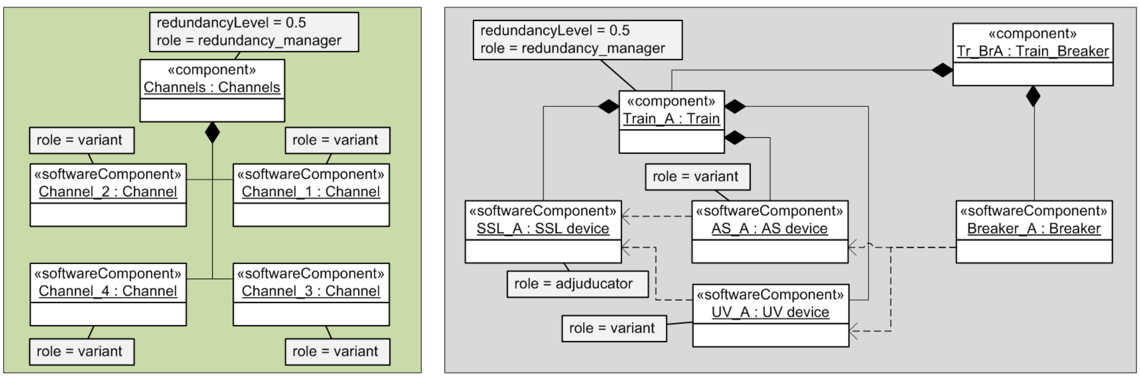

Figure 7.3: *The object diagram of the RPS components*

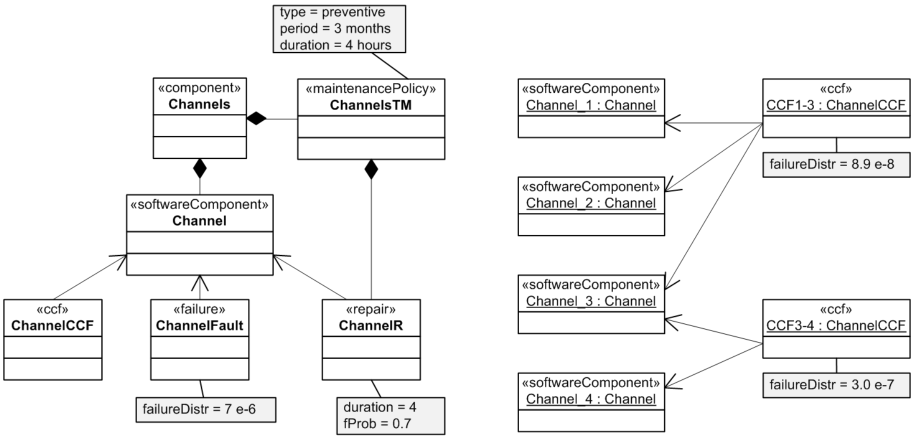

Figure 7.4: *The UML diagrams for the Channel segments*

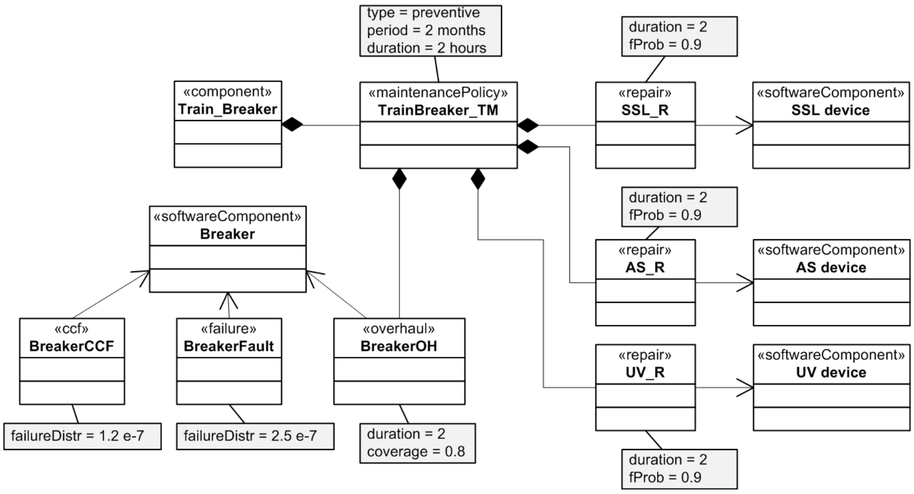

Figure 7.5: *The UML diagrams for the Train-Breaker segments*

Every device in the train segment has different random and common cause failure rates which are modeled as classes marked with the **failure** stereotype. These elements are illustrated on Figure 7.6.

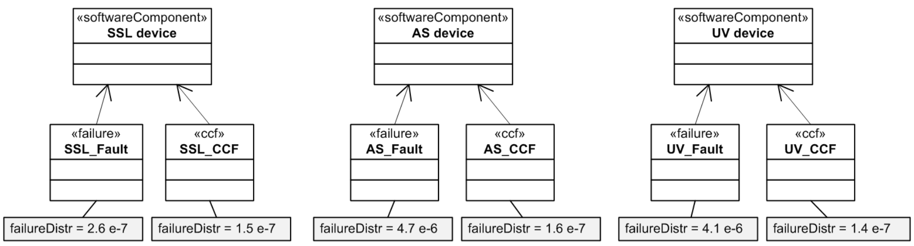

Figure 7.6: *The UML diagrams for the devices in the train segment*

7.1.2 IM model created from the UML models of the RPS

After executing the transformation from the UML models to the intermediate model, the structural part of the RPS can be illustrated as shown on Figure 7.7. The hierarchy of the system is easily recognizable along with the associations representing the roles of the components in the fault-tolerance structures. The redundancy manager components are represented as a **FTS** and a **Node** element.

