Centro de Informática U·F·P·E

Pós-Graduação em Ciência da Computação

Systematic Model-Based Safety Assessment via Probabilistic Model Checking

Por

Adriano José Oliveira Gomes

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, NOVEMBRO/2010

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ADRIANO JOSÉ OLIVEIRA GOMES

"SYSTEMATIC MODEL-BASED SAFETY ASSESSMENT VIA PROBABILISTIC MODEL CHECKING"

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR(A): ALEXANDRE CABRAL MOTA

RECIFE, NOVEMBRO/2010

Catalogação na fonte
Bibliotecária Jane Souto Maior, CRB4-571

Gomes, Adriano José Oliveira
Systematic model-based safety assessment via probabilistic model checking - Recife: O Autor, 2010.
88 folhas : il., fig., tab.

Orientador: Alexandre Cabral Mota.
Dissertação (mestrado) - Universidade Federal de Pernambuco. CIn, Ciência da Computação, 2010.

Inclui bibliografia e apêndice.

1. Ciência da Computação. 2. Engenharia de software. 3. Métodos formais. I. Mota, Alexandre Cabral (orientador). II. Título.

004 CDD (22. ed.) MEI2011 – 079

Dissertação de Mestrado apresentada por **Adriano José Oliveira Gomes** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**Systematic Model-Based Safety Assessment via Probabilistic Model Checking**", orientada pelo **Prof. Alexandre Cabral Mota** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Juliano Manabu Iyoda
Centro de Informática / UFPE

Prof. Enrique Andrés López Droguett
Departamento de Engenharia de Produção / UFPE

Prof. Alexandre Cabral Mota
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 30 de agosto de 2010.

Nelson Souto Rosa

Prof. Nelson Souto Rosa
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Acknowledgments

I would like to thank God, for His love and provision. He made me strong to keep me healthy and able to carry out this work. Also, to my fiancée Isis, my parents and my family, for their love and care. They give me support and encouragement in difficult moments that allowed me to conclude this work.

I am very grateful to my supervisors, Alexandre Mota and Augusto Sampaio, for their support, enthusiasm, patience and guidance during this research. They always give me support and incentive since my undergraduate course. I am very pleased to have them as teachers, supervisors, and friends.

Thanks also to all my friends, those that studied with me in the undergraduate course, and those that I meet in this graduate course, for the emotional support and the nice moments that we shared together. In particular, to Joabe and Farley. Both explain and support me on the “pyramid challenge”. Also, to my colleagues from the Comment Lab company, that make support and good moments. In particular, to my partner and dear friend Vitor, who “held the bar” of the company during the difficult times of this work.

I would also like to thank the people who gave me important contributions for the development of this work. From Embraer, Felipe Ferri, Julio Buzzi and Edson Watanabe for answering my several questions about the *Safety Assessment Process*, and for the interactions about the Simulink and FTA. From the RISCTEC Group, Marcio Moura, for answering my questions about system reliability, and for discussions about the FTA and probabilistic models.

Finally, I would also like to thank Embraer, for the *Safety Assessment Process* resources and the support for a technical visit; and the National Council of Technological and Scientific Development (CNPq), FACEPE and the Brazilian Space Agency, for financial support that made possible the development of this research.

Resumo

A análise da segurança (*Safety Assessment*) é um processo bem conhecido que serve para garantir que as restrições de segurança de um sistema crítico sejam cumpridas. Dentro dele, a análise de segurança quantitativa lida com essas restrições em um contexto numérico (probabilístico).

Os métodos de análise de segurança, como a tradicional *Fault Tree Analysis* (FTA), são utilizados no processo de avaliação da segurança quantitativo, seguindo as diretrizes de certificação (por exemplo, a ARP4761 – Guia de Práticas Recomendadas da Aviação). No entanto, este método é geralmente custoso e requer muito tempo e esforço para validar um sistema como um todo, uma vez que para uma aeronave chegam a ser construídas, em média, 10.000 árvores de falha e também porque dependem fortemente das habilidades humanas para lidar com suas limitações temporais que restringem o âmbito e o nível de detalhe que a análise e os resultados podem alcançar. Por outro lado, as autoridades certificadoras também permitem a utilização da análise de Markov, que, embora seus modelos sejam mais poderosos que as árvores de falha, a indústria raramente adota esta análise porque seus modelos são mais complexos e difíceis de lidar. Diante disto, FTA tem sido amplamente utilizada neste processo, principalmente porque é conceitualmente mais simples e fácil de entender.

