



**UNIVERSITA' DELLA CALABRIA**

Dipartimento di Ingegneria Informatica, Modellistica, Elettronica e Sistemistica (DIMES)

**Scuola di Dottorato**

Ingegneria dei Sistemi, Informatica, Matematica e Ricerca Operativa (ISIMR)

**Indirizzo**

Ingegneria dei Sistemi e Informatica

*La presente tesi è cofinanziata con il sostegno della Commissione Europea, Fondo Sociale Europeo e della Regione Calabria. L'autore è il solo responsabile di questa tesi e la Commissione Europea e la Regione Calabria declinano ogni responsabilità sull'uso che potrà essere fatto delle informazioni in essa contenute*

**CICLO**

XXVI

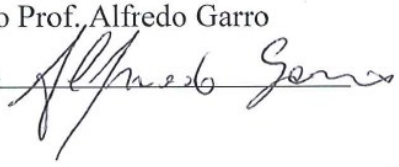
# Model-based and Simulation-driven Methods for the Reliability and Safety analysis of Systems

**Settore Scientifico Disciplinare: ING-INF/05**

**Direttore:** Ch.mo Prof. Sergio Greco

Firma 

**Supervisore:** Ch.mo Prof. Alfredo Garro

Firma 

**Dottorando**: Dott. Andrea Tundis

Firma 

# Abstract (English)

In several industrial domains such as automotive, railway, avionics, satellite, health care and energy, a great variety of systems are currently designed and developed by organizing and integrating existing components (which in turn can be regarded as systems),that pool their resources and capabilities together to create a new system which is able to offer more functionalities and performances than those offered by the simple sum of its components. Typically, the design and management of such systems, whose properties cannot be immediately defined, derived and easily analyzed starting from the properties of their parts when they are considered in stand-alone, require to identify and face with some important research issues.

In particular, the integration of system components is a challenging task whose criticality rises as the heterogeneity and complexity of the components increase. Thus, suitable engineering methods, tools and techniques need to be exploited to prevent and manage the risks arising from the integration of system components and, mainly, to avoid their occurrence in the advanced phases of the system development process which may result in a significant increase in the entire project costs. To overcome these issues the adoption of the Systems Engineering approach represents a viable solution as it provides a wide set of methods and practices which allow the definition of the system architecture and behavior at different abstraction level in terms of its components and their interactions. Moreover, systems requirements are constantly traced during the different system development phases so to clearly specify how a system component concurs to the fulfillment of the requirements. However, in the Systems Engineering field, even though great attention has been devoted to functional requirements analysis and traceability, there is still a lack of methods which specifically address these issues for non-functional requirements. As a consequence, the analysis concerning if and how non-functional requirements are met by the system under development is not typically executed contextually to the design of the system but still postponed to the last stages of the development process with a high risk of having to revise even basic design choices and with a consequent increase in both completion time

and development costs. Among all system requirements, Reliability and Safety are important non-functional requirements. Especially for mission-critical systems, there is a strong demand for new and more powerful analysis tools and techniques able not only to verify the reliability indices and safety of a system but also to flexibly evaluate the system performances and compare different design choices.

In this context, the research aimed to promote the use of flexible methods for the analysis of non-functional requirements by focusing on the definition of: (i) model-based method for system reliability analysis centered on popular SysML/UML-based languages for systems modeling and on *de-facto* standard platforms for the simulation of multi-domain dynamic and embedded systems (Mathworks Simulink); (ii) a methodological process for supporting the safety analysis, along with an approach for performing the Fault Tree Analysis of cyber-physical systems, mainly based on the Modelica language and OpenModelica simulation environment. Furthermore, in order to support the representation of system requirements and thus enable their verification and validation during the design stages, a meta-model for modeling requirements of physical systems as well as different approaches for extending the Modelica language have been proposed. Moreover, an algorithm, which allows trace and evaluate requirements violation through simulation, has been defined.

Finally, the effectiveness of the proposed methods and approaches, especially in the modeling and analysis of both the expected and dysfunctional system behavior, is the result of an intensive experimentation in several industrial domains such automotive, avionics and satellite.

Arcavacata di Rende (CS), November 2013 *Andrea Tundis*

# Abstract (Italian)

In diversi settori industriali quali automobilistico, ferroviario, avionico, satellitare, assistenza sanitaria ed energia, una grande varietà di sistemi sono attualmente progettati e realizzati organizzando e integrando componenti esistenti (che a loro volta possono essere considerate come sistemi), che mettono insieme le proprie risorse e capacità per originarne uno nuovo, che sia in grado di offrire sinergicamente maggiori funzionalità e prestazioni migliori rispetto a quelle che ne deriverebbero dalla semplice somma di quelle offerte dalle sue componenti. Tipicamente, la progettazione e la gestione di tali sistemi, le cui proprietà non possono essere immediatamente definito, derivate e facilmente analizzate a partire dalle proprietà delle loro parti quando essi sono considerati in modo isolato, richiedono di identificare ed affrontare alcune importanti problematiche di ricerca.

In particolare, l'integrazione di componenti di sistema è un compito molto impegnativo la cui criticità cresce all'aumentare della complessità e dell'eterogeneità delle componenti. Di conseguenza, strumenti metodologici e tecniche ingegneristiche dovrebbero essere impiegiegate per prevenire e gestire eventuali rischi derivanti dall'integrazione delle componenti di sistema e, principalmente, per impedirne il loro verificarsi nelle fasi più avanzate del processo di sviluppo, con un impatto negativo sui costi di progetto. Per affrontare questi problemi l'adozione di approcci tipici dell'Ingegneria dei Sistemi rappresentano una soluzione praticabile in quanto forniscono una vasta gamma di metodi e pratiche ben consolidate che consentono la modellazione del sistema in termini della sua architettura e l'analisi del suo comportamento a diversi livelli di astrazione nonchè la rappresentazione delle interazioni tra le singole parti. Inoltre, i requisiti di sistema sono costantemente tracciati durante le varie fasi di sviluppo del sistema in modo da indicare chiaramente come un componente di sistema concorre al soddisfacimento di quali requisiti. Tuttavia, nel settore dell'Ingegneria dei Sistemi, sebbene grande interesse è stato dedicato all'analisi e alla tracciabilità dei requisiti funzionali, poca attenzione, invece, è stata rivolta ai requisiti non funzionali, dovuta ad una mancanza di strumenti metodologici, nell'affrontare specificamente questi temi.

Di conseguenza, l'analisi relativa a se e come i requisiti non funzionali siano soddisfatti da un sistema, non è tipicamente affrontata contestualmente alla progettazione del sistema stesso, ma demandata alle ultime fasi del processo di sviluppo, con un elevato rischio di dover rivalutare persino scelte di progettazione di base, con un conseguente aumento sia in termini di tempi di completamento che di costi di sviluppo. Tra tutti i requisiti di sistema, l'affidabilità (reliability) e la sicurezza (safety) rappresentano due requisiti non funzionali di primaria importanza. Soprattutto per sistemi mission-critical, infatti, vi è una forte richiesta di nuovi ed efficaci strumenti e tecniche di analisi in grado di verificare non solo gli indici di affidabilità e sicurezza di un sistema durante il design, ma anche di valutare in modo flessibile le prestazioni di sistema e confrontare le diverse scelte progettuali.

In questo contesto, l'attività di ricerca è stata dedicata a promuovere l'uso di metodi innovativi per l'analisi dei requisiti di sistema non funzionali, concentrandosi in particolare sulla definizione di: (i) un metodo, basato su modelli, per l'analisi di affidabilità (reliability) di sistemi centrato sul ben noto linguaggio OMG SysML/UML per la parte di modellazione di sistema e su Mathworks-Simulink come piattaforma di simulazione, poichè considerato uno standard *de facto* per la simulazione di sistemi dinamici multi-domain ed embedded systems; (ii) un processo metodologico, basato sul linguaggio Modelica, in grado di supportare, durante le fasi di progettazione, l'analisi di sicurezza (safety) di un sistema impiegando come ambiente di simulazione OpenModelica, insieme ad un approccio in grado di integrarne la Fault Tree Analysis. Inoltre, al fine di consentire la modellazione di requisiti di sistema e permetterne la loro verifica e validazione mediante simulazione, è stato proposto un meta-modello per la loro rappresentazione insieme ad alcuni approcci che ne abilitino la loro integrazione nel design e definendo specifiche estensioni del linguaggio Modelica. La semantica di validazione di tali requisiti e la logica per la loro tracciabilità è stata specificata attraverso un algoritmo di tracciabilità.

Infine, l'efficacia dei metodi e degli approcci proposti, con particolare riguardo alla modellazione e all'analisi sia del comportamento atteso che quello disfunzionale di sistema, è il risultato di una intensa ed accurata sperimentazione in diversi settori industriali quali quello automobilistico, avionico e satellitare.

Arcavacata di Rende (CS), Novembre 2013 *Andrea Tundis*

# Acknowledgments

Questo lavoro di Tesi non sarebbe mai stato realizzato senza il contributo ed il supporto delle molte persone, che mi sono state vicine e mi hanno aiutato nei diverisi momenti di questo importante percorso.

Prima di tutto voglio ringraziare il mio supervisore, il Prof. Alfredo Garro, un esempio per me, mio maestro in questo duro, ma gratificante cammino di vita, che stimo ed ammiro moltissimo, e che mi ha sempre guidato con i suoi consigli nella giusta direzione, trasmettendomi inoltre la passione della ricerca e l'entusiasmo di affrontare nuove sfide con grinta e determinazione.

Voglio inoltre ringraziare tutti i colleghi dell'Università di Linköping in Svezia, dove ho trascorso un anno del mio Dottorato di Ricerca, ed in particolare il Prof. Peter Fritzson per avermi dato la grande opportunità di collaborare e di far parte del suo gruppo di ricerca, il PELAB (Programming Environment Laboratory), ed in modo particolare con la Dott.ssa Lena Buffoni-Rogovchenko con la quale ho strettamente cooperato.

Questo percorso non è stato semplice, al contrario ha richiesto molti sacrifici, non solo da parte mia ma anche alla mia famiglia. Per questo voglio ringraziare dedicando questo lavoro di Tesi ai miei meravigliosi genitori, Anna e Franco, che mi hanno sempre sostenuto nelle mie idee ed incoraggiato nelle decisioni e nelle scelte fatte. Mai obbligato, ma sempre e solo consigliato. Li ringrazio per avermi insegnato a ragionare, ed a prendere le giuste decisioni, nel rispetto altrui, senza mai avermi influenzato in alcun modo.

Inoltre voglio ringraziare mio fratello Gerardo, per essere sempre stato presente e per la grande pazienza che ha avuto con me; per avermi aiutato, in ogni modo, ed ogni volta che ne ho avuto bisogno.

Tante sono le parole che vorrei dire, e molte altre le persone a cui vorrei rivolgere un persiero, forse un pò troppe, per questo dico semplicemente, ma con tanto affetto e con il cuore in mano, grazie a tutti voi per essermi state vicino.

Novembre 2013 *Andrea Tundis*

# Contents

# List of Tables

# List of Figures

# 1

# Introduction

## 1.1 Reference context, Motivations and Objectives

Engineering disciplines and their related activities are strongly centered on the concept of *system*, whose definition is neither unique nor generally shared; a starting point can be provided by the definition proposed in [100] and adopted by the International Council on Systems Engineering (INCOSE)[55]:

*"A system is a construct or collection of different elements that together produce results not obtainable by the elements alone. The elements, or parts, can include people, hardware, software, facilities, policies, and documents; that is, all things required to produce systems-level results. The results include system level qualities, properties, characteristics, functions, behavior and performance. The value added by the system as a whole, beyond that contributed independently by the parts, is primarily created by the relationship among the parts; that is, how they are interconnected".*

In several application domains ranging from automotive to aerospace, a great variety of modern systems are currently designed and developed by organizing and integrating existing components which in turn can be regarded as systems. This design approach potentially offers many advantages in terms of time and cost reductions as promote the (re)usability of existing components and enable a natural parallel work organization in the system realization; in fact, system components can be selected, customized, realized separately and then integrated so to obtain the overall system. However, the integration of system components is a challenging task whose criticality increases as the heterogeneity and complexity of the components increase, as a consequence suitable engineering methods, tools and techniques need to be exploited to prevent and manage the risks arising from the integration of system components and, mainly, to avoid their occurrence in the late phases of the system development process which may result in a significant increase in the development costs.

In this context, the discipline that aims at providing an integrated and methodological approach to deal with the development and management of large-scale and complex systems is indicated as Systems Engineering (SE) which, according to the INCOSE organization, is defined as [56]:

*"an interdisciplinary approach and means to enable the realization of successful systems. It focuses on defining customer needs and required functionality early in the development cycle, documenting requirements, then proceeding with design synthesis and system validation while considering the complete problem: operations, performance, test, manufacturing, cost & schedule, training & support, and disposal. Systems engineering integrates all the disciplines and specialty groups into a team effort forming a structured development process that proceeds from concept to production to operation. Systems engineering considers both the business and the technical needs of all customers with the goal of providing a quality product that meets the user needs".*

In particular, the Model Based System Engineering (MBSE) paradigm, which is defined in [57] as:

*"the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases"*,

is becoming one of the main approaches for Systems Engineering (SE) as it allows to increase the quality both of the development process and of the system to be realized, by organizing the development activities and using formalized representations of systems, called models.

Indeed, MBSE represents a viable solution as it provides a wide set of methods and practices which allow the definition of the system architecture and behavior at different abstraction level in terms of its components and their interactions [48]. Moreover, systems requirements (called also properties [62]) are constantly traced during the different system development phases so to clearly specify how a system component concurs to the satisfaction of the requirements. However, even though great attention has been devoted to functional requirements analysis and traceability, there is still a lack of models, methods and practices which address these issues for non-functional requirements. Indeed, the analysis concerning if and how non-functional requirements are met by the system under development is not typically executed contextually to the design of the system but still postponed to the last stages of the development process (e.g. system verification) with a high risk of having to revise even basic design choices and with a consequent increase in both completion time and development cost [58, 80, 122]. To overcome these issues

several research efforts, which are centered on the exploitation of model-based design approach as well as on the employment of simulation techniques [88], are currently on going.

In this context, the research presented in this Thesis has been focused on two important non-functional requirements, reliability and safety [68, 42], with the aim of defining methods for supporting their analysis and verification contextually to the system design phase, and by employing computer simulation to allow not only the evaluation of system performances, but also to compare different design choices.

The research activity has been conducted in cooperation with the Programming Environment Laboratory (PELAB)[89] of the Linköping University (Sweden) where I spent one year of my Ph.D. program.

## 1.2 Main Results

Starting from the research objective above described, the main contributions resulting from the research activity presented in this Thesis concern the definition of:

1. a model-based method for the reliability analysis of cyber-physical systems, called RAMSAS, mainly based on the SysML modeling language (SysML)[116] and the MATLAB-Simulink simulation environment [74];

2. a methodological process for supporting the safety analysis of cyber-physical systems which is mainly based on the Modelica language[25, 76] and the OpenModelica simulation platform [87].

Concerning the first contribution, RAMSAS is a model-based method which supports the reliability analysis of systems through simulation by providing a classical iterative process consisting of four main phases: Reliability Requirements Analysis, System Modeling, System Simulation, and Results Assessment. For each phase input work-products are defined as well as its role and the internal process for generating specific output work-products. Specifically, in the Reliability Requirements Analysis phase the objectives of the reliability analysis are specified and the reliability functions and indicators to evaluate during the simulation are defined. In the System Modeling phase, the structure and behavior (both expected and dysfunctional) of the system are modeled in SysML (OMG Systems Modeling Language) by using zooming in-out mechanisms; in the System Simulation phase, the previously obtained models of the system are represented in terms of the constructs offered by the target simulation platform, then simulations are executed so to evaluate the reliability performance of the system also on the basis of different operating conditions, failure modes and design choices.

Concerning the second contribution, the result is a model-based method, inspired by the ISO-26262 [59], for the development and the analysis of systems with hard safety requirements. Since it is centered on the Modelica language, this allows to reduce the need for transformations between different formalisms and reducing, consequently, the costs of maintenance and modification of the simulation code, benefiting, additionally, of the variety and capabilities offered by the simulation environments based on Modelica. Moreover, the object-oriented features of Modelica promote a modular design and a hierarchical structure of system models. In the different phases of the method, well-known model-based formalisms are combined in a global framework; in particular, the ModelicaML visual modeling language [77], is used in the phases of System Requirements Analysis and System Modeling, while the Modelica language is adopted in the process of Virtual Testing. The ModelicaML models are first transformed into Modelica source code and then analyzed through simulation by using an extension of the OpenModelica platform for supporting requirements validation.

The experience on requirements representation and verification gained during the definition of the above described methods, has allowed to face with the more general issue of Requirements Modeling. Specifically, in order to support the representation of system requirements and thus enable their verification and validation during the design stages, a meta-model for modeling requirements of cyber-physical systems as well as different approaches for suitable extending the Modelica language has been proposed. Moreover, an algorithm, which allows to support traceability and evaluate requirements violation and through simulation, has been defined. Finally, an approach for supporting the Fault Tree Analysis (FTA) [14] of a system design based on the Modelica language has been proposed.

## 1.3 Thesis Overview

This Thesis is organized as follows. In Chapter 2 a background of both Reliability and Safety properties is provided. In particular, in Section 2.1 a mathematical definition of reliability and safety is reported, then well-known analysis techniques are illustrated in Section 2.2. In Section 2.3 some efforts concerning model-based methods for the reliability and safety analysis of systems are described, whereas, the main International Standards in the context of safety and reliability analysis are reported in section 2.4; finally, the main open issues are introduced and discussed in Section 2.5.

In Chapter 3, RAMSAS, a model based method for the reliability analysis of systems through simulation, is presented in terms of its phases and related work-products. In particular, in Section 3.1 the Reliability Analysis Require-

ments phase is introduced; in Section 3.2 the System Modeling phase and its activities concerning the modeling of system structure and behavior are presented, with particular emphasis on the representation of the system dysfunctional behavior; then the Simulation phase, along with some best practices of model to model transformation, is shown in Section 3.3. Finally, how simulation results can be analyzed and organized in technical reports is presented in Section 3.4.

Chapter 4 provides an experimentation of the RAMSAS method, concerning the Reliability Analysis of an Attitude Determination and Control System (ADCS), which has been jointly conducted with the Institute of Statics and Dynamics of Aerospace Structure of the University of Stuttgart. This experience has been chosen among several experimentations that have been performed in different application domains to evaluate the suitability of RAMSAS for supporting the reliability analysis of systems through simulation and thus obtain useful insights to improve its effectiveness. Concerning the structure of the Chapter, after a brief introduction on the satellite domain, described in Section 4.1, the other sections are dedicated to illustrate the concrete exploitation of the different phases of the RAMSAS process.

Chapter 5 presents a methodological process, inspired by the international standard ISO-26262, which is dedicated to the automotive domain, for supporting the safety analysis of cyber-physical systems through simulation. The method exploits some proposed extensions of the Modelica language both for collecting and representing system requirements as well as for enabling their verification though simulation. In Section 5.1 the overall process is presented. Then, in Section 5.2, a comparison between the proposed process and the ISO-26262 standard is reported; finally, a case study concerning the safety analysis of a Airbag System is shown in Section 5.3.

In Chapter 6 further research contributions on requirements modeling are presented. Specifically, a conceptualization of system requirements through a meta-model is presented in Section 6.1; the description of a Tank System, which is used in the subsequent Sections as a reference example, is provided in Section 6.2; then, three different approaches for the modeling of system requirements and their integration into a Modelica-based design are shown in Section 6.3. In Section 6.4 some proposed Modelica extensions for modeling system requirements along with an algorithm for their traceability are discussed; finally, how the proposed Modelica extensions, enriched by a Probability Model, along with their exploitation for automatically generating a Fault Tree Analysis of a Modelica-based design are presented respectively in Section 6.5 and in Section 6.6.

In Chapter 7 the contributions of this Thesis are summarized and ongoing and future works delineated.

## 1.4 Selected and Relevant Publications

Contents of this thesis were published and presented in international Workshops and Conferences as well as in research Journals; some selected and relevant publications are the following:

- A. Garro, J. Groß, M. Riestenpatt Gen. Richter, and A. **Tundis**. "Reliability Analysis of an Attitude Determination and Control System (ADCS) through the RAMSAS method". Journal of Computational Science (Elsevier), 2013.

- L. Rogovchenko-Buffoni, A. **Tundis**, M. Z. Hossain, M. Nyberg, and Fritzson P. "An Integrated Toolchain For Model Based Functional Safety Analysis". Journal of Computational Science (Elsevier), 2013.

- A. Garro, and A. **Tundis**. "Enhancing the RAMSAS method for Systems Reliability Analysis through Modelica". Proceedings of 7th MODPROD Workshop on Model-Based Product Development. Linköping University, Sweden, 5-6 February, 2013.

- L. Rogovchenko-Buffoni, P. Fritzson, A. Garro, A. **Tundis**, and M. Nyberg. "Requirement Verification and Dependency Tracing During Simulation in Modelica". Proceedings of the 8th EUROSIM Congress on Modelling and Simulation (EUROSIM). Cardiff, Wales, United Kingdom, 10-13 September, 2013.

- A. Garro, A. **Tundis**, L. Rogovchenko-Buffoni, and P. Fritzson. "From Safety Requirements to Simulation-driven Design of Safe Systems". Proceedings of the 12th International Conference on Modeling and Applied Simulation (MAS). Athens, Greece, 25 - 27 September, 2013.

- A. **Tundis**, L. Rogovchenko-Buffoni, P. Fritzson, A. Garro. "Modeling System Requirements in Modelica: Definition and Comparison of Candidate Approaches". Proceedings of the 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT). University of Nottingham, UK, 19 April, 2013.

- A. Garro, and A. **Tundis**. "A Model-Based method for System Reliability Analysis". Proceedings of the Symposium On Theory of Modeling and Simulation (TMS), part of the SCS SpringSim 2012 conference, Orlando, FL, USA, 26-29 March, 2012.

- A. Garro, and A. **Tundis**. "Enhancing the RAMSAS method for System Reliability Analysis: an exploitation in the Automotive Domain". Proceedings of the International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH). Rome, Italy, 28-31

July, 2012.

- A. Garro, A. **Tundis**, J. Groß, and M. Riestenpatt Gen. Richter. "Experimenting the RAMSAS method in the reliability analysis of an Attitude Determination and Control System (ADCS)". Proceedings of the International Workshop on Applied Modeling and Simulation (WAMS), jointly held with the NATO CAX FORUM 2012, Rome, Italy, 24 -27 September, 2012.

- A. Garro, and A. **Tundis**. "Modeling and Simulation for System Reliability Analysis: The RAMSAS Method". Proceedings of the IEEE SoSE 2012 7th International Conference on System of Systems Engineering (SoSE). Genoa, Italy, 16-19 July, 2012.

- A. Garro, A. **Tundis**, and N. Chirillo. "System reliability analysis: a Model-Based approach and a case study in the avionics industry". Proceedings of the 3rd Air and Space International Conference (CEAS). Venice, Italy, 24-28 October, 2011.

- A. Garro, W. Russo, and A. **Tundis**. "Developing Service-Oriented Applications: A Method Engineering Based Approach". Proceedings of the International Conference on Semantic Web and Web Services (SWWS). Las Vegas, Nevada, USA, 18-21 July, 2011.

- F. De Luca, A. **Tundis**, and A. Garro. "PROCE: an agent-based PROcess Composition and execution Environment". Proceedings of the 12th Workshop on Objects and Agents (WOA). Rende, Italy, 04-06 July, 2011.

*Publications under revision*:

- A. Garro, and A. **Tundis**. "On the Reliability Analysis of Systems and Systems of Systems: the RAMSAS method and related extensions". Submitted to IEEE Systems Journal.

- A. Garro, and A. **Tundis**. "RAMSAS4Modelica: a Simulation-driven Method for System Dependability Analysis centered on the Modelica language and related tools". Submitted to the Symposium On Theory of Modeling and Simulation (TMS), part of the SCS SpringSim 2014 conference, Tampa, FL, USA, 13-16 April, 2014.

# 2

# Background and Challenges

In many industrial domains and, in particular, in those contexts which deal with mission critical systems, there is the need to guarantee suitable level of system operation and performances. The collective term used to describe such important characteristics is called Dependability; one of the most popular definition of Dependability, provided by the IEC (International Electrotechnical Commission) [68], is:

*"The collective term used to describe the availability performance of a system and its influencing factors: reliability performance, maintainability performance and maintenance support performance".*

The engineering discipline [42] which aims at providing an integrated and methodological approach to deal with system dependability is commonly indicated by the acronym RAMS (Reliability, Availability, Maintainability and Safety) and RAMS analysis indicates the engineering task whose main objective is to identify causes and consequences of system failures. In particular, RAMS engineering deals with making systems: (i)Reliable - low probability of failing; (ii)Available - high probability of working at a given time or when required; (iii) Maintainable - easy to maintain and keep in optimum condition; (iv) Safe - low risk of causing hazardous events.

In this Chapter the attention is mainly directed towards Reliability and Safety properties. In particular, in Section 2.1 a mathematical definition of Reliability is provided along with a deep description of the concept of Safety; in Section 2.2 classic methods, available in literature, that are mostly exploited for performing statically reliability and safety analysis are described; then the main efforts concerning model-based approaches for analyzing reliability and safety properties of systems by adopting modern analysis techniques and centered on Modeling and Simulation are described in Section 2.3; finally, some open issues are highlighted in Section 2.4.

Reliability represents the ability of a system to perform its required functions under stated conditions, identified during its design, for a specified period

of time [19]. It is an important property to be considered for a wide range of systems, and, it becomes even crucial, when systems called mission-critical are considered, that is to say when the failure of a system may lead to catastrophic losses in terms of cost, environmental damages, or even human lives, as in several industrial and service domains such as avionics, railway, automotive, energy and health care. The formal definition of Reliability relies on the important concept of time-to-failure which is modeled as a random variable T and by the following related functions:

1. Reliability function (2.1), which indicates the probability that the system fails after time t:
$$R(t) = Pr(T > t) \tag{2.1}$$
2. Failure distribution (2.2), which is a cumulative distribution function and indicates the probability that a system fails before time t:
$$F(t) = Pr(T \leq t) = 1 - R(t) \tag{2.2}$$
3. Failure density function (2.3), which indicates how the probability of failure is distributed over the life of the system::
$$f(t) = \frac{d}{dt}F(t) = \dot{F}(t) \tag{2.3}$$
4. Hazard function (or hazard rate) (2.4), which indicates the probability of a system failure between t and t + $\Delta t$ given that it was operating at time t, and becomes the instantaneous failure rate as *Delta* t tends to zero:
$$\lambda(t) = \lim_{\Delta t \to 0} \frac{R(t) - R(t + \Delta t)}{R(t) \cdot \Delta t} = \frac{\dot{R}(t)}{R(t)} \tag{2.4}$$
For a wide range of mechanical systems the function lambda (t) assumes a typical shape called Bathtub curve which shows three different stages of failures during the system life (see Figure 2.1):
   - early failures, the failure rate is high due to the infant mortality of system components but rapidly decreasing;
   - random failures, the failure rate is low and constant (useful life of the system);
   - wear-out failures, the failure rate increases as age and wear take their toll on the system components.
5. Mean Time To Failure (MTTF) (2.5), which represents the expected value of the time-to-failure random variable T:
$$MTTF = E(T) = \int_0^\infty f(t)dt \tag{2.5}$$

In Table 2.1 the expressions of the above introduced functions in terms of each of them are reported. The valuation and evaluation of these reliability

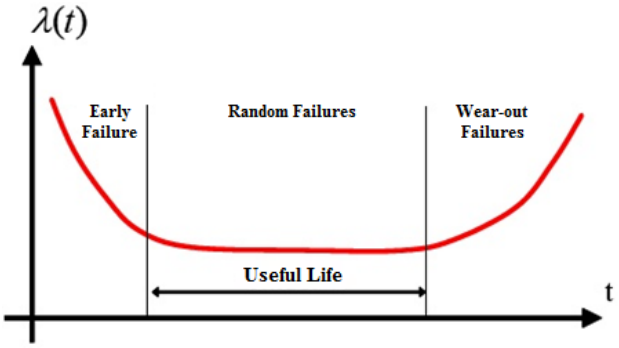



**Fig. 2.1.** Bathtub curve

functions represent a crucial task for a quantitative analysis of the reliability properties of a system. Beside quantitative analyses, qualitative analyses, which aim to identify the possible system failures, their rate of occurrence, and (local and global) effects on the system, should be also executed [85] so to complement the information obtained from the quantitative ones.

**Table 2.1.** Main Reliability Functions

| | ***f(t)*** ***Failure density function*** | ***F(t)*** ***Failure distribution*** | ***R(t)*** ***Reliability function*** | ***λ(t)*** ***Hazard function*** | ***MTTF*** ***Mean Time to Failure*** |
|---|---|---|---|---|---|
| ***f(t)*** | | $\int_0^t f(\tau)\,d\tau$ | $\int_t^\infty f(\tau)\,d\tau$ | $\frac{f(t)}{1-\int_0^t f(\tau)\,d\tau}$ | $\int_0^\infty t\,f(t)dt$ |
| ***F(t)*** | $\frac{d\,F(t)}{dt}$ | | $1-F(t)$ | $\frac{d\,F(t)}{dt} * \frac{1}{1-F(t)}$ | $\int_0^\infty (1-F(t))dt$ |
| ***R(t)*** | $\frac{-d\,R(t)}{dt}$ | $1-R(t)$ | | $\frac{-d\,R(t)}{dt} * \frac{1}{R(t)}$ | $\int_0^\infty R(t)dt$ |
| ***λ(t)*** | $\lambda(t)e^{-\int_0^t \lambda(\tau)d\tau}$ | $1-e^{-\int_0^t \lambda(\tau)d\tau}$ | $e^{-\int_0^t \lambda(\tau)d\,\tau}$ | | $\int_0^\infty e^{-\int_0^t \lambda(\tau)d\,\tau}\,dt$ |

In life data analysis and accelerated life testing data analysis, as well as other testing activities, one of the primary objectives is to obtain a life distribution that describes the times-to-failure of a component, subassembly, assembly or system. This analysis is based on the time of successful operation or time-to-failure data of the item (component), either under use conditions or from accelerated life tests. For any life data analysis, the analyst chooses a point at which no more detailed information about the object of analysis is known or needs to be considered. At that point, the analyst treats the object of analysis as a "black box". The selection of this level (e.g. component, subassembly, assembly or system) determines the detail of the subsequent

analysis. In this kind of system reliability analysis a model of a system is built from these component models, that in turn concerns with the construction of a model in terms of life distribution. The life distribution represents the times-to-failure of the entire system based on the life distributions of the components, subassemblies and/or assemblies ("black boxes") from which it is composed, as illustrated in Figure 2.2.

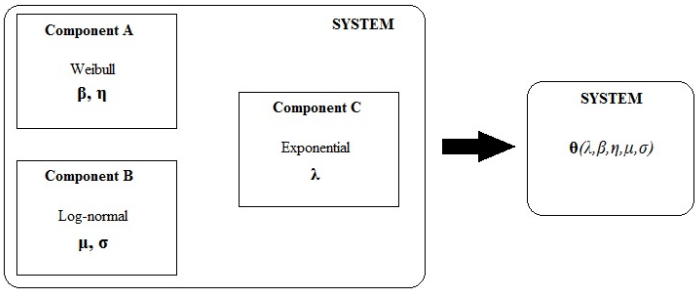



**Fig. 2.2.** Hierarchical composition of a System model

To accomplish this, the relationships between components are considered and decisions about the choice of components can be made to improve or optimize the overall system reliability, maintainability and/or availability. There are many specific reasons for looking at component data to estimate the overall system reliability. One of the most important is that in many situations it is easier and less expensive to test components/subsystems rather than entire systems. Many other benefits of the system reliability analysis approach also exist and will be presented throughout this reference [75].

From the other hand, Safety Analysis is a discipline of Safety Engineering whose aim is to ensure that engineered systems provide acceptable levels of safety through the identification of safety related risks, eliminating or controlling them by design and/or procedures, based on acceptable system safety precedences [80, 81, 23]. Indeed the system safety concept calls for a risk management strategy based on identification, analysis of hazards and application of remedial controls using a systems-based approach [45].

System safety uses systems theory and systems engineering approaches to prevent foreseeable accidents and minimize the effects of unforeseen ones. It considers losses in general, not just human death or injury. Such losses may include destruction of property, loss of mission and environmental harm. Safety of systems needs to be planned in an integrated and comprehensive engineering framework that requires experience in the application of safety engineering principles by exploiting well-known analysis techniques to perform safety analysis for the identification and the management of hazards. The general definition of Safety is based on the main concept of risk which is the

combination of the probability of a failure event and the severity resulting from the failure.

Among non-functional requirements, Safety, which represents an important requirement to be satisfied for a wide range of systems [68], becomes even more crucial in several industrial domains such as nuclear plants, medical appliances, avionics, automotive and satellite [42, 67, 101]. In particular, in the automotive domain, although Safety has always played a key role, the importance that is attributed to it has become far greater in recent times [47, 83]. In modern automobile design, Safety Requirements can be generally categorized in three main classes: (i) Passive safety, which aims to minimize the severity of an accident; examples of passive safety elements are seatbelts, crumple zones, airbags; (ii) Active safety, which aims to avoid accidents and to minimize their effects if they occur; examples of active safety elements are: predictive emergency braking, seatbelt pre-tensioning, anti-lock braking systems and traction control; (iii) Functional safety, which aims to ensure that both the electrical and electronic systems (such as power supplies, sensors, communication networks, actuators, etc.), also including all active safety related systems, function correctly. In other words, Functional safety aims to guarantee the absence of unacceptable risk due to hazards caused by malfunctioning behavior of electrical and electronic systems.

Many interests in such disciplines, from both academic institutions and research centers as well as military and civil international organizations such as NASA (The National Aeronautics and Space Administration) [79], ESA (European Space Agency) [21], EADS (European Aeronautic Defence and Space Company) [20], is even more increasing. Indeed some efforts about the standardization of methods, processes, tools and practices with particular focus on the reliability and safety are currently under consideration as it is shown in the next subsections.

## 2.1 Classical Methods and Techniques for System Reliability analysis and Safety

A first classification about Methods and Techniques for System Reliability analysis and Safety is about performing qualitative and/ or quantitative analysis, as a consequence the following main categories of analysis and related techniques have been identified:

Quantitative analysis techniques (such as Series-Parallel and Markov Chains) are based on the identification and modeling of physical and logical connections among system components and on the analysis of their reliability through statistical methods and techniques, but very often probabilistic information is not so relevant or desired, for example, when one wants to study the reachability of a state of the system, as a consequence Qualitative analysis techniques are often preferred [85, 102].