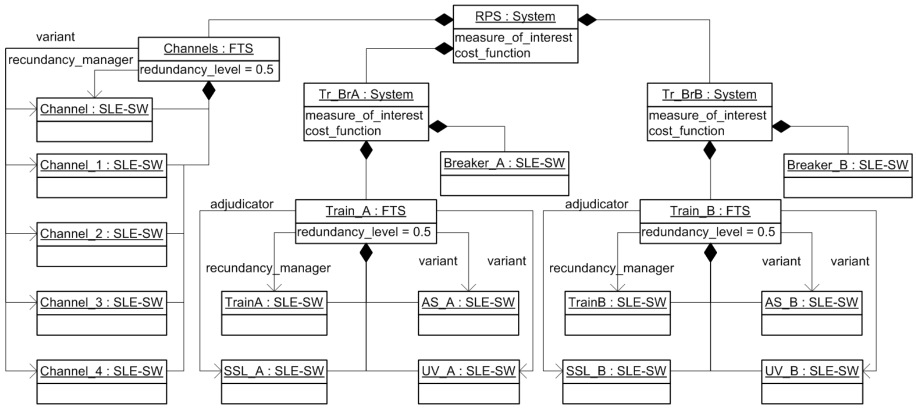

Figure 7.7: *The structure part of the IM created from the UML model*

The maintenance policies and activities defined for the channels are illustrated on Figure 7.8 along with the various failures damaging the channels. It is important to note that only a part of the common cause failures are illustrated in order to keep the diagram readable. However the complete intermediate model contains every common cause failure combination defined for the components. Furthermore the failures are contained in the fault-trees of the **FTS** element but this composition is illustrated later.

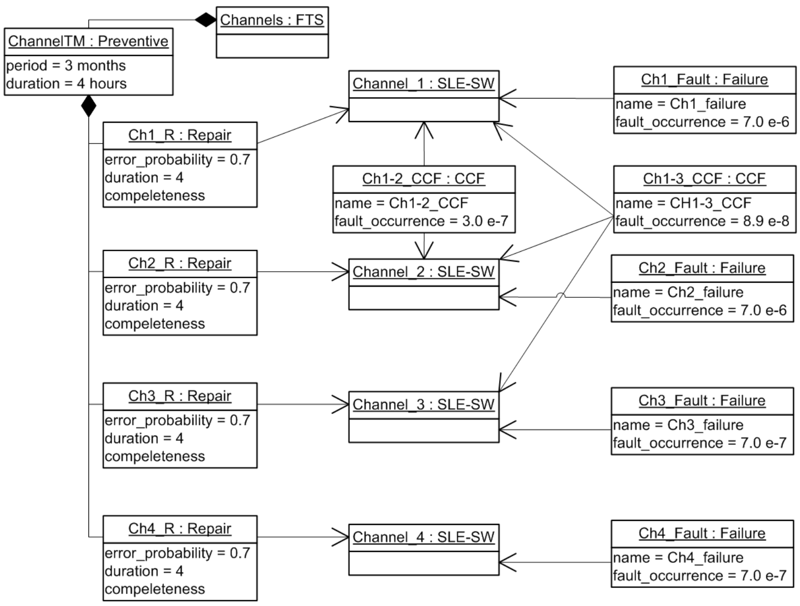

Figure 7.8: *The channel part of the IM created from the UML model*

The fault-trees of the **Channels** fault-tolerance structure are illustrated on Figure 7.9 though only a part of the fault-trees are shown. Note that the top-level **OR gates** are defined for the various combinations of channel failures that cause the channel segment to fail. The **AND gates** on the lower level represent the different failure combinations which correspond to the event represented by the top-level gate. For example the channels 1,2 and 3 can fail either separately, by a common cause failure affecting all three of them or only two are affected by a common cause failure and the third fails separately.

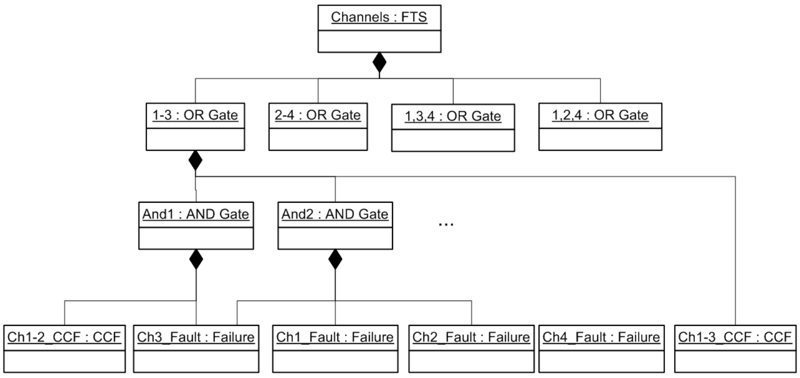

Figure 7.9: *The fault-trees of the channels in the IM*

Similarly to the channels segment, the preventive maintenance policy and its activities are defined for the train-breaker segments. The intermediate model part is illustrated on Figure 7.10 along with the failures defined for the various elements. Note that only the SSL device is illustrated among the devices in the trains segment as the AS and UV devices are modeled in the exact same way apart from the attribute values.