À medida que a complexidade e o *time-to-market* dos sistemas aumentam, o interesse em abordar as questões de segurança durante as fases iniciais do projeto, ao invés de nas fases intermediárias/finais, tornou comum a adoção de projetos, ferramentas e técnicas baseados em modelos. Simulink é o exemplo padrão atualmente utilizado na indústria aeronáutica. Entretanto, mesmo neste cenário, as soluções atuais seguem o que os engenheiros já utilizavam anteriormente. Por outro lado, métodos formais que são linguagens, ferramentas e métodos baseados em lógica e matemática discreta e não seguem as abordagens da engenharia tradicional, podem proporcionar soluções inovadoras de baixo custo para engenheiros.

Esta dissertação define uma estratégia para a avaliação quantitativa de segurança baseada na análise de Markov. Porém, em vez de lidar com modelos de Markov diretamente, usamos a linguagem formal Prism (uma especificação em Prism é semanticamente interpretada como um modelo de Markov). Além disto, esta especificação em Prism é extraída de forma sistemática a partir de um modelo de alto nível (diagramas Simulink anotados com lógicas de falha do sistema), através da aplicação de regras de tradução. A verificação sob o aspecto quantitativo dos requisitos de segurança do sistema é realizada utilizando o verificador de modelos de Prism, no qual os requisitos de segurança tornam-se fórmulas probabilísticas em lógica temporal.

O objetivo imediato do nosso trabalho é evitar o esforço de se criar várias árvores de falhas até ser constatado que um requisito de segurança foi violado. Prism não constrói árvores de falha para chegar neste resultado. Ele simplesmente verifica de uma só vez se um requisito de segurança é satisfeito ou não no modelo inteiro.

Finalmente, nossa estratégia é ilustrada com um sistema simples (um projeto-piloto), mas representativo, projetado pela Embraer.

Palavras-chave: Análise Quantitativa de Segurança, Prism, Verificador de Modelos Probabilístico, Métodos Formais, Análise de Markov, Sistemas Aeronáuticos

Abstract

Safety assessment is a well-known process to assure that safety constraints on a critical system are met. It includes quantitative safety assessment that deals with safety constraints stated in numerical (probabilistic) terms.

Safety analysis methods, such as the traditional Fault-Tree Analysis (FTA), are used in the quantitative safety assessment process, following certification guidelines (For instance, the ARP4761- Aerospace Recommended Practice). However, this method is usually expensive and requires much time and effort to validate an entire system, since for an aircraft it can be constructed, on average, 10,000 fault-trees mainly because it strongly depends on human skills for dealing with time limitations that constrain the scope and level of detail that the analysis and results can reach. Certification authorities also allow the use of Markov analysis. Although Markov models are more powerful than fault-trees, industries rarely use this analysis because Markov models are more complex to be handled. Therefore, FTA has been widely used during this process mainly because it is conceptually more simple and easy to understand.

As complexity and time-to-market pressure increases, the interest in addressing safety issues during the early design phases, rather than during the intermediate/final, popularized the use of model-based design notation, techniques, and tools. Simulink is the current de facto standard in the aerospace industry. But, even in this scenario, current solutions follow what engineers were using previously. On the other hand, Formal Methods, which are languages, tools and methods based on logic and discrete mathematics and do not follow traditional engineering approaches, can provide innovative cost-effective solutions to engineers.

This dissertation defines a strategy for quantitative safety assessment based on Markov analysis. But instead of dealing with Markov models directly we use the Prism specification language (a Prism specification is semantically interpreted as a Markov model). Furthermore, a Prism specification is extracted systematically from a high-level model (Simulink diagram annotated with failure logic) via the application of translation rules. The verification of quantitative safety requirements is performed using the Prism model-checker, where safety requirements become probabilistic temporal logic formulas.