Qualitative analysis techniques aim to identify the possible system failures, their rate of occurrence and (local and global) effects on the system so to individuate corrective actions; two main techniques are currently exploited: FMECA (Failure Modes Effects and Critical Analysis) [43] and FTA (Fault Tree Analysis) [63]. To perform quantitative and qualitative RAMS analyses, several techniques are currently available which are mainly based on statistical and probabilistic tools and on the hierarchical decomposition of the system in terms of its components.

Most techniques were originally conceived mainly for electromechanical systems but, with the increasing adoption of software components in many modern systems, some extensions for embedded and software intensive systems have been proposed (e.g. S-FMECA, S-FTA) [68] along with specific software-oriented techniques (e.g. HSIA, SCCFA, PSH) [22]. Furthermore several techniques for performing quantitative and qualitative safety analyses are currently available. The most adopted techniques for performing quantitative and qualitative RAMS analyses are reported in subsequent paragraphs (see Table 2.2).

**Table 2.2.** Reliability Analysis Techniques

| | Quantitative Analysis | Qualitative Analysis | Suitable for Software Intensive Systems |
|---|---|---|---|
| Series-Parallel (RBD) | $x$ | – | – |
| Markov Chains | $x$ | – | – |
| FMEA/FMECA | – | $x$ | $x(S-FMEA$, $S-FMECA)$ |
| FTA | – | $x$ | $x(S-FTA)$ |
| HAZOP | – | $x$ | $x$ |
| HSIA | – | $x$ | $x$ |
| SCCFA | – | $x$ | $x$ |
| PSH | – | $x$ | $x$ |

The *Fault Hazard Analysis (FHA)* is a deductive method of analysis that can be used exclusively as a qualitative analysis or, if desired, expanded to a quantitative one [94]. The Fault Hazard Analysis requires a detailed investigation of the subsystems to determine component hazard modes, causes of these hazards, and resultant effects to the subsystem and its operation. This type of analysis belongs to a family of reliability analysis techniques which comprehends FMEA/FMECA (Failure Mode and Effects Analysis/Failure mode effects and criticality analysis). The main difference between the FMEA/FMECA and the Fault Hazard Analysis is a matter of depth. Wherein the FMEA or FMECA looks at all failures and their effects, the Fault Hazard Analysis deals only with those effects that are safety related.

*Reliability Block Diagrams (RBDs)* use logical blocks to link a complex system state to the states of its components [71]. A block, representing a component, can be viewed as a "switch" that is "closed" when the block is operating and "open" when the block is failed. System is operational if a path of closed "switches" is found from the input to the output of the diagram. Blocks can be connected in series (to represent components that are all required for system functioning), in parallel (to represents blocks of which at least one is required), in a *k-of-n* structure (when at least $k$ *out of* $n$ components are required). The overall structure can be composed of all of these kinds of connection, leading either to series-parallel RBDs (that can be solved by simple series-parallel reductions) or to non-series-parallel RBDs (that can be solved by state enumeration, factoring, conditioning, or binary decision diagrams (BDDs).

*Fault Tree Analysis (FTA)* is a popular and productive hazard identification tool [14, 82]. A FTA is a deductive or backward logic representation which involves specifying a top event to analyze (a system failure), followed by identifying all of the associated elements in the system that could cause that top event to occur. It provides a standardized discipline to evaluate and control hazards. The FTA process is used to solve a wide variety of problems ranging from safety to management issues. This tool is used by the professional safety and reliability community to both prevent and resolve hazards and failures. Both qualitative and quantitative methods are used to identify areas in a system that are most critical to safe operation. The output is a graphical presentation providing a map of "failure or hazard" paths.

*Event Tree Analysis (ETA)* is an analysis technique for identifying and evaluating the sequence of events in a potential accident scenario following the occurrence of an initiating event [65]. ETA is an inductive or forward logic representation, which starts from an initiating event and includes all possible paths, whose branch points represent successes and failures. The objective of ETA is to determine whether the initiating event will develop into a serious mishap or if the event is sufficiently controlled by the safety systems and procedures implemented in the system design. An ETA can result in many different possible outcomes from a single initiating event and it provides the capability to obtain a probability for each outcome.

*Common Cause Failure Analysis (CCFA)* is an extension of FTA to identify "coupling factors" that can cause component failures to be potentially interdependent [72]. Primary events of minimal cut sets from the FTA are examined through the development of matrices to determine if failures are linked to some common cause relating to the environment, location, secondary causes, human error, or quality control. A cut set is a set of basic events (e.g. a set of component failures) whose occurrence causes the system to fail. A minimum cut set is one that has been reduced to eliminate all redundant "fault paths". CCFA provides a better understanding of the interdependent relationship between FTA events and their causes. It analyzes safety systems for "real" redundancy.

*Sneak Circuit Analysis (SCA)* is a method for the evaluation of electrical circuits [95]. SCA employs recognition of topological patterns that are characteristic of all circuits and systems. The purpose of this analysis technique is to uncover latent (sneak) circuits and conditions that inhibit desired functions or cause undesired functions to occur, without a component having failed. The process converts schematic diagrams to topographical drawings and searches for sneak circuits.

The *Energy Trace* is a hazard analysis approach addresses all sources of uncontrolled and controlled energy that have the potential to cause an accident [5]. Examples include utility electrical power and aircraft fuel. Sources of energy causing accidents can be associated with the product or process. The purpose of energy trace analysis is to ensure that all hazards and their immediate causes are identified. Once the hazards and their causes are identified, they can be used as top events in a fault tree or used to verify the completeness of a fault hazard analysis. Consequently, the energy trace analysis method complements but does not replace other analyses, such as fault trees, sneak circuit analyses, event trees, and FMEAs.

Even though the above mentioned techniques are fairly popular for the safety static analysis of systems, nowadays, with the increase of complexity and heterogeneity of modern systems, more dynamic and flexible analysis techniques, based on simulation methods as well as compliant with international safety standards for specific domains, are even more required. The main and more significant contributions are presented in the next paragraph.

## 2.2 Model-based Methods for the Reliability and Safety analysis of Systems

From a MBSE point of view, a Method can consist of a set techniques for performing a task. At any level, process tasks are performed using methods. However, each method is also a process itself, with a sequence of tasks to be performed for that particular method.

For several years many SE methods, that describe how using modeling approaches to carried out system design tasks, have been proposed, but few of them include approaches to deal with verification of requirements, and in particular with reliability and safety design and analysis of systems in a comprehensive and integrated way in order to improve the realization and the performances of a system during the System Engineering (SE) Development process and its early design phases. In the following some of me most adopted and popular model based method present in literature are described. Nowadays new emerging techniques for system reliability and safety analysis, which are centered on model-based approaches [84, 108] and incorporate the use of simulation [93], are emerging.

One of these method is MéDISIS [16], a deductive and iterative approach that aims at facilitating crucial reliability analysis and enhancing the use of

the diverse tools and languages used for dysfunctional behavior validation. It is used within an industrial project to design a hypersonic aircraft which is a relevant complex and critical system. MéDISIS applies model based techniques and combining the advantages of established modeling tools with more formal analysis methods by providing a tool-based solution and it is domain-specific oriented. In MéDESIS the inputs to the models are expressed in SysML and a repository registering and managing the knowledge raised by the activities performed in a structure modeled in this language has been built, in particular a model based dysfunctional behind the FIDES [24] guide has been designed as a reliability database. The main assumption that was made for the constitution of MéDISIS, was to consider that the method used by the designer to construct the functional model of the system is let totally free. This assumption was taken to make it possible to integrate MéDISIS to others MBSE methods tackling the other tasks of system development. MéDISIS includes the following steps: (i)deduction of the dysfunctional behavior with an FMEA, identification of the impacted requirements; (ii) construction of a model integrating functional and dysfunctional behaviors with a formal language; (iii) analysis and quantification of dysfunctional behavior. To perform the steps of MéDISIS several tools and analysis routines have been defined to support each phase and optimize the rapidity and quality of the reliability studies. These developments are made to construct a complete System Development Environment (SDE) supporting MéDISIS and system design. Analysis techniques and a tool have defined to support FMEA realization. The results of this study raise the dysfunctional behavior of each component. They are capitalized in a dysfunctional models repository and reused to construct a formal representation of the system using the AltaRica Data Flow [8] language. The construction of this formal model, mandatory for system validation, is also helped by analyses techniques systematizing the creation of this reliability-oriented view. A service to support embedded systems analysis, which proposes to generate AADL [111] models exploitable for real time application studies using Cheddar tool, is available in it .

Another effort in the context of model-based engineering and reliability analysis is centered on AltaRica language. Altarica is a language used to describe critical systems whose base component of an Altarica model is called node. Its structure may comprise the following sections: sub, state, event, init, flow, trans, assert, sync, extern. An Altarica model [3, 99] is a hierarchical graph composed of nodes. At the same level of the hierarchy, nodes communicate through flows and synchronizations. The hierarchy yields a tree structure, where two types of nodes are possible: component which represents a single process of the system, it cannot contain definition of subnodes or synchronizations; equipment which represents a container for nodes; it may contain declarations of subnodes and synchronizations, but it cannot have state variables. This structure imposes that the component nodes represent the leafs, whereas the equipment nodes are containers for the components. Moreover, there is a special equipment node called main, which represents

the root of the full Altarica model. The Altarica language was designed to specify the behavior of a system when faults occur, and and Altarica model can be evaluated by complementary tool, such as fault tree generator and model checker. From the Fault model view, a failure mode state of a node can be achieved using a transition that takes the particular failure event. The semantics of the Altarica model is defined in terms of Interfaced Transition Systems (ITSs) [2]. Intuitively, the ITS associated with a component is given straightforwardly by the state variables (which define the states), the initial condition, the transitions, the events and flow variables (which define the observations) of the node. The ITS associated to an equipment node is given by the composition of the ITSs associated with the subnodes taking into account synchronizations. The evolution of an Altarica system can be further constrained by associating events with special laws and priorities. By default, events are considered stochastic. These events are typically used to model component failures and can be optionally associated with a probability distribution law (e.g. Exponential law). These laws are used to establish interoperability with commercial RAMS (Reliability, Availability, Maintainability and Safety) analysis tools and do not affect qualitative behavior of the system. An Altarica fault tree generator takes in input an Altarica model along with some unexpected event and generates a fault tree for non-temporal failure conditions.

In the framework of the probabilistic safety analysis of the Paluel nuclear power plant, EDF has developed software packages allowing the automation construction and assessment of reliable models. Our concern for unification of the software packages, explanation of the reliability expert's modeling choices, and generality has led us to design a unique system modeling language (the FIGARO language) [8, 7] which is independent from the processing method used afterwards. This language has been worked out in order [6]: (i) to give a suitable formalism for setting up knowledge bases (with generic component descriptions), (ii) to be more general than all conventional reliability models, (iii) to make the best possible compromise between modeling power (or generality) and processing tractability, (iv) to be as readable as possible, (v) to be easily associated with graphic representations. On the basis of a FIGARO language modeling, different compilers and translators allow to deduce automatically the data which are necessary for the classical reliability model processing codes: fault trees, Markov chains, Petri nets, etc. Models in Figaro are objects, and may have properties such as conditions, constraints and relationships to other objects. Figaro model classes are created by inheriting functionality from existing classes. Figaro provides a modular, compositional Metropolis-Hastings algorithm, and gives the modeler optional control over it. FIGARO is part of the so-called "hybrid" languages with specific object-oriented type syntax and semantics, that is to say it takes some of its features from the object-oriented languages and models the behavior of an object through order 1 production rules. The use of rules offers two advantages (i) The rules are close to the natural language if their syntax is selected appropriately: their

use will improve the model readability; (ii) EDF has got the mastery of different tools in this field and, in particular, worked on the validation of 0 order production rule bases. Conventional model, such as a fault tree, a reliability diagram, would be built more rapidly than a FIGARO based model, which obliges to structure and formalize the concepts being handled more or less consciously in the production of the specific model. In return, it won't be at all reusable for carrying out a second study of the same type. FIGARO gives a very satisfying answer to this request through the possibility it offers to create "knowledge micro-bases" corresponding to the classical models. These bases allow graphic model acquisition. More generally, it is important to notice that any simple graphic language can be supported by means of a small, quickly written FIGARO knowledge base. Besides, the FIGARO based modelling allows to access the full available processing set: for example it is possible to assess the reliability of a system represented through a fault tree by a Monte Carlo simulation, which is feasible whatever the FIGARO model, or by the analytical calculations of GSI whereas most of the fault tree codes do not permit such a calculation (for a repairable system).

Another effort has been performed in a method for the virtual verification of system designs against system requirements (vVDR) by means of simulation is proposed [107, 70]. It is based on the Modelica language Modelica is an object-oriented equation-based modeling language primarily aimed at physical systems [87, 25]. It points out that this method strongly depends on the design models that are planned to be created and that not all type of requirements can be evaluated using this method. In the vVDR approach, formalized requirements, system design and test cases are defined in separate models and can be reused and combined into test setups in an efficient manner. In this methodology It is assumed that the requirements from the customer have been elicited as requirement statements according to common standards in terms of quality, e.g. according to the work in [51] stating that the individual requirements should be unique, atomic, feasible, clear, precise, verifiable, legal, and abstract, and the overall set of requirements should be complete, non-redundant, consistent, modular, structured, satisfied and qualified. The methods to achieve this have been well defined and can be considered to be established.

## 2.3 Importance of Regulation and main International Standards

Despite a general consensus on the advantages that could derive from the exploitation of model-based approaches for system reliability analysis and safety, the use of these techniques has been traditionally unusual and has not been recommended by international standards until recently. Nowadays, indeed being compliant to international standards for specific domains is even

more required, and often it is very crucial during all the different development phases of a system. In the following paragraphs an overview of the most common reference standard provided and/or exploited by international organization such as IEC (International Electrotechnical Commission), NASA, ESA as well as EADS company [54, 79, 21, 20], is reported.

The standard IEC-61508 [53] includes all aspects of Electrical, Electronical and Programmable electronical Systems, for safety related function and usability as well as how systems have to be developed, tested, used and maintained according national and internationals standards. It enables a systematic and risk based methodology for safety related problems. The standard IEC/EN 61508 is detailed in seven chapters, but only the first four, as listed below, present normative requirements for the development:

- IEC/EN 61508-1: General Requirements;
- IEC/EN 61508-2: Hardware Requirements;
- IEC/EN 61508-3: Software Requirements;
- IEC/EN 61508-4: Notation and abbreviations;
- IEC/EN 61508-5: Example for calculating different safety integrity levels (SIL);
- IEC/EN 61508-6: Application guidelines for IEC/EN 61508-2 and IEC/EN 61508-3;
- IEC/EN 61508-7: Overview of techniques and actions.

ISO-26262 represents the reference standard in automotive domain [59]. Its basis resides in the more generic IEC-61508 [53] which has a broad field of application (industrial process, control and automation, oil/gas, nuclear, etc.). However, ISO-26262 is totally dedicated to the automotive sector and allows car manufacturers to indemnify themselves from liability in case a malfunction remains undetected when following the standard [67]. At the process level, this standard allows to follow a clear guidance on the development and validation of electrical and electronic systems, avoiding errors in the design and implementation, which could otherwise induce more expensive production activities and delay during the development [119]. Moreover, a well-defined and standardized development process, which goes from the Requirements Analysis phases up to the System Testing phases, allows supporting the traceability of Safety Requirements during all the intermediate development stages. As an example, the Process Deployment Advisory Service defined on ISO-26262 in order to help identifying gaps in the development processes, including requirements traceability and requirements based-testing, is fully supported by popular tools such as MATLAB-Simulink [1, 106, 113, 114] .

The RTCA - DO 254 is another important standard prepared by RTCA Special Committee 180 and approved by the RTCA Program Management Committee on April 19, 2000 [98]. This document provides guidance for design assurance of airborne electronic hardware from conception through initial certification and subsequent post certification product improvements to ensure continued airworthiness. It was developed based on showing compliance

with certification requirements for transport category aircraft and equipment but parts of this document may be applicable to other equipment. The guidance of this standard is applicable, but not limited to, the following hardware items: Line Replaceable Units (LRUs), Circuit Board Assemblies, Custom micro-coded components such as Application Specific Integrated Circuits (ASICs) and Programmable Logic Devices (PLDs) including any associated macro functions, Integrated technology components such as hybrids and multichip modules and Commercial-Off-The-Shelf (COTS) components.

RTCA - DO 178B are software consideration in airborne systems and equipment certification [96]. This standard was jointly prepared by EUROCAE Working Group 12 and RTCA Special Committee 167, ad accepted by the Council of EUROCAE on December 10, 1992. The purpose of this document is to provide guidelines for the production of software for airborne systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements. In particular these guidelines are in the form of: Objectives for software life cycle processes, Descriptions of activities and design considerations for achieving those objectives, Description of the evidence which indicate that the objective have been satisfied.

RTCA - DO 178C Software Considerations in Airborne Systems and Equipment Certification is the title of the recently published document from RTCA, Incorporated, in a joint effort with EUROCAE [97]. It was completed in November 2011 and approved by the RTCA in December 2011. It has been released to provide clarification and address inconsistencies in Do 178B, as well as introduce technology advancement in the areas of certifiable software development through technology supplements: (i) Software Tool Qualification considerations (in DO-330), (ii) Model-based Development and Verification Supplement (in DO-331), (iii) Object Oriented Technology and related Techniques Supplement (in DO-332) and Formal Methods Supplement (in DO-333). Tool Qualification plays a key role in RTCA - DO 178C standard where the terms "development tool" and "verification tool" have been replaced by three tool qualification criteria that determine the applicable tool qualification level (TQL) in regards to software design assurance level.

ECSS-Q80-03 is a standard defined by the European Space Agency (ESA) concerning Methods and techniques to support the assessment of software dependability and safety [22]. This Standard was prepared by the ECSS-Q-80-03 Working Group, reviewed by the Product Assurance Panel, and approved by the ECSS Steering Board. It is one of the series of ECSS Standards intended to be applied together for the management, engineering and product assurance in space projects and applications. The scope of this standard is limited to assessment aspects not including development and implementation techniques for dependability (e.g. fault tolerance techniques, or development methods like coding standards are not covered). It provides support for the selection and application of reliability and safety methods and techniques that can be used

for the assessment of software intensive systems and the inputs to be provided for the system level analysis.

NASA/SP-2007-6105 is another efforts which have been accomplished by the National Aeronautics and Space Administration (NASA) [80]. It provides top-level guidelines for good systems engineering practices. The approach provided the opportunity to obtain best practices from across NASA and bridge the information to the established NASA systems engineering process. It consists of six core chapters: (1) systems engineering fundamentals discussion, (2) the NASA program/project life cycles, (3) systems engineering processes to get from a concept to a design, (4) systems engineering processes to get from a design to a final product, (5) crosscutting management processes in systems engineering, and (6) special topics relative to systems engineering.

## 2.4 Conclusion

The increase in both system complexity and accuracy required in the reliability and safety analysis often goes beyond the capabilities of the above mentioned techniques, which are usually employed for a static/structural analysis of systems. Moreover, their integration in typical system development processes [48], and especially in the design phases, is quite difficult and then their use is often postponed to the later development stages [61, 110]. Despite a general consensus on the advantages that could derive from the exploitation of model-based approaches which incorporate the use of simulation techniques to analysis the behavior of systems and their evolution, the delay in the adoption is mainly due to the lack of methods able to integrate available modeling languages, tools and techniques in a consistent modeling framework. So, the need of emerging methods, which are centered on model-based approaches, so to benefit from the available modeling practices and which incorporate the use of simulation to flexibly evaluate during the design the system reliability and safety performance and compare different design choices [88, 66], are nowadays even more required. As a consequence, the present Thesis work has been conceived by starting from these considerations.

# 3

# RAMSAS: a model-based method for the reliability analysis of systems

In this Chapter, RAMSAS, the proposed model-based method for the reliability analysis of systems through simulation, is described [29]. RAMSAS aims contributing to fill the lack of methods specifically conceived for addressing the analysis and verification of non-functional requirements. In particular, the attention is focused on system reliability which is a key requirement to satisfy especially for mission critical systems where system failures could cause even human losses. RAMSAS is the result of an intensive experimentation in several application domains (avionics, automotive, satellite) [27, 28, 30, 31, 34] which allowed improving the effectiveness of the method especially in the modeling of both the expected and dysfunctional system behavior.

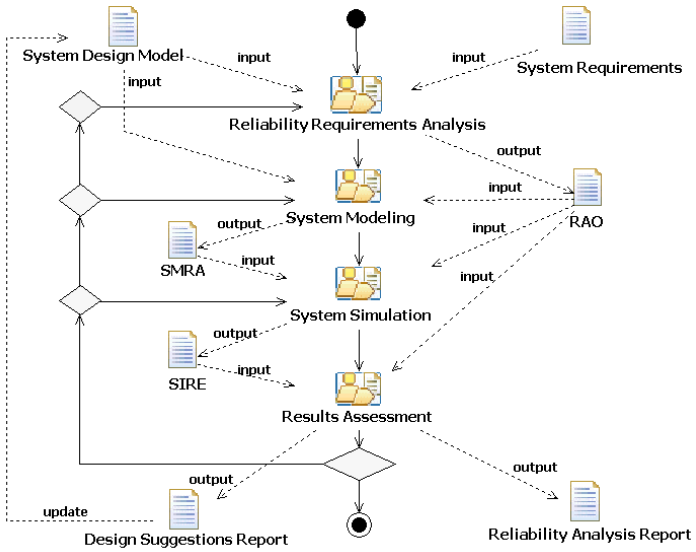



**Fig. 3.1.** The RAMSAS Method - A process view

RAMSAS is based on a classical iterative process consisting of four main phases (see Figure 3.1): *Reliability Requirements Analysis, System Modeling, System Simulation*, and *Results Assessment*. In the first phase (*Reliability Requirements Analysis*), the objectives of the reliability analysis are specified and the reliability functions and indicators to evaluate during the simulation are defined. In the *System Modeling* phase, the structure and behavior of the system are modeled in SysML (OMG Systems Modeling Language) by using zooming in-out mechanisms [78]; moreover, beside the intended system behaviors, specific dysfunctional behaviors and related tasks, which model the onset, propagation and management of faults and failures, are introduced. In the *System Simulation* phase, the previously obtained models of the system are represented in terms of the constructs offered by the target simulation platform (currently MATLAB-Simulink). Finally, simulation results are analyzed with respect to the objectives of the reliability analysis; if necessary, new partial or complete process iterations are executed. RAMSAS is defined as a Method Fragment [9, 10, 46] and, as a consequence, it can be integrated in various phases of a typical System Development Process (e.g. in a V-Cycle process, see Figure 3.2): (i) in the verification phases to support the verification of system reliability after the actual realization of the system; (ii) in the design phases to support the evaluation and validation of configuration scenarios and settings of system parameters so to guide and suggest design choices before the realization of the system.

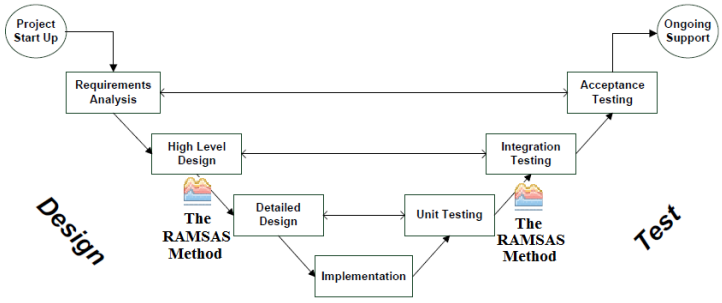



**Fig. 3.2.** The RAMSAS method plugged into a V-Model

The contributions provided in this Section is related with Model-based Methods described in Section 2.2 and, in particular, with MeDISIS [16], a deductive and iterative approach that aims at facilitating crucial reliability analysis. Both the RAMSAS method and MeDISIS approach rely on simulation based analysis and verification against description of the required behavior, but RAMSAS provides a more methodological solution whereas MeDISIS approach provides a tool-based solution and it is domain-specific oriented. In the following sections a more detailed description of the RAMSAS Method is provided in terms of models, phases and related work-products.

## 3.1 Reliability Analysis Requirements

In the *Reliability Requirements Analysis* phase the objectives of the system reliability analysis are specified. The inputs of this phase are the work-products typically resulting from previous System Design phases. Starting from this documentation, the scenarios to be analyzed, the functions that the system has to perform, the related operative conditions, and the reference time horizons should be clearly individuated along with the reliability functions and indicators to be derived from the analysis of the simulation results. In particular, the *Reliability Requirements Analysis* phase takes as input a description of the system under consideration in term of both *System Requirements (SR)* and *System Design Model (SDM)*.
*SR* includes functional (*FR*) and non-functional requirements (*NFR*), whereas *SDM* provides a system representation in terms of its architecture and behavior. Among the *NFR*, the *Reliability Requirements (RR)* specify the ability required for the system in performing the functions identified by the FR under specific stated conditions and for a given period of time.
In addition, a Failure Modes and Effects Analysis (FMEA) can be also provided to highlight the potential failure modes of the system along with their severity and likelihood. Starting from the above mentioned SR, in the *Reliability Analysis Objectives (RAO)* work-product, the reliability indicators and the scenarios of interests are identified along with the main analysis techniques to be applied to the data gathered from simulation. In the *RAO*, a visual representation of the *SR* can be also provided through SysML *Requirements Diagrams* along with the allocation of these requirements (especially the reliability ones) to main system components.

## 3.2 System Modeling

In the *System Modeling* phase the structure and both the intended (also called nominal or normal or expected) and dysfunctional behavior of the system under consideration are represented in SysML by executing four modeling activities (see Figure 3.3): *System Structure Modeling, Intended Behavior Modeling, Dysfunctional Behavior Modeling* and *Behavior Integration*.

The specifications concerning the structure and intended behavior of the system are derived from the *System Design Model (SDM)* resulting from previous design phases; these activities can be straightforward if during the system design similar structural and behavioral reference models have been adopted along with a UML based modeling notation. With reference to the *dysfunctional behavior* of the system, its modeling can be based on the results reported in the *RAO* document concerning the analysis of the failure modes of the system (see Section 3.1). Finally, the intended and dysfunctional system behaviors are suitable integrated so to provide a complete system specification for the subsequent simulation phase. In the following Subsections, each of these modeling activities is discussed more in details.

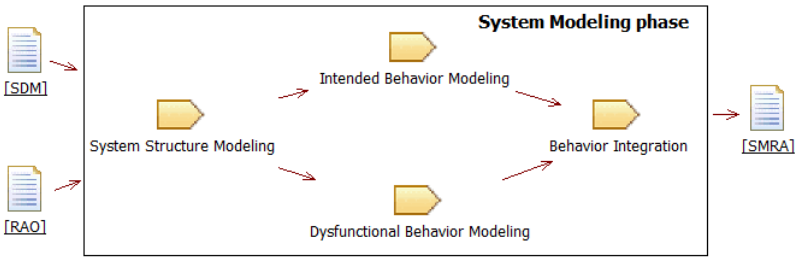



**Fig. 3.3.** System Modeling phase

### 3.2.1 System Structure Modeling

In the *System Structure Modeling* activity, the system structure is modeled by using SysML Blocks following a top-down approach. To this aim, several decomposition levels should be considered by applying *in-out zooming* mechanisms [78] such as *system, subsystems, equipment*, and *components*; however to allow system analysis at the desired level of details, further abstraction levels along with different and deeper hierarchies can be also introduced. Each system entity is defined by both a *Block Definition Diagram (BDD)* and an *Internal Block Diagram (IBD)*. Specifically, for a given abstraction level, a *BDD* describes a block with its interfaces, attributes, operations, constraints, parts and relationships with other blocks; whereas, an *IBD* provides a description of the block internal structure in terms of the organization of its component blocks.

### 3.2.2 Intended Behavior Modeling

In the *Intended Behavior Modeling* activity the intended behavior of the system is represented following also a layered approach but combining the *top-down* with a *bottom-up* strategy. The reference model is service and task-oriented: the behavior of each entity is modeled in terms of the *services* (or *functions*) that the entity is able to provide and which are performed through tasks. In order to specify the behavior of the system and its component entities, two levels of decomposition are considered: *leaf level* (e.g. component level) and *non-leaf level* (e.g. equipment, subsystem or system level). In particular, for each entity at the *leaf decomposition level* (the lowest decomposition level):

- the services (or functions) provided by the entity, in terms of their input and output work-products along with pre and post conditions, should be specified;
- each task (flow of activities/actions) performed by the entity for providing a specific service (or function) has to be specified through an *Activity Diagram*; each task can be *scheduled* or *triggered* by incoming events/messages;

- for the tasks of the entity that can be naturally represented in terms of equations, *Parametric Diagrams* are introduced;
- the exchange of messages between the entity and the external environment (which can be another entity at the same or at a higher decomposition level) should be represented through *Sequence Diagrams*;
- in case the behavior of the entity depends on its internal state, a state machine which models the entity life cycle can be specified through a *Statechart Diagram*.

Moving from the *leaf decomposition level* to higher decomposition levels (*non-leaf decomposition levels*), the representation of the entity behavior is similar; however, how the component entities (i.e. sub-entities) participate and determine the behavior of the considered enclosing entity should be taken into account; as a consequence, for each entity at a *non-leaf decomposition level*:

- beside the services (or function) provided by the entity, how these services can be obtained by composing the services provided by the sub-entities should be also specified;
- each task (flow of activities/actions) performed by the entity for providing a specific service (or function) has to be specified by an *Activity Diagram* possibly highlighting through *swim-lanes* the responsibility of each sub-entity in carrying out the activities of the task;
- the exchange of messages between the entity and the external environment should be represented through *Sequence Diagrams* possibly highlighting the role played by its sub-entities in each interaction (e.g. by explicitly representing them as participants in the diagrams);
- in case the behavior of the entity depends on its internal state, a state machine that models the entity life cycle can be specified through a *Statechart Diagram*; the diagram can adopt advanced constructs, such as *composite states*, *AND/OR-decomposition* and *History pseudo-states*, for representing how the behavior of the entity is related to the behavior of its sub-entities.

### 3.2.3 Dysfunctional Behavior Modeling

In the *Dysfunctional Behavior Modeling* activity, the focus is on the modeling of *faults* and *failures*, which are key concepts of the system reliability analysis [64, 19]. Specifically, the behavior, concerning faults and failures of each system entity (i.e. the *dysfunctional behavior*), is specified as a set of specific tasks (which can be modeled through *Activity Diagrams*). The reference model of a generic system entity, regardless on the considered abstraction level, is shown in Figure 3.4. An entity is represented by a SysML Block which provides a set of services/functions; the tasks performed by the entity for providing these services are modeled during the *Intended Behavior Modeling* phase (see Section 3.2.2).

Beside the intended behavior specified through these tasks, a set of *dysfunctional tasks* are added so to model the *dysfunctional behavior* of the entity [11, 41]. In particular, each block could receive as input a set of failure events (e.g. due to the failures of other blocks) and could, in turn, produce in output other *failure events* due to its *failure*; moreover, internal *faults* (represented as *fault events*) can be generated and treated inside the block possibly producing block *failures* (and thus output *failure events*). With reference to the above described behavioral reference model (see Figure 3.4), six templates of *dysfunctional tasks* have been individuated (see Table 3.1): *Fault Generation, Failure Generation, Failure Management, Fault Management, Failure Propagation* and *Failure Transformation*.

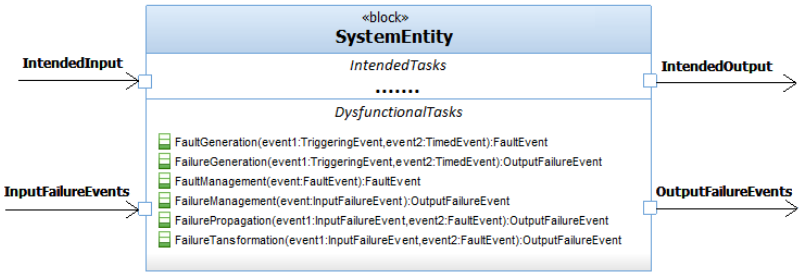
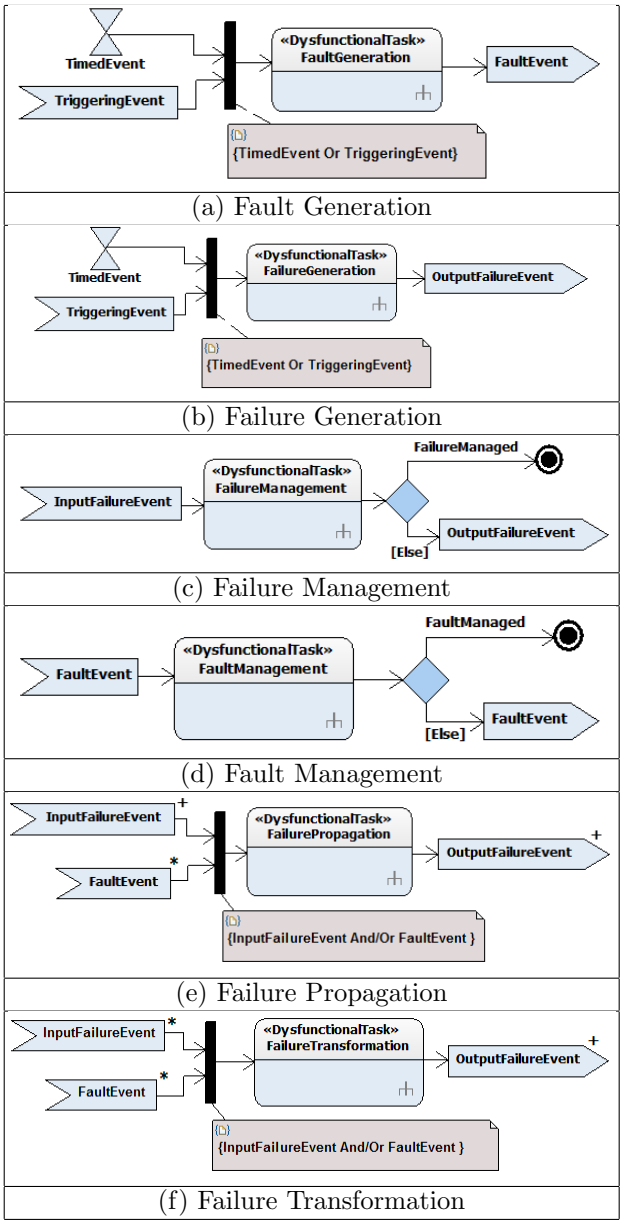



**Fig. 3.4.** The reference Behavioral Model of a system entity

Tasks of the *Fault* and *Failure Generation* type (see Table 3.1.a and 3.1.b respectively) can generate a fault/failure as a result of specific causes occurring according to a given probability function. In order to allow driving, during the simulation, the processes of fault/failure generation so as to study the reliability performances of the system, these tasks can be scheduled or triggered by specific events. However, whereas a *failure* is directly associated to an *output failure event* (see Table 3.1.b), to produce a *failure event* starting from a *fault*, the fault has to be taken in input from either a *Failure Propagation* or a *Failure Transformation task*. Incoming *failures* as well as internally generated *faults* can be (successfully or not) handled by tasks of the *Failure* and *Fault Management* type (see Table 3.1.c and 3.1.d respectively). Finally, tasks of the *Failure Propagation* and the *Failure Transformation* type take dysfunctional events in input and produce *dysfunctional events* in output. In particular, tasks of the *Failure Propagation* type (Table 3.1.e) generate *output failure events* either through the propagation of incoming failure events or by combining such incoming failures with internal faults. Tasks of *Failure Transformation* type (Table 3.1.f) produce *output failure events* derived from the transformation or combination either of incoming failure events or of internal faults.