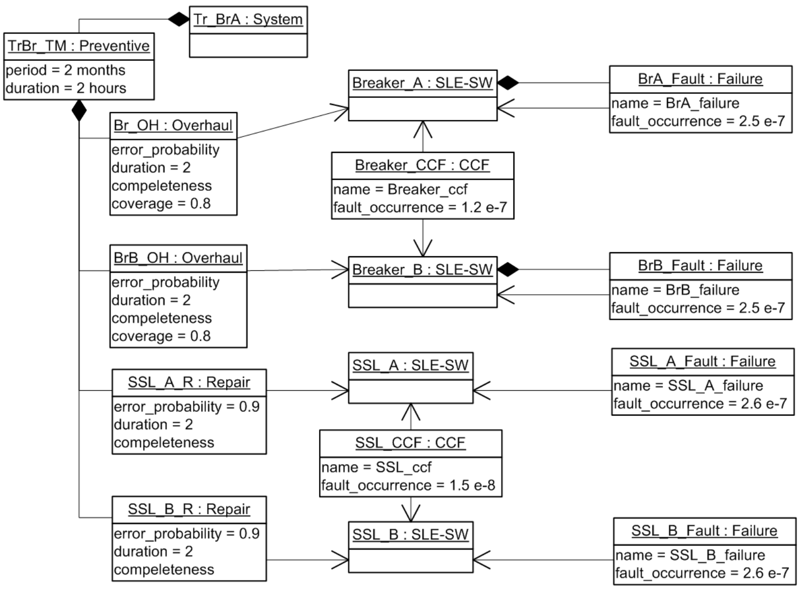

Figure 7.10: *The train-breaker part of the IM created from the UML model*

The fault-trees of the train segment are illustrated on Figure 7.11. The train segment fails if either the SSL device fails or if both the AS and UV devices fail. This option is represented by the top-level **OR gates** which contain several **AND gates** representing the different failure combinations which can occur in the system.

Note that although it is not represented in the model explicitly at the moment, the **RPS** system is operational as long as the channels segment and one of the train-breaker segments are operational. This additional information can be included either by refining the UML model and defining a fault-tolerance structure for the train-breaker segments or by manually including the refined failure and repair subnets in the dependability model.

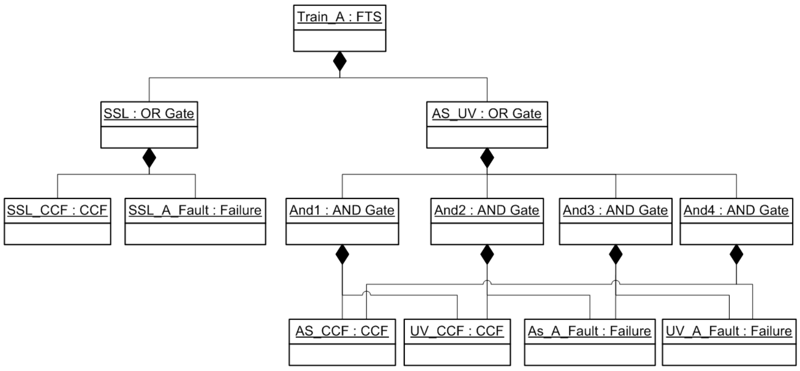

Figure 7.11: *The fault-trees of the trains in the IM created*

7.1.3 Dependability model created from the IM model of the RPS

Phase Net The maintenance policies defined for the RPS case study are used in Section 5.1 as an example for the illustration of the Phase Net creation algorithm, the generated Phase Net is shown on Figure 5.2. The normal operation is repeatedly interrupted with maintenance periods which are defined for the channel and train-breaker segments of the system.

System Net Given the complexity of the case study and the similarity among the components only a small part of the System Net is illustrated here. Figure 7.12 shows the preventive maintenance policy subnets for the channels and one of the train-breaker segments, the basic and maintenance subnets of one of the breakers along with the subnet controlling the occurrence of common cause failures affecting the breakers.

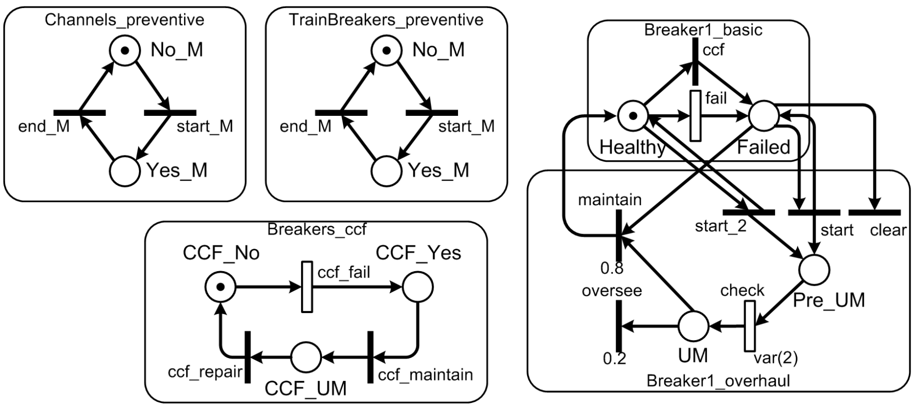

Figure 7.12: *The policy and the Channel basic and maintenance subnets*

As an example for the representation of fault-trees in the dependability model the fault-tree corresponding to the top-level system element is illustrated on Figure 7.13. As discussed earlier the system itself fails if either the channels segment or both the train-breaker segments fail. This is also represented in the repair subnet which shows that the system returns to correct operation if the channels segment and at least one of the train-breaker segments are working.