The immediate contribution of our work is a process that avoids the creation of several fault-trees until a safety requirement is violated. Prism does not build fault trees to reach this result. It just checks whether a safety requirement is satisfied or not in the entire model.

Finally, our strategy is illustrated with a simple (a pilot project) but representative system designed by Embraer.

Keywords: Quantitative Safety Assessment, Probabilistic Model-Checking, Formal Methods, Prism, Markov Analysis, Aircraft Systems

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Critical systems are increasingly being controlled by complex computer solutions. This imposes an even stronger requirement on reliability and safety. The occurrence of failures in these systems is almost unacceptable because failures can result in loss of human lives, financial losses or damage to the environment.

For instance, to allow the operation of an aircraft (civil or military), the authorities of this sector, as the FAA (Federal Aviation Administration) of the USA and ANAC (National Civil Aviation Agency) of Brazil, require stability in the control and enforcement functions of an aircraft [2, 4, 12]. The guarantee of stability depends on all systems and their subsystems and components and how they are related in the plane (see Fig. 1.1).

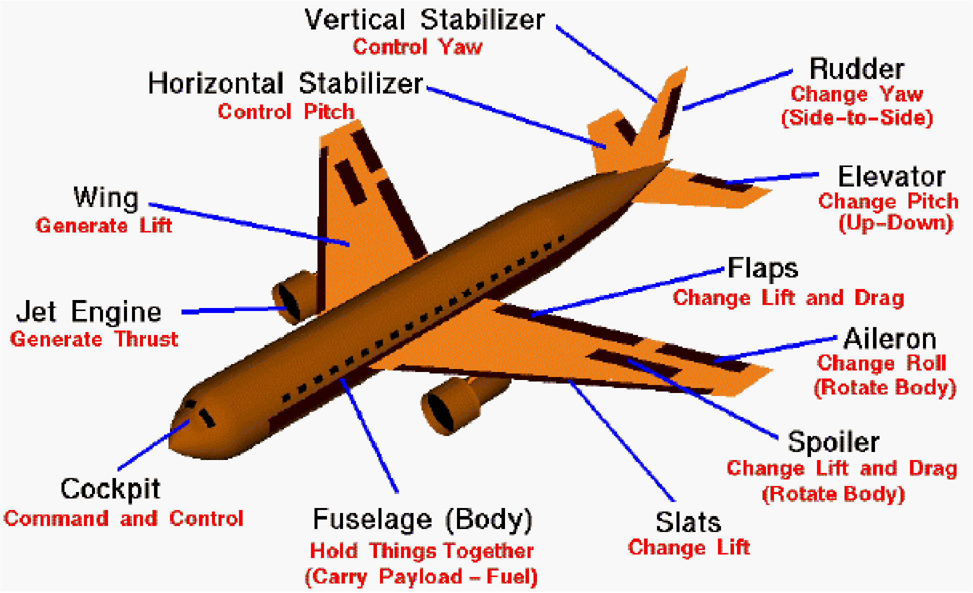


Fig 1.1 - Airplane parts and functions

During an aircraft development, a major challenge is designing a guaranteed system architecture that conceives the functional aspects to operate safely under the several hazard situations that can occur. Demonstrating that a solution (design) tackles such a challenge is mandatory for the certification authorities to approve the system.

Therefore, a safety assessment process is followed by the airborne industry in order to ensure the correct construction of a safe aircraft. This process is guided by rigorous norms and patterns such as DO 178B [12] and ARP 4761 (Aerospace recommend Practice) [2] that propose a well-established set of guidelines and methods for civil aircraft systems[1]. Although the safety assessment process determines a common framework for the aeronautics industry to handle the safety issues of aircraft systems, the fulfillment of this process involves long and arduous engineering tasks, given the complexity and magnitude of the projects involved. These tasks are based, in most cases, on engineers’ judgment and can present problems and limitations [4].