**Table 3.1.** Templates of dysfunctional tasks




To further support this crucial modeling activity, a set of *patterns* to associate to each of the above discussed six types of *dysfunctional tasks* should be defined.

The definition of each of these patterns should take into account the type of the dysfunctional task (e.g. Failure Generation, Propagation or Transformation) as well as the specific nature of the fault/failure to which the pattern refers to; in other word, a basic pattern is associated to a couple (*dysfunctional task type; fault/failure type*). As a consequence, for the definition of these patterns, beside the individuated six *dysfunctional task types* (see Table 3.1), a (possibly hierarchical) classification of faults/failures needs to be introduced.

A first solution could consider the following fault/failure types [41]: (i) *reaction too late*; (ii) *reaction too early*; (iii) *value failure*; (iv) *commission*; and (v) *omission*. By combining the individuated six dysfunctional task types with these five fault/failure types, thirty different basic fault/failure behavioral patterns can be defined.

The modeling of the dysfunctional behavior of each system entity, in terms of a set of dysfunctional tasks of the above described types and possibly based on available fault/failure behavioral patterns, is essential to evaluate through simulation the dysfunctional behaviors of the system and analyze the possible consequences of failures as well as feasible solutions for their management in order to improve system reliability.

### 3.2.4 Behavior Integration

This is the last activity of the *System Modeling* phase of the RAMSAS method. In the *Behavior Integration* activity, the nominal/intended behaviors and the dysfunctional behaviors modeled in the previous modeling activities (see Sections 3.2.2 and 3.2.3) are integrated in order to obtain an overall behavioral model of the system and its component entities. As an example, starting from the *Activity* and *Sequence diagrams* which have been used to model both the intended and dysfunctional behaviors of the system entities, a complete and integrated definition of the life cycle of each entity, regardless on the considered abstraction level, can be obtained and represented through a *Statechart diagram*. This activity closes the System Modeling phase by delivering the *System Model for Reliability Analysis (SMRA)* work-product.

## 3.3 System Simulation

The objective of the *System Simulation* phase is to evaluate, through simulation, the reliability performance of the system and, possibly, compare different design alternatives and parameters settings; in particular, the following three main activities are performed (see Figure 3.5): *Model Transformation, Parameters Setting* and *Simulation Execution*.

In the *Model Transformation* activity the previously obtained models of the System in the *SMRA* are represented in terms of the constructs offered by the target simulation platform which is, currently, MathWorks Simulink, so producing an *Executable System Model (ESM)*. This transformation is based

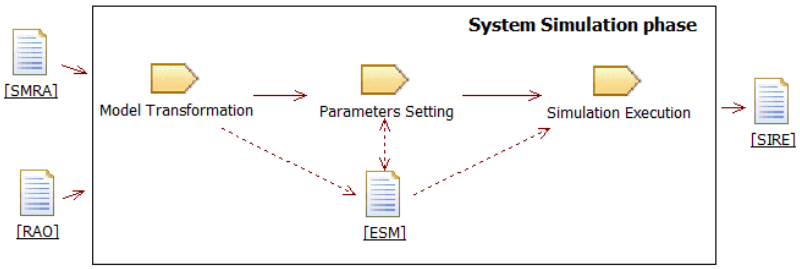



**Fig. 3.5.** System Simulation phase

on a mapping between the basic SysML and Simulink constructs (see Table 3.2) [31, 110], where the following mapping has been exploited: (i) a (simple) SysML Block is transformed into a Simulink Block; (ii) a (composite) SysML Block, consisting of other blocks (its parts), is transformed into a Simulink Subsystem Block; (iii) SysML FlowPorts are transformed into Input and Output Simulink Blocks; (iv) SysML Flow Specifications, used to type FlowPorts, are transformed into Simulink Bus Objects.

It is worth to notice that not only the intended behavior of the system but also the dysfunctional tasks (see Section 3.2.3), which are essential for analyzing during the simulation the reliability performance of the system, are generated. Indeed, this allows to suitably *injecting* faults and failures during the simulation and setting the parameters of the related generation, management, propagation and transformation tasks (see Table 3.1).

**Table 3.2.** Mapping among SysML and Simulink main constructs

| Entity | SysML | Simulink |
|---|---|---|
| System, Subsystem, Equipment, Component | Block<br>Part | Block<br>Subsystem Block |
| Behavior and Constraint | Activity diagram<br>Sequence diagram<br>Activity diagram<br>Stachart diagram | S-Function<br>Stateflow diagram |
| Input/Output Interface | Flow Port<br>Flow Specification | I/O Simulink Block<br>Simulink Bus Object |
| Association/Binding | Connection | Signal |

In the *Parameters Setting* activity the *ESM* is refined so to allow the flexible setting of system configuration and simulation parameters which can be tuned according to both the characteristics of the operative scenario to simulate and the dysfunctional behaviors to analyze.

In the *Simulation Execution* activity the *ESM* is executed by varying the desired parameters according to the analysis objectives reported in the *RAO*. Specifically, the *ESM* is executed by Simulink according to a synchronous reactive model of computation: at each step, Simulink computes, for each block, the set of outputs as a function of the current inputs and the block state, then it updates the block state. During the simulation *faults* and *failures* can be injected and/or caused to stress and analyze the reliability performance of the *system*. At the end, the data generated from the simulations are reported in the *Simulation Results (SIRE)* work-product to be analyzed in the next phase.

## 3.4 Results Assessment

In the *Results Assessment* phase, the results reported in the SIRE are elaborated with reference to the objectives of the reliability analysis identified in the initial phase of the process so to obtain important information on the reliability properties of the system under consideration. A large part of these analyses can be directly performed in Simulink, whereas more advanced analyses can be performed by external tools by exporting the obtained results through the MATLAB workspace. As a result, the following two work-products are produced in output:

- *Reliability Analysis Report (RAR)*, which provides a detailed analysis about the reliability performance of the system under consideration;
- *Design Suggestions Report (DSR)*, which provides a set of suggestion to improve the design of the system and/or choose among different design choices. It is worth to note that the *DSR* exploits typical FMECA and FTA notations and representation formats so to make easier the use of RAMSAS in conjunction with classical RAMS techniques.

As for any iterative process, new (partial or complete) iterations of RAMSAS can be executed for achieving new or missed analysis objectives.

## 3.5 Conclusion

This Chapter has presented RAMSAS, a Model-Based method for the Reliability Analysis of Systems, which combines in a unified framework: (i) the strengths of powerful visual modeling languages (such as OMG SysML), suitable to flexibly model the architectural and behavioral aspects of complex, dynamics, and heterogeneous systems; (ii) mature and popular tools (such as Mathworks Simulink), suitable for the simulation and analysis of multi-domain systems.

RAMSAS aims at filling the lack of methods which specifically address the analysis and verification of non-functional requirements. It is the result of an

intensive experimentation in several application domains (such as avionics, automotive and satellite) which allowed improving its modeling and simulation features especially in the support provided to the modeling of the dysfunctional system behaviors.

The proposed method is not intended to be an alternative to other traditional RAMS techniques (e.g. FMECA, FTA, RBD) but rather a complement to them able to provide additional analysis capabilities due to the jointly exploitation of Systems Engineering modeling and simulation languages, tools and techniques. Moreover, these distinctive features make the proposed method particularly suitable to be integrated in various phases of a typical System Development Process, especially in the design phases. This allows supporting the fulfillment and traceability of an important non-functional requirement, such as reliability, in the early stages of a development process with considerable time and cost reductions respect to more traditional reliability analyses techniques which are often carried out in the latest stages of the development, with the risk of having to revise even basic design choices.

# 4

# Experimenting the RAMSAS Method

Reliability analysis of modern large-scale systems is a challenging task which could benefit from the jointly exploitation of recent model-based approaches and simulation techniques, such as the RAMSAS Method, to flexibly evaluate the system reliability performances and compare different design choices. To prove the suitability of RAMSAS to support the reliability analysis of systems through Simulation, and thus obtain useful insights and feedback to improve its effectiveness, several experimentations in various application domains have been performed; in particular, RAMSAS has been experimented:

- in the avionics domain for the reliability analysis both of a Landing Gear System [34] and of a Flight Management System [31];
- in the automotive domain for the reliability analysis of an Electronic Stability Control (ESC) system [30].

In this Chapter, an actual exploitation of the RAMSAS Method, for the reliability Analysis of an Attitude Determination and Control System (ADCS) of a Satellite, is presented. This experimentation has been carried out in cooperation with the *Institute for Statics and Dynamics of Aerospace Structures* of the University of Stuttgart [27].

## 4.1 Reliability analysis of an Attitude Determination and Control System (ADCS)

In the following an Attitude Determination and Control System (ADCS) of a satellite [120, 44], is briefly described and then analyzed through the RAMSAS method in order to improve, where necessary, its design and/or its performances before its actual realization.

After the launch of a satellite, there is almost no possibility for the maintenance and repair of it. Despite the expensive maintenance missions to the Hubble telescope [50], satellite missions depend on the faultless operation of

the system over the whole lifecycle. The analysis of the system reliability is therefore a major aspect in satellite design. Due to the necessarily high reliability figures, the reliability analysis and the consequent redundancy in hardware and software are a major cost driver in satellite design. To ensure the high reliability for satellite components, the components must prove the high reliability through different tests. As example, in [15] the multiple test procedure for thrusters of an Attitude and Determination Control System is explained. To analyze the behavior of the system and the subsystems for example statistical approaches are used. An example for a statistical approach and the results of this approach are described in [12]. These results are based on the examination of over 1500 satellite missions, with the analysis of the failures and anomalies. A similar approach is described in [115] pointing out that in satellite design, increasing reliability is even more important than shortening development times.

The statistical data from [12] have a binary scope on the state of the system - "Complete failure?" or "full functionality". Based on these data in [11] "degraded states" between the polarized states of [12] are introduced. From this starting point, a more distinguished view on the system is given and a step in direction to the qualitative analysis technique as the Fault Tree Analysis (FTA) is made (see Chapter 2, Section 2.1). The Fault Tree Analysis is widely used for the reliability analysis of satellites. This fact is also represented by the *NASA Handbook for Fault Tree Analysis* [82], where the use of the FTA for Aerospace Applications is described. An example for the actual usage of the FTA in the reliability analysis of a satellite is given in [4], in the context of the *Hermes CubeSat* mission. For the different subsystems of a satellite, different analysis techniques are applied. In [91] the reliability analysis of a satellite structure with a *Finite-Element* model is described. Therefore, parametric and non-parametric models were used.In the satellite domain, the reliability analysis is not finished with the calculation of the reliability figures. Due to the impossibility of repairing satellite failures in orbit, the analysis of failures is implemented in the satellite software system itself.

The method of *Failure Detection, Isolation and Recovery (FDIR)* for satellite components is a field of active research [50, 49, 86]. The *FDIR* method is intended to firstly detect failures, then isolate the erroneous component and afterwards, recover the correct functionality of the satellite system to regain full operational capabilities. In this context, as RAMSAS can be exploited during the design of the satellite system for evaluating its reliability performances through simulation, it could also support the definition and evaluation of design choices which envisage the use of components dedicated to perform *FDIR* tasks, as described in the following Sections.

## 4.2 System Description

The satellite under survey is the hypothetical *FireSat* mission from literature based on [69] with a refined system design from [38, 39, 40]. The mission objectives of FireSat are to detect, analyze and monitor forest fires. Therefore the Attitude Determination and Control System (ADCS) of the satellite has to provide (among other modes) the ability for the satellite to scan the area below the satellite on the earth surface to detect fires. The corresponding mode is called *nadir-pointing mode*, which means that the satellite is pointing towards the center of the earth. The satellite is orbiting the earth at an altitude of 700 km over ground, which is called a low-earth orbit (LEO). Resulting from its altitude, the satellite has to turn with a constant angular velocity once it is aligned to nadir pointing. In Figure 4.1 the activation of the nadir-pointing mode is shown. After the acquisition of the attitude, the *payload camera* of the satellite surveys a swath of a certain width on the ground below the satellite.

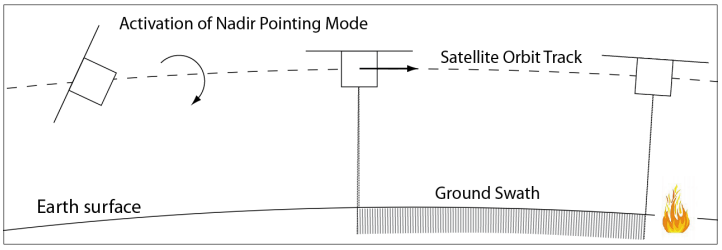



**Fig. 4.1.** Sketch of the satellite flying over a fire in Nadir pointing mode

The necessary adjustment of the satellite's attitude and its angular velocity is attained by applying torques on the satellite. To this aim, the Attitude Determination and Control System (ADCS) contains *thrusters* which imply a torque on the satellite in orbit. By firing different thrusters at the corners of the box-shaped satellite, the satellite can be turned around all axes (see Figure 4.2).

## 4.3 Reliability Requirements Analysis

The ADCS of the satellite has to fulfill the functional requirements for the alignment of the satellite. The system consists of sensors, actuators and the on-board computer controlling the system. The thrusters are used as the actuators in the system. Due to their mission-critical role there are redundant thruster packs (see Figure 4.2). The sensors determine the angular velocity and the attitude angles of the satellite. The on-board computer has a navigation unit, which calculates with the sensor data the target alignment of the satellite. Afterwards, the commands for the actuators are calculated, based on the

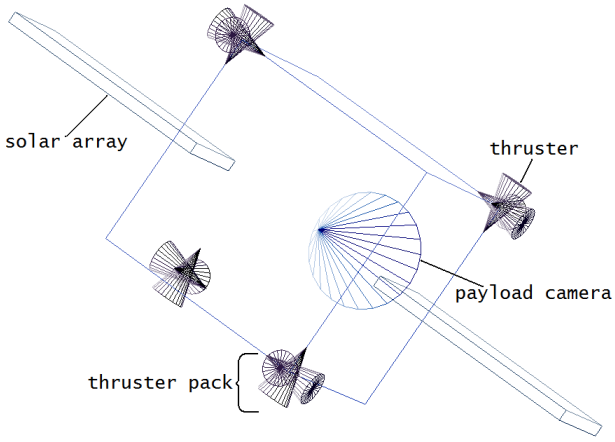



**Fig. 4.2.** Layout of the Satellite

resulting data of the navigation. This chain of activities (determine attitude, calculate action, execute commands) has to be fulfilled over the whole lifetime of the satellite by the ADCS. Since the functional requirements for attitude control can be reached by several combinations of the redundant thruster packs, the evaluation of non-functional requirements for system reliability has to be coupled with the functional analysis of the system.

## 4.4 System Modeling

In the *System Modeling* phase the structure and both the intended and dysfunctional behavior of the system under consideration are represented in SysML by executing four modeling activities (see Chapter 3): *System Structure Modeling, Intended Behavior Modeling, Dysfunctional Behavior Modeling* and *Behavior Integration*. Each of these activities will be described in the following sub-sections with reference to the ADCS, and supported by IBM Rational Rhapsody [52].

### 4.4.1 System Structure Modeling

In the *System Structure Modeling* activity, the system structure is modeled by using SysML Blocks following a top-down approach so to obtain a hierarchical decomposition of the system (e.g. system, subsystems, equipment, and components). Specifically, each system entity is represented by a SysML *Block* and modeled by both a *Block Definition Diagram* (BDD) and an *Internal Block Diagram* (IBD). As an example, the BDD of the ADCS system of Figure 4.3 shows that the ADCS consists of the following subsystems: *FlightSoftware, Actuators, Sensors, VehicleDynamics*, and the *PointingMode*. For each system

block, its input and output interfaces are specified according to the following template: ¡*SourceBlock_DestinationBlock_PortName_InputOrOutputPortType*¿.

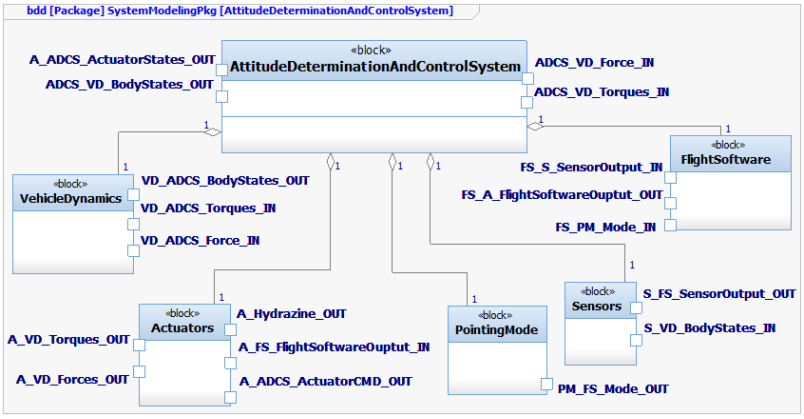



**Fig. 4.3.** Block Definition Diagram of the ADCS

For providing a description of the internal structure of a block in terms of the organization of its component blocks an IBD is introduced. As an example, the internal structure of the ADCS is reported in Figure 4.4 in which the component subsystems, their connections and interaction paths along with their operations and attributes, are represented.

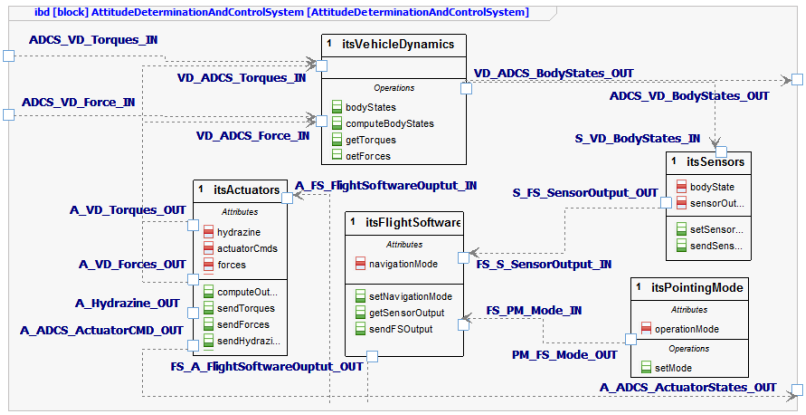



**Fig. 4.4.** Internal Block Diagram of the ADCS

By applying *zooming-in mechanisms* the system block identified after the first decomposition (see Figure 4.3) can be further decomposed so to reach a deeper level of decomposition. As an example, the structure of the Actuator

subsystem in terms of its components (*ThrustersControl* and *ComputeBodyForces*) is shown by the BDD diagram in Figure 4.5 whereas the connections among them are highlighted in the IBD diagram in Figure 4.6.

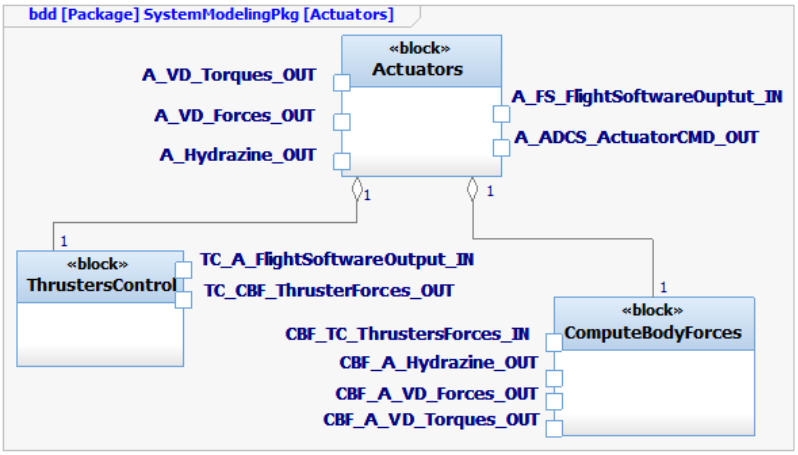



**Fig. 4.5.** Block Definition Diagram of the Actuators subsystem

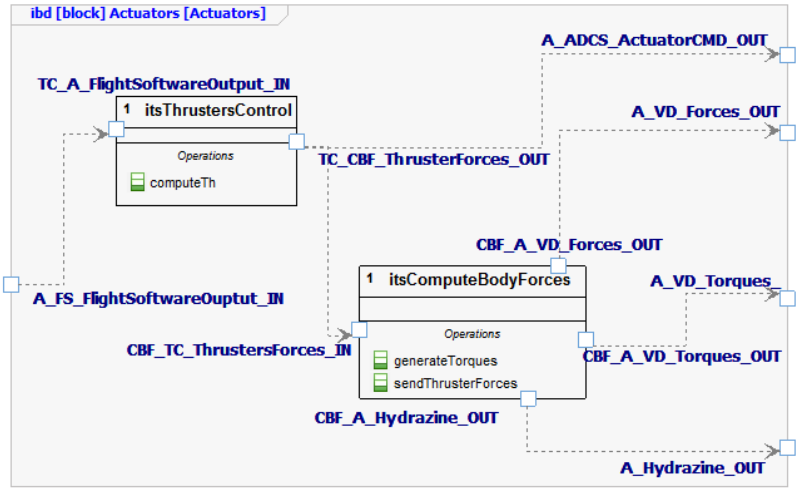



**Fig. 4.6.** Internal Block Diagram of the Actuators subsystem

In Figure 4.7 the structure of the FlightSoftware subsystem in terms of its components (*Navigation* and *AttitudeControl*) is reported exploiting a BDD, whereas its internal structure is highlighted in Figure 4.8 through an IBD.

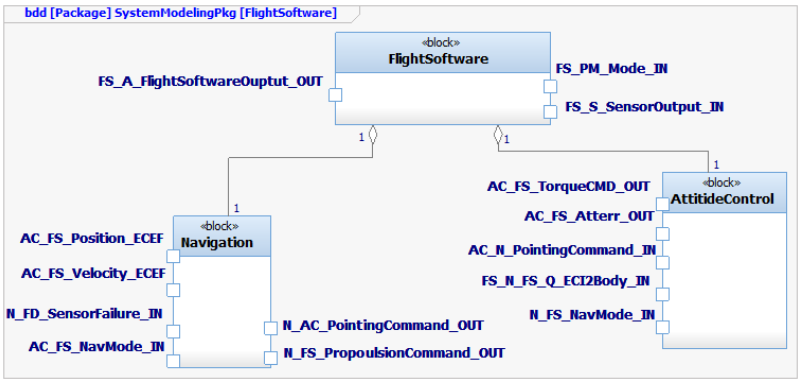



**Fig. 4.7.** Block Definition Diagram of the FlightSoftware subsystem

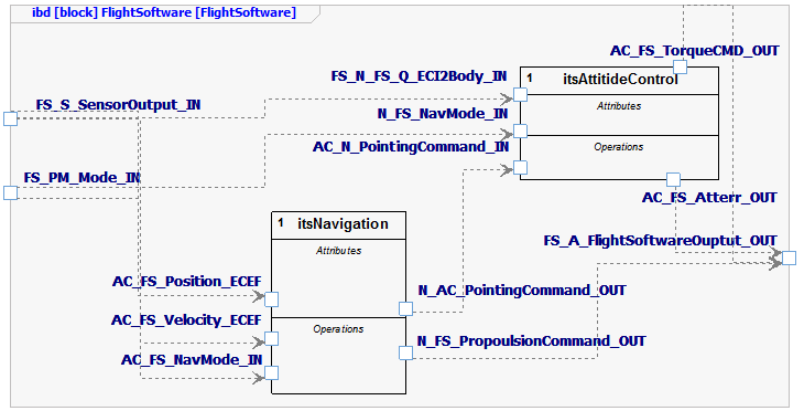



**Fig. 4.8.** Internal Block Diagram of the FlightSoftware subsystem

### 4.4.2 Intended Behavior Modeling

The *Intended Behavior Modeling* activity takes as input the hierarchical structure of the system as obtained during the *System Structure Modeling* activity (see Section 4.4.1) and specifies the intended behavior of the system by following a *bottom-up* approach. Specifically, the behavior of the system entities at the lowest level in the hierarchy, or *leaf level* (e.g. component level), are first specified; then the behavior of the entities at higher levels of abstraction, or *non-leaf levels* (e.g. subsystem and system level), are modeled by specifying how the enclosed entities participate and determine the behavior of each considered enclosing entity.

Depending on both, the characteristics of the behavior and the abstraction level to represent, different type of SysML diagrams can be exploited to model the behavior of a given entity: Activity, Sequence, Parametric, and Statechart Diagrams (see Chapter 3, Section 3.2.2).

With reference to the ADCS, its behavior depends on the behavior of its subsystems (*FlightSoftware, Actuators, Sensors, PointingMode, VehicleDynamics)* and their interactions. In particular, the *FlightSoftware* subsystem is the brain of the system as it takes the decisions to control the satellite system; whereas, the *Actuators* subsystem is used to apply a torque on the satellite. In turn, the behavior of the *FlightSoftware* depends on the behavior of both the Navigation and *AttitudeControl* components; whereas the behavior of an *Actuators* subsystem depends on the behavior of both the *ThrustersControl* and *ComputeBodyForces* components, and so on.

In Figure 4.9 the intended behavior of the *ThrustersControl* component is shown using a SysML Activity diagram. In particular starting from the *torque_cmds* command (or signal), which is received from the *FlightSoftware*, if this command is, in one or more axes, over a *torque_thresh* threshold, then the appropriate thrusters are set on. If the command falls below *-1*torque_thresh* threshold, then the thrusters are set off; in particular, if the thruster is set off, then the relative valve for the fuel connection is closed. If the thruster is set on, then the appropriate valve is open. Because the satellite has at 4 of its 8 corners one thruster pack consisting of 3 thrusters ($x$,$y$,$z$), see Figure 4.2, the cycle is executed 12 times. At the end of this task, a signal of *thruster_forces*, which is composed by the complete settings for each of the four *thruster packs*, is produced and sent in output to be processed by the *ComputeBodyForces* component.

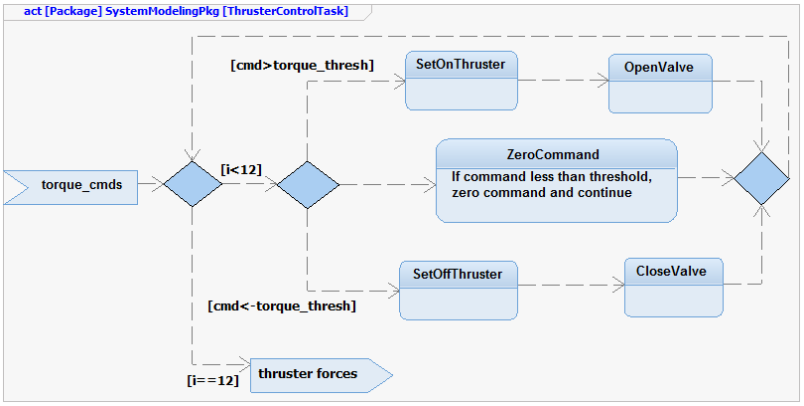



**Fig. 4.9.** Intended Behavior of the ThrustersControl component

In the following, the behavior of the *ComputeBodyForces* component is described; moreover, one part of it is also represented through the exploitation of a *Parametric Diagram* (see Figure 4.10). Such behavior is defined as a set of equations which take as input:(i) the *thruster_forces* signal in terms of their single packs *xyz_pack_i* for i1,...,4 (where *xyz_pack_i* is a three-element vector which indicates which thruster of the thruster pack $i$ is on/off); (ii)

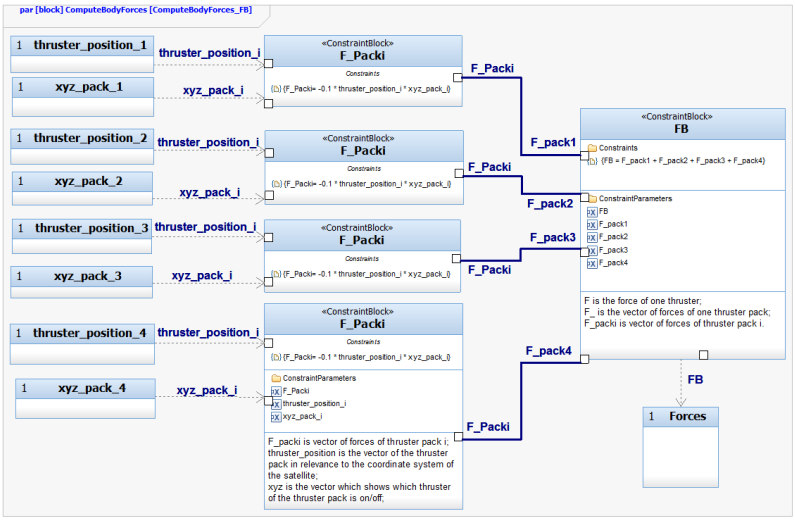



**Fig. 4.10.** Intended Behavior of the ComputeBodyForces component

the *thruster_position_i* for i=1,...,4 (where *thruster_position_i* is the coordinate vector of the thruster pack $i$ in relevance to the coordinate system of the satellite). All those signals are used to compute *F_Packi* for i=1,...,4 (where *F_packi* is the vector of forces of the thruster pack $i$) which in turn are exploited to produce in output a part of the *Forces* signals.

As described above, after defining the intended behavior of the entities at the *leaf level* (e.g. the *ThrustersControl* and the *ComputeBodyForces* component for the *Actuators* subsystem), the behavior of the entities at the *non-leaf levels* is specified. As an example, the intended behavior of the *Actuators* subsystem can be derived and represented through a *Sequence* diagram (see Figure 4.11) which highlights the iterations among the involved entities; the behavior specified in Figure 10 and in Figure 11 is invoked by the *computeTh(torques_cmds)* and *computeForce (ThrusterForces)* messages respectively.

It is worth noting that, as for the above mentioned case of the *Actuators* subsystem, *Sequence* diagrams can be exploited for representing the exchange of messages among parts of a composite block (i.e. a block at a *non-leaf level* in the structural decomposition of the system). This information complements that provided by the IBD of the composite block, that highlights the internal structure of the block in terms of the organization of its component blocks, their connections and interaction paths along with their operations and attributes. Moreover, as well as the behavior of a component block at *leaf level* can be described by using *Activity, Statechart* and/or *Parametric Diagrams* (depending on the type of the component), if necessary, also the behavior of a composite block can be further represented by using these kinds of diagrams.

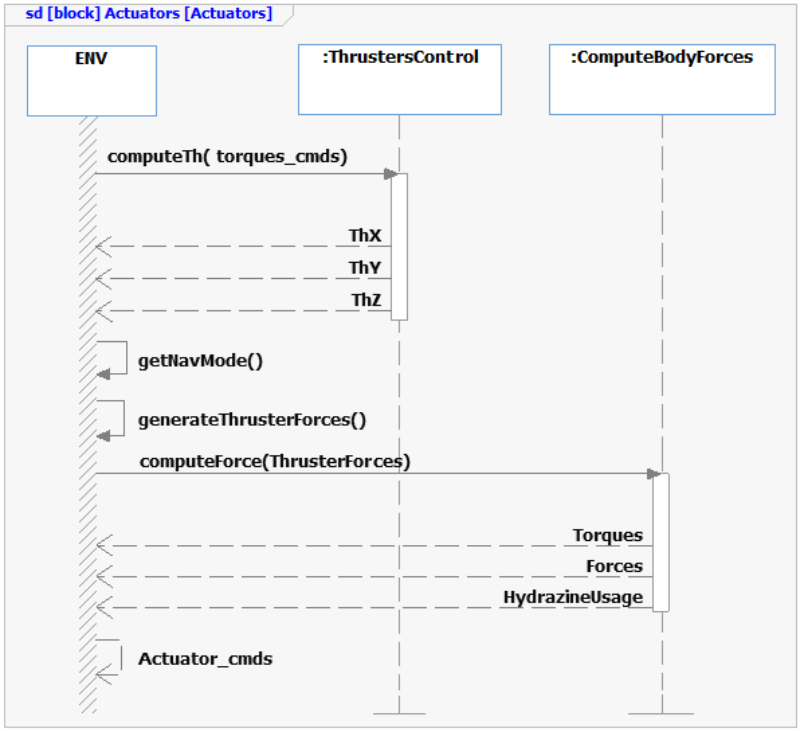



**Fig. 4.11.** Intended Behavior of the Actuators subsystem

By applying this modeling approach, which relies on the SysML best practices and is strongly related to that proposed in [16, 93], a SysML-based representation of the intended behavior of the whole system has been derived.

### 4.4.3 Dysfunctional Behavior Modeling

In the *Dysfunctional Behavior Modeling* activity, the focus is on the modeling of faults and failures, which are key concepts of the system reliability analysis. Specifically, for each entity represented by a SysML Block (see Section 4.4.1), beside the intended behavior, the behavior concerning faults and failures (i.e. the dysfunctional behavior) is specified as a set of dysfunctional tasks (see Figure 3.4). These tasks could receive as input a set of *failure events* (e.g. due to the failures of other blocks) and could, in turn, produce as output other *failure events* due to the *failure* of the block; moreover, internal *faults* (represented as *fault events*) can be generated and treated inside the block possibly producing block failures (and thus output *failure events*). For specifying these dysfunctional tasks six templates have been individuated (see Table 3.1): *Fault Generation, Failure Generation, Failure Management, Fault Management Failure Propagation*, and *Failure Transformation*. Moreover, five

fault/failure types could be considered [41]: (i) reaction too late; (ii) reaction too early; (iii) value failure; (iv) commission; and (v) omission. By combining the individuated six dysfunctional task types with these five fault/failure types, thirty different basic fault/failure behavioral patterns can be derived [29].