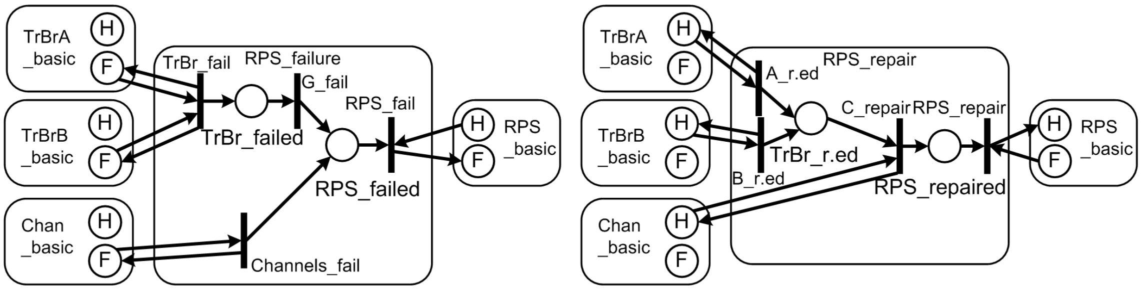

Figure 7.13: *The fault-trees of the top-level System element*

Dependability properties Various properties can be defined for the RPS system using reward functions that can be specified in the DEEM tool as measures. In Section 1.3 the definition of several properties were given for illustration purposes the *availability* property is used here. In the following the availability of the RPS is specified on the dependability model as:

- **RPS available if at least two channels and one of the train-breaker segments is available.** The train breaker segment is available if either the AS or the UV device and both the SSL device and the breaker are working correctly. This can be specified as a reward measure in DEEM using the markings of the basic subnets representing the components.

- DEEM is capable of performing both *instantaneous* and *cumulative* analysis for the defined reward functions. The reward function can be specified as described in (7.1) where $m(Subnet.Place)$ refers to the marking of the given place in the subnet.

$$A(t) = IF((m(Ch1.H) + m(Ch2.H) + m(Ch3.H) + m(Ch4.H) > 1)AND$$
$$(((m(BrA.H) + m(BrB.H) * m(TrBrA.Yes_T M) > 0)AND(m(SSL_A.H) = 1)$$
$$AND((m(AS_A.H) = 1)OR(m(UV_A.H) = 1)))OR((m(BrB.H) + m(BrA.H)*$$
$$m(TrBrB.Yes_T M) > 0)AND(m(SSL_B.H) = 1)AND((m(AS_B.H) = 1)OR$$
$$(m(UV_B.H) = 1))))THEN(1)ELSE(0) \quad (7.1)$$

- If the fault-trees are precisely modeled in the dependability model then the (7.1) formula can be expressed simply by stating that the channels and at least one train-breaker is working correctly (7.2).

$$A(t) = IF((m(Channels.H) = 1)AND((m(TrBrA.H) = 1)$$
$$OR(m(TrBrB.H) = 1)))THEN(1)ELSE(0) \quad (7.2)$$

- If the dependability model also includes the refined fault-trees for the top-level system then the *availability* of the system can be expressed simply with the formula $A(t) = IF(m(RPS.H) = 1)THEN(1)ELSE(0)$ which is important because its simplicity provides easier automated generation as the availability of the system can be expressed as the state of the top-level **System** element. However it requires precise fault-trees to be defined at every level of the model.

7.2 Financial Case Study

In Section 3.7 the extension for non-functional service contracts was introduced and in Section 4.3 the steps required to generate the IM elements from the UML model of the contracts were defined. In this section the Financial case study is used for illustrating the use of the method on a concrete UML model. First the UML model for the actual service contract is described which is later used for the generation of the IM part that can be transformed to the dependability model. Finally the measures required to evaluate the availability of the service is defined.

UML models using extended UML4SOA

The dependability and maintenance characteristics for the service contract between the Credit request service and the Balance validation service are defined on Figure 7.14. The service is marked as stateless which means error recovery and detection is not applicable to the service.