Furthermore, according to FAR 25.1309 (Federal Aviation Regulations) [24], which defines the requirements for certifying the systems and software loaded into an aircraft, there is a classification of the aircraft's functions and their systems with respect to the losses they can generate for the aircraft itself, its passengers and crew (from level A that means a catastrophic effect - most critical, to level E that means a no effect - least critical). A failure is the inability of a system to perform its required functions within specified performance requirements. The greater the criticality of a failure condition (hazard situation), the lower must be the probability of its occurrence (risk). This derives safety requirements that must be defined and satisfied under qualitative or quantitative analyses.

More specifically, the relation between the criticality of a failure and its probability of occurrence within the system exposition time defines the risk of an accident. Hazard is the potential to cause harm; risk on the other hand is the likelihood of harm (in defined circumstances, and usually qualified by some statement of the severity of the harm). Hence, the qualitative analysis refers to the characterization of the behavior of different faults (abnormal condition that may cause a reduction in, or loss of, the capability of a functional unit to perform a required function) that may result in a hazard, whereas the quantitative analysis refers to reliability predictions for system components that may cause or contribute to this hazard (based on a risk evaluation). The safety assessment process must take into account both analyses to assess the system architecture under all safety requirements that can be foreseen.

Concerning the quantitative aspect of the safety assessment process of aircraft systems, it is traditionally addressed using Fault Tree Analysis (FTA) [1]. This method is frequently used in industrial applications and it is also indicated by certification authorities. The main reason for its practical acceptance is that FTA is conceptually simple and easy to understand [2]. However, certification authorities also accept the use of Markov Analysis (MA) [16] to assure safety requirements of a system design.

Both FTA and Markov models use system failure logic information derived from well-known analysis techniques such as Failure Mode and Effect Analysis (FMEA) and Failure Hazard Analysis (FHA) presented in the ARP 4671 [2]. Based on this information, the analysis methods evaluate the probabilities of the undesired failure conditions to check whether a safety requirement is satisfied or not. Each technique executes this analysis using different mathematical representations; FTA uses static event-based trees and Markov analysis uses stochastic processes. Although Markov models are more powerful than fault-trees [2], they are more complex to be handled;

---

[1] MIL-STD-882D [3] covers the same purpose in the military domain.

thus, they are scarcely adopted in industry. Furthermore, in practice, they are created in a non-systematic ad hoc fashion [6, 23].

In recent decades, quantitative analysis has appeared as an arduous and expensive process, mainly due to the complexity and variety of systems involved as well as the several different situations of hazard under which they are evaluated. Even guided by well-defined standards and methods, the major representatives of the airborne industry such as Boeing, Embraer and Airbus still have several difficulties to handle this process efficiently [4, 6, 17]. Given this scenario, investment in tools and methods to support the development of safe systems on the established time and budget is still a big challenge [6, 13]. In particular, time constraints on the operation of these systems, that determine their correct operation, require the use of methods and tools that guide the entire process of design and validation. Such methods need to be rigorous, systematic and have a practical focus on validation, verification and quality assurance. Moreover, the quality of this product should act as a competitive advantage without affecting demand and time to market.

Currently, promising initiatives are directing towards proposals of advanced model-based design techniques to support the development process mainly at initial stages of the development during which engineers have more flexibility to evaluate different solutions and to propose improvements. The advent of high-level tools like Matlab/Simulink [9], SCADE [10] and Statemate [58] makes possible to model large complex systems using hierarchical diagrams, while preserving structural and functional aspects of the intended design. In the safety analysis view, these approaches have been extended to include the failure logic of a system inside its own diagrammatic model. On the qualitative analysis side, parameterized verification algorithms have been developed to identify failure causes and consequently constructing the corresponding fault trees automatically [5, 14, 40]. However, as described previously, qualitative analysis methods alone are not sufficient for addressing completely all safety aspects. The safety assessment process also demands probability constraints to be addressed by quantitative analysis.