For each system entity, the instantiation of these fault/failure behavioral patterns requires to specify all the aspects concerning the fault/failure generation process as well as fault/failure management and propagation rules and policies. As an example, if the failure rate of a component increases with time due to material stress and/or component wear then suitable time-dependent failure rate functions should be used for the fault/failure generation tasks of the component; obviously, these functions (and/or their parameters) can be also left as parameters that can be set during the *Parameters Setting* activity of the *System Simulation* phase (see Section 4.5.2).

As an example, with reference to the system under consideration, both the *FailureGeneration* and *FailurePropagation* templates have been exploited to model the failure generation and failure propagation events of the *FlightSoftware* subsystem. In particular, a *FailureGeneration* task is activated by a *TimedEvent* (manually or by a clock) according to a set of *StepFunctions* having specific *function values* and *delay times* (see Figure 4.12).

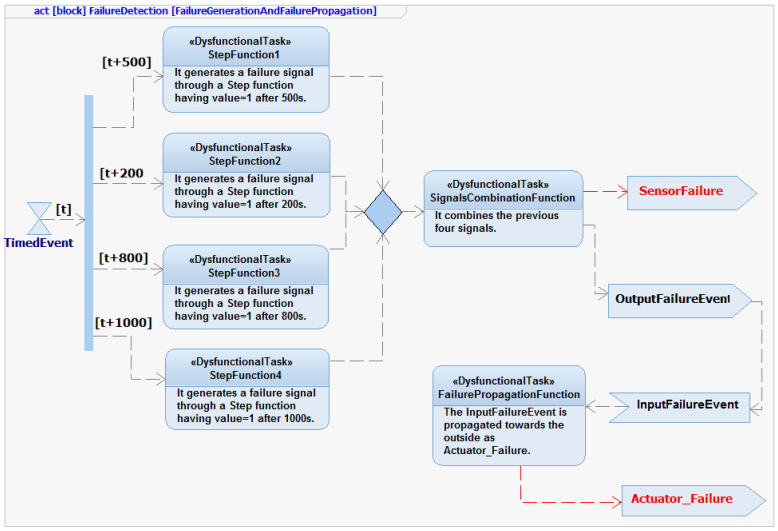



**Fig. 4.12.** Dysfunctional Behavior of the FlightSoftware subsystem

Then, two *OutputFailureSignals* are produced: (i) a *SensorFailure* signal which is directly sent to the *Navigation* component, and (ii) an *Actuator_Failure* signal, which is propagated outside towards the Actuators subsys-

tem, by applying a *FailurePropagation* task. When the *Actuator_Failure* signal reaches the *Actuators* subsystem, it is propagated towards the *ThrusterControl* component. Beside the intended behavior of the *TrusterControl* component, a *FailureManagement* task has been also implemented which, starting from the *Actuator_Failure* signal in input, is able to handle such *InputFailureSignal* or produces an *OutputFailureSignal* that, in turn, simulates the crash of a whole thruster pack after a specific time.

Another example of dysfunctional behavior that will be analyzed during the simulation (see Section 4.5.3) is that represented by the fault of the valve between the fuel connection and the related thruster (there is one valve for each thruster). In this case, two types of fault can be generated: *blocking* of a valve (while opening) and *sticking* of a valve (while closing).

### 4.4.4 Integrated Behavior

In the *Behavior Integration* activity, both the intended behaviors and the dysfunctional behaviors modeled in the previous modeling activities are integrated to obtain an overall behavioral model of the system and its component entities. As an example, in order to integrate both the *FailureGeneration* and *FailurePropagation* task in the intended behavior of the *FlightSoftware* subsystem, a new software component, called *FailureDetection*, has been introduced (see Figure 4.13) which implements the dysfunctional behavior represented in Figure 4.12.

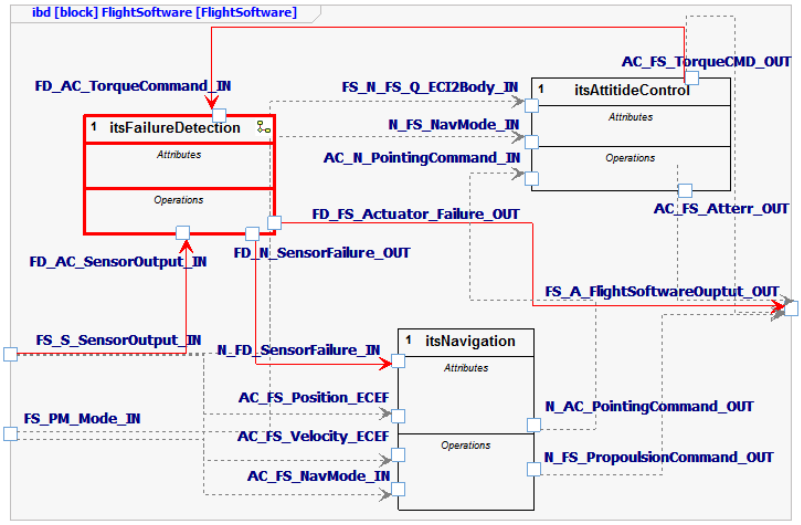



**Fig. 4.13.** Behavior Integration into the FlightSoftware subsystem

In particular, the *FailureDetection* component takes as input two signals: (i) *sensor_outputs* coming from the Sensors subsystem and (ii) *torque_commands* which is in a feedback. Then, the *FailureDetection* component produces as output (through the *FailureGeneration* task) two signals: (i) *sensor_failure* which is sent to the *Navigation* component and (ii) an *actuator_failure* which is sent as output (through the *FailurePropagation* task) to the *Actuators* subsystem (towards the *ThrusterControl* component). A similar model has been derived for the *Actuators* subsystem.

This *Behavior Integration* activity closes the *System Modeling* phase by delivering the *System Model for Reliability Analysis (SMRA)* work-product.

## 4.5 System Simulation

The objective of the *System Simulation* phase is to evaluate through simulation the reliability performance of the system and, possibly, compare different design alternatives and parameters settings. In particular, the following three main activities are performed: *Model Transformation, Parameters Setting*, and *Simulation Execution*. Each of these activities is described in the following Subsections.

### 4.5.1 Model Transformation

The *Model Transformation* activity is the first step of the *System Simulation* phase and aims to obtain an *executable* model of the system (*ESM, Executable System Model*) represented in terms of the constructs offered by the target simulation platform. Indeed, in the current version of RAMSAS, the system under consideration and, in particular, its reliability requirements, structure, functional and dysfunctional behaviors, are modeled in SysML. Then, the simulation is performed in MathWorks Simulink which represents a *de-facto* standard for the simulation of multi-domain dynamic and embedded systems. As a consequence, it is necessary to translate the SysML-based model of the system (*SMRA, System Models for Reliability Analysis*) into a Simulink model to be simulated. This model-based approach allows to keep the (*conceptual*) Modeling of the system (which is supported by SysML and thus is platform independent) distinct from its Simulation, which depends on the specific target simulation platform (Simulink in this case). The transformation of the SysML-based model of the system into the Simulink-based simulation model is currently performed manually based on a mapping between the basic SysML and Simulink constructs (see Table 3.2, Chapter 3); thus, it is not supported by a specific tool and the consistency among these models has to be guarantee by the expert manually. Finally, the SysML behavioral diagrams which model the intended and the dysfunctional system behavior are, in the current version of RAMSAS, manually transformed in Simulink functions and/or Stateflows according to specific transformation rules.

As an example, Figure 4.14 shows an *ESM* model which has been derived from the ADCS system represented, through a SysML notation, in Figure 4.3 and in Figure 4.4.

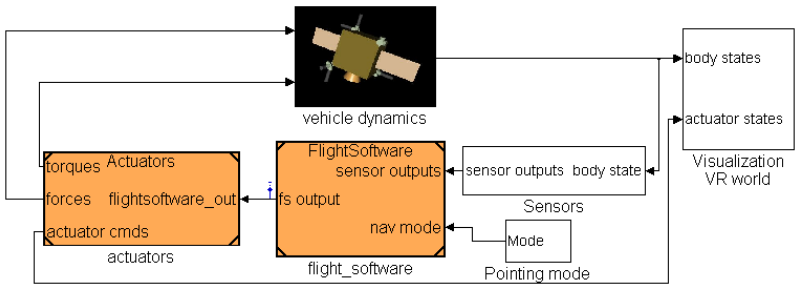



**Fig. 4.14.** Executable System Model of the ADCS system

Figure 4.15 represents the full *ESM* model for the Intended Behavior of the *ComputeBodyForces* component, which has been derived from the *Parametric diagram* in Figure 4.10.

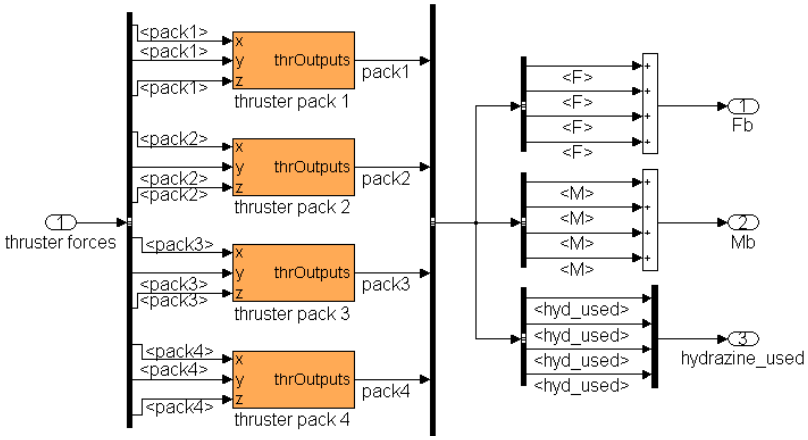



**Fig. 4.15.** Executable System Model for the Intended Behavior of the ComputeBodyForces component

Figure 4.16 and Figure 4.17 show the behavior of the thruster packs through Stateflows after the *Behavior Integration* activities. In particular, the Stateflow of the *ThrusterLogic* (Figure 4.16) simulates the behavior of the thruster packs. The function, shown in Figure 4.17, is derived from the behavior of a thruster pack, shown in Figure 4.9. This function includes a

*FailureManagement* task (in combination with the Stateflow Data, see Figure 4.16), beside the intended and dysfunctional behavior of the thruster packs.

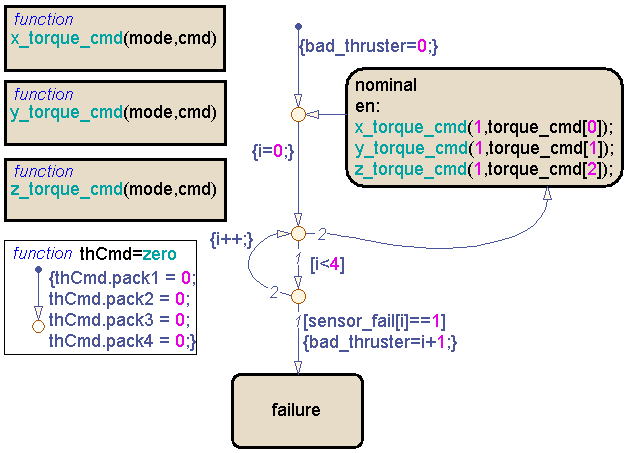



**Fig. 4.16.** Stateflow of the Behavior of the Thruster Pack

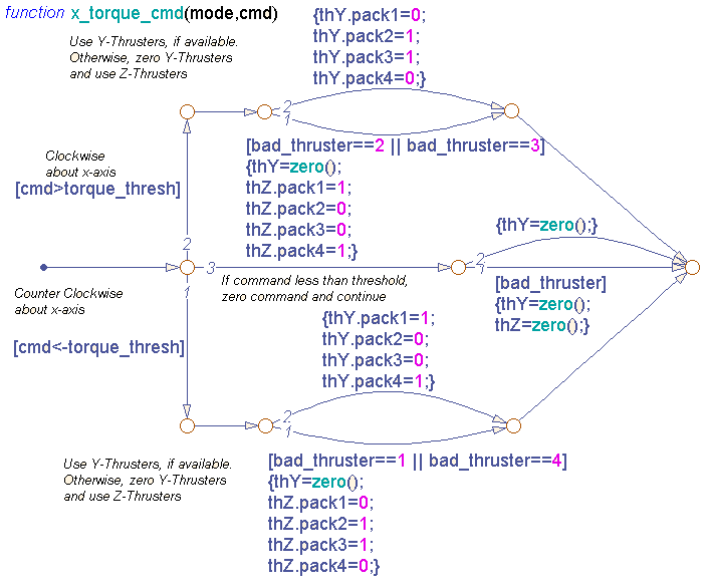



**Fig. 4.17.** Thruster Pack failure management for x-torque commands

In Figure 4.18 the Stateflow for the simulation of the behavior of the valve between the fuel connection and the related thruster (there is one valve for each thruster) is reported. The Stateflow simulates both the intended and dysfunctional behavior of the valve (see Section 4.4.3).

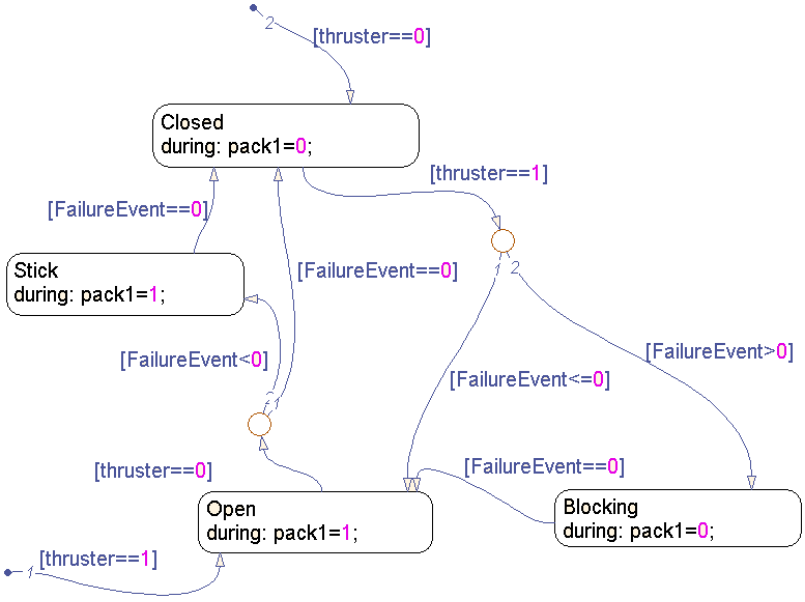



**Fig. 4.18.** Stateflow of the Behavior of a Valve

### 4.5.2 Parameters Setting

Before starting the simulation, several system and configuration parameters can be set to evaluate system reliability performance in different simulation scenarios. In the *Parameters Setting* activity, the *ESM* is refined so to allow the flexible setting of system configuration and simulation parameters which can be tuned according to both, the characteristics of the operative scenario to simulate and the failure modes to analyze (by acting on the component failure rates and the other settings of the faults and failures generation, propagation and management tasks). It is worth noting that when simulating different operative conditions with different parameters settings, depending on the complexity of model, probably a large number of scenarios will need to be simulated. In this case, to master the combinatorial problem, some kinds of Monte Carlo or statistic methods could be exploited. The current version of RAMSAS does not provide an explicit support for addressing this issue leaving the expert free to exploit the features of automatic definition of simulation

scenarios provided by the Simulink environment. For the ADCS different parameters can be set. One parameter is the *torque_thresh* of which variation determines the operating range of the thrusters. Further, the specific impulse (*Isp*) of the thrusters can be changed, which represents the variation of the thruster and/or fuel. The position of the thruster packs can also be changed. This variation changes the lever for the torque calculation. These are only a few examples, explicit for the actuators, which show the flexibility range of the system.

As an example, Figure 4.19 shows the Model Explorer panel of Simulink by which the main parameters of the ADCS system can be tuned opportunely.

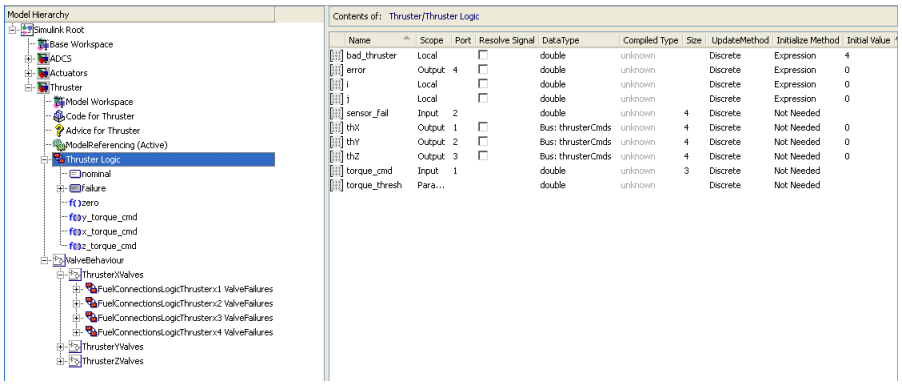



**Fig. 4.19.** A Screenshot of the Parameters Setting activity

In the next Sections some of the main simulations executed to evaluate the reliability performances of the modeled ADCS are presented and then the obtained simulation results analyzed and discussed.

### 4.5.3 Simulation Execution

In the *Simulation Execution* activity the resulting *ESM*, which is a complete executable Simulink model, is executed according to a synchronous reactive model of computation: at each step, Simulink computes, for each block, the set of outputs as a function of the current inputs and the block state, then it updates the block state. During the simulation *faults* and *failures* are injected (by *TimedEvent* or *TriggeringEvent*) and/or caused to stress and analyze the behavior of the ADCS *system*. At the end of this activity, the data generated from the simulations are reported in the *Simulation Results (SIRE)* work-product to be analyzed in the next phase.

Executing the system allows, beside parameter variation, the simulation of the system behavior, during the failure of some components. Specifically, first the intended behavior of the ADCS has been simulated; then, according to

the dysfunctional behaviors introduced in Section 4.4.3, the following *failure modes* have been simulated: (i) the failure of a thruster pack; (ii) the failure of a valve between the fuel connection and the relative thruster. The simulations related to these failure modes are described below.

In Figure 4.20 the intended behavior of the ADCS is illustrated.

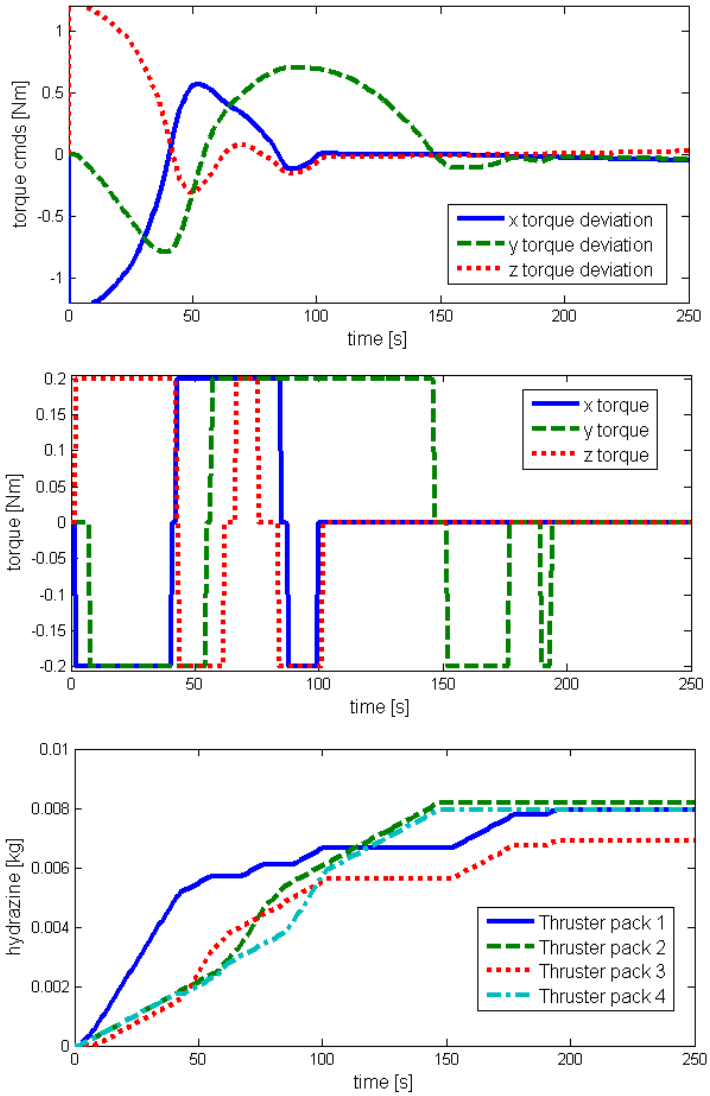



**Fig. 4.20.** Diagrams for the Intended System Behavior

The simulation begins with a start alignment of the satellite. The results of the simulation are illustrated within the three diagrams plotted over the simulation time. The topmost diagram shows the torque command, calculated by the *FlightSoftware* subsystem. The torque threshold is 0.2 Nm. This means the maximum difference between the required torque for a target alignment and the actual alignment in an axis, which is allowed, is 0.2 Nm. If the threshold is exceeded, then the thrusters should create a torque to reach the intended alignment. The three different curves are the separation in the three different directions (x, y, and z). The second diagram in Figure 4.20 shows the torque and the direction of the torque which acts on the satellite. It is the sum of the torques of the different thrusters. The different curves show the different directions. Therefore, if the torque command in one direction exceeds the threshold, then the actuators counteract (the same colors in both diagrams indicate the same direction). The third diagram shows the summarized hydrazine usage of the four thruster packs.

The failure of a thruster pack with all three thrusters out of order has been simulated. The failure is compensated through the use of other thrusters (see Figure 4.17). The intended choice for raising the torque is with the thrusters in y-axis. However, if one of the used thrusters is defective, then the failure management tries to use the thrusters aligned in z-axis to fulfill the command. In Figure 4.21 the diagrams for the failure of thruster pack 1 are shown. The simulation was executed with the same parameters as in the simulation of the intended behavior (Figure 4.20). The diagrams show that the failure of thruster pack 1 can be compensated and the system is still fulfilling its task. Furthermore, it is visible that the compensation of the start alignment takes longer and, at the beginning, is not as exact as in the fully functional case. However, the diagrams show also that the hydrazine usage is lower than with all thruster packs (curve for the hydrazine usage of the thruster pack 1 is equal to zero). The lower hydrazine consumption results from the lower angular velocities used to align the satellite in this case.

Moreover the failure of the valve between the fuel connection and the related thruster has been simulated. As reported in Figure 4.18, while opening, the valve could block and stay closed. Further, while closing, the valve could stick and stay open so two failure modes should be evaluated: *blocking* of a valve and *sticking* of a valve.

The failure mode *blocking* of a valve causes that the effected thruster cannot be used anymore. Due to the redundant thrusters (each thruster pack has three thrusters, one for each axis) the only impact is an increased maneuver duration, because only half of the thrust for creating the torque will be available. The system itself will however not fail. An opposite failure is the failure mode *sticking* of a valve. This failure mode causes a constant fuel flow and thruster use. Due to this fault a different failure will happen: the system will counteract and therefore, the opposite thrusters will be activated. This holds the satellite in position, but the whole time fuel will be required, until the tank is empty. This failure leads to a fail of the whole system. Looking on these

failures, a system design with an extra valve in front of the whole thruster could be developed. To enable the failure detection, a sensor surveying the fuel flow is also required.

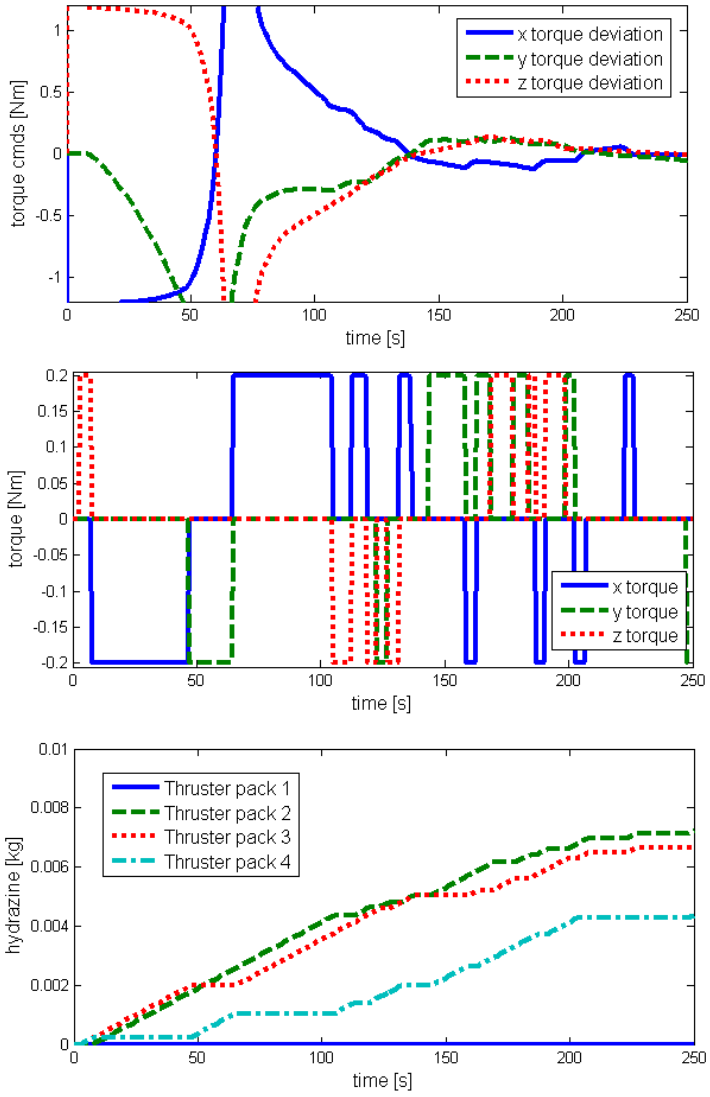



**Fig. 4.21.** Diagrams for the Dysfunctional Behavior: "Failure of a thruster pack"

## 4.6 Results Assessment

In the *Results Assessment* phase, the simulation results (*SIRE*) are elaborated with reference to the objectives of the reliability analysis identified in the initial phase of the process so to obtain important information on the reliability properties of the system under consideration. In particular, the analysis of the resulting graphs or of the obtained data can be conducted. The use of domain experts should not be underestimated to obtain a good analysis of the results and their evaluation, since an effective re-design of the system is also an outcome of a deep knowledge of the domain. These analyses are able to give information about the reliability performances of the ADCS system under consideration as reported in the *Reliability Analysis Report (RAR)* document; moreover, they also provide suggestions to improve the reliability of the system proposing alternative design solutions as reported in the *Design Suggestions Report (DSR)* document.

As shown in the above discussed simulations, great part of these analyses can be directly performed in Simulink, whereas more advanced analyses can be also performed by external tools after exporting the results obtained through the MATLAB environment. Moreover, for letting the expert to mainly focus on problematic simulation traces, additional Simulink blocks for the automatic indication of requirements violations during the simulation can be introduced during the definition of the *ESM* (see Section 4.5.1), starting from the reliability requirements formalized in SysML (by using *Requirements Diagrams*) during the *Reliability Requirements Analysis* phase of RAMSAS and traced (by using the related SysML constructs) in the *System Modeling* phase.

The ADCS with its intended and dysfunctional behavior, which is the system under consideration, can be improved by analyzing the results of the simulation execution. A variation of the parameters expands the understanding of the reliability of the system. Some failures, which were modeled and simulated, have a deep impact on the system; therefore, a failure management task should be introduced to solve these problems, while other failures have almost no impact on the reliability of the system (see Section 4.5.3).

Furthermore, the reliability simulations lead to investigate other aspects which could be considered. As an example, the whole system fails when the fuel tank is empty. On the other hand the system has a longer lifetime, if less fuel is used. The simulation of the dysfunctional behavior (see Section 4.5.3) showed that with longer acquisition times allowed for the system, the fuel consumption sinks: the satellite will reach its target but with more time required for the target acquisition and with larger deviations from the target attitude. The attitude accuracy is required by the payload camera looking towards the earth surface under a specific angle. The availability of the camera drops if the acquisition time increases. This results lead to new requirements which could lead to a change in the mission specification.

## 4.7 Conclusion

In this Chapter, a concrete experimentation of the RAMSAS Method has been shown. RAMSAS has been exploited for the reliability analysis of an Attitude Determination and Control System (ADCS) of a satellite. Specifically, according to the RAMSAS method, the definition of a SysM-based model, both for the intended and dysfunctional system behavior, along with the subsequent derivation of a Simulink based simulation model have been shown. One of the main reason for the choice of SysML as the reference modeling language and of MathWorks Simulink as the reference simulation environment was to lower the learning curve of the RAMSAS method by Systems and Dependability engineers.

The concrete exploitation of RAMSAS has allowed appreciating its effectiveness and suitability both in the system structural and behavioral modeling and in the evaluation through simulation of the system reliability performances. Moreover, as SysML is one of the standard modeling languages for Systems Engineering, its exploitation in conjunction with wide adopted simulation environments, such as Simulink, allows for a seamless model-based design process: SysML allows to represent in an integrated way both the system requirements and the intended and dysfunctional system behavior, whereas Simulation makes it possible to assess the fulfillment of the requirements, evaluate system performances and compare different design choices. As an example, with reference to the presented case study, from the simulation of both, the intended and dysfunctional system behavior, it was found that the ADCS system is more fuel efficient in one of the possible failure modes. The combined simulations can thus lead to interesting insights about the system design. In particular, further simulations results are under analysis for the comparison of different design choices so as to improve the reliability and overall performances of the ADCS.

# 5

# A Modelica-based Method for supporting the Safety Analysis of Physical Systems

*System safety* is an important aspect of *System Dependability* which should be taken in consideration during the whole system life cycle. However, often systems are built by considering mainly their functional aspects whereas *safety requirements* are verified and validated in the latest stages of the development process. For this reason and due to the deep integration of modern systems in the daily life of people, regulatory standards have been defined and have to be applied during the development of critical systems in order to guarantee a minimum and acceptable level of safety. Moreover, the significant increase in the complexity and autonomy of the systems renders the verification of the (non)functional safety of each individual component as well as of the entire system a complex task and underlines the need for integrated and model based approaches that would assist this process [92, 121]. In this context, a model-based process, inspired by the ISO-26262 standard (see Chapter 2, Section 2.3), which provides a methodological support, based on Modelica language and tools, for the verification and validation of safety requirements of systems, has been proposed [104, 33, 32]. In particular, the Method exploits as an integrated chain of tools both the ModelicaML profile [77] during the *System Modeling* phase for representing systems and then, in the *Virtual Testing* phase, the OpenModelica environment [87] as the simulation platform for the execution of the system model generated in the previous phase. Some proposed Modelica extensions are exploited in all the process phases both for collecting and representing system requirements as well as for supporting their verification through communication and propagation mechanisms among system requirements (see Chapter 6).

The contribution provided in this Section is related with model-based methods described in Chapter 2, Section 2.2 and, in particular, with the vVDR method in [107, 70], which is centered on the Modelica language where, however, neither communication processes nor evaluation mechanisms among system requirements (properties) have been specified in order to enable the propagation of assessments among them.

In Section 5.1, the proposed simulation-driven design process for supporting the safety analysis of physical systems is presented; then in Section 5.2 the main relationships between the ISO-26262 standard and the proposed process are highlighted; finally, in Section 5.3 the method is exemplified through a case study in the automotive domain concerning the safety analysis of an Airbag System.

## 5.1 Methodological Process From Safety Requirements a Simulation-Driven System Design

In this Section the proposed methodological process for the development of safe systems, based on the validation of the design through simulation, is presented. As it is shown in Figure 5.1, such process, inspired by the ISO-26262 standard, is defined in terms of three main iterative phases: *Requirements Analysis, System Modeling* and *Virtual Testing*, that aim to provide a methodological support according to the ISO-26262 standard.

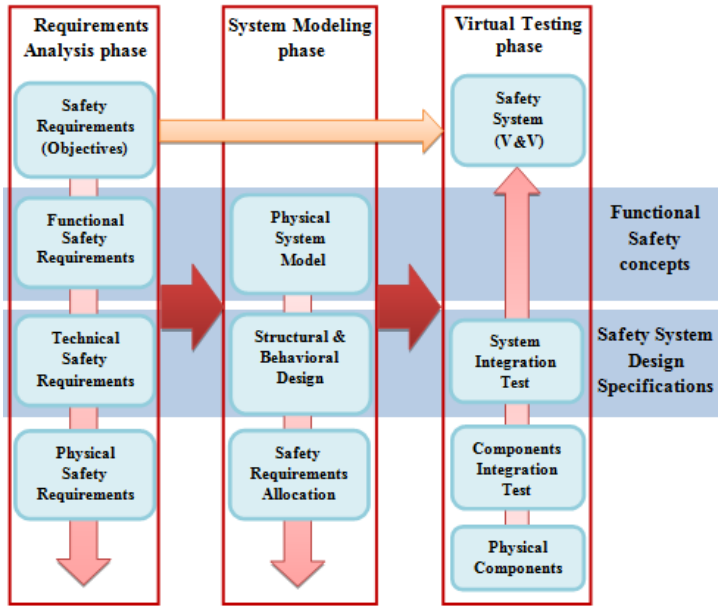



**Fig. 5.1.** Main phases of the proposed simulation-driven process for the design of safe systems

In the *Requirements Analysis* phase the system safety objectives are analyzed and Safety requirements, in terms of Functional, Technical and Physical requirements, are identified [103, 112]. They may consist of properties and

safety performances to be considered in order to eliminate the risk or to reduce it to an acceptable level. Specifically, a process for their elicitation, definition, formalization and validation is defined according to a reference *meta-model* proposed by the author in [117] (see Chapter 6).

In particular, the first step consists in the requirements elicitation that, according to the proposed meta-model, is obtained through *RequirementAssertions*. An iterative process between the user and the analyst is typically executed in order to state all the requirements, as much as possible, by associating to each of them a *Name* for their identification along with a possible *Description* in a text format by using the natural language in order to provide an explanation of specific or salient aspects, characteristics, or features (e.g. functional, technical or physical) of the system in a detailed way. At the end of this step the so called *User Requirements (URs)* are generated according to the meta-model.