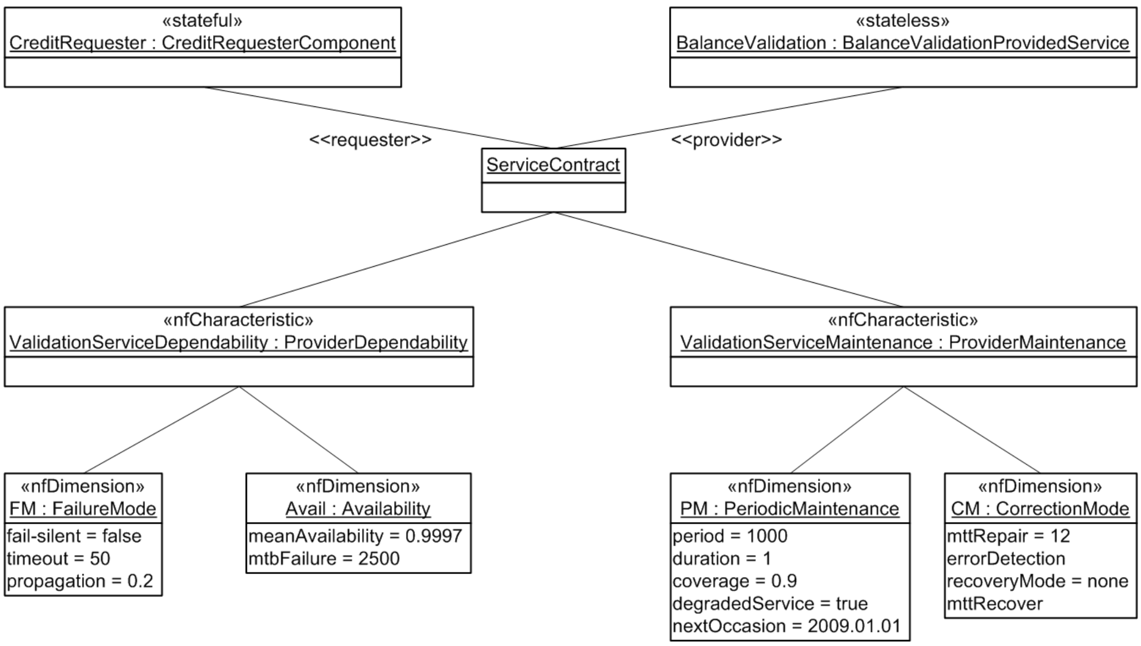

Figure 7.14: *The UML model using the extended contract definitions*

7.2.1 Intermediate model of the Balance Validation service

The values given in the service contract of the concrete configuration can be used to create the Intermediate Model elements in order to include the service in the dependability evaluation. The service itself is modeled as a software node, in the case of the example, an **SLE-SW** element. The **CorrectiveMode** dimension is represented by a **Corrective** element and a **Repair** element. The attributes are copied from the values of dimension attributes. Similarly the **PreventiveMaintenance** dimension is represented by a **Preventive** and an **Overhaul** element, with the attributes getting the proper values from the dimension. The maintenance policies are contained in the **SLE-SW** node representing the service and the defined activities are associated with it as well. Finally the **Failure** element representing the possible failure of the service is created. The fault occurrence rate is equal to the reciprocal of the *mtbFailure* attribute as the time between failures has to be transformed into the number of failures in a given time.

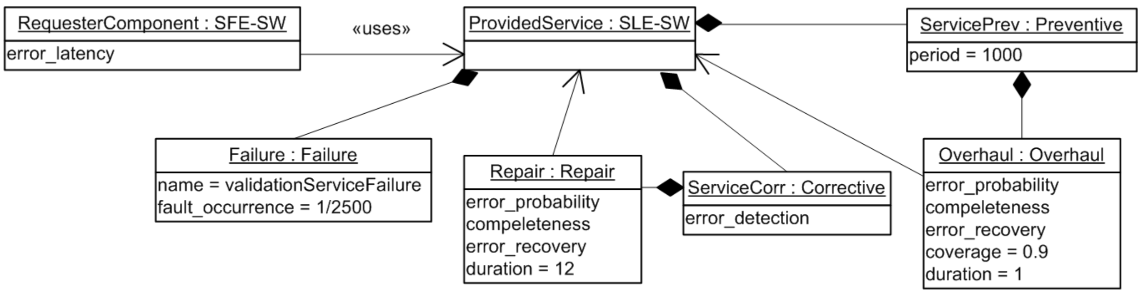

Figure 7.15: *The generated IM model part from the service contract*

7.2.2 Dependability model of the Balance Validation service

The System Net of the generated dependability model contains the subnets corresponding to the elements of the intermediate model. On Figure 7.16 the basic, repair and overhaul subnets are illustrated for the service. The firing delay of the exponential transitions of the subnets is determined by the attributes of the intermediate model elements which are in turn taken from the UML models.

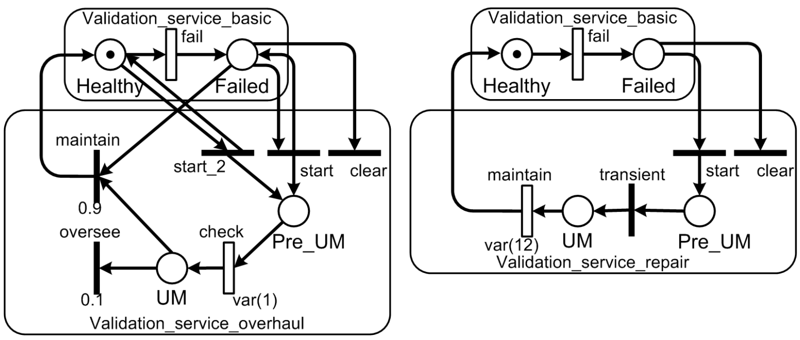

Figure 7.16: *The basic, repair and overhaul subnets for the service*

The service has both a preventive and a corrective policy defined which are represented by the policy subnets on Figure 7.17. The preventive policy also means that the Phase Net is generated using the algorithm defined with the policy of the service included.

Finally the propagation subnet is illustrated on Figure 7.18 which is corresponding to the relation between the requester component and the service. The error propagation subnet is activated when the service is in the **Failed** state. The error propagates to the component according to the probability defined in the service contract. The repair constraints are not

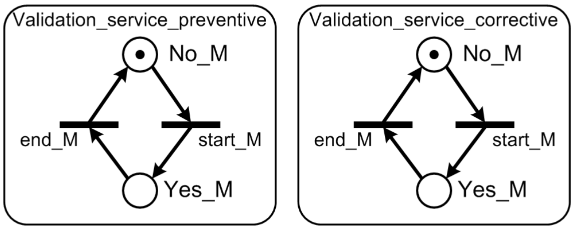

Figure 7.17: *The maintenance policy subnets for the service*

illustrated but they are included in the model according to Section 5.2.4.