Despite several automatic model-based approaches being proposed for FTA [11, 13, 14, 15] using a high-level tool like Simulink or SCADE, their treatment of quantitative parameters still depend of some human intervention. This can introduce errors in the analysis. Moreover, they are not cost-effective, because the probability of each failure condition (top event) must be evaluated singly (just one failure condition at a time), requiring more effort to undertake the analysis of the whole system. Thus, for each of the thousands of possible failure conditions of an aircraft, a fault-tree must be constructed and analyzed qualitatively and quantitatively to ensure that the probability of occurrence of such failure conditions is in accordance with the safety requirements. Considering that about 10,000 fault-trees are built during the safety assessment process of an aircraft, containing hundreds of basic events and whose depth we can reach many of the levels (see Fig. 1.2), the analysis of these fault-trees can be very stressful. This has a direct influence on development time of an aircraft that can reach 5 years and involve about 300 to 400 engineers [4, 17]. Furthermore, integrating quantitative analysis into a model-based solution, considering probabilistic models such as Markov chains in a semantically sound manner, is still a challenge. Nevertheless, a link to effective probabilistic verification tools has not been established so far.

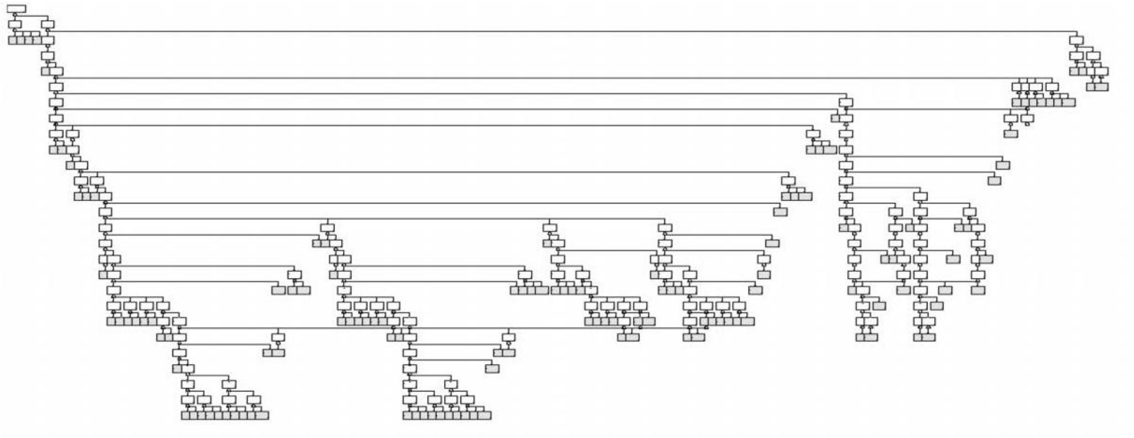


Fig 1.2 - Distant view of a fault tree for one failure condition

On the other hand, formal methods comprise mathematically founded methods designed to describe the properties of a system (requirements) in a precise and non-ambiguous way as well as allows one to assure that corresponding implementations satisfy such properties. Formal methods are equipped with a formal specification language that has a well-defined semantics. Formal specifications can be analyzed by model checkers or theorem provers to demonstrate that system design models meet the requirements. Consequently, this can greatly increase the confidence in the safety and correctness of the corresponding system. Examples of formal languages are CSP [59], Prism [7], Z [60] and Probmela [57]. Given the clearly importance of validating and verifying the safety requirements of the system throughout the safety assessment process, probabilistic models should be created using formal specifications in order to offer reliability predictions for the system that can properly support the process of quantitative analysis [6, 7, 13].

Despite the best practices provided by formal methods and their current conquered maturity, they are not successfully integrated into many development processes. The principal issues are related to a typical aversion on the part of the development teams in dealing with formal notations because such notations are not usual to them. Moreover, formal verification tools typically have their specific variants of the original language, which implies a bigger learning curve.

1.1 Context and Objectives

Our research has been developed in the context of a cooperation with Embraer (Empresa Brasileira de Aeronáutica S.A.). A central motivation is the fact that safety assessment can be improved by more systematic solutions instead of following just checklists as well as informal guidelines. Embraer has used the standards of its sector, briefly discussed here, as a means of fulfilling requirements for system development and certification. Particularly, Embraer heavily uses model-based solutions, through Simulink based tools provided by plug-ins.