The second step consists in the refinement of the *URs* in order to generate *System Requirements (SRs)*. This step is very crucial to make *URs* machine readable and executable in order to enable their verifiability during the simulation, as a consequence, it is really important what to represent and how to do it as well as when to use such requirements. First of all a *RequirementAssertion* could be involved in several verification tasks grouped in different *RequirementModel*, so the membership of each requirement to at least one of those *RequirementModels* must be identified. Then, the output values, associate to the evaluation of requirements, for describing if a requirement has been not violated, violated, and so on, have to be fixed. At the end, a *Metric* needs to be specified for each *RequirementAssertion*. In particular, it specifies the purpose of a *RequirementAssertion* in terms of verification mechanism. In Figure 5.2 the relationships among the User Requirements, System Requirements and Safety Requirements are represented.

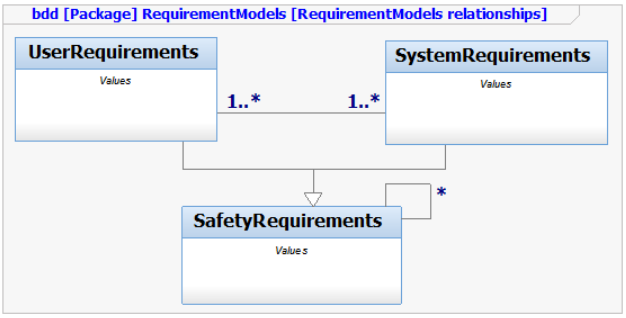



**Fig. 5.2.** Relationships among User Requirements System Requirements and Safety Requirements

The representation of requirements is carried out by using Requirement diagrams available in SysML [116] and ModelicaML [77] profiles for modeling systems in order to enable model-based systems engineering. It is worth to notice that not all the requirements can be formalized into something computable such as "a cable must be well connected", if the term "well connected" is not represented in a machine readable formalism.

In the *System Modeling* phase, a possible physical model of the real system in terms of its components is defined; in particular, the Structural and the Behavioral views are generated by breaking down the system in (sub)components. Specifically the first step, according to the *Physical* side of the proposed meta-model (see Chapter 6 Section 6.1), consists in building a possible *PhysicalSystemModel*, of the actual *PhysicalSystem* by specifying the *models* of its physical components (*PhysicalComponentModels*) and the related *Attributes* and, then, defining the *relationships* among them as well as their *behaviors*. In particular, the structural part of the system is described by using Block Definition Diagrams and Internal Block Diagrams in a top-down fashion. The behavior of the system, which is modeled by following a bottom-up approach, can be defined in terms of Activity, Sequence or Parametric diagrams in order to model the internal behavior of each system components as well as the flows of actions and interactions between components.

Then *SRs* belonging to the *RequirementModel* concerning Safety Requirements, can be further formalized in order to make them machine executable. In particular, a formal *Measure*, and its expected input and output values, can be associated to the defined *Metric*. Specifically, a *Measure* can be expressed by adopting an appropriate *ComputationalModel* which in turn could be represented through an *Algorithm*, a *Finite Automata*, a *Function*, a set of *Equations* or by their combination to enable the computational process. Finally, the allocation between the Safety Requirements and the *PhysicalSystemModel* is performed. Furthermore, inputs, required from the *Measure* of a *RequirementAssertion* for its evaluation, are explicitly included in the *PhysicalComponentModels*.

In the Virtual Testing phase, the Models of the system under consideration are transformed into executable models and represented in terms of the constructs offered by the OpenModelica platform [87, 25, 76]. In particular, physical components are defined and integrated in order to build the *PhysicalSystemModel* and then the safety requirements to be verified are introduced into the overall model by exploiting some extensions of the Modelica languague proposed by the author [105]. Then, different simulation scenarios are set and simulations are executed; finally, simulation results can be analyzed on the basis of the system safety requirements identified in the first process phase. This analysis allows to evaluate the safety properties of the system, to compare different design choices for improving, possibly, the safety of the system under consideration.

As the process is iterative, if necessary, new partial or complete process iterations can be executed.

## 5.2 Relationships between the ISO-26262 standard and the proposed process

The above described process is inspired by the IEC-61508 standard [53] and, in particular, by the ISO-26262 [59] whose goal is to demonstrate the capability to develop certain products with acceptable risks. ISO-26262 is organized in 10 parts as following:

- *Part 1 - Vocabulary*: which specifies the terms, definitions and abbreviated terms for application in all parts of ISO 26262;
- *Part 2 - Management of Functional Safety*: which specifies the requirements for functional safety management for automotive applications, including (i) project-independent requirements with regard to the organizations involved (overall safety management), and (ii) project-specific requirements with regard to the management activities in the safety lifecycle (i.e. management during the concept phase and product development, and after the release for production);
- *Part 3 - Concept phase*: which specifies the requirements for the concept phase for automotive applications (e.g. item definition, functional safety concept, etc.);
- *Part 4 - Product Development at system level*: which specifies the requirements for product development at the system level for automotive applications, such as the system design and system integration and testing;
- *Part 5 - Product Development at hardware level*: which specifies the requirements for product development at the hardware level for automotive applications (e.g. hardware design and hardware architectural metrics, hardware integration and validation);
- *Part 6 - Product Development at software level*: which specifies the requirements for product development at the software level for automotive applications such as software architectural design, software unit design and implementation, software integration and testing;
- *Part 7: Production and Operation*: which specifies the requirements for production, operation, service and decommissioning.
- *Part 8: Supporting Processes*: which specifies the requirements for supporting processes through qualified tools, system engineering approaches and best practices;
- *Part 9: Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analyses*: concerning the measures required to avoid unreasonable risks.
- *Part 10: Guidelines on ISO-26262.*

In the Table 5.1 the matching between ISO-26262 parts and the phases of the proposed process are shown, by indicating in which phase of the process a specific part of such standard should be considered. In particular *Vocabulary* and *Management of Functional Safety Concept phase* can be considered in the

*Requirements Analysis phase* for the definition, the organization and categorization of requirements; then *Product Development at system level, Product development at the hardware level* and *Product development at the software level* can be taken into account in the *System Modeling phase*, when the design of the system is under definition, whereas the *Supporting Process* part can be considered during the *Virtual Testing phase* of the proposed process.

**Table 5.1.** Matching between ISO-26262 and the proposed process

| Parts of the Standard ISO-26262 | Simulation-Driven Process for the Design of Safe Systems |
|---|---|
| *Vocabulary*<br>*Management of Functional Safety*<br>*Concept phase* | *Requirements*<br>*Analysis*<br>*phase* |
| *Product development at system level*<br>*Product development at the hardware level*<br>*Product development at the software level* | *System*<br>*Modeling*<br>*phase* |
| *Supporting*<br>*phase* | *Virtual*<br>*Testing*<br>*phase* |

## 5.3 Performing Safety Analysis of an Airbag System: A Case Study in automotive domain

In this paragraph, a case study in the automotive domain concerning the modeling of an airbag system, and the validation and evaluation of its design according to the safety requirements through simulation, is analyzed following the proposed process. After a brief descriptive introduction of the system under consideration, its safety analysis is performed. In particular, airbags are one of the most important components of a motor vehicle system for the occupant protection. It is used along with and as a supplement to the seatbelt restraint system to provide passenger protection in case of collision In addition to the standard airbags for the driver and front passenger, an increasing number of specialized airbag variants (such as curtain airbags, kneebags, etc.) are used.

Each airbag should be specifically designed and optimized for its intended purpose. In addition to the deployment technology, which can in principle be based on the uniform pressure approach or the more recent corpuscular method, this includes the selection of the inflow method (such as Wang-Nefske or hybrid approaches) as well as the verification and validation of the associated inflow data. Moreover, the deployment behavior is also determined by

the correct adjustment of contact, discharge opening and porosity parameters. As a consequence a sensible and comprehensive simulation of airbag behavior as part of a simulation of the entire restraint system is indispensable.

An airbag is typically made of synthetic material and equipped with holes in the rear; it is usually composed by different subsystems such as:

- a *sensor* that detects the abrupt deceleration of the vehicle caused by an impact and the pressure;
- an *Airbag Control Unit (ACU)* that monitors the readiness of the entire airbag system;
- a *detonator* that triggers the substance contained in the explosive capsule through an electric current or a bump of a ferrule;
- a possible second capsule (GasSource) that contains pre-compressed inert gas which inflates the airbag;
- a *warning light* which is illuminated if a fault is detected.

Specifically the *ACU* receives the signal of the *sensor*, processes it and sends the command to switch on a *detonator*; which in turn blows up the capsule of the detonator by developing a large amount of gas, to inflate the container.

### 5.3.1 Requirements Analysis phase

In this phase of the proposed process all the possible user requirements need to be identified and elicited- By following the process described in the Section 5.1, in the Requirement Analysis phase all the possible user requirements are identified and elicited As an example, in the following some *URs* are reported:

- (*Req1*) when the car decelerates very quickly, as in a head-on crash, the electrical circuit has to be turned on for initiating the process of inflating the airbag;
- (*Req2*) the process, from the initial impact of the crash to full inflation of the airbags, takes less than 40 milliseconds;
- (*Req3*) when a sensor detects a collision an immediate trigger should be sent to enable the deployment of the airbag;
- (*Req4*) in order for the airbag to cushion the head and torso with air for maximum protection, the airbag must begin to deflate (i.e., decrease its internal pressure) by the time the body hits it, otherwise, the high internal pressure of the airbag would create a hard surface instead of a protective cushion;
- (*Req5*) the airbag is ignited within a well-define threshold.

Starting from the collected *URs* the next step consist into their rewriting in SRs for making them more formal and by identifying their belonging *RequirementModel*. For example:

- *AbruptDeceleration (Req1)*: when the deceleration d is greater than a *threshold*, a signal to switch on the electronic circuit has to be sent;
- *InflationTime (Req2)*: The time to inflate the airbag has to take less than *40ms*, $inflationTime \leq 40$;
- *CollitionDetection (Req3)*: when the collision is detected by the sensor, a *collitionSignal* has to be generated;
- *DeflationTime (Req4)*: the airbag has to be able to deflate in a deflation-Time lesser than a deflation threshold.
- *Activation (Req5)*: after a crash the airbag is deployed in $delayTime = 45ms$.

Specifically, the relationships among the above mentioned safety requirements are represented in Figure 5.3. In particular the status of the *Deflation-Time* is not violated if at least the status of the requirement *InflationTime* is not violated. In turn the status of the *InflationTime* is not violated if at least the status of the *Activation* requirement is fulfilled at least by both the *AbruptDeceleration* requirement and the *CollitionDetection* requirement. That is to say, the status of both AbruptDeceleration and CollitionDetection must be not violated. Moreover different scenarios can be analyzed, such as:

1. the airbag is not ignited or is inflated too late even though a critical crash occurred;
2. the airbag is deployed unintentionally, which means that it is ignited even though no crash at all or only a non-critical crash has occurred;

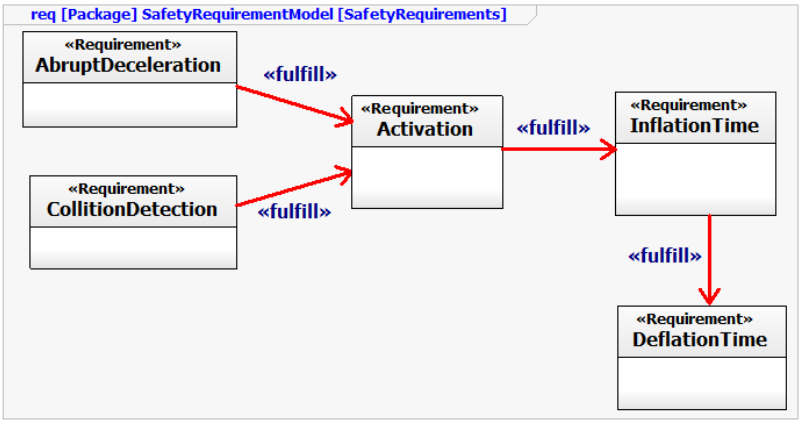



**Fig. 5.3.** Safety System Requirement relationships

### 5.3.2 System Modeling phase

In this phase both the physical structure of the system is built by composing components and then the behavior of each single component is specified. As it is shown in Figure 5.4, a Block Definition Diagram (BDD) of an Airbag System is depicted, in terms of its subsystems and ports. Then, the interactions among these components are better specified by using the Internal Block Diagram (IBD), as it is shown in Figure 5.5.

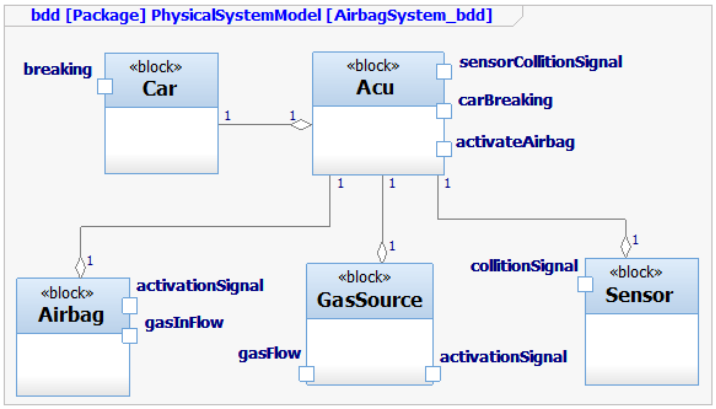



**Fig. 5.4.** Physical System Model: Components of the Airbag System

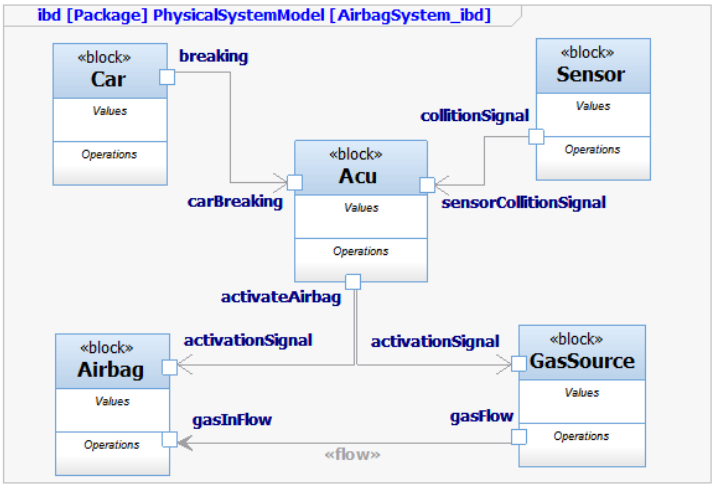



**Fig. 5.5.** Physical System Model: Components interactions of the Airbag System

After the structure is built, Parametric diagrams are employed for representing the behavior of each subsystem as well as dynamic interactions among them, by exploiting a *Computational Model* based on *EquationsSet*. As an example, in Figure 5.6 the diagram concerning the behavior of the Airbag component is reported. In particular, in the first section of the diagram, the parameters taken in input from the model are defined, secondly a brief description about the use of such parameters is reported; then the behavior of the Airbag component, which exploits such input parameters, is represented in terms of equations.

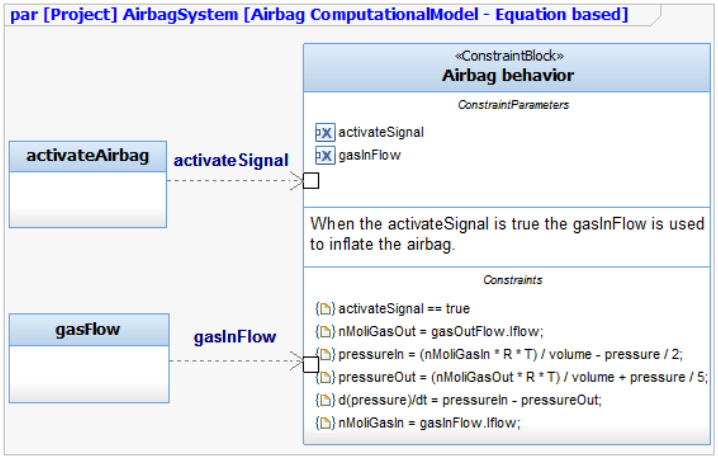



**Fig. 5.6.** Computational Model of the Airbag component

Finally, requirements modeled in the previous phase, which need to be verified, are allocated to (i) a single physical component in order to check its behavior or (ii) a set of physical components in order to check if the interaction among them is or is not consistent as expected. In Figure 5.7 the allocation of some requirements to the airbag physical system model, is shown. In particular, such a scenario wants to verify, the *InflationTime* of the airbag when a car-crash occurs. Specifically, the requirement is not violated when the status of the *Activation* requirement is not violated and both the *Acu* component and the *Airbag* component fulfill the internal rules specified by the *InflactionTime*.

Figure 5.8 shows the design of the Airbag System under consideration in OMEdit, the graphical editor of OpenModelica, which has been extended to enable the modeling and verification of requirements.

In particular, a new panel has been introduced in OMEdit of OpenModelica which contains:(i) a button for graphically model the *fulfill* relationship; (ii) a button that allows to hide/visualize a *RequirementAssertion* and all its relationships between the *System Design* and the *RequirementsModel*; (iii) a

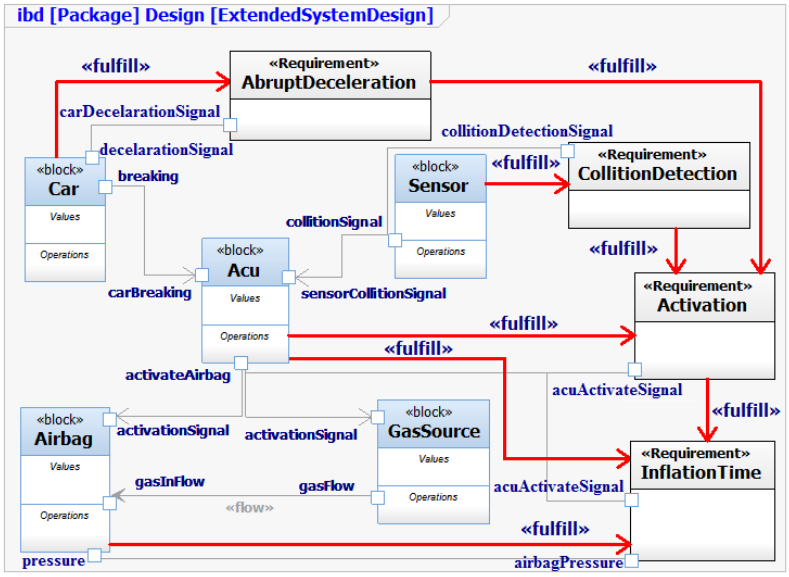



**Fig. 5.7.** Allocation of Safety Requirements to the Airbag Physical System Model

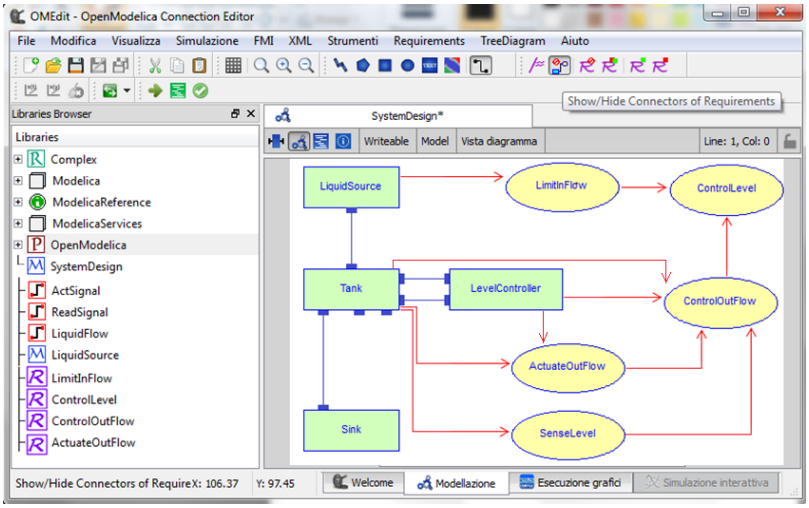



**Fig. 5.8.** An extension of OMEdit, an Open Source graphical editor in OpenModelica, supporting Requirements

button that allows to hide/visualize connections defined through the *connect* contruct between *RequirementAssertion* and *PhysicalComponentModels*, only in order to make more readable the graphic representation; (vi) other two buttons that allow to enable and disable requirements, in order to choose which of

them will take part of a specific simulation run, as a consequence for defining which requirement will be evaluated.

### 5.3.3 Virtual Testing phase

In this phase the virtual testing is executed by exploiting the simulation in order to study the behavior of the system under consideration and analyze interesting aspects or, possibly, to discover some issues that are not immediately obvious when applying static analysis techniques. In order to enable the simulation, models generated in the previous phase need to be translated into the desired simulation platform in order to make them executable. In this case the OpenModelica environment has been chosen as simulation platform since: (i) it is equation based (by implementing the Modelica Language) and, as a consequence, compliant with the *Computational Model* which has been used to represent the behavior of the overall Airbag System; (ii) it is open source, thus allowing the possibility to extend both the language and the tool, to enable modeling of requirements and introduce allocation mechanisms.

In Figure 5.9, a fragment of source code in Modelica language, which represents the structure of the AirbagSystemDesign, is reported.

```
model AirbagSystemDesign
 import AirbagPhysicalComponentModel.*;
 Car car;
 Acu acu;
 Sensor sensor;
 Airbag airbag;
 GasSource gasSource(gasGain = 0.73);
 GasOut gasDestination(gasGain = 0.02);
equation
 connect(car.breaking,acu.carBreaking);
 connect(sensor.collitionSignal,acu.sensorCollitionSignal);
 connect(sensor.collitionSignal,acu.carBreaking);
 connect(acu.activateAirbag,airbag.activationSignal);
 connect(airbag.gasInFlow,gasSource.gasFlow);
 connect(airbag.gasOutFlow,gasDestination.gasFlow);
end AirbagSystemDesign;
```

**Fig. 5.9.** Airbag System Design in Modelica

As it is shown by looking at the source code, the transformation between ModelicaML notation and Modelica constructs, is almost direct. In particular, each *ModelicaML block* can be represented as a *Modelica Model*, whereas connections among ModelicaML blocks can be enabled by the *connect* construct, which is already available in the Modelica standard language. In Figure 5.10, a fragment of Modelica source code concerning the implementation of the behavior of the airbag component, is shown.

As it is shown in the picture, the component behavior, which is described in the *System Modeling* phase (see Figure 5.6), has been is translated in a set

```
model Airbag
 Boolean activationSignal(start = false);
 GasFlow gasInFlow "Connector-flow";
 GasOut gasOutFlow "Connector-flow";
 Real nMoliGasIn(unit = "mol", start = 0);
 Real nMoliGasOut(unit = "mol", start = 0);
 constant Real volume(unit = "l") = 9.0;
 constant Real R(unit = "l*atm/mol*K") = 0.0821;
 constant Real T(unit = "K") = 295.0;
 Real pressureIn(start = 0.0, unit = "atm");
 Real pressureOut(start = 0.0, unit = "atm");
//Airbag Internal Pressure
 Real pressure(start = 0.0, unit = "atm") ;
equation
 if activationSignal == true then
   der(pressure) = pressureIn - pressureOut;
   nMoliGasIn = gasInFlow.lflow;
   nMoliGasOut = gasOutFlow.lflow;
   pressureIn = (nMoliGasIn * R * T) / volume - pressure / 2;
   pressureOut = (nMoliGasOut * R * T) / volume + pressure / 5;
 ...
 ...
 end if;
end Airbag;
```

**Fig. 5.10.** Representation of the behavior of the Airbag component in Modelica

of equations by using the Modelica language. Similarly, the requirements identified in the Requirements Analysis phase are formalized by exploiting some extensions of the Modelica language, proposed by the author in [105]; specifically: (i) the *requirement* keyword is used for their representations, (ii) the *fulfill* relationship is used both for their allocation to the physical system and for their traceability, (iii) the *precondition equation* section is used to specify the conditions when the evaluation of the requirement has to be performed. In particular, the source code of the formalized requirement *InflationTime* is reported in Figure 5.11, where the evaluation is based on the inflation time that the airbag takes to reach a specific safety level of pressure after the airbag is activated.

The source code of the extend system design is reported in Figure 5.12, where the *fulfill* keyword is employed for creating the matching among the requirements as well as between requirements and physical components of the airbag system.

After obtaining the executable models, the tuning of the simulation parameters is performed in order to reach a safe working state of the system according to the specified requirements. Several simulations have been executed for testing virtually the System in different scenarios and evaluating its behavior. Moreover three possible values can be reached by a requirement.

In the considered experimentations (see Figure 5.13 and Figure 5.14) a pressure level (*safePressureLevel* in yellow color) of 2.5 atmospheres (atm) has been considered as minimum safe threshold, coupled with a maximum inflation time (*maxInflationTime*) of 40 milliseconds (ms) in order to reach

```
requirement InflationTime
  Real airbagPressure(unit="atm");
  Real safePressureLevel(unit="atm")=2.5;
  Boolean activateAirbag(start=false);
  constant Real maxInflationTime(unit="ms")=40;
  constant Real activationTime(unit="ms")=20;
precondition equation
 activateAirbag=true;
 (time < activationTime + maxInflationTime)=true;
equation
 (airbagPressure >= safePressureLevel)=true;
end InflationTime;
```

**Fig. 5.11.** Formalization of the InflationTime requirement by using Modelica language extensions

```
model ExtendedSystemDesign
      import SafetyRequirementModel.*;
 // System Design
 AirbagSystemDesign asd;
 // Safety Requirements
 AbruptDeceleration ad;
 CollitionDetection cd;
 Activation ac;
 InflationTime it;
equation
 // Fulfill equations
 {asd.car} fulfill ad;
 {asd.sensor} fulfill cd;
 {asd.acu, ad, cd} fulfill ac;
 {asd.airbag, asd.acu, ac} fulfill it;
 // Connect equations between Requirements
 // and PhysicalComponents
 connect(asd.airbag.pressure,it.airbagPressure);
 connect(asd.acu.activateAirbag,it.acuActivateSignal);
 connect(asd.car.decelarationSignal,ad.carDecelarationSignal);
 connect(asd.sensor.collitionSignal,cd.collitionDetectionSignal);
 connect(asd.acu.activateAirbag,ac.acuActivateSignal);
end ExtendedSystemDesign;
```

**Fig. 5.12.** Formalization of the ExtendedSystemDesign by using Modelica language extensions

such safe pressure level of the airbag, after that the activation airbag signal (*activateAirbag*) is arrived.

As it is shown in Figure 5.13, even though the *AbruptDeceleration* (in dark blue color) and *CollitionDetection* (in dark green color) requirements are satisfied as well as the *Activation* (in brown color) requirement, the *requirementStatus* of the *InflactionTime* (in red color) is negative, that is to say it is violated. Indeed, as we can see, the *airbagPressure* (in light blue color) is not able to reach the *safePressureLevel* (in yellow color) within the *maxInflationTime* (in 40 milliseconds) as expected.

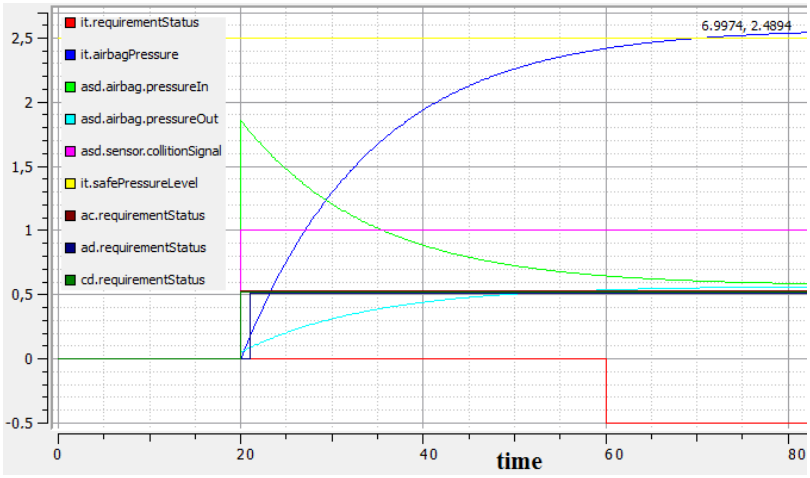



**Fig. 5.13.** Violation of the InflationTime requirement

As it is shown in Figure 5.14, by setting opportunely some parameters of the airbag system, for example, by increasing the pressure to be provided in input to the airbag (*pressureIn* in light green color), it is possible to reach the necessary parameters tuning which fulfills all the requirements in the considered scenario.

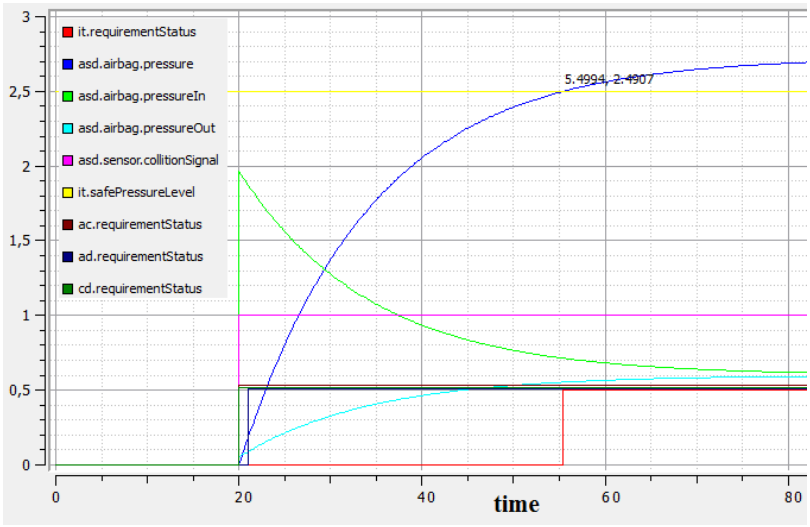



**Fig. 5.14.** Fulfillment of the InflationTime requirement

## 5.4 Conclusion

This Chapter has presented a model-driven process for supporting the safety analysis of systems which is inspired by ISO-26262 standard and exploits simulation techniques. Two powerful languages for modeling systems have been combined in a comprehensive system engineering framework; specifically, ModelicaML has been exploited for platform independent representation of the system; whereas, the Modelica language has been exploited for the executable representation of the systems according to an equation based paradigm. A prototype of the OpenModelica simulation platform, able to support both the modeling of requirements and their allocation, according to a well-defined reference meta-model, has been used for the simulation. Finally, a concrete experimentation has been conducted in the automotive domain which has allowed to point out both the flexibility and the effectiveness of the overall proposed process for safety analysis.

# 6

# Further Contributions on Modeling of System Requirements

The experience gained during the definition of the two methods for Reliability and Safety analysis described in Chapter 3 and 5 respectively, along with the research activities performed also in the context of a European Research project (MODRIO project - ITEA 2) [60], allowed me to focus on the more general problem concerning the *modeling of system requirements* and their formal representation in recently proposed and popular modeling language.

In particular, the main contribution is related to the representation of constraints and requirements that have an impact on the behavior of the system so as to enable their verification through real or simulated experiments [26]. It is strongly related with the work in [62] where the representation of requirements is closely bounded and restricted to the exploitation/implementation of a software library, whereas the contribution described in this Chapter aims at defining some extensions of modeling languages (such as Modelica language) so as to allow the requirements modeling as well as to define a mechanism to enable their traceability in order to support their verification process through simulation.

The Chapter is structured as follows: in Section 6.1, the proposed reference *meta-model* for representing requirements of physical systems is reported [117]; then the description of a Tank System, which is used in the subsequent Sections as a reference example, is provided in Section 6.2; in Section 6.3 different *approaches* for integrating the modeling of system requirements and their verification during the simulation are proposed in the context of the Modelica language [117]; then, some Modelica extensions along with an *algorithm for dependency tracing of requirements* during simulation is described in Section 6.4 [105]; finally, in Section 6.5 and Section 6.6, a *Probability Model* for further enabling static analysis of a Modelica-based design by exploiting the proposed Modelica extensions, is shown [118].

## 6.1 A Meta-model for representing System Requirements as RequirementAssertions

In this Section a proposed meta-model for representing system requirements is described. Since the notions of model and meta-model are crucial when we talk about representation and modeling, often these terms generate confusion, so it is necessary to clarify the difference between them and the context for the use of each of them [13].

Firstly the concept of *subject* can be defined as the main thing that we want to think/reason about and on which the experiments have to be performed. This usually belongs to the real world. To solve a problem we construct a simplified representation of the subject, called *model*, to which different experiments can be applied, in order to answer questions aimed at the subject. Since a model captures only a part of the complete subject, it is possible to define many models which represent the same subject but that are able to capture different characteristics, aspects, variables and parameters. In order to perform reasoning on a model it is necessary to know exactly which variables are available, furthermore, it is necessary to know the structure of the model. Such information can be expressed through *meta-data* by defining a higher abstraction level called *meta-model*. Hence, a meta-model is a model that defines the structure of valid models (see Figure 6.1).

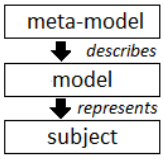



**Fig. 6.1.** Meta-model, model and subject abstraction levels

In the following the definition and description of the proposed *meta-model* (see Figure 6.2) is provided. It is a combination of two main parts: the *Physical Meta-Model* (in the left-side) and the *Requirement Meta-Model* (on the right-side).

As previously stated, before defining *System Requirements*, it is necessary to build a representation of the physical model. Thus, the *meta-data* of the *Physical Meta-Model* are used to describe and to represent one among all the possible physical models of a specific actual system, whereas the *meta-data* of the *Requirement Meta-Model* are exploited to represent *System Requirements* in terms of *requirement assertions* by defining a possible *requirement-model* on a specific physical model representation.

Starting from the *Physical Meta-Model* side, the main concept is the Attribute, which represents a characteristic (i.e. temperature, pressure, level of liquid, age) of an entity (i.e. a system, a sub-system, a component); in the

proposed meta-model, it is defined by (i) a *Name* (by which it is referred) (ii) a *Type* (type of value which is expected), (iii) a *Value* (a possible value among all the range of values related to a specific *Type*) and (iv) (optionally) a *Unit* of measure. Each *Attribute* is associated with one specific *Variability* which in turn can be (i) *Constant* which means that its *Value* never changes, (ii) *Variable* which means that its *Value* depends on other attributes, (iii) *Parameter* which means that its *Value* can be properly tuned. Moreover, each *Attribute* has to specify its access level called *Visibility* which, according to the meta-model, could be either *Private*, if accessible only internally to the component in which it has been defined, *Protected*, if accessible by the descendants, or *Public*, if accessible externally.