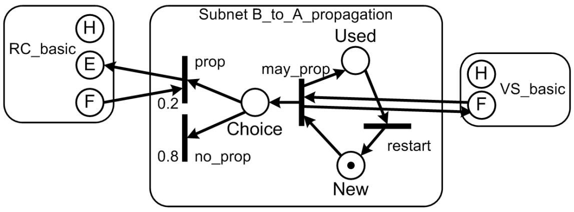

Figure 7.18: *The propagation subnet for the service*

7.2.3 Dependability property for the case study

As already discussed in Section 7.1.3 reward functions can be specified in the DEEM tool as measures to evaluate the properties of the modeled system. In the following the reward functions for the availability (*instantaneous* or *mean*) of the service and the system its part of are defined as an example:

- **Service availability** can be expressed simply by the expression (7.3).

$$A(t) = IF(m(Serv.H) = 1)THEN(1)ELSE(0) \tag{7.3}$$

  This property can be used to assure that the service contract indeed describes a service with the required dependability level.

- **System availability** can be expressed by extending the expression created according to Section 7.1.3 with the availability of the service if the failure of the service causes the system to fail (7.4)

$$A(t) = IF((m(Sys.H) = 1)AND(m(Serv.H) = 1))THEN(1)ELSE(0) \tag{7.4}$$

In this chapter the application of the defined method was illustrated on two case studies. The complete model of the Reactor Protection System is defined and transformed to first the intermediate model then to the dependability model. The intermediate model created from the non-functional service contract of the Balance Validation service is described along with the dependability model parts corresponding to the intermediate model elements. Furthermore the definition of dependability characteristics in the DEEM tool is described briefly using the RPS and availability as an example.

Chapter 8

Conclusion and future work

8.1 Concluding remarks

The systems designed and developed today (such as embedded, mission-critical or service-oriented architectures) have to live up to high expectations both in availability and performance while their complexity grows beyond the understandable. In order to be able to grasp the numerous aspects of these systems engineers use high abstraction level modeling languages like UML to describe the systems often using domain-specific profiles. An example for domain-specific profiles is MARTE which covers the modeling of real-time and embedded systems. These modeling languages provide a powerful tool in specifying a system of arbitrary complexity by using hierarchic architecture and common representation.

However the qualitative and quantitative evaluation of various nun-functional properties of complex systems have to be executed before the production phase, during design-time. Precise mathematical models are used for modeling the system in question at a low abstraction level before analyzing these models with various model checking tools. It is important to note that the creation of these models require a deep understanding of the theory behind model checking.

The huge break between the high level system description languages and the low-level mathematical models calls for novel methods to bridge the gap. These methods must provide the designers with the ability to evaluate their system models without the hardships of creating the mathematical models by hand.

The paradigm of Model-Driven Architecture and the technique of model transformation are particularly suited for the definition and execution of such methods. A key aspect of model-driven engineering is the independence from a concrete system and the ability to support a whole domain of systems.

Over the years numerous methods were introduced to aid the automatic generation of evaluation models each capable of handling a given domain of systems. However most of these methods exclude the modeling of the maintenance and monitoring of systems even though these play a huge role in high-availability systems and their specification should be included in the system models.

Additionally there is an ever-increasing applications developed using the Service-Oriented

Architecture paradigm while there are only a few approaches which support the analysis of such systems. The Service-Oriented Profile provides engineers with the means to exploit the full potential of these domain-specific concepts.

In this document a novel method is defined to provide a framework for the dependability analysis of UML-based system designs. The **conceptual contributions** include the following:

- **The support of a new profile including maintenance and monitoring** that extends the state-of-the-art UML profile MARTE with several additional elements.
- **The extended definition of an intermediate model** which provides a transition between the UML language and the mathematical models used for the analysis. The intermediate model includes the explicit definition of *fault-trees* for redundancy structures, possible *failures* of system components and *maintenance policies and activities.*
- **Detailed definition of the transformation steps needed to generate** the *intermediate model* from the *UML models* and the *dependability model* from the *intermediate model.*
- **Definition of a novel algorithm for creating the Maintenance Schedule Table** and the perfectly staggered maintenance schedule for the concurrent use of different preventive maintenance policies.
- **Extension for the non-functional service contracts of the UML4SOA profile** to provide support for the inclusion of services as components in the dependability evaluation.

The **other contributions** include the:

- **The detailed definition of the method implementation** which uses the state-of-the-art model transformation framework VIATRA2 as a powerful model generation and transformation engine. The metamodels of the various models are defined as well as the reference models connecting the corresponding elements of the different models. The transformations are defined conforming to the steps defined in the conceptual method.
- **The application of the method is illustrated on case studies** including the Westinghouse Reactor Protection System and the Financial Case Study of the SENSORIA project.

8.2 Possible future enhancements and extensions

The method defined in this document is barely a first step towards a more complete framework for the dependability evaluation of UML-based designs. The following extensions and enhancements are seen as possible:

- **The inclusion of the complete profile** with the definition of *monitoring* could be used for creating a more precise dependability model of the designed system. While the *analysis results of other tools* and methods also represented with the help of the profile could be used to drive the generation of the dependability model with parameters.
- **The definition of redundancy structures** in the intermediate model can be extended with reliability block diagrams, event-trees or other redundancy representations.
- **The tool-integration of the method into a framework** to provide dependability evaluation capabilities from the UML modeling environment.
- **Other analysis tools** can be used for evaluation by either creating a proper exporter if they can handle DSPN or by creating an alternate transformation from the intermediate model otherwise.
- **Back-annotation of evaluation results** can be implemented to represent the dependability analysis results at the UML level in a format that a designer understands.
- **Complex models could be analyzed hierarchically** by taking advantage of the fact that the results of a dependability evaluation of a lower-level structure can be used to represent the structure as a single component during the evaluation of the higher level structure.
- **The "weak points" of the system could be automatically selected** by examining the results of the dependability evaluation and showing the engineer the parts of the system that has the biggest impact on its dependability properties.