This work is an initial effort to integrate quantitative safety assessment into a model-based solution considering formal models equipped with stochastic events. Such languages have specialized probabilistic verification tools that allows us to check the safety requirements of a system.

Therefore, this work is part of a larger project presented in [43] that defines a methodology that integrates a functional (qualitative) analysis [42] with a non-functional (quantitative) analysis over the system design with the support of formal methods. In this dissertation we detail the non-functional analysis strategy [65], with focus on the systematic model generation and analysis from Simulink diagrams.

We propose a strategy for quantitative safety assessment based on the Prism language [7, 8, 53]. By using this language, one can deal with Markov models indirectly and using a high-level and modular specification language (Prism). Additionally, Prism provides a probabilistic model checker which allows us to check probabilistic temporal logic formulas. And checking such formulas mean performing a quantitative analysis on the underlying Markov model in a high-level and versatile way, obtaining a lot of different analysis (compared to the traditional ones employed by engineering) easily. This is one of the most powerful advantages of Prism.

The additional effort and cost to create formal models in a traditional safety assessment process has been a significant barrier. Manually creating models aiming at formal analysis is labor intensive, because this requires significant skills of formal methods notations as well as those models must be kept faithfully synchronized to justify the results of the analysis. Consequently, there is a need to offer formal verification techniques available in notations common to engineers, such as Simulink.

Our proposed strategy addresses these problems by hiding the interaction with Prism. This is achieved by using translation rules that take a Simulink diagram, annotated with failure conditions and logic [5, 10], as input and produce a Prism model and CSL formulas [8] (to check safety requirements) in such a way that we are able to report to the user only those safety requirements that are not satisfied. The Simulink's notations have straightforward formal treatment. This means that it is possible to use the models designed in these notations as the basis for formal analysis, removing the incremental labor for constructing formal model. It is worth noting, however, that in this work we do not provide the implementation of our proposed framework.

Hereby, a model-driven safety assessment approach combined with formal methods can provide more efficient means to assess the validity of the safety requirements in line with the system architecture.

A large amount of work has been done for quantitative safety assessment which is based mainly on a model-based approach with the support of formal methods. An example of an effort in this direction is the use of the MRMC [37] model checker to compute the failure conditions probability. This model checker is used in the COMPASS project [14] that aims at developing an alternative design language, based on the architectural description language AADL [37, 38]. Another relevant effort is the Probabilistic FMEA [18], a fault injection approach where Prism is used for modeling and safety analysis. Although the recent proposed approaches, many positive results and reasonable technical advantages in this context, our work is relevant in the sense that it proposes a systematic integration with a well-established model-based design tool such as Simulink, allowing it to be used directly in industry. Such systematization also allows us to prove that our translation is valid and always works. Moreover our strategy prevents that the formal notation is exposed to the user, avoiding that engineers, not familiar with this notation, have any impact for its adoption.

1.1.1 Main contributions

The main contributions of this dissertation are:

- A (occult) Markov-based quantitative model-based safety assessment process;
- Translation rules that systematically transform Simulink diagrams (tabular structures), into Prism models augmented with CSL formulas that can automatically verify the quantitative requirements of the system;
- The use of a single model from which it is possible to check any stipulated failure condition for the system;
- A case study that illustrates the overall approach.

1.1.2 Development of a case study

Our case study, although simple, was provided by Embraer and is a common subsystem found in aircrafts. But in the near future, Embraer itself intends to use the results provided by our work in several other case studies to measure its practical feasibility. Considering this, we intend to develop a plug-in for Simulink to automate our systematic approach. In this sense, the outcome of this project will have the potential to increase the quality of the products developed as well as the productivity; reducing development costs and generating competitive advantages.