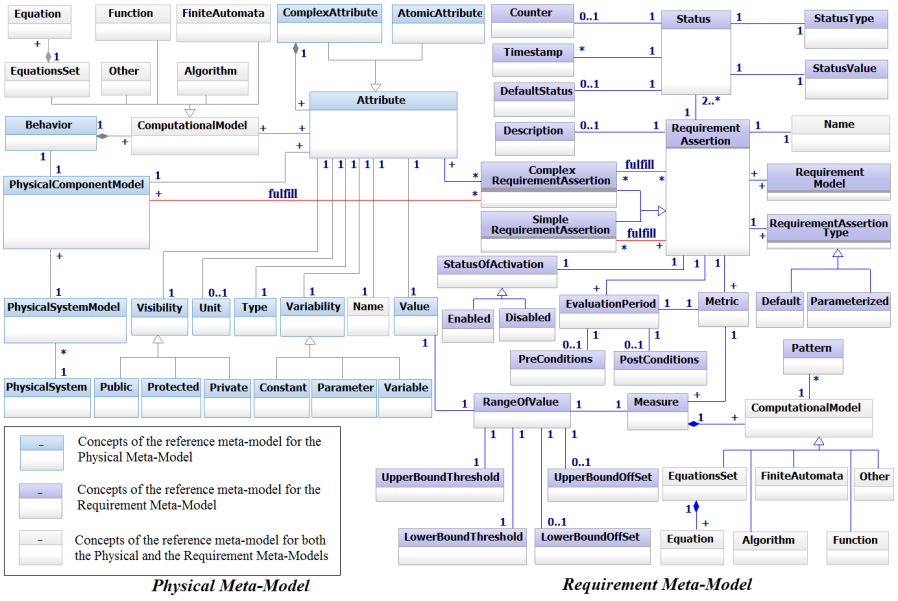



**Fig. 6.2.** A meta-model for modeling System Requirements

An *Attribute* can be (i) an *AtomicAttribute*, which means it cannot be further decomposed, (ii) a *ComplexAttribute*, that is, composed by other attributes.

A *ComputationalModel*, which could be represented through an *Algorithm*, a *FiniteAutomata* (e.g. Timed Automata, Hybrid Automata, etc.), a *Function*, an *EquationsSet* (i.e. a set of *Equation* concepts), or by their combination as well as by *Other* kinds of computational models, defines the behavior of a *PhysicalComponentModel*. An *Attribute* has to belong at least to one *ComputationalModel* as well as a *ComputationalModel* has to use at least one *Attribute*. One or more *PhysicalComponentModels* compose a *PhysicalSystem*-

*Model*, which in turn is one of the many possible models to describe an actual system called *PhysicalSystem*.

While the *meta-data* on the left side of the figure is used for the description of the physical model, moving to the right side of the *meta-model*, we can see the concepts used for the modeling of *System Requirements*. Among these, the main concept is the *RequirementAssertion*, which is used to describe a *Requirement* of a system. A *RequirementAssertion* can be (i) a *SimpleRequirementAssertion*, that means it does not receive any input from any *PhysicalComponentModel*, (ii) a *ComplexRequirementAssertion*, which is connected directly to at least one *Attribute* and to one *PhysicalComponentModel*; this means that a *ComplexRequirementAssertion* is based on at least a *PhysicalComponentModel* and it is able to receive one or more input values coming from several attributes of the physical model; moreover, a *RequirementAssertion* (*SimpleRequirementAssertion* or *ComplexRequirementAssertion*) could be defined in terms of other *RequirementAssertions* whereas on a single *PhysicalComponentModel*, different *RequirementAssertions* can be defined.

According to the meta-model a *RequirementAssertion* belongs at least to one possible *RequirementModel* as well as a *RequirementModel* has to define at least one *RequirementAssertion*; each *RequirementAssertion* being characterized by:

- a *Name* and a possible *Description* in a text format by using the natural language;
- a *RequirementAssertionType* which specifies the type of the role played by the *RequirementAssertion*; in particular a *RequirementAssertion* can have (i) a *Default* behavior type: it is allowed only to monitor a *PhysicalComponentModel* without influencing its evolution; (ii) a *Parameterized* behavior type: it is able to alter the value of a *PhysicalComponentModel* and influence its evolution (the *RequirementAssertion* has both *read* and *write* capabilities);
- at least two *Status* in order to represent the status of fulfillment of the requirement, which in turn is defined in terms of a *StatusType* and a *StatusValue*. The first concept defines the type of value that a state can take (i.e. a Boolean type, a real type, etc.) whereas the second one represents the value which is related to a specific *StatusType* (such as True/False for a Boolean or NotEvaluated/Satisfied/NotSatisfied for a three valued logic, etc.). Each *Status* could be associated to both a *Counter* counting how many times the *RequirementAssertion* has gone in a specific state and a *Timestamp* in order to register each occurrence of the event. Moreover, a status can be defined as a *DefaultStatus* (useful, for example, in the initialization phase when none value is still provided to the *RequirementAssertion*). A *RequirementAssertion* has a *StatusOfActivation*, that means it can be *Enabled* and *Disabled* in order to decide if it takes/does not take part in a specific scenario or simulation run.

- at least one *EvaluationPeriod* to indicate when the *RequirementAssertion* has to be evaluated according to possible *PreConditions* and *PostConditions* that could be based on temporal values or on values coming from *Attributes*. Moreover for each *EvaluationPeriod* a *Metric* must be associated.
- at least a *Metric* to describe the objective to be verified for which the *RequirementAssertion* has been defined (e.g. Mean Time To Failure for the Reliability); each metric has to define a way which objectively allows its evaluation in terms of *Measure* (e.g. the number of failures in a period of time to measure the Mean Time To Failure). Specifically, a *Measure* can be expressed by adopting an appropriate *ComputationalModel*; moreover, one or more *Patterns* could be applied for representing such *ComputationalModels* when a sort of recurrent structure occurs (e.g. a threshold pattern, a derivative pattern, a delay pattern, etc.). Furthermore, each measure should define a *RangeOfValue*, within the *Value* of the *Attribute* which is related to, in which it is valid. Such *RangeOfValue* is specified by: (i) a *LowerBoundThreshold*: minimum value of validity in the range; (ii) *UpperBoundThreshold*: maximum value of validity in the range; moreover, further thresholds as *LowerBoundOffSet* and *UpperBoundOffSet* can be exploited when the *Value* of a *RequirementAssertion* is respectively below/above the *LowerBoundThreshold* and *UpperBoundThreshold* for a limited time.

*RequirementAssertions* can describe the state and the *intended behavior* [29, 41] of *PhysicalComponentModels*, i.e. the expected behavior for which components are designed. Both *Physical Meta-Model* and *Requirement Meta-Model* are jointly exploited to describe the overall model (hereafter called *Extended System Design - ESD*) of an actual system.

To further clarify the meta-model above described, a simple exemplification is provided below, where some of the above described concepts are illustrated in order to define an *requirement model* upon a *physical model* in compliance with the proposed meta-model.

The *PhysicalSystem* under consideration is a Water System whose model, i.e. one among all possible *PhysicalSystemModels*, called WaterSystemModel is simply composed by a single *PhysicalComponentModel* of a Tank. The Tank is modeled through different *Attributes* such as the current level of liquid *levelInTank* as well as the height of the tank *tankHeight* (both as a Real *type* and *unit*="m"). Such attributes can be accessed externally (*Public Visibility*), whereas other *Attributes* can be used by the descendants of the Tank (*Protected Visibility*). All those *Attributes* (both with *Public* and *Protected Visibility*) are exploited into a *ComputationalModel* which is defined through different equations (*EquationsSet*) in order to model the Behavior of the *Tank*.

Let us assume to define a *RequirementModel* on this specific *PhysicalSystemModel* (the above described WaterSystemModel), in order to verify the following *RequirementAssertion* of a Tank (hereafter we refer to the model of

the Tank), whose *Description* is: "The level of liquid in the tank shall never exceed 80% of the tank height" and its *Name* is "LevelOfLiquidInTank". According to the meta-model the status of activation (*StatusOfActivation*) of this *RequirementAssertion* is enabled (*Enabled*) for all the simulation time, and its evaluation period (*EvaluationPeriod*) has a duration equal to the duration of the simulation run without further specific *PreConditions* or *PostConditions*. The Status of the *RequirementAssertion* has a *StatusType* set to Boolean, consequently, the allowed status value (*StatusValue*) will range between true and false (or Satisfied and NotSatisfied).

The fulfillment of this *RequirementAssertion* is defined by a metric (*Metric*) based on the current level of fluid in the Tank, which is measured (*Measure*) as a percentage according to the maximum height of the tank. Consequently, the definition of the *RequirementAssertion* exploits the *levelInTank* and *tankHeight* that are both two *Public Attributes* of the Tank, moreover, an internal parameter, equal to 0.8, is used to express the percentage. Finally, this *Measure* is expressed by adopting as *ComputationalModel* a set of equations (*EquationsSet*). In particular, in this case by a single Equation, which is defined according to a threshold Pattern (e.g. levelInTank<0.8*tankHeight); a fragment of the possible Modelica (pseudo) code is reported below (see Figure 6.3).

```
requirement LevelOfLiquidInTank
  Real levelInTank(unit="m");
  Real tankHeight(unit="m");
  parameter Real limit (start=0.8);
equation
  levelInTank<limit*tankHeight;
end LevelOfLiquidInTank;
```

**Fig. 6.3.** A fragment of the Modelica Code

In the following, after a brief introduction of a Tank System as reference example, some approaches for modeling *System Requirements* through *RequirementAssertions*, based on the above presented meta-model and centered on the Modelica language, are described.

## 6.2 Description of a Tank system

In the subsequent Sections a case-study, concerning a Tank System, is first described and then used to show how a reference example. The Tank System is composed by four main physical components: a *Source* component, a *Tank* component, a *LevelController* component and a *Sink* component. The *Source* component produces a flow of liquid, which is taken in input by the *Tank*

component. The *Tank*, which is managed through the *LevelController* component, provides in output a liquid flow according to the control law defined by the *LevelController*. The *Sink* is the component where a part of liquid is sent.

After an analysis of the *URs*, the following main *SRs* (and many others) have been identified:

- *System_Requirement_1*: the system has to be composed by one Source Component, one Sink Component, at least one Tank Component and at least one LevelCotroller Component;
- *System_Requirement_2*: each tank has to provide one port called qIn in order to receive flow from another possible Tank Component (or from the Source component if it is the first Tank component in the chain);
- *System_Requirement_3*: each tank has to be connected to its own Level-Controller component;
- *System_Requirement_3_1*: each Tank component has to provide a port called *tSensor* in order to provide signal to the LevelController component;
- *System_Requirement_4*: the Source component has to provide a flow port called *qOut*;
- *System_Requirement_4_1*: the liquid flow produced by the Source component has to be equal three times the initial flow after 150 seconds;
- *System_Requirement_5_1*: the liquid flow produced by the Source component should be less than 10 $m^3$/s.
- *System_Requirement_5_2*: the role of the LevelController should be verified by exploiting both the $h$ level from the Tank component and the *qOut* flow.
- *System_Requirement_5_3*: the validity of both the *tActuator* (Out-flow) and the *outFlowArea* values should be checked according to a specified function;
- *System_Requirement_5_4*: both the $h$ level and the *tSensor* should provide the same values;
- *System_Requirement_5_5*: the $h$ level coming from the Thank should be checked according to a specified function.

Starting from the *SRs* above described, the main Components of the Tank System as well as one possible *SD* have been defined and their Modelica source code is reported, respectively in Figure 6.4 and in Figure 6.5, whereas in Figure 6.6, the System Design of the Tank System, is represented in ModelicaML.

Then a set of *RequirementAssertions* to be verified, can be defined on the *SD* of the Tank System; in particular:

- *RequirementAssertion_1*: *LimitInFlow*, which takes in input the value of the *qOut* port of the *Source* component. It is satisfied if the liquid flow produced by the *Source* component is less than a specific "maxLevel" (i.e. liquidFlow¡=maxLevel, in our case maxLevel =10).

```
package PhysicalComponentModel
model Source;
  LiquidFlow qOut;
  parameter Real flowLevel=0.02;
equation
  qOut.lflow = if time>150 then
  3*flowLevel else flowLevel;
end Source;
model Tank
  ReadSignal tSensor;
  ActSignal tActuator;
  LiquidFlow qIn;
  LiquidFlow qOut "Connector, flow (m3/s)
  through output valve";
  parameter Real area(unit="m2")=0.5;
  parameter Real flowGain(unit="m2/s")=
  0.05;
  parameter Real minV = 0,maxV = 10;
  Real actuatorControllerV;
  Real outFlowArea(unit="m");
  Real h(start=0.0, unit="m");
equation
  der(h)=(qIn.lflow-qOut.lflow)/area;
  actuatorControllerV=flowGain*
  tActuator.act;
  qOut.lflow = LimitValue(minV, maxV,
  actuatorControllerV);
  tSensor.val=h;
  outFlowArea=-qOut.lflow/flowGain;
end Tank;
…
end PhysicalComponentModel;
```

**Fig. 6.4.** Modelica code of Components of the Tank System

- *RequirementAssertion_2*: *ControlOutFlow*, which takes in input the **h** level from the Tank component and the *qOut* flow to validate the role of the *LevelController*; moreover, to be valid it must be fulfilled by both the *RequirementAssertion_3* and the *RequirementAssertion_4*.
- *RequirementAssertion_3*: *ActuateOutFlow*, which takes in input both the *tActuator* (Out-flow) and the *outFlowArea*, checks if the *outFlowArea* value is proportional at the *tActuator* signal.
- *RequirementAssertion_4*: *SenseLevel*, which takes in input both the *h* level and the *tSensor*, checks if the sensor output is equals to the *h* level (i.e. lLevel=sensorOuput).
- *RequirementAssertion_5*: *ControlLevel*, which takes in input the *h* level coming from the Tank component, checks if hLevel¡9 and hLevel¿5; moreover, to fulfill the *RequirementAssertion_5*, both the state of *RequirementAssertion_1* and of *RequirementAssertion_2* have to be satisfied.

```
model SystemDesign
  import TankComponents.*;
  Tank tank1(area = 1);
  Sink sink1;
  LevelController levelController(ref = 0.25);
  LiquidSource source(flowLevel = 0.02);
equation
  connect(source.qOut,tank1.qIn);
  connect(tank1.tSensor,levelController.cIn);
  connect(tank1.tActuator,levelController.cOut);
  connect(tank1.qOut,sink1.qIn);
end SystemDesign;
```

**Fig. 6.5.** Modelica code of the System Design of the Tank System

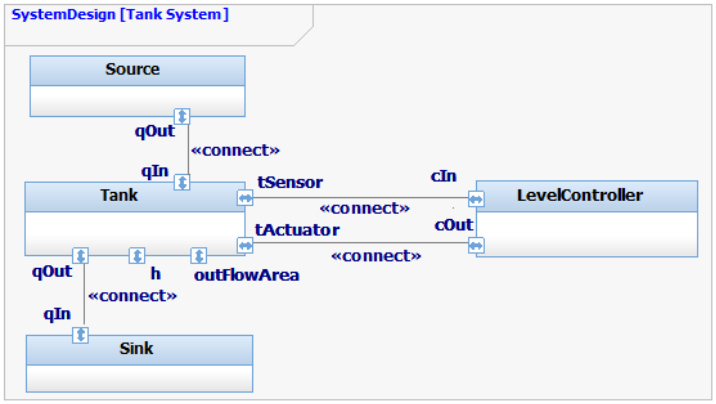



**Fig. 6.6.** The System Design (SD) of the Tank System

## 6.3 Candidate Approaches for representing System Requirements in Modelica language

In this Section different approaches for modeling system requirements and how they can be used to verify the intended behavior of the system and validate it through simulation are described. All the approaches are based upon the Modelica language and ModelicaML.

Although both Modelica and ModelicaML are expressly designed for modeling systems in a coherent framework based on an equation approach, they do not yet provide concepts to be used in order to represent and trace the occurrence of dysfunctional/abnormal behavior (such as faults and failures), that is to say, an observable deviation from the intended behavior at the system boundary [16, 29, 41].

In this perspective, the exploitation of the meta-model presented in the previous Section 6.1, upon the Tank System as a reference case study (described in Section 6.2), can be used to enrich both the Modelica language and ModelicaML to provide them with the capability of modeling system re-

quirements and to enable model checking. In particular, different approaches are presented and discussed in the following subsections based on the main concept of requirement assertion (see Section 6.1).

### 6.3.1 Approach A

In this approach the concepts of *requirement* and *fulfill* are defined as follows:

- *requirement*: which is represented by a *RequirementAssertion* able to validate the *behavior* of a specific *PhysicalComponentModel* which is related to, or to validate interactions among different *PhysicalComponentModels* (according to the *SRs* and the *SD*).
- *fulfill*: which expresses the entailment relationship between *PhysicalComponentModels* and a *requirement*, as well as among *requirements*. Moreover, it provides the propagation process of an assessment among *RequirementAssertions*.

An example model, which illustrates these concepts, is shown in Figure 6.7.

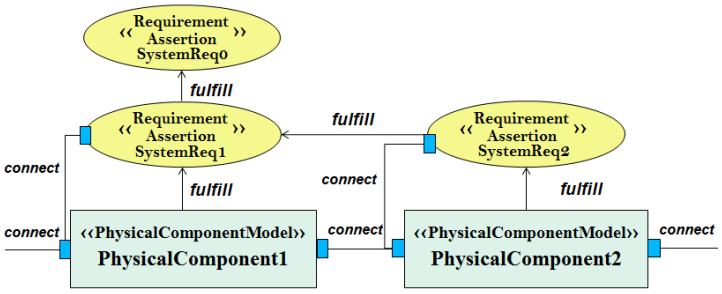



**Fig. 6.7.** A verification model based on requirement assertion and fulfill

In particular, after the declaration of the instances of both *PhysicalComponentModels* and *RequirementAssertions* their relationship is established according to the following *five connection-rules*:

1. the connections enabled through the *connect* construct among *PhysicalComponentModels* are defined to build the *SD* of the *PhysicalModel*;
2. the connections enabled through the *connect* construct among a *PhysicalComponentModel* and a *RequirementAssertion* are used to provide outputs coming from *PhysicalComponentModels* in input to *RequirementAssertions*.
3. the exploitation of the *fulfill* keyword is used to define which instance of an *RequirementAssertion* has to be satisfied/related from at least one specific instance of a *PhysicalComponentModel*.

4. the exploitation of the *fulfill* keyword is used among *RequirementAssertions* to enable the propagation mechanisms of assessment among them;
5. if $A_1, .., A_n$ are *RequirementAssertions* and $C_1, .., C_m$ are *PhysicalComponentModels*, then we can define $(A_2, .., A_n, C_1, .., C_m)fulfill(A_1)$, where $A_1$ is satisfied if and only if $C_1, .., C_m$ satisfy $A_1$ as well as $A_2, .., A_n$ are all satisfied (*fulfill* follows the rule of the And logic).

As we can see in Figure 6.7, the *connect* construct, which is already available in the Modelica language, is used not only among *PhysicalComponentModels* but also between a *RequirementAssertion* and a *PhysicalComponentModel*. Even though the connect construct allows to define connections among attributes of two or more components in an acausal way [25], in this approach some restrictions are defined on it. As an example, the connection is only able to provide inputs from a physical component towards a *RequirementAssertion*. The reason for such a restriction is to prevent a *RequirementAssertion* from providing input to a *PhysicalComponentModel* and consequently affecting its behavior.

Figure 6.8 shows an example of ModelicaML-based notation for the different concepts, then some code fragments of the *RequirementModel* and, in particular, the implementation of *RequirementAssertion_1* and of *RequirementAssertion_5*, through the introduction of the new keyword *requirement*, are reported, by exploiting the *A Approach* in Figure 6.9, whereas the *ExtendedSystemDesign (ESD)* is shown in Figure 6.10.

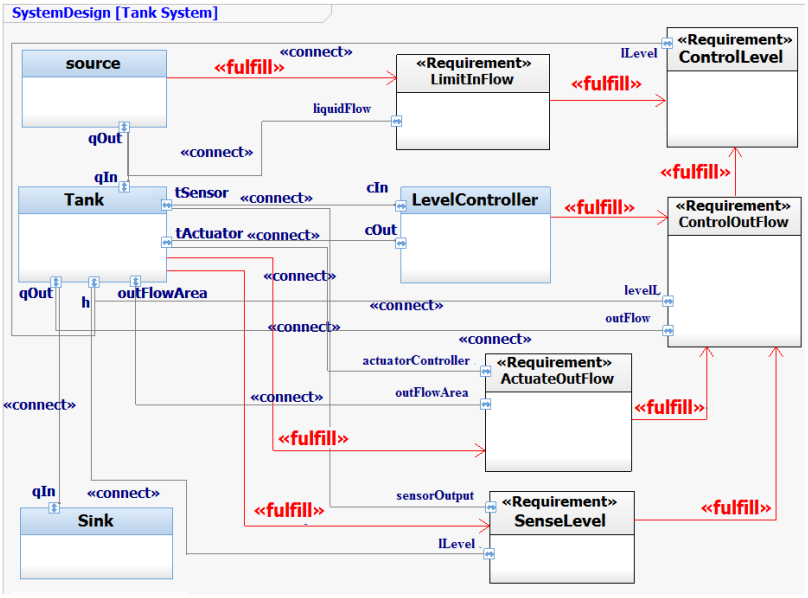



**Fig. 6.8.** Approach A for modeling requirements of the Tank System

```
package RequirementModel
requirement Requirement1
  Real liquidFlow; "qOut of Source"
  parameter Real maxLevel=10;
equation
  if liquidFlow<=maxLevel then
    Status.satisfied;
    …
end Requirement1;
    …
requirement Requirement5
  Real lLevel;
  parameter Real Lmin=5, Real Lmax=9;
equation
  if lLevel<Lmax and lLevel>Lmin then
    …
end Requirement5;
end RequirementModel;
```

**Fig. 6.9.** Modelica code for Requirement Modeling according to the A approach

```
model ExtendedSystemDesign
  //PhysicalComponentModels
  Source source;
  Tank tank1(area=1);
  …
  //RequirementComponents
  Requirement1 limitInFlow;
  …
  Requirement5 controlLevel;
equation
  //Connection among PhysicalComponents
  connect(source.qOut,tank1.qIn);
  …
  //fulfill connections
  (source)fulfill(limitInFlow);
  (tank1)fulfill(actuateOutFlow);
  (tank1)fulfill(senseLevel);
  (limitInFlow,controlOutFlow)
  fulfill(controlLevel);
  (levelController,actuateOutFlow,
  senseLevel)fulfill(controlOutFlow);
  //connection between physical
  //components and requirements
  connect(tank1.h,controlLevel.L);
  connect(tank1.h,senseLevel.lLevel);
  connect(source.qOut,limitInFlow.
  liquidFlow);
  …
end ExtendedSystemDesign;
```

**Fig. 6.10.** Modelica code for the ESD according to the A approach

By adopting this approach, the *RequirementModel* is completely decoupled from the *PhysicalSystemModel* of the system under consideration. Indeed, a requirement model only requires input values of specific types, regardless of the type and the number of components that the values come from. This means that a requirement model could be re-used to validate physical components belonging to different *SD*, although the semantics of such physical components could be completely different. The link between the *RequirementModel* and *PhysicalSystemModel*, occurs only in the *ESD*, through the *fulfill* relationships which govern the assignment of a component to a requirement, while the inputs to be sent to the requirement are provided by the connect construct.

### 6.3.2 Approach B

Whilst the above mentioned approach allows to model requirements in a simple and intuitive fashion, with the help of a minimal set of new concepts (i.e. *requirement assertion* and *fulfill*), the addition of extra connections between *requirement assertions* and components through connect, could make the *ESD* overly verbose and difficult to read from a visual representation point of view, thus complicating the maintainability of the source code.

Therefore, an alternative approach is a variant of the previous one in which along with the keyword *requirement*, another concept (and another keyword) called *On*, which is only visible in the source code of a *RequirementAssertion*, is introduced. Similar to the extends construct, but with some restrictions on the inherited elements, the *On* keyword enables a *requirement* to be defined on specific *PhysicalComponentModels*. Such a *requirement* will inherit the *attributes* on which it will carry out the processing.

The process to build the *ESD* follow the *five-connection-rules*, which have been described in Section 6.3.1 except for the rule number *2*; in this way:

- it allows to avoid the exploitation of extra *connect* (between *PhysicalComponentModels* and *RequirementAssertions*) in order to provide input values coming from *constants*, *parameters* or *variables* of physical components towards a *requirement*. Indeed, such relationships are established during the definition of the *RequirementModel* through the exploitation of the *On* keyword;
- it allows to avoid of having too many connections into a graphical representation, as it is in Figure 6.11, by also reducing the lines of the source code of the *Extended System Design*.

The concept of *fulfill* is that explained in Section 6.3.1.

In this example the *B Approach* is exploited to represent the reference Tank System including the *RequirementAssertions*. Figure 6.12 shows the related ModelicaML-based notation of such a modeling approach. As it is represented, the picture is less crowded with connections and consequently easier to read.

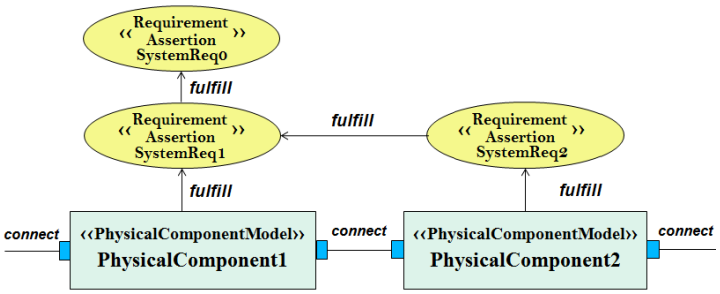



**Fig. 6.11.** Modeling Requirements using the On construct

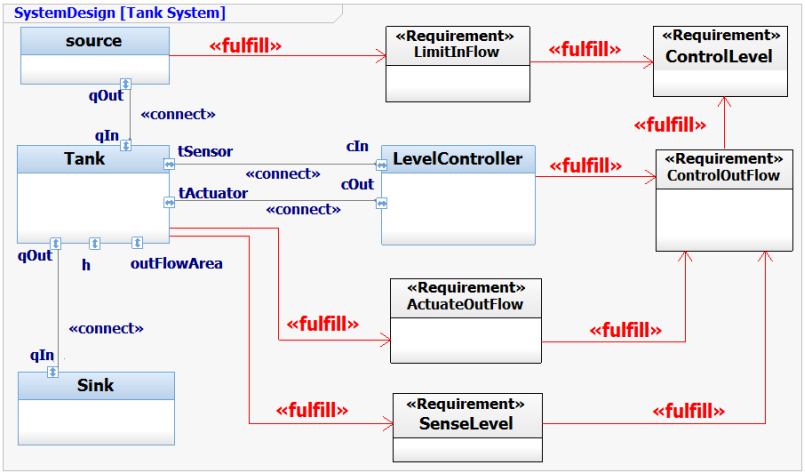



**Fig. 6.12.** Approach B for modeling requirements of the Tank System

Furthermore, as it is shown in the next code fragments (see Figure 6.13) illustrating the source code of *Requirement_Assertion_1* and of *Requirement_Assertion_5*, both the keyword *requirement* along with the *On* keyword are combined for the definition of each requirement. Specifically, starting from the *Source* model, *Requirement1* is defined on it; this means that *Requirement1* is able to use (read-only) all the *Public attributes*, which have been defined by the *Source* model. In particular, the *qOut* attribute of the *Source* model can be used by *Requirement1* without further referencing or connections with the *Source* model.

As for the previous example a fragment of source code combining both the *PhysicalSystemModel* and the *RequirementModel* is presented. As we can see, no connections which use the connect construct between a *PhysicalComponentModel* and a *requirement* component, are present in the source code of the *ESD* model (see Figure 6.14).

```
package RequirementModel
requirement Requirement1 On Source
  parameter Real maxLevel=10;
equation
  if Source.qOut<=maxLevel then
    Status.satisfied;
  else
    Status.notSatisfied;
  end if;
  …
end Requirement1;
  …
requirement Requirement5 On Tank
  parameter Real Lmin=5, Lmax=9;
equation
  if Tank.h<Lmax and Tank.h>Lmin then
  …
end Requirement5;
```

**Fig. 6.13.** Modelica code for Requirement Modeling according to the B approach

```
model ExtendedSystemDesign
  //PhysicalComponentModels
  Tank tank1(area=1);
  Sink sink1;
  …
  //RequirementComponents
  Requirement1 limitInFlow;
  …
  Requirement5 controlLevel;
equation
  //Connections among PhysicalComponents
  connect(source.qOut,tank1.qIn);
  …
  //fulfill relationships
  (source)fulfill(limitInFlow);
  (tank1)fulfill(actuateOutFlow);
  (tank1)fulfill(senseLevel);
  (levelController,actuateOutFlow,
  senseLevel)fulfill(controlOutFlow);
  (limitInFlow,controlOutFlow)
  fulfill(controlLevel);
end ExtendedSystemDesign;
```

**Fig. 6.14.** Modelica code for the ESD according to the B approach

By adopting this approach, the *RequirementModel* is not completely decoupled from the *PhysicalSystemModel* (this makes *requirement assertions* less flexible and less reusable) as it knows *Public Attributes* that are defined in the *PhysicalSystemModel*. On the other hand, it allows for a more immediate exploitation making the *ESD* model easier to read by hiding the details of the matching between the *PhysicalSystemModel* and the *RequirementModel*. Indeed, as it is shown both in Figure 6.12 and through the code of the *ESD*, only

the *fulfill* relationships are visible, while the connection (through the *connect* construct) among *PhysicalComponentModels* and *RequirementAssertions* are not part of the *ESD*.

### 6.3.3 Approach C

Often, it is necessary to have additional mechanisms for generating dysfunctional/abnormal behavior in a physical component, so as to assess the consequences on the whole system. To this end, the *C Approach* proposes the possibility of altering the values of the components starting from the B approach and adding the new notions of *tester entity/component entity* and the *supersede* keyword. The tester entity can be seen as a specific component that is defined on a *PhysicalComponentModel* and which is able to generate outputs (e.g. signals, events or values) according to specific functions and inject them into the *PhysicalComponentModel* in order to alter its intended/nominal behavior (expected values). The *supersede* keyword enables the mechanism to create a reference between an instance of a *tester entity* and an instance of a *PhysicalComponentModel*. In particular, the following rules define the semantics of the *supersede* keyword and how to use it:

1. the exploitation of the *supersede* keyword is used to define which specific instance of a *PhysicalComponentModel* could be compromised by which specific instance of a *Tester component*;
2. if $T_1, .., T_n$ are *Tester components* and $C$ is a *PhysicalComponentModel*, then we can define $(T_1, .., T_n)supersede(C)$, where the operation work of $C$ could be influenced only by one among the $T_1, .., T_n$ *Tester components* (*supersede* follows the rule of the XOr logic).

*RequirementAssertions* can monitor the occurrence of abnormal/dysfunctional behavior in physical components; the *fulfill* relationship is exploited by the *RequirementAssertion* to check the impact and the consequently propagation of possible unexpected values in a component on other components (see Figure 6.15). The *On* keyword enables both *RequirementAssertions* and *Tester components* to have access directly to the attributes of the physical component models on which they are defined.

This approach is adopted to model the previously described *requirement assertions* on the Tank System. Additionally, the possibility of modeling entities that alter the intended behavior of components, and consequently of the system, is taken into account by exploiting *tester entities/components*. In this section, three *tester components* have been defined in order to illustrate their use:

- *AlterSourceFlow* and *AlterSourceFlow2* on the *Source* component, respectively producing the double of the liquid in the first case and producing a negative value of liquid in the second case.
- *AlterOut* on the *Tank* component, where the LimitValue function has been removed from the behavior of the tank.

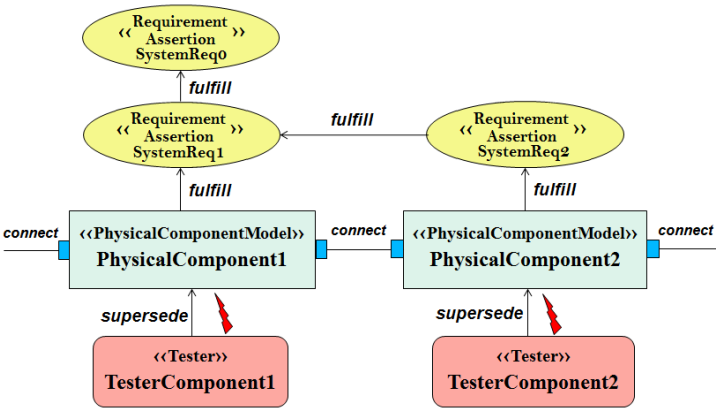



**Fig. 6.15.** Requirements and Tester component for the dysfunctional behavior analysis

In the following, some code fragments describing the *TesterModel* and, specifically, the source code of the *AlterSourceFlow* and of the *AlterOut* are reported (see Figure 6.16).

```
package TesterModel
tester AlterSourceFlow On Source
  parameter Real flowLevel=0.04;
  …
equation
  qOut.lflow=flowLevel;
end AlterSourceFlow;
tester AlterOut On Tank
  …
equation
  actuatorControllerV=-
  flowGain*tActuator.act;
  qOut.lflow = actuatorControllerV;
  tSensor.val = h;
  outFlowArea=-qOut.lflow/flowGain;
end AlterOut;
end TesterModel;
```

**Fig. 6.16.** Modelica code for Requirement Modeling according to the C approach

As we can see in the source code below (see Figure 6.17), the link between *PhysicalSystemModel* and *TesterModel* is defined in the *ESD* through the keyword *supersede*. In Figure 6.18 a ModelicaML-based notation for such a modeling approach, introducing both *Requirement* and *Tester components* as well as physical components is depicted.

```
model ExtendedSystemDesign
  //PhysicalComponentModels
  Tank tank1(area=1);Source source;
  …
  //RequirementComponents
  …
  //TesterComponents
  AlterSourceFlow alterSourceFlow;
  AlterSourceFlow2 alterSourceFlow2;
  AlterOut alterOut;
equation
  //supersede relationships
  (alterSourceFlow,
  alterSourceFlow2)supersede(source);
  (alterOut)supersede(tank1);
  //fulfill relationships
  …
end ExtendedSystemDesign;
```

**Fig. 6.17.** Modelica code for the ESD according to the C approach

It is worth noting that one possible variant of the *C Approach* consists in defining the relationships between a *PhysicalComponentModel* and a *Tester component* in the *ESD* by using the construct *connect*, in order to avoid the exploitation of the *On* keyword during the definition of the tester components in the *TesterModel*. By adopting this version (similar to the *A Approach*), the *PhysicalSystemModel* will be completely decoupled from both the *RequirementModel* and the *TesterModel*.