Nomenclature

AS	Auto Shunt trip
ASM	abstract state-machines
Assign	value setting functions (graph transformation rule)
CCF	Common Cause Failure
Cond	logical conditions (graph transformation rule)
DEEM	Dependability Evaluation of MPS
DSPN	Deterministic and Stochastic Petri Nets
EMF	Eclipse Modeling Framework
FTS	Fault-tolerance structures
GSPN	Generalized Stochastic Petri Nets
IM	Intermediate Model
LHS	left hand side (graph transformation rule)
MARTE	Modeling and Analysis of Real-time and Embedded systems
MPS	Multiple-Phased Systems
MST	Maintenance Schedule Table
Neg	negative application condition (graph transformation rule)
OMG	Object Management Group
Par	descendance function (graph transformation rule)
PIM	platform-independent model
PN	Petri Nets
PSM	platform-specific model
RHS	right hand side (graph transformation rule)

RPS	Reactor Protection System
RTES	real-time and embedded systems
SFE-HW	Stateful Hardware
SFE-SW	Stateful Software
SLE-HW	Stateless Hardware
SLE-SW	Stateless Software
SOA	Service-Oriented Architecture
SPT	Schedulability, Performance and Time Specification
SSL	Solid State Logic
T&M	test and maintenance
UML	Unified Modeling Language
UML4SOA	Service-Oriented Profile
UV	Under Voltage
VIATRA	Visual Automated model Transformations
VPM	Visual and Precise Metamodeling
VTCL	VIATRA Textual Command Language
VTML	VIATRA Textual Metamodeling Language

Bibliography

[1] *The Viatra 2 Model Transformation Framework Users Guide*, 2008. `http://dev.eclipse.org/viewcvs/indextech.cgi/gmt-home/subprojects/VIATRA2/index.html`.

[2] N. Addouche, C. Antoine, and J. Montmain. UML models for dependability analysis of real-time systems. In *SMC*, pp. 5209–5214. IEEE, 2004.

[3] M. Albini. Un profilo UML 2.0 per la descrizione di strategie di manutenzione in sistemi critici e la sua applicazione. (Masters thesis), 2009.

[4] M. Alessandrini and D. Dost. D8.3.a: Requirements modeling and analysis of selected scenarios, Finance Case Study, August 2007. SENSORIA Deliverables Month 22.

[5] A. Balogh and D. Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *ACM Symposium on Applied Computing — Model Transformation Track (SAC 2006)*, pp. 1280–1287. ACM Press, Dijon, France, 2006.

[6] W. C. Bars, C. Telefonaktiebolaget, and L. Ericsson. OMG Unified Modeling Language (OMG UML), Superstructure, V2.2 OMG Available Specification, 2009.

[7] S. Bernardi, S. Donatelli, and G. Dondossola. A class diagram framework for collecting dependability requirements in automation systems. In *International Symposium on Leveraging Applications of Formal Methods.*, vol. TR-2004-6. Department of Computer Science, University of Cyprus, 2004.

[8] S. Bernardi and J. Merseguer. QoS Assessment via Stochastic Analysis. *IEEE Internet Computing*, vol. 10(3):pp. 32–42, 2006.

[9] 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(4):pp. 509–522, 2004.

[10] A. Bondavalli and R. Filippini. Modeling and Analysis of a Scheduled Maintenance System: a DSPN Approach. *The Computer Journal, BCS*, vol. 47:pp. 634–650, 2004.

[11] A. Bondavalli, I. Majzik, and I. Mura. HIDE Deliverable 2: Transformations, From Structural UML diagrams to timed Petri nets, 1998.

[12] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, 2003.

[13] M. D. Cin. Extending UML towards a useful OO-language for modeling dependability features. In *WORDS Fall*, pp. 325–330. IEEE Computer Society, 2003.

[14] G. Clark, T. Courtney, D. Daly, D. Deavours, S. Derisavi, J. M. Doyle, W. H. Sanders, and P. Webster. The Mobius modeling tool. In *Petri Nets and Performance Models, 2001. Proceedings. 9th International Workshop on*, pp. 241–250. 2001.

[15] V. Cortellessa and A. Pompei. Towards a UML profile for QoS: a contribution in the reliability domain. In J. J. Dujmovic, V. A. F. Almeida, and D. Lea (eds.), *WOSP*, pp. 197–206. ACM, 2004.

[16] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal*, vol. 45(3):pp. 621–645, 2006. URL `http://www.research.ibm.com/journal/sj/453/czarnecki.html`.

[17] M. R. D'Ambrogio A., Iazeolla G. A method for the Prediction of Software Reliability. In *Proc. Software Engineering and Applications Conference 2002*. Cambridge, MA, USA, 2002.

[18] Eclipse Modeling, Model Development Tools, UML2, 2009. `http://www.eclipse.org/modeling/mdt/?project=uml2`.

[19] T. Erl. *Service-Oriented Architecture : Concepts, Technology, and Design*. Prentice Hall PTR.

[20] S. Gilmore and J. Hillston. The PEPA Workbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. In *In Proceedings of the Seventh International Conference on Modelling Techniques and Tools for Computer Performance Evaluation, number 794 in Lecture Notes in Computer Science*, pp. 353–368. Springer-Verlag, 1994.

[21] L. Gönczy and V. Dániel. Design and Deployment of Service Oriented Applications with Non-Functional Requirements. In J. Suzuki (ed.), *Methodologies for non-functional requirements in Service Oriented Architecture*. IGI Global. Under review.