1.2 Related Work

The remarkable interest in providing support for the safety assessment process by the introduction of formal methods and model-based approaches is evident in various relevant works. An example of an effort in this direction is the use of FTA to compute the failure conditions probability such as the HAZOP [14], which provides a design developed in Simulink; another relevant effort is the ISAAC project where SCADE is used for modeling and safety analysis [11, 13]. It is also worth mentioning FSAP/NuSMV-SA [15], a fault injection approach developed in the ESACS project.

Due to the limitations of static FTA methods, as discussed in Chapters 1 and 2, more recently, approaches considering dynamic reliability have been proposed, based on timed-probabilistic models that perform quantitative safety assessment based mainly on a previous qualitative analysis [23, 36, 68, 69]. Other interesting works have also incorporated coverage modeling (the probability that a system can automatically recover from a fault, given that a failure occurs), the failures on demand (that is, failure of a component to intervene), human intervention, the role of control/protection systems, expert judgment, and also the ordering of events during accident propagation [33]. These different approaches usually include BDD-based techniques for the evaluation of static fault trees and state transitions of semi-Markov models [31, 32], Stochastic Petri Nets [35], dynamic fault trees [34] and direct simulation via Monte Carlo analysis [29, 30]. Also, they can use a hybrid stochastic model that takes into consideration the mutual interactions between the hardware components of a plant and the physical evolution of its process variables by the integration of continuous time semi-Markov processes and Bayesian belief networks, for instance [28]. Our approach, which is concerned with the systematic generation of Markov models, differs from the previous works in the sense that they are more concerned with the manual modeling and direct evaluation of a given plant whereas we focus at mechanization (and ideally at automatization) of correct solutions as well as integration with accepted design tools, such as Simulink. Furthermore, we use model checking to support automatic verification of arbitrary CSL properties (in particular, safety properties). In the following section we highlight some works that are more closely related to ours.

1.2.1 FSAP/NuSMV-SA

FSAP/NuSMV-SA [15, 27] is a tool developed as part of the project ESACS [16] to automate the generation of fault trees. The methodology of ESACS aims to integrate the design with the safety analysis of the systems. The FSAP tool requires that the system model be specified in the NuSMV-SA language and uses its model checker on temporal logic to generate a fault tree from a particular top event. After the failure modes are defined, the user can automatically inject faults in the system model to create a new extended model. The model of the extended system adds a degraded performance compared to the original system, corresponding to the failure modes defined. This model can be used to assess the safety of the system.

A significant advantage of the FSAP automatic analysis tool is that it eliminates the need for manual creation of fault trees, since the system and failure model are specified. NuSMV-SA also provides a trace for every minimum cut it generates. The trace shows how the top event is reached, given a particular configuration of fault determined by the minimum cut set. FSAP/NuSMV-SA can also automatically perform analysis of events considering ordering a top event and a minimum cut set. Traditionally FTA is restricted to static analysis but using FSAP it is possible to investigate the influence of failure modes in dynamic situations.

Although FSAP is a very powerful tool, it has drawbacks, which may limit its practical applicability. A fault tree generated by the FSAP has a fixed structure, in the style "or-and", that is, it is a disjunction of all minimal cutsets, where each minimal cutset is as a product of basic events. A fault tree generated by a traditional manual analysis is usually more intuitive to read because the engineers create the fault tree corresponding to the structure of the system. Moreover, we note that there is not much flexibility in defining the fault model --- no reasonable way to indicate the fault propagation, simultaneous/dependent failures, or persistent/intermittent faults.

1.2.2 COMPASS Approach

In the COMPASS project [22], the model-based safety assessment approach results from the combination of the NuSMV [67] symbolic model checker and the MRMC [37] probabilistic model checker which allow the analysis of aircraft systems. The model is specified in the SLIM (System-Level Integrated Modeling) [12] design language, which is inspired by AADL [37, 38], architecture-based and model-driven top-down and bottom-up engineering.

The approach allows the extension of the nominal model of the system adding probabilistic fault behavior, providing a precise characterization of them and describing the system error propagation, recovery mechanisms, timing and probability based on a formal semantics. The analysis is based on a set of verification tools (FSAP/NuSMV, RAT, Sigref, and MRMC) which allow verifying safety/dependability aspects and quantitative analyses (probabilistic analysis of dynamic FTA).