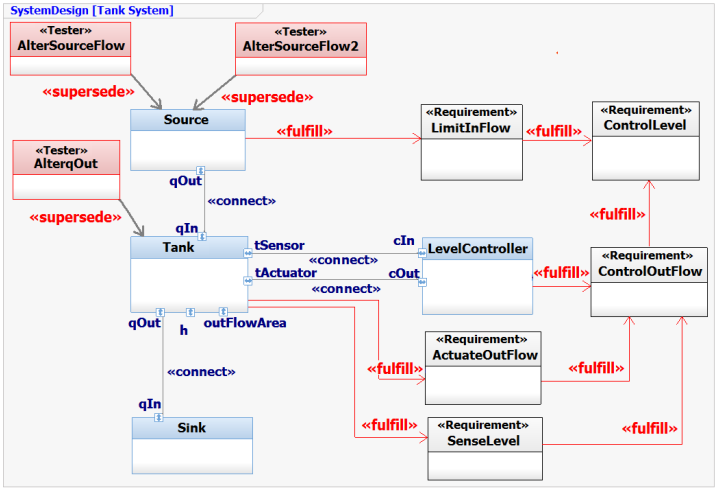



**Fig. 6.18.** Approach C for modeling requirements of the Tank System

## 6.4 Modelica Extensions, Requirements Verification and Dependency Tracing

Compliant with the meta-model proposed in the Section 6.1, starting from an actual system named *Physical System*, different *Physical System Models*, representing the specific aspects that each model wants to capture, can be defined. In particular, following the object oriented paradigm (OOP), a given *Physical System Model* can be defined by (re)using new or already defined *Physical Component Models*. As a reference, the architectural framework shown in Figure 6.19 has been followed with the aim of extending the Modelica-based model by introducing, beside the *Physical System Model*, which provides one possible representation of the *Physical System*, two more key elements: (i) *Requirement Model*, which formalizes the expected behavior of the system; (ii) *Monitoring Models*, in which possible relationships between the *Physical System Model* and *Requirement Model* are defined.

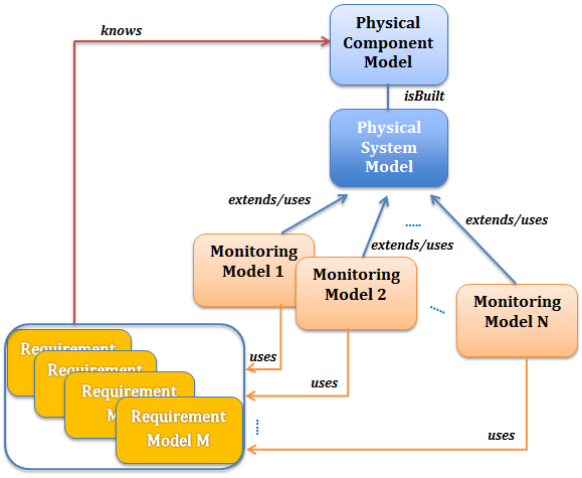



**Fig. 6.19.** A reference architectural framework based on Modelica

From the point of view of the models:

1. The goal of the *Monitoring Model* is to connect together a *Physical System Model* and a set of *requirements* to be verified. For a given *Physical System Model*, different *Monitoring Models* with different configurations and different *requirement* sets can be defined.
2. *Requirement Models* are used to describe the expected state and behavior of the *Physical System Model*. In order to make verification intuitive, flexible and to benefit from the power of the OpenModelica tools, it has been chosen to represent requirements in the same formalism by extracting them from their textual specification, and formalize them in Modelica.

From the point of view of the language (and in particular on Modelica language):

1. To represent requirements, a new type of class, a *requirement* class, has been introduced. This allowed to adopt a modular development approach to requirement modeling, wherein the paradigms of object-oriented programming such as inheritance and hierarchical structuring can be exploited. A textual requirement is therefore formalized into a computable form, by defining the expected behavior of the system as a set of Modelica equations. A *requirement* class reads system values through classic Modelica ports, however by its definition it does not feed back to the system, as requirement verification should not influence the behavior of the system
2. A *precondition equation* section can be also contained in a *requirement*. The goal of a precondition is to define the situations in which the requirement applies/acts (has to be evaluated/considered).
3. Moreover, because the dependencies between the different components and requirements are not necessarily deductible from the data flow relationships between the requirements and the related components, an extension of Modelica language (called *fulfill*) for explicitly declaring such relations has been also define. The following equation (6.1):

$$\{r_1, ..., r_n, c_1, ..., c_m\}\textbf{fulfill}r \tag{6.1}$$

states that the verification of requirement $r$ depends on requirements $r_1$ to $r_n$ and components $c_1$ to $c_m$. This relationship also allows to adopt a refinement approach to requirement modeling, as bigger requirements can be fractioned into more refined sub requirements, and their relationships can be expressed through *fulfill* equations.

The two steps to requirement verification are the verification of each individual requirement and the verification of the *fulfill* relationships. A requirement class contains equation sections, like a normal Modelica model. However these equations are not interpreted in the same way. Instead of being used to solve the overall model, they can be viewed as conditions on the different values in the system that must be verified. To simplify the debugging, analysis process, a requirement in a system model can either be active, or inactive. An inactive requirement is not verified at simulation. However a requirement can only be de-activated if no active requirements depend on it. Specifically a requirement can be evaluated to three possible states:

- *Violated*, meaning that under the current simulation parameters the requirement was not validate;
- *Not violated*, meaning that under the current simulation parameters the requirement hold;
- *Not evaluated*, meaning that in the given scenario, the preconditions of the requirement are never fulfilled, thus is never evaluated.

Different simulation configurations are possible: a requirement can either be verified at each step of the simulation, registering the number of violations, or the simulation can stop at the first violation of a requirement. Once the individual requirement values are computed, the fulfill relationships must be verified. In the following a first effort of requirement verification and traceability process in Modelica through an algorithm, has been formalized. This algorithm aims to provide an answer to the following questions:

- When is a *RequirementAssertion* violated?
- How can the component responsible for violating the *RequirementAssertion* be identified?

A requirement can be *violated*, not only when a single component fails, but also when the interactions among two or more components are wrong. Basically, the verification and traceability process requires to trace all the fulfill relationships for a given requirement, to reach a set of components that the requirement depends on. This set can be then analyzed, to detect that (i) either a component is not properly working or (ii) the interaction among a set of components is not consistent as expected. Two types of *RequirementAssertions (RAs)* are distinguished:

- A *RA* upon a single *Physical Component*, that defines its expected behavior;
- A *RA* upon a two or more *Physical Components* that defines the expected behavior in terms of interactions among a set of *Physical Components*.

As a consequence, it can be identified a single component as a possible *culprit* or a set of components within which one or more components are responsible for violating the requirement. To illustrate the difference between these two notions, let us suppose that two components are working properly when considered individually, meaning that they are well-defined. However, when they are used together, their interaction could not work as expected. In such a case this means that this part of the physical model should be redesigned/reviewed, because the behavior of components is correct but their interaction is not. In order to address both issues, the following *algorithm* has been proposed:

Let $A$ be a set of *RequirementAssertions (RAs)* $\{A_1, A_2, ..., A_i, ..., A_n\}$ with $1 \leq i \leq n$ ; Let $C$ a set of *PhysicalComponentModels* $\{C_1, C_2, ..., C_j, ..., C_m\}$ to be verified using $A$, with $1 \leq j \leq m$. Let $C'$ a subset of $C$, and $A'$ a subset of $A$, such that $card(C') > 1, card(A') > 1, A_i \notin A'$ and $C_j \notin C'$. Moreover the functions *holds* and *preconditions* have been defined, that for a given $RA$ can check whether the equations in its equation and precondition sections, respectively, are verified.
For each $A_i \subset A$ the status is initialized to **not evaluated**. Then, for each $A_i \subset A$ if $preconditions(A_i)$ are valid,then $check(A_i)\{$

```
CASE 1: Cj fulfill Ai
        if holds(Ai) then return {NOT violated, {}};
        else return{violated, Cj};
        end if;
CASE 2: C' fulfill Ai
        if holds(Ai) then return {NOT violated, {}};
        else return{violated, C'};
        end if;
CASE 3: A' fulfill Ai
  {state, culprit} is { NOT violated, {}};
        for each Ak ⊂ A'
          check(Ak) is {status', culprit'}
          if status' is violated then
          {state, culprit} is {violated, culprit ∪ culprit'}
        end if;
        end for;
        return {state, culprit};
CASE 4: C', A' fulfill Ai
        if holds(Ai) then
          {state, culprit} is { NOT violated, {}};
        else {state, culprit} is {violated, C'};
        end if;
        for each Ak ⊂ A'
        if check(Ak) is {status', culprit'}
          if status' is violated then
          {state, culprit} is {violated, culprit ∪ culprit'}
         end if;
        end for;
        return {state, culprit};
}
```

Using verification at simulation, we can only assert that a given component has not violated a given set of requirements under the current runtime configuration. Therefore for each requirement this algorithm returns its state, as evaluated with the given simulation parameters, and if it was violated, a set of components possibly responsible for the violation.

The two steps to requirement verification are the verification of each individual requirement and the verification of the *fulfill* relationships. For simplicity the relationships among requirements and physical component are briefly reported in the follow:

1. liquidSource **fulfill** limitInFlow;
2. {tank1, levelController} **fulfill** actuateOutFlow;
3. tank1 **fulfill** senseLevel;
4. {limitInFlow, controlOutFlow} **fulfill** controlLevel;
5. {tank1, levelController, actuateOutFlow, senseLevel} **fulfill** controlOutFlow;

The algorithm, which has been just described, is applied on the Modelica design of the Tank system, for each of the possible cases, by analyzing the five *fulfill* relationships which have been reported above.

Case 1: This evaluation is only based on the *inputs/variables*, provided by a single component, which must fulfill some *rules/equations* defined inside a requirement class. As an example (considering the *fulfill* relationship 1) *limitInFlow* is going to verify whether the component *liquidSource* is working according to some *rules/equations* specified in *limitInFlow* by exploiting only the input provided by *liquidSource*, so *limitInFlow* is *violated* if such *rules/equations* are *violated*, otherwise *limitInFlow* is *not violated*; as well as (considering the *fulfill* relationship 3) *senseLevel* is going to verify only if the component *tank1* is working according to some *rules/equations* specified in *senseLevel* by exploiting only the input provided by *tank1*, so *senseLevel* is *violated* if such *rules/equations* are *violated*, otherwise *senseLevel* is *not violated*.

Case 2: This evaluation is based on the *inputs/variables*, provided at least by two different components, which must fulfill some *rules/equations* defined inside a requirement class. As an example, considering the *fulfill* relationship 2, *actuateOutFlow* is going to verify only if the interactions between *tank1* and *levelController* happen according to some *rules/equations* specified in *actuateOutFlow* by exploiting the inputs provided by *tank1* and *levelController*, so *actuateOutFlow* is *violated* if those *rules/equations* are *violated*, otherwise *actuateOutFlow* is *not violated*.

Case 3: This evaluation is only based on the *Status* of each single requirement in the left side of the *fulfill* equation. As an example, considering the *fulfill* relationship 4, *controlLevel* does not take any *inputs/variables* from any component, so no *rules/equations* are specified in *controlLevel*. As a consequence *controlLevel* is *not violated* if and only if both *limitInFlow* and *controlOutFlow* are *not violated*.

Case 4: This evaluation is based both on the *inputs/variables*, provided at least by one component, which must fulfill some *rules/equations* defined inside a requirement class as well as on the *Status* of each single requirement in the left side of the *fulfill* equation. In this case, considering the *fulfill* relationship 5, *controlOutFlow* verifies that both the interactions between *tank1* and *levelController* happen according to some *rules/equations* specified in *controlOutFlow* by exploiting both the inputs provided by *tank1* and *levelController*. Furthermore, since *actuateOutFlow* and *senseLevel* are on the left side of the fulfill equation, *controlOutFlow* is fulfilled if both the *actuateOut-*

*Flow* and the *senseLevel* are fulfilled. As a consequence, *controlOutFlow* is *violated* when some *rule/equation*, defined in *controlOutFlow*, is *violated* OR when at least the *Status* of a requirement (in this case *actuateOutFlow* or *senseLevel*) in the left side of the fulfill equation computes to *violated*, otherwise *controlOutFlow* is *not violated*.

By running the simulation for the monitoring model described in this Section, with a configuration that respects the constraints defined by *limitInFlow*, *controlOutFlow* and *controlLevel*, it can be seen that the three requirements are valid throughout the simulation (see Figure 6.20).

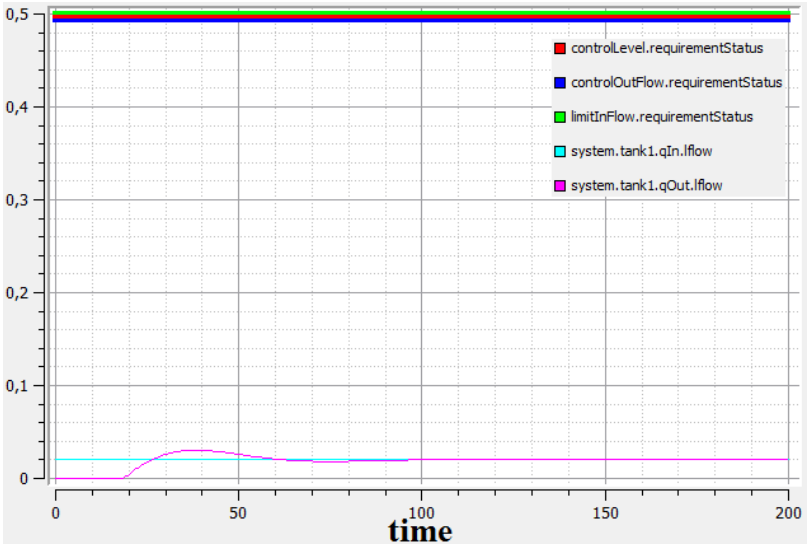



**Fig. 6.20.** Red, blue and green lines represent controlLevel, controlOutFlow and limitlInFlow requirements respectively. They are all evaluated to non violated throughout the execution

Then by modifying the flow level from the source triple after 150 seconds (see Figure 6.21), we can see that the system reports a violation of the requirement *limitInFlow*, which in turn means that *controlLevel* is also violated. Currently, the fulfill relationships are only being used to verify the requirements, however in the future they could be used to assist the designer in analyzing and debugging his model by integrating tool support for tracing the failures to the components that must fulfill the requirements.

Although the requirement model is used for dynamic verification at runtime, and due to the classic reliability and safety techniques are widely adopted, in the next Section is shown how has been natural exploits the proposed Modelica models and related extensions to generate diagram of System for performing further static analysis.

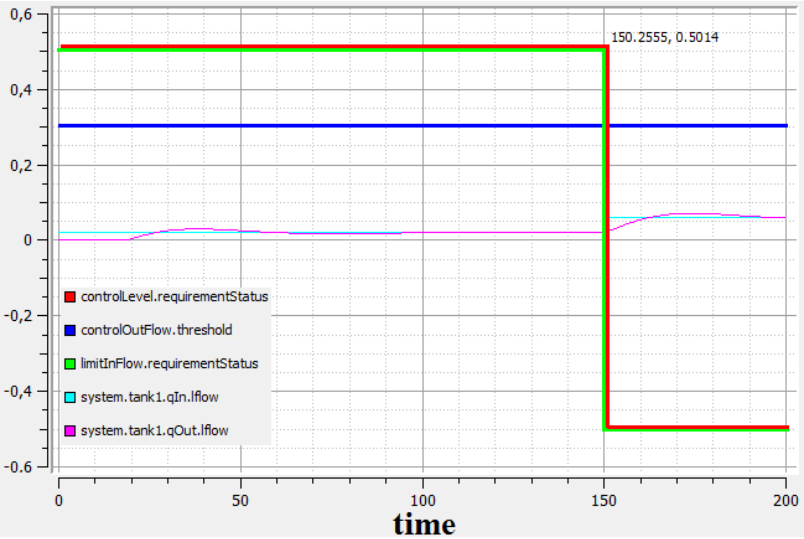



**Fig. 6.21.** After 150 seconds, the Status of controlLevel and limitInFlow requirements changes to violated

## 6.5 Extending the Modelica language through a Probability Model

In this Section a formalism for modeling probabilities in Modelica and its exploitation to analyze the behavior of physical systems under specific operative conditions is shown in the context of an integrated modeling process. According to the process represented in Figure 6.22, which can be integrated into the *System Modeling* phase of the methods presented in Chapter 5, the system to be analyzed is defined in terms of:

- *PhysicalComponentModels* which represent basic or complex elements of a *PhysicalSystem*;
- *RequirementAssertions*, used to model requirements, which in turn constitute a *RequirementsModels* that a *PhysicalSystem* has to satisfy.

Starting from a set of *PhysicalComponentModels (PCMs)* and *RequirementsModels (RMs)*, a specific *ExtendedSystemDesign (ESD)* to be analyzed can be generated. Then, a *Probability Model (PM)* is introduced into the *ESD* model through *annotations*. This composite model can be used to generate data for static analysis. For instance, *Fault Tree Diagrams (FTDs)* can be exported, by exploiting *OpenModelica APIs*, in different machine readable XML formats which can be used by external tools for performing *Fault Tree Analysis*.

The *ExtendedSystemDesign (ESD)* is a composite model which is defined by integrating both the *PCMs* of the *PhysicalSystem* and the *RMs* to be ver-

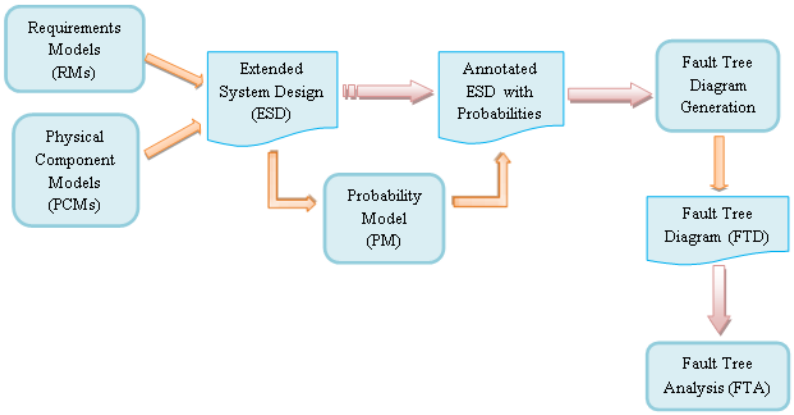



**Fig. 6.22.** The reference process for supporting Fault Tree Analysis in Modelica

ified. In particular, the *PCMs* of the PhysicalSystem are generated by using standard Modelica language, whereas the *RMs* are expressed by exploiting some proposed extension of the Modelica language called *RequirementAssertions* for modeling *requirements* (see Section 6.1).

In Figure 6.23 a template of a requirement represented as a RequirementAssertion is shown.

```
requirement <name>
    <Attribute at1>
    ...
    <Attribute atK>
precondition equation
    <pre-equation pre-eq1>
    ...
    <pre-equation pre-eqN>
equation
    <equation eq1>
    ...
    <equation eqM>
end name;
```

**Fig. 6.23.** Structure of a requirement represented as a RequirementAssertion

This template contains a new type of Modelica *class* defined through the keyword *requirement*, the class has a *precondition equation* section that defines the situations in which the requirement applies and an *equation* section that specify how the requirement has to be evaluated. Moving to the *ESD* model, the concept of *fulfill* is used for tracing requirements. A representation of the

*ESD*, which is built using the proposed extensions of the Modelica language, is shown in Figure 6.24.

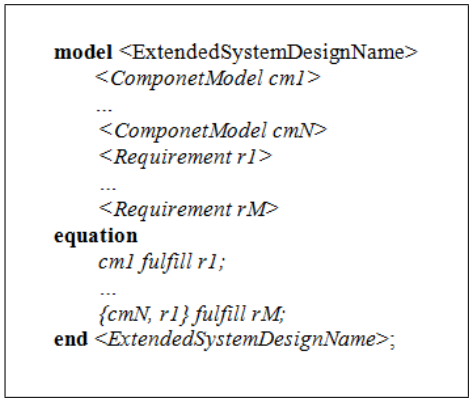



**Fig. 6.24.** Exploitation of the fulfill relationship into the ESD model

The *fulfill* relationships are exploited both to allocate *RMs* to *PCMs* and to generate the hierarchical structure of the *Fault Tree Diagram*.

Then, a *Probability Model (PM)*, which allows to introduce additionally information and as a consequence to enrich a Modelica-based system design, is introduced. A *PM* consists of a list of possible states in which a component can be (i.e. operating, operating in degraded mode, broken) and, eventually, of a set of characteristics associated with each state. In order to avoid the definition of this model for every individual component, it is defined separately and then referenced by every component that needs to use it; such a model allows:

- identifying the cause(s) of a failure;
- identifying the weaknesses in a system design;
- assessing a proposed design for its reliability or safety;
- assessing the effects of human errors;
- prioritizing the contributors to failure;
- making effective upgrades to a system;
- quantifying the failure probability and contributors.

The model is represented by using the Modelica standard language (see Figure 6.25); this makes it reusable in different simulation tool implementing the Modelica specifications.

A *PM* could be introduced without any reference to an *ESD*. Starting from the *PCMs* and the *RMs*, where models of components and requirements are defined as a sort of "component library", a PM can be associated to them as a kind of initialization ("Probability Initialization" or *PI* for short). This means

```
package ProbabilityModel
    model ProbabilityModelComponent1
        parameter real status1 (start = 0.50);
        parameter real status2 (start = 0.50);
    equation
        status1 + status2 = 1.0;
    end ProbabilityModelComponent1;
    ...
    ...
    model ProbabilityModelComponentN
        ...
    end ProbabilityModelComponentN;
end ProbabilityModel;
```

**Fig. 6.25.** Definition of a Probability Model based on equations by using the Modelica languague

that all the instances, derived from models with the same *PI*, will have the same probability distribution and, as a consequence, the same initial values.

*PhysicalComponentModels* and *ExtendedSystemDesign* models generated by combining through the *fulfill* relationships both *RequirementsModels* and *PhysicalComponentModels* can be augmented by introducing a *Probability Model (PM)*.

In particular, Modelica *Annotations* are used to combine the *Probability Model* with a Modelica-based Design. In Figure 6.26 a fragment of code which exemplifies how to use *probability_annotation* is reported.

```
package PhysicalComponentModel
    import ProbabilityModel.*;

    probability_annotation(schema="ProbabilityM
    odels. ProbabilityModelComponent1");
    model ModelComponent1
        ...
        ...
    end ModelComponent1;
    ...
    probability_annotation(schema="ProbabilityM
    odels. ProbabilityModelComponentN");
    model ModelComponentN
        ...
    end ModelComponentN;
package PhysicalComponentModel;
```

**Fig. 6.26.** Exploiting the Probability Model for annotating the Modelica-based System Components

As an example, the *PhysicalComponentModel ModelComponent1*, which is defined in the *PhysicalComponentModel* package, is annotated with the *PM* called *ProbabilityModelComponent1* which is defined in the *ProbabilityModel* package; similarly the *PhysicalComponentModel* called *ModelComponentM*, which is defined in the *PhysicalComponentModelN* package, is annotated with the *PM* called *ProbabilityModelComponentN* which in turn is defined in the *ProbabilityModel* package.

Furthermore, as a component can work in different environments and under specific working conditions which can influence its actual life cycle (for example the temperature), the probability model of any component can be defined/modified during its instantiation. As a consequence, a second annotation step can occur during the definition of the *ESD* as it is shown in Figure 6.27.

```
model ExtendedSystemDesign
      ModelComponent1 component1_1;
      probability_annotation(schema="ProbabilityM
      odels.              ProbabilityModelComponent1",
      status1=0.80, status2=0.20);
      ModelComponent1 component1_2;
      ...
      ...
      ModelComponentN componentN_1;
equation
      ...
      ...
end ExtendedSystemDesign;
```

**Fig. 6.27.** Exploiting the Probability Model for annotating the Modelica-based ExtendedSystemDesign

If there is also a probability model associated with the type of the component that associated with the instance will override it (see the *probability_annotation* of *ModelComponent1* in Figure 6.27 and the example reported in Figure 6.32). The generation of *Fault Tree Diagrams (FTDs)*, is enabled in OMEdit through the implementation of Modelica APIs. This generation is completely automatic and allows the choice among different formats (in the current prototype an XML format, based on the GeNIe tool [37], has been implemented); specifically: (i) the top event is the event to analyze; (ii) the middle events are the intermediate ones; (iii) a basic event is represented by the probability of a failure of a component; (vi) the logical relationships follow basically the And-logic.

The diagram is generated by exploiting two main information sources: (i) the *fulfill* relationships (see Section 6.3), for extracting the structure of the Fault Tree Diagram from the ExtendedSystemDesign; (ii) the information

provided by the *Probability Model* concerning both the operating states and the related values of probability for each component of the Modelica-based design.

In Figure 6.28 a fragment of the template of a FTD in XML is shown.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<smile version="1.0" id="Unnamed"
numsamples="1000" discsamples="10000">
    <nodes>
        <cpt id=" component1_1">
            <state id="status1" />
            <state id="status2" />
            <probabilities>0.80 0.20</probabilities>
        </cpt>
        <cpt id=" component1_2">
            <state id="status1" />
            <state id="status2" />
            <probabilities>0.50 0.50</probabilities>
        </cpt>
    ...
    ...
        <cpt id=" componentN_1">
            <state id="status1" />
            <state id="status2" />
            <parents> component1_1</parents>
            <probabilities>0.5 0.5 0.5 0.5
        </probabilities>
        </cpt>
    </nodes>
    ...
    ...
</smile>
```

**Fig. 6.28.** The XML template of a Fault Tree Diagram generated from a Modelica-based System Design

## 6.6 Performing Fault Tree Analysis of the Modelica-based Design of the Tank System

In order to show how to exploit the *Probability Model* presented in Section 6.5 the Tank System, which is described in Section 6.2, has been considered. It is composed by four *PhysicalComponentModels (PCMs)* (i. g.*Source* component, a *Tank* component, a *LevelController* component and a *Sink* component), furthermore five *RequirementAssertions (RAs)* (*LimitInFlow, ControlOutFlow, ActuateOutFlow, SenseLevel, ControlLevel*) are considered in such analysis.

It is worth noting that further details about the *PhysicalComponentModels* and the *RequirementsModel* of the tank system are not required for the *FTA*

analysis of the Modelica-based design because the hierarchical organization and the structure of the system is sufficient for performing this analysis.

```
model ExtendedSystemDesign
//PhysicalComponentModels
    Tank tank1
    Sink sink1;
    LevelController levelController;
    LiquidSource source;
//RequirementModels
    LimitInFlow limitInFlow;
    ActuateOutFlow actuateOutFlow;
    SenseLevel senseLevel;
    ControlLevel controlLevel;
    ControlOutFlow controlOutFlow;
equation
//Fulfill Relationships
    source fulfill limitInFlow;
    {tank1, levelController} fulfill
    actuateOutFlow;
    tank1 fulfill senseLevel;
    {limitInFlow, controlOutFlow} fulfill
    controlLevel;
    {tank1, levelController,
    actuateOutFlow, senseLevel} fulfill
    controlOutFlow;
//Connections
    connect(source.qOut,tank1.qIn);
    connect(tank1.qOut,sink1.qIn);
    connect(tank1.tSensor,levelController.
    cIn);
    connect(tank1.tActuator,levelControlle
    r.cOut);
    ...
    ...
end ExtendedSystemDesign;
```

**Fig. 6.29.** An example of the Modelica-based System Design extended through the RequirementsModels and by exploiting the fulfill relationship

In order to make the Modelica system design ready to be analyzed statically, the *ESD* (see Figure 6.29 is further enriched through a *ProbabilityModel*. In this phase a *Probability Model* for the Tank System is defined; in particular, the definition of a *PM* is provided for each *PhysicalComponentModel* as well as for each *RequirementModel*.

In Figure 6.30 a fragment of source code of the *PM* for the physical side of the system is reported; in this case each model defines two possible states:

- *Success*: which represents the probability of success of a component performing its work.
- *Failure*: which represents the probability of failure of a component performing its work.

In the *PM* of Figure 6.30, the probabilities associated to each status of each probability model have the same initial value (0.50); in general, the number of states as well as their probability values can be freely initialized (i.e. *Success*=0.75, *Failure*=0.25); the only constraint is that the sum of these probabilities has to be equal to 1.0.

```
package ProbabilityModel
   model LevelControllerProbabilityModel
     parameter real success (start=0.50);
     parameter real failure (start=0.50);
   equation
      success + failure = 1.0;
   end SourceProbabilityModel;
   model LiquidSourceProbabilityModel
     parameter real success (start=0.50);
     parameter real failure (start=0.50);
   equation
     success + failure = 1.0;
   end LiquidSourceProbabilityModel;
   model TankProbabilityModel
     parameter real success (start=0.50);
     parameter real failure (start=0.50);
   equation
     success + failure = 1.0;
   end TankProbabilityModel;
   model SinkProbabilityModel
     parameter real success (start=0.50);
     parameter real failure (start=0.50);
   equation
     success + failure = 1.0;
   end SinkProbabilityModel;
end ProbabilityModel;
```

**Fig. 6.30.** An example of a Probability Model based on the Modelica language

### 6.6.1 Augmenting the Tank System design through the Probability Model by using Annotations

In this phase the Modelica design can be enriched through the Probability Model in two steps: (i) during the definition of both *PhysicalComponentModels* and *RequirementsModels*; (ii) during the instantiation of the *PhysicalComponentModels* or of the *RequirementsModels*.

In the first case the *Probability Model* defined for a specific component is valid for all its instances; as an example, in Figure 6.31 the definition of the *LiquidSource* component model, that is annotated with its probability model, is shown.

In the second case the validity of the *Probability Model* is limited only to a specific instance of a component model; as an example, with reference to

```
@probability_annotation(schema="Probabili
tyModels.LiquidSourceProbabilityModel");
model LiquidSource;
  LiquidFlow qOut;
  parameter Real flowLevel=0.02;
equation
  qOut.lflow = if time>150 then
  3*flowLevel else flowLevel;
end LiquidSource;
```

**Fig. 6.31.** A PhysicalComponentModel annotated with a Probability Mode

the *ExtendedSystemDesign* of the Tank System in Figure 6.32, a specific instance of the *LiquidSource* component is annotated through a new *Probability Model* by overriding the default values of probabilities (i.e. *Success*=0.50 and *Failure*=0.50) with new values of probabilities (i.e. *Success*=0.80 and *Failure*=0.20). In this case, these new values of probabilities are valid only for such specific instance of the *LiquidSource* component model.

```
model ExtendedSystemDesign
//PhysicalComponentModels
    Tank tank1
    Sink sink1;
    LevelController levelController;

    probability_annotation(schema="Probab
    ilityModels.LiquidSourceProbabilityMod
    el", success=0.80, failure=0.20);
    LiquidSource source;
//RequirementModels
    …
    …
end ExtendedSystemDesign;
```

**Fig. 6.32.** An instance of PhysicalComponentModel annotated with a Probability Model

### 6.6.2 Generation of the Fault Tree Diagram

Finally the *Fault Tree Diagram (FTD)* of the Modelica-based design augmented with the *Probability Model* is generated by using the OpenModelica simulation environment [87].

In Figure 6.33 the user interface of OMEdit, the Open Source graphical editor of OpenModelica, which has been extended for supporting the generation of FTDs in an XML format, is shown.

Specifically, OMEdit has been extended by introducing a minimum set of features supported by *APIs* and some GUI buttons which allow to annotate components with the proposed *Probability Model* and then to export the Modelica design in terms of *FTD*.

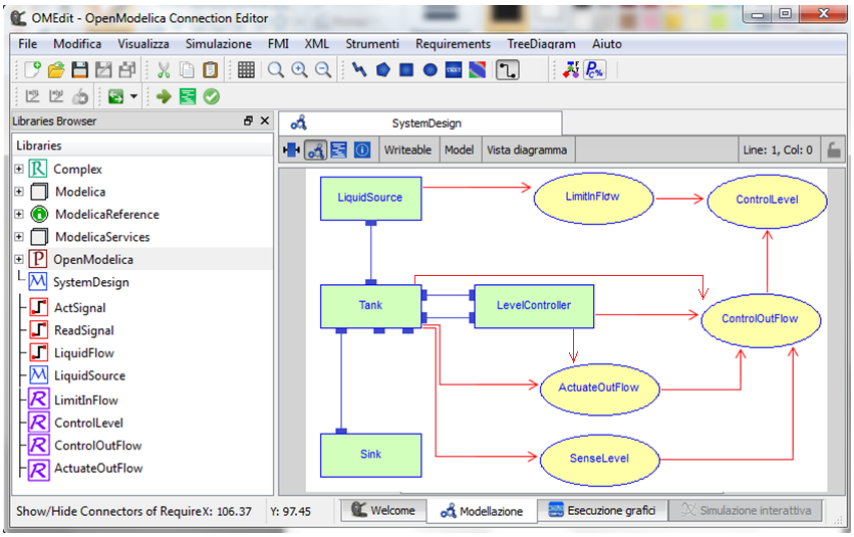



**Fig. 6.33.** An extension of OMEdit, for supporting modeling and generation of Fault Tree Diagrams

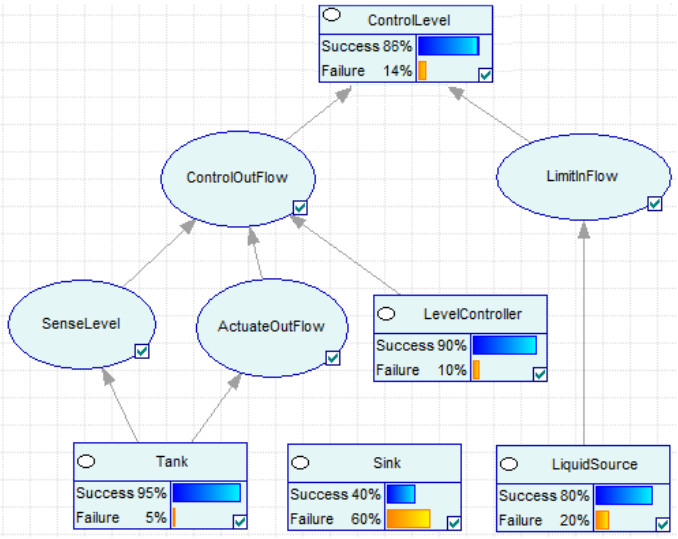



**Fig. 6.34.** Graphical representation of the ExtendedSystemDesign of the Tank System by using the GeNIe analysis environment

Above, a fragment of the generated XML code, representing the *Fault Tree Diagram* of the Modelica design, is reported. Then starting from the *FTD* generated in the previous step (see Figure 6.35), its graphical representation

for the Fault Tree Analysis in the GeNIe environment is obtained (see Figure 6.34). This generation is completely automatic and compliant with the GeNIe [37] tool format; the so generated *FDT* is ready to be simulated without need for further modifications.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<smile version="1.0" id="Unnamed" numsamples="1000" discsamples="10000">
<nodes>
		<cpt id="LevelController">
			<state id="Success"/>
			<state id="Failure"/>
			<probabilities>0.9 0.09999999999999998</probabilities>
		</cpt>
		<cpt id="Tank">
			<state id="Success"/>
			<state id="Failure"/>
			<probabilities>0.95 0.05000000000000004</probabilities>
		</cpt>
		<cpt id="Sink">
			<state id="Success"/>
			<state id="Failure"/>
			<probabilities>0.4 0.6</probabilities>
		</cpt>
		<cpt id="ActuateOutFlow">
			<state id="Success />
			<state id="Failure"/>
			<parents>Tank</parents>
			<probabilities>0.5 0.5 0.5 0.5</probabilities>
		</cpt>
		<cpt id="SenseLevel">
			<state id="Success"/>
			<state id="Failure"/>
			<parents>Tank</parents>
			<probabilities>0.5 0.5 0.5 0.5</probabilities>
		</cpt>
		<cpt id="LiquidSource">
			<state id="Success"/>
			<state id="Failure"/>
			<probabilities>0.8 0.2</probabilities>
		</cpt>
		<cpt id="LimitInFlow">
			<state id="Success"/>
			<state id="Failure"/>
			<parents>LiquidSource</parents>
			<probabilities>0.5 0.5 0.5 0.5</probabilities>
		</cpt>
		<cpt id="ControlOutFlow">
			<state id="Success"/>
			<state id="Failure"/>
			<parents>SenseLevel ActuateOutFlow LevelController</parents>
			<probabilities>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
0.5 0.5</probabilities>
		</cpt>
		<cpt id="ControlLevel">
			<state id="Success"/>
			<state id="Failure"/>
			<parents>ControlOutFlow LimitInFlow</parents>
			<probabilities>0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5</probabilities>
		</cpt>
	</nodes>
	<extensions>…</extensions>
</smile>
```

**Fig. 6.35.** A fragment of the generated XML code representing the Fault Tree Diagram

It is worth noting that in order to make more readable the picture represented in Figure 6.34 both physical components (i.e. the *Tank*, the *Sink*, the

*LiquidSource* and the *LevelController*) and the top *requirement* to be verified (i.e. the *ControlLevel*) are depicted by using a square shape, whereas all the other intermediate requirements are depicted by using a round shape.