[22] K. Goseva-Popstojanova, A. E. Hassan, A. Guedem, W. Abdelmoez, D. E. M. Nassar, H. H. Ammar, and A. Mili. Architectural-level risk analysis using UML. *IEEE Trans. Software Eng*, vol. 29(10):pp. 946–960, 2003.

[23] 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(4):pp. 528–558, 2007.

[24] A. Hassan, K. Goseva-Popstojanova, and H. Ammar. UML based severity analysis methodology. In *Reliability and Maintainability Symposium, 2005. Proceedings. Annual*, pp. 158–164. 2005.

[25] J. Hillston. PEPA: Performance enhanced process algebra. Technical Report CSR-24-93, University of Edinburgh, Edinburgh, Scotland, 1993.

[26] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, vol. 23:pp. 279–295, 1997.

[27] I. (Idaho national engineering and environmental laboratory). Reliability study: Westinghouse Reactor protection system, 1999. Lockheed Martin Idaho technologies company. NUREG/CR-5500.

[28] Electropadia: The World's Online Electrotechnical Vocabulary: dependability, 2009. `http://dom2.iec.ch/iev/iev.nsf/display?openform&ievref=191-02-03`.

[29] Electropadia: The World's Online Electrotechnical Vocabulary: reliability, 2009. `http://dom2.iec.ch/iev/iev.nsf/display?openform&ievref=191-12-01`.

[30] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring distributed systems. *ACM Transactions on Computer Systems*, vol. 5:pp. 121–150, 1987.

[31] J. Jürjens. Developing Safety-Critical Systems with UML, 2003.

[32] J. Jürjens, S. Wagner, and T. U. München. Component-based Development of Dependable Systems with UML. In *ComponentBased Software Development for Embedded Sytems. An Overview on Current Research Trends*. Springer, 2005.

[33] N. Koch, P. Mayer, R. Heckel, L. Gönczy, and C. Montangero. D1.4.a: UML for Service-Oriented Systems, October 2007. SENSORIA Deliverables Month 24.

[34] M. Kovacs, P. Lollini, I. Majzik, and A. Bondavalli. An Integrated Framework for the Dependability Evaluation of Distributed Mobile Applications. In *RISE/EFTS Joint International Workshop on Software Engineering for REsilieNt systEms (SERENE 2008)*, pp. 29–38. 2008.

[35] J.-C. Laprie and K. Kanoun. Software reliability and system reliability. In *Handbook of software reliability and system reliability*, pp. 27–69. McGraw-Hill, Inc., Hightstown, NJ, USA, 1996.

[36] I. Majzik, A. Pataricza, and A. Bondavalli. Stochastic dependability analysis of system architecture based on uml models. In *Architecting Dependable Systems LNCS–2667*, pp. 219–244. Springer-Verlag, 2003.

[37] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. Modelling with Generalized Stochastic Petri Nets. *SIGMETRICS Perform. Eval. Rev.*, vol. 26(2), 1998.

[38] UML Profile for Modeling and Analysis of Real-time and Embedded Systems (MARTE), 2009. `http://www.omg.org/technology/documents/profile_catalog.htm`.

[39] M. Martinez, J. Davis, J. Scott, J. Sztipanovits, and G. Karsai. Integrated Analysis Environment for High Impact Systems, 1997.

[40] A Proposal for an MDA Foundation Model, 2005. `http://www.omg.org/cgi-bin/doc?ormsc/05-04-01`.

[41] S. Mustafiz, X. Sun, J. Kienzle, and H. Vangheluwe. Model-driven assessment of system dependability. *Software and System Modeling*, vol. 7(4):pp. 487–502, 2008.

[42] Catalog of UML Profile Specifications, 2009. `http://www.omg.org/technology/documents/profile_catalog.htm`.

[43] G. J. Pai and J. B. Dugan. Automatic Synthesis of Dynamic Fault Trees from UML System Models. In *ISSRE*, pp. 243–256. IEEE Computer Society, 2002.

[44] A. Pataricza. From the General Resource Model to a General Fault Modeling Paradigm? In *Workshop on Crititcal Systems Development with UML at UML 2002*, pp. 163–171. 2002.

[45] A. Pataricza, T. Bartha, G. Csertán, S. Gyapay, I. Majzik, and D. Varró. *Formális módszerek az informatikában*. Typotex, 2004. In Hungarian.

[46] A. Pataricza, I. Majzik, G. Huszerl, and G. Várnai. UML-based design and formal analysis of a safety-critical railway control software module. In *Formal Methods for Railway Operation and Control Systems*, pp. 125–132. 2003.

[47] Petri Nets Tools Database Quick Overview, 2009. `http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/quick.html`.

[48] SENSORIA (Software Engineering in Service-Oriented Overlay Computers) EU FP6 Project, 2005. `http://sensoria-ist.eu`.

[49] N. Shankar. Symbolic Analysis of Transition Systems. In Y. Gurevich, P. W. Kutter, M. Odersky, and L. Thiele (eds.), *ASM 2000*, no. 1912 in LNCS, pp. 287–302. Springer-Verlag, Monte Verità, Switzerland, 2000.

[50] A. A. Ucla, A. Avizienis, J. claude Laprie, and B. Randell. Fundamental Concepts of Dependability, 2001.

[51] Unified Modeling Language, 2009. `http://www.uml.org/`.

[52] UML Specification Version 1.3, 2001. `http://www.omg.org/spec/UML/1.3/`.

[53] UML Specification Version 2.0, 2005. `http://www.omg.org/spec/UML/2.0/`.

[54] The Weibull Distribution, 2006. `http://www.weibull.com/LifeDataWeb/the_weibull_distribution.htm`.