Starting from the probability values associated to the Tank System model, the *Fault Tree Analysis* of the system with the top level represented by the requirement *ControlLevel* shows that the probability of *Failure* is equal to 14%, whereas the probability of *Success* is equal to 86%; these values are obtained by considering the propagation through the system of the initial probability values of Failure and Success associated to the PhysicalComponentModels of the Tank System.

## 6.7 Conclusion

This Chapter has described further contributions focused on the modeling of requirements in an equation-based context. In particular, a reference *meta-model* for representing *System Requirements* in terms of *RequirementAssertions* has been presented along with three different *approaches* for the modeling of System Requirements that adhere to the proposed meta-model. All of them aim to provide support for model verification by defining extensions of the Modelica language, and, one of them also aims to extend such model verification by supporting the modeling of system failures and thus allowing to analyze the behavior of the system in presence of faults, according to a proposed pseudo-algorithm for tracing requirements.

Then, a *model-driven process* for performing *Fault Tree Analysis* of a Modelica-based System Design by modeling probabilities through a *Probability Model* has been presented. In particular the methodological process aims to support the reliability analysis of systems by combining the benefits of graphical editors for the system design and *annotations*, such as OMEdit, with dedicated analysis tools, such as GeNIe. The possibility to combine through an annotation mechanism the system design model with a *Probability Model* allows to support the analysis of important system properties and thus to reduce incoherences and simplify model modification and reuse. Specific annotations have been introduced to integrate a Modelica-based system design with the proposed *Probability Model*; then, dedicated *OpenModelica APIs* have been defined and implemented for generating *Fault Tree Diagrams*. Such contributions have been experimented by considering a reference example consisting of a Tank System as well as by prototyping a specific OpenModelica version in order to experiment the above mentioned extensions.

# 7

# Conclusions

## 7.1 Main contribution

The research presented in this Thesis aimed at contributing to fill the lack of methods which specifically address the analysis and verification of non-functional requirements. The research has been focused on the Reliability and Safety properties of systems using Model-Based System Design (MBSD) approaches and Simulation techniques to support their representation and analysis. In particular, the benefits deriving from the adoption of a MBSD approach have concerned the improvement of the system development process with a significant reduction of both time and development costs as well as models integration, thanks, also, to its modularly approach that enabling (re)usability of models; whilst the strength of Simulation techniques have been employed for making more effective the verification process of the system in order to validate its design against requirements, as well as for studying its evolution. In this context, two main contributes have been provided:

- RAMSAS, a model-based method for system reliability analysis through simulation;
- a Modelica-based method for supporting the safety analysis of cyber-physical systems.

RAMSAS allowed leveraging the strengths both of powerful visual languages, suitable to flexibly model the architectural and behavioral aspects of complex, dynamics, and heterogeneous systems, and of mature and popular tools, suitable for the simulation and analysis of multi-domain systems. Indeed, it combines in a unified framework the benefits of popular OMG modeling languages (UML, SysML) with the widely adopted Mathworks simulation and analysis environment (MATLAB-Simulink), so as to keep lower the learning curve of the method. RAMSAS is the result of an intensive experimentation in several application domains (avionics, automotive, satellite) which allowed improving its modeling and simulation features especially in

the support provided to the modeling of the dysfunctional system behaviors, where the focus is on the modeling of faults (a defect in a component) and failures (an observable deviation from the intended behavior at the system boundary). In particular, six templates of dysfunctional tasks have been identified (Fault Generation, Failure Generation, Fault Management, Failure Management, Fault Propagation and Failure Propagation) as well as five fault/failure types for support this crucial modeling activity; then a set of patterns have been introduced as a combination of dysfunctional task type and fault/failure type. Finally, RAMSAS is defined as a Method Fragment and, as a consequence, it can be integrated in various phases of a typical System Development Process (e.g. in a V-Cycle process): (i) in the verification phases to support the verification of system reliability after the actual realization of the system; (ii) in the design phases to support the evaluation and validation of configuration scenarios and settings of system parameters so to guide and suggest design choices before the realization of the system.

The second contribution has concerned the definition of a Modelica-based method, inspired by the ISO-26262 standard, for the development, the analysis and the verification of systems with hard safety requirements. The method is mainly centered on the Modelica language by adopting an equation based style with acausal features, so as to allow defining hybrid physical systems models (e.g. systems containing mechanical, electrical, electronic, hydraulic, thermal, control, electric power components) in a declarative manner; furthermore the strength of well-known simulation techniques is employed. In particular, the Method exploits, as an integrated chain of tools, both the ModelicaML profile during the Requirement Analysis and the System Modeling phases for representing systems and then, in the Virtual Testing phase, the OpenModelica environment as the simulation platform for the execution of the system model generated in the previous phases. The exploitation of a common language/formalism in the different phases of the method allows to reduce the need for transformations between different formalisms and reducing, consequently, the costs of maintenance and modification of the simulation code, benefiting, additionally, of the capabilities offered by the simulation environment based on the Modelica language. Moreover, the object-oriented features of the approach promote the modular design and the hierarchical organization of the models. Some extensions of the Modelica language, that allow to represent the system requirements according to the classic design model adopted by Modelica , have been proposed. In particular, a new type of class (*requirement* class) has been introduced, to support not only the definition of a system requirements, but also to validate the behavior of one or more components associated to a requirement. Furthermore, for expressing the relationships between system components and requirements as well as among requirements, another new construct (*fulfill* equation) has been introduced. An algorithm, which specifies the semantic of fulfillment of requirements and their relationships, has been defined as well as a mechanisms which allows their assessment propagation and traceability has been formalized. The effectiveness of this

process has been evaluated through a case study in the automotive domain concerning the safety analysis of an Airbag System.

The experience gained during the definition of the above mentioned methods allowed to face the more general issue concerning the Modeling of System Requirements. In this context, a meta-model for modeling requirements of cyber-physical systems as well as different approaches for integrating the modeling of system requirements according to the Modelica model have been defined. These approaches have the goal of providing support for the allocation, traceability and verification of requirements through appropriate extensions of the Modelica language; in addition, the proposed Modelica-based method for safety analysis has been enhanced for supporting the Fault Tree Analysis of a system design based on the Modelica language. In particular, it has been enriched through a Probability Model, which has been integrated into the design of the Physical System Model by using standard Modelica annotations, for representing the probability of Success or Failure of components. By exploiting specific Modelica APIs, the obtained Physical System Model can be exported in several formats so to enable its Fault Tree Analysis through different external tools. This approach allows to (re)use design models without additional efforts, in terms of time and development costs, to be spent in (re)defining models of physical systems, already represented in Modelica language.

## 7.2 Ongoing and Future Work

The results presented in this Thesis constitute a starting point for ongoing and future research activities.

One of these concerns how to extend RAMSAS for supporting the reliability analysis of Systems of Systems (SoS) [17]. The current version of RAMSAS, has been conceived for large-scale systems, i.e. systems which are constituted by a multitude of components organized so to form a whole with clearly defined boundaries. Examples of this kind of systems are military and commercial aircraft, spacecraft, satellites, power plant automobiles, etc. The reliability analysis of these systems is challenging task which, as proved by RAMSAS, could benefit both from model-based approaches and from simulation. However, although the structure of these large-scale systems is rather complex (or better complicated) it remains quite the same during the system life cycle; moreover, a great part of the system components manifest a reactive behavior and pro-activeness is limited to a narrow subset of components. Moving from large-scale system to the System of Systems (SoS) context, these assumptions typically do not hold [17, 73]. Indeed, a SoS (e.g. a Coast Guard Integrated Deep-water System or an Air Traffic Management System) is constituted by a set of interconnected, and often geographically distributed, systems which interact for achieving common goals and are capable of autonomous and independent behaviors; moreover, the set of involved systems typically changes

during the SoS life as new systems join the SoS and others dynamically leave it. In this context, some aspects are currently under investigation. First of all the definition of Reliability needs to be clearly defined. Indeed, in case of a system (even complicated such an aircraft) is normally clear when the system fails to perform its mission and thus the system failure modes and consequent effects could be clearly identified (e.g. by an FMECA analysis); whereas, in case of a SoS, the concept of failure is not so easily identifiable. Then the modeling of the SoS structure, which is fundamental for performing its reliability analysis, has to be identified. Indeed, differently to what typically happens for a system (such as for an aircraft), the structure of a SoS changes dynamically (new entities could take part into the SoS as well as others could leave it) so as to its configuration; moreover, the boundaries of a SoS are often not well identifiable as it is not clear when to consider an entities as part of the SoS and when belonging to the environment in which the SoS is situated [109]. How to represent the organizational structure of a SoS and of the norms, rules and protocols that govern the SoS operation and its evolution should be addressed. Indeed, SoSs are typically characterized by distributed organizational structures (organization charts, command chains, etc.) which envisage roles, permissions, responsibilities, rights, and so on [18, 90]. Moreover the modeling of dynamic and goal-driven behaviors of the entities, which are typically characterized by autonomous and goal-oriented behaviors, as well as how to represent both indented and dysfunctional behavior of the SoS, which could require distributed approach, need a deeper research activity. On the light of the above open discussion upon the reliability analysis of SoSs, if and how to extend the RAMSAS method is under investigation [35]

Another future work includes the improvements of the proposed Modelica-based method and its extensions towards a version RAMSAS-like, by introducing some approaches and possible patterns for representing dysfunctional behavior and fault injection to be integrated into the development process along with software tools able to fully support the entire development and verification process through simulation, in order to analyze the system in presence of faults and/or failures. It should open RAMSAS to the adoption of different free and commercial tools which strongly rely on the Modelica language. This could improve the tool support for the different phases of the method as well as a more seamless and automatic transition from the conceptual to the simulation model of the system. As a starting point, a full version of RAMSAS, called *RAMSAS4Modelica*, is currently under definition [36].

# References


1. H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner-fischer, and S. Leue. "Safety Analysis of an Airbag System using Probabilistic FMEA and Probabilistic Counterexamples". Proceedings of the 6th International Conference on the Quantitative Evaluation of Systems (QEST), Eger, Hungary, September 17-18, 2009.
2. A. Arnold, A. Griffault, G. Point, and A. Rauzy. "The AltaRica formalism for describing concurrent systems". Fundamenta Infomaticae, vol. 40, pp. 109-124, 2000.
3. Altarica language. Available: http://altarica.labri.fr/wp/
4. F. Bidner. "Fault Tree Analysis of the HERMES CubeSat". University of Colorado at Boulder, USA, 2010.
5. M. Booya, S. Arghami, H. Asilian, and S. Mortazavi. "Safety analysis of a corn processing industry by energy trace and barrier analysis method: a case study". Iran Occupational Health Journal, vol. 4 (3), 27-34, 2007.
6. M. Bouissou, H. Bouhadana, and M. Bannelier. "Un moyen d'unifier diverses modélisation pour les études probabilistes: le langage FIGARO". HT-53/90-42A, EDF report, 1990.
7. A. Villemeur, M. Bouissou, and A. Dubreuil-Chambardel. "Accident sequences: methods to compute probabilities". Proceedings of the International topical conference on PSA and Risk Management, Zurich, Switzerland, 1987.
8. M. Bouissou, H. Bouhadana, M.Bannelier, and N. Villatte. "Knowledge modelling and reliability processing: presentation of the FIGARO language and associated tools". Proceedings of the International Conference on Computer Safety, Reliability and Security (SafeComp), Trondheim, Norway, November 1991.
9. S. Brinkkemper. "Method Engineering: engineering of information systems development methods and tools". Information and Software Technology, vol. 38, n. 4, pp. 275-280, Elsevier B. V., Amsterdam, The Netherlands 1996.
10. S. Brinkkemper, K. Lyytinen and R. Welke. "Method engineering: principles of method construction and tool support". Springer-Verlag, Berlin Heidelberg, Germany, 1996.
11. J. F. Castet and J. H. Saleh. "Beyond reliability multistate failure analysis of satellite subsystems A statistical approach". Reliability Engineering and System Safety, pp. 311-322, 95, Elsevier Ltd., 2010.

12. J. F. Castet, and J. H. Saleh. "Satellite and satellite subsystems reliability: Statistical data analysis and modeling". Reliability Engineering and System Safety, pp. 1718-1728, 94, Elsevier Ltd., 2009.
13. T. Clark, P. Sammut, and J. Willans. "Applied metamodelling: a foundation for language driven development". Second Edition, 2008.
14. E. Clifton. "Fault tree analysis - a history". Proceedings of the 17th International Systems Safety Conference, pp. 1-9, Orlando (Florida, USA), August 16-21, 1999.
15. K. Coste. "Qualification Test Methods for Satellite ACS Thrusters". Jet Propulsion Laboratory, National Aeronautics and Space Administration, Pasadena, USA, 2004.
16. R. Cressent, V. Idasiak, F. Kratz, and P. David. "Mastering safety and reliability in a model based process". Proceedings of Reliability and Maintainability Symposium (RAMS), Lake Buena Vista (Florida, USA), January 24-27, 2011,
17. W. A. Crossley. "System of Systems: An Introduction of Purdue University Schools of Engineering's Signature Area". Proceedings of the Engineering Systems Symposium, Cambridge, MA, USA, March 2004.
18. H. R. Darabi, A. Gorod, and M. Mansouri. "Governance Mechanism Pillars for Systems of Systems". Proceedings of the 7th International Conference on System of Systems Engineering (SoSE), Genoa, Italy, July 2012.
19. B. Dodson, and D. Nolan. Practical Reliability Engineering, John Wiley & Sons Ltd, England, 2001.
20. EADS (European Aeronautic Defence and Space Company). Available: http://www.eads.com/
21. ESA (European Space Agency). Available: http://www.esa.int/
22. ECSS-Q80-03. Space product assurance: Methods and techniques to support the assessment of software dependability and safety. ESA Publications Division, 2006
23. FAA. System Safety Handbook, 2000.
24. FIDES Group. Reliability Methodology for electronic systems - FIDES Guide. May 2009.
25. P. Fritzson. Principles of Object-Oriented Modeling and Simulation with Modelica 2.1, Wiley IEEE Press, 944 pages, February 2004.
26. P. Fritzson. "Integrated UML-Modelica Model-Based Product Development for Embedded Systems in OPENPROD". Proceedings of the 1st Workshop on Hands-on Platforms and tools for model-based engineering of Embedded Systems (Hopes), Paris, June 15, 2010.
27. A. Garro, J. Groß, M. Riestenpatt Gen. Richter, and A. Tundis. "Reliability Analysis of an Attitude Determination and Control System (ADCS) through the RAMSAS method". Journal of Computational Science (Elsevier), 2013.
28. A. Garro, A. Tundis, J. Groß, and M. Riestenpatt Gen. Richter. "Experimenting the RAMSAS method in the reliability analysis of an Attitude Determination and Control System (ADCS)". Proceedings of the International Workshop on Applied Modeling and Simulation (WAMS), jointly held with the NATO CAX FORUM, pp. 24-27, Rome, Italy, September, 2012.
29. A. Garro, and A. Tundis. "Modeling and Simulation for System Reliability Analysis: The RAMSAS Method". Proceedings of the IEEE SoSE 2012 7th International Conference on System of Systems Engineering (SoSE), pp. 155-160, Genoa, Italy, July 16-19, 2012.

30. A. Garro, and A. Tundis. "Enhancing the RAMSAS method for System Reliability Analysis: an exploitation in the Automotive Domain". Proceedings of International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH), pp. 328-333, Rome, Italy, 28-31 July, 2012.
31. A. Garro, and A. Tundis. "A Model-Based method for System Reliability Analysis". Proceedings of the Symposium On Theory of Modeling and Simulation (TMS'12), part of the SCS SpringSim 2012 conference, Orlando, FL, USA, March 26-29, 2012.
32. A. Garro, and A. Tundis. "Enhancing the RAMSAS method for Systems Reliability Analysis through Modelica". Proceedings of the 7th MODPROD Workshop on Model-Based Product Development, Linköping University, Sweden, February 5-6, 2013.
33. A. Garro, and A. Tundis, L. Rogovchenko-Buffoni and P. Fritzson. "From Safety Requirements to Simulation-driven Design of Safe Systems". Proceedings of the 12th International Conference on Modeling and Applied Simulation (MAS), Athens, Greece, September 25-27, 2013.
34. A. Garro, A. Tundis, and N. Chirillo. "System reliability analysis: a Model-Based approach and a case study in the avionics industry". Proceedings of the 3rd Air and Space International Conference (CEAS), pp. 1587-1595, Venice, Italy, October 24-28, 2011.
35. A. Garro, and A. Tundis. "On the Reliability Analysis of Systems and Systems of Systems: the RAMSAS method and related extensions". Submitted to IEEE Systems Journal.
36. A. Garro, and A. Tundis. "RAMSAS4Modelica: a Simulation-driven Method for System Dependability Analysis centered on the Modelica language and related tools". Submitted to the Symposium On Theory of Modeling and Simulation (TMS), part of the SCS SpringSim 2014 conference, Tampa, FL, USA, April 13-16, 2014.
37. GeNIe (Graphical Network Interface). Available: http://genie.sis.pitt.edu/
38. J. Groß, and R. Rudolph. "Generating Simulation Models from UML A FireSat Example". Proceedings of the Symposium on Theory of Modeling and Simulation (TMS), Orlando, Florida, USA, March 26-29, 2012.
39. J. Groß, and R. Rudolph. "Dependency Analysis in Complex System Design using the FireSat Example". Proceedings of the INCOSE Symposium, Rome, Italy, 2012.
40. J. Groß, and R. Rudolph. "Hierarchie von Entwurfsentscheidungen im modellbasierten Entwurf komplexer Systeme". Tag des Systems Engineering der GfSE, Hamburg, Germany, 2011.
41. L. Grunske, and B. Kaiser. "Automatic Generation of Analyzable Failure Propagation Models from Component-Level Failure Annotations". Proceedings of the 5th International Conference on Quality Software (QSIC), Melbourne, Australia, September 2005.
42. R. Guillerm, H. Demmou, and N. Sadou, "Engineering dependability requirements for complex systems - A new information model definition". Proceedings of the 4th Annual IEEE Systems Conference, 2010.
43. L. Gullo, and D. Raheja. "Design Failure Modes, Effects, and Criticality Analysis". Book Chapter of Design for Reliability, Wiley-IEEE Press, pp. 67-86, 2012.

44. D. M. Harland and R.D. Lorenz. "Space Systems Failures, Disasters and Rescues of Satellites, Rockets and Space Probes", Springer Berlin, Germany, 2005.
45. E. Harold Roland, and Brian Moriarty. System Safety Engineering and Management. John Wiley & Sons, 1990.
46. Henderson-Sellers. "Method engineering for OO systems development". Communications of the ACM, 46, 2003.
47. T. Herpel, and R. German. "A simulation approach for the design of safety-relevant automotive multi-ECU systems". Proceedings of the 4th IEEE International Conference on System of Systems Engineering (IEEE SoSE), Albuquerque, New Mexico, USA), May 30 - June 03, 2009.
48. H. P. Hoffmann. "System Engineering Best Practices with Rational Solution for Systems and Software Engineering", February 2011. Available: http://www.ibm.com
49. N. Holsti, and M. Paakko. "Towards Advanced FDIR Components". Data Systems in Aerospace, DASIA, Nice, France, May 2001.
50. Hubble Space Telescope Servicing Mission 4. NASA, Greenbelt, USA, 2008.
51. E. Hull, K. Jackson, and J. Dick. "Requirements engineering". Springer Verlag, 2005.
52. IBM Rational Rhapsody. Available: http://www-01.ibm.com/software/awdtools/rhapsody/
53. IEC-61508. Functional safety of electrical/electronic/programmable electronic safety-related systems, Parts 1-7, 2010.
54. IEC (International Electrotechnical Commission). Available: http://www.iec.ch/
55. INCOSE - International Council on Systems Engineering. Available: www.incose.org
56. INCOSE - Systems Engineering Handbook (INCOSE-TP-2003-002-03 v. 3), "A Guide For System Life Cycle Processes and Activities", June 2006.
57. INCOSE - Systems Engineering Vision 2020 (INCOSE-TP-2004-004-02 Version 2.03), 2007.
58. INCOSE - Requirements Management Tools Survey. Retrieved April 19, 2013. Available: www.incose.org/productspubs/products/rmsurvey.aspx.
59. ISO-26262. Software Compliance: Achieving Functional Safety in the Automotive Industry, 2011.
60. ITEA 2 - MODRIO project. Available: http://www.itea2.org/
61. F. Iwu, A. Galloway, J. McDermid and I. Toyn. "Integrating safety and formal analyses using UML and PFS". Reliability Engineering and System Safety, pp. 156-170, 92, 2007.
62. A. Jardin, D. Bouskela, T. Nguyen, N. Ruel, E. Thomas, R. Schoenig, S. Loembé, and L. Chastanet. "Modelling of System Properties in a Modelica Framework". Proceedings of the 8th International Modelica Conference, TU Dresden, March 20-22, 2011.
63. C. Jianguo, L. Maotin, Y. Zuobin, W. Zongshuai, and X. Fengjian. "Equipment systems reliability analysis based on FTA". Proceedings of the International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering (ICQR2MSE), pp. 293-296, Chengdu, June 2012.
64. D. Kececioglu. "Reliability Engineering Handbook", Prentice-Hall, Englewood Cliffs, New Jersey, USA, 1991.
65. R. Kenarangui. "Event-tree analysis by fuzzy probability". IEEE Transaction on Reliability, 40 (1), 120-124, 1991.

66. J. Krause, E. Hintze, S. Magnus and C. Diedrich. "Model based specification, verification and test generation for a safety fieldbus profile". Proceedings of the 31st International Conference on Computer Safety, Reliability and Security (SafeComp), Magdeburg, Germany, September 25, 2012.
67. J. Lahtinen, M. Johansson, J. Ranta, H. Harju, and R. Nevalainen. "Comparison between IEC 60880 and IEC 61508 for certification purposes in the nuclear domain". Proceedings of the 29th International Conference on Computer Safety, Reliability and Security (SafeComp), Vienna (Austria), September 14-17, 2010.
68. J. C. Laprie. Dependability: Basic Concepts and Terminology, Springer-Verlag, 1992.
69. W. J. Larson, and J. R. Wertz. "Space mission analysis and design". 3rd ed., Microcosm Press, El Segundo, California, USA, 1999,
70. F. Liang, W. Schamai, O. Rogovchenko, S. Sadeghi, M. Nyberg, and P. Fritzson. "Model-based Requirement Verification: A Case Study". Proceedings of the 9th International Modelica Conference (Modelica'2012), Munich (Germany), September 3-5, 2012.
71. Y. Liu and W. Wu. "Research on the system of reliability block diagram design and reliability prediction". Proceedings of the International Conference on System Science, Engineering Design and Manufacturing Informatization (ICSEM), pp. 35-38, Guiyang, October 2011.
72. X. Liudong and W. Wendai. "Probabilistic common-cause failures analysis". Proceedings of Annual Reliability and Maintainability Symposium, pp. 354-358, Las Vegas, NV, USA, January 28-31,2008.
73. M. W. Maier. "Architecting Principles for Systems of Systems". Systems Engineering, pp. 267-284, vol 1, No. 4, 1999.
74. Mathworks MATLAB-Simulink. Available: http://www.mathworks.com/
75. X. Min, P. Kim-Leng, and D. Yuan-Shun. "Computing System Reliability: Models and Analysis", 2004.
76. Modelica and the Modelica Association. Available: https://www.modelica.org/
77. ModelicaML (OpenModelica Project) - A UML Profile for Modelica. Available: www.openmodelica.org/modelicaml
78. A. Molesini, A. Omicini, A. Ricci and E. Denti. "Zooming multi-agent systems". Proceedings of the 6th International Workshop on Agent-Oriented Software Engineering, Utrecht, The Netherlands, July 2005.
79. NASA (The National Aeronautics and Space Administration). Available: http://www.nasa.gov/
80. NASA. Systems Engineering Handbook, Revision 1, NASA/SP-2007-6105, NASA.
81. NASA. Available: http://askmagazine.nasa.gov/pdf/pdf31/NASA/APPEL/ASK/31i/introduction/to/system/safety.pdf
82. NASA. Fault Tree Analysis Handbook for Aerospace Applications. Washington, USA, 2002.
83. V. K. Navinkumar and R. K. Archana. "Functional safety management aspects in testing of automotive safety concern systems (electronic braking system)". Proceedings of the 3rd International Conference on Electronics Computer Technology (ICECT), India, April 8-10, 2011.
84. G. Nicolescu and P.J. Mosterman. Model Based Design for Embedded Systems (Computational Analysis, Synthesis and Design of Dynamic Systems), CRC Press, 2009.

85. P. O'Connor. Reliability Engineering Handbook (Quality and Reliability), CRC Press, 1999.
86. X. Olive. "FDIR for satellite at Thales Alenia Space How to deal with high availability and robustness in Space domain". Proceedings of the Conference on Control and Fault Tolerant Systems, Nice, France, October 2010.
87. OpenModelica - Open Source Modelica Consortium (OSMC). Available: https://www.openmodelica.org/.
88. T. I. Oren, and L. Yilmaz. "Synergy of Systems Engineering and Modeling and Simulation". Proceedings of the International Conference on M&S Methodology, Tools, Software Applications (M&S MTSA), Calgary (Alberta, Canada) July 31 - August 2, 2006.
89. Programming Environment Laboratory (PELAB). Available: https://www.ida.liu.se/labs/pelab/
90. J. Pavon, C. Sansores and J. J. Gómez-Sanz. "Modelling and simulation of social systems with INGENIAS". International Journal of Agent-Oriented Software Engineering, pp. 196-221, vol 2, no 2, 2008.
91. M. Pellissetti, E. Capiez-Lernout, H. Pradlwarter, C. Soize, and G. I. Schuëller. "Reliability analysis of a satellite structure with a parametric and a non-parametric probabilistic model". Computer Methods in Applied Mechanics and Engineering, pp. 344-357, Elsevier Ltd., 2008.
92. M. Peraldi-Frati and A. Albinet. "Requirement traceability in safety critical systems". Proceedings of the 1st Workshop on Critical Automotive applications: Robustness & Safety (CARS), Valencia, Spain, April 27, 2010.
93. D. Pierre, V. Idasiak, and F. Kratz. "Reliability study of complex physical systems using SysML". Reliability Engineering and System Safety, vol. 95 pp. 431-150, 2010.
94. I. Pomeranz and S.M. Reddy. "Hazard-Based Detection Conditions for Improved Transition Fault Coverage of Functional Test Sequences". Proceedings of the 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 358-366, Chicago, IL, USA, October 7-9, 2009.
95. C. J. Price and N. Hughes. "Effective automated sneak circuit analysis". Proceedings of Annual Reliability and Maintainability Symposium, pp. 356-360, Seattle, WA, USA, January 28-31, 2002.
96. RTCA/DO 178B - Software Considerations in Airborne Systems and Equipment Certification, December 1, 1992.
97. RTCA/DO 178C - Software Considerations in Airborne Systems and Equipment Certification, December 2011.
98. RTCA/DO 254 - DESIGN ASSURANCE GUIDANCE FOR AIRBORNE ELECTRONIC HARDWARE, April 19, 2000.
99. A. Rauzy. "Mode automata and their compilation into fault trees". Reliability Engineering and System Safety, vol. 78, pp. 1-12, 2002.
100. E. Rechtin, and W. Maier. "The Art of Systems Architecting", Second Edition, CRC Press LLC, 2000.
101. L. Rierson. Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance, CRC Press, 2013
102. J. L. Rouvroye and E. G. Van den Bliek. "Comparing safety analysis techniques". Journal of Reliability Engineering & System Safety, Elseiver, 75 (3), pp. 289-294, 2002.

103. D. Rubio, S. Ponce and F. Madrid. ISO/IEC 17025 technical requirements in electrical safety laboratory for electromedical devices. Proceedings of Pan American Health Care Exchanges (PAHCE), Rio de Janeiro, Brazil, March 28 - April 1, 2011.
104. L. Rogovchenko-Buffoni, A. Tundis, M.Z. Hossain, M. Nyberg and P. Fritzson. "An Integrated Toolchain For Model Based Functional Safety Analysis". Journal of Computational Science (Elsevier), 2013.
105. L. Rogovchenko-Buffoni, P. Fritzson, A. Garro, A. Tundis and M. Nyberg. "Requirement Verification and Dependency Tracing During Simulation in Modelica". Proceedings of the 8th EUROSIM Congress on Modelling and Simulation (EUROSIM), Cardiff, Wales, United Kingdom, September 10-13, 2013.
106. SAE International. Airbags and Safety Test Methodology, Society of Automotive Engineers, 2003.
107. W. Schamai, P. Helle, P. Fritzson, and C. Paredis, "Virtual Verification of System Designs against System Requirements". Proceedings of the 3rd International Workshop on Model Based Architecting and Construction of Embedded Systems (ACES'2010), Oslo (Norway), October 2010.
108. D. C. Schmidt. Model Driven Engineering. IEEE Computer, vol. 39(2), pp.25-31, 2006.
109. C. E. Siemieniuch and M.A. Sinclair. "Socio-technical considerations for Enterprise System Interfaces in Systems of Systems". Proceedings of the 7th International Conference on System of Systems Engineering (SoSE), Genoa, Italy, July 2012.
110. A. Sindico, M. Di Natale and G. Panci. "Integrating SysML with Simulink using Open-Source model transformations". Proceedings of the 1st International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH), Noordwikerhout, The Netherlands, July 2011.
111. Society of Automotive Engineers. "SAE Architecture Analysis & Design Language". Specification V2, 2009.
112. I. Sommerville and P. Sawyer. "Requirements Engineering: A good practice guide". Wiley, 2003.
113. R. F. Stapelberg. Handbook of Reliability, Availability, Maintainability and Safety in Engineering Design, 1st ed. Spinger-Verlag, 2008.
114. D. E. Struble. Advances in Side Airbag Systems, Society of Automotive Engineers Inc., 2005.
115. N. Sultan. "Cheaper, Faster is Better only if Safer: Need for "FIRST": Failure Information Reliability Space Tool to retrieve & analyze orbital Failures". Proceedings of the 19th International Communications Satellite Systems Conference (AIAA), Toulouse, France, 2001.
116. Systems Modeling Language (SysML). Available: http://www.omgsysml.org/
117. A. Tundis, L. Rogovchenko-Buffoni, P. Fritzson and A. Garro. "Modeling System Requirements in Modelica: Definition and Comparison of Candidate Approaches". Proceedings of 5th International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT), pp. 15-24, University of Nottingham, United Kingdom, April 19, 2013,
118. A. Tundis, L. Rogovchenko-Buffoni, A. Garro, M. Nyberg and P. Fritzson. "Performing Fault Tree Analysis of a Modelica-based System Design through a Probability Model". Proceedings of the International Workshop on Applied Modeling and Simulation (WAMS), Buenos Aires, Argentina, November 24-27, 2013.

119. H. Täubig, U. Frese, C.Hertzberg, C. Lüth, S.Mohr, E. Vorobev, and D. Walter. "Guaranteeing functional safety: design for provability and computer-aided verification". Journal Autonomous Robots, 32, Springer, pp. 303-331, 2012.
120. J. Wertz. "Spacecraft Attitude Determination and Control". D. Reidel Publishing Company, Dordrecht, Netherland, 1978.
121. G. Yu, Z. Xu, and J. Du. "An Approach for Automated Safety Testing of Safety-Critical Software System Based on Safety Requirements". Proceedings in the International Forum on Information Technology and Applications (IFITA), Chengdu, China, May 15-17, 2009.
122. M. Zamparelli. "Using a formal requirements management tool for System Engineering: First results at ESO". Proceedings of SPIE - The International Society for Optical Engineering, vol. 6271, 2006.