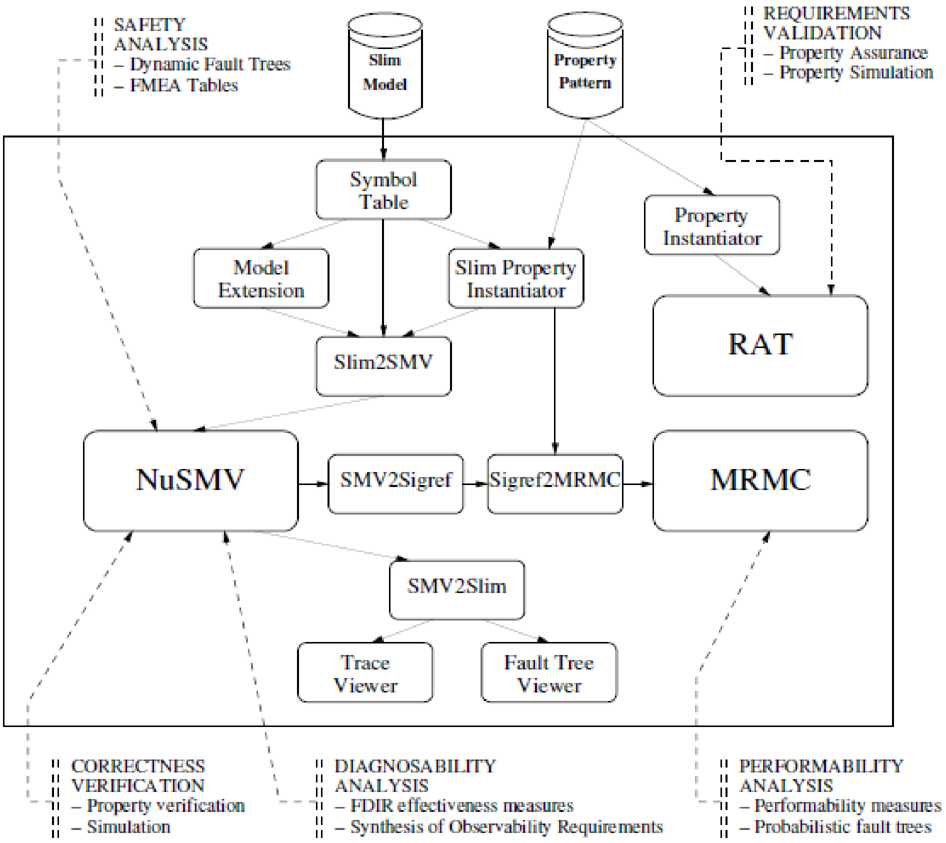


Fig 1.3 - Architecture of the Compass Toolset [22]

Fig. 1.3 shows the architecture of the COMPASS tool. It receives as inputs the SLIM model and the properties patterns. The latter describes the properties of the system, which are expressed in a user-friendly pattern, called ProProST [66]. ProProST converts these properties to its respectively CSL or CTL formulas. These inputs are processed and the tool generates several artifacts as output. For instance, the NuSMV checks the system's correctness by property verification, generating counterexample traces when some property is violated. Furthermore, the SLIM models can be adjusted and resulting in an interactive Markov chain, allowing that performance requirements are analyzed with MRMC. Moreover, the MRMC also computes the probability of the top events in fault-trees.

The completeness and consistency of this approach qualify it as a promising solution, but the formal modeling language adopted is exposed to the user (except the properties notation), demanding that engineers to be familiar with this notation. Thus, the impact for the adoption of this solution might be significant; our approach follows the hidden formal methods view.

1.2.3 Probabilistic FMEA

The work reported in [18] (which proposes pFMEA or Probabilistic FMEA) also uses the Prism model-checker to support quantitative analysis. The approach integrates the failure behavior into the system model described in CTMC via failure injection. An overview of pFMEA approach is illustrated in Fig. 1.4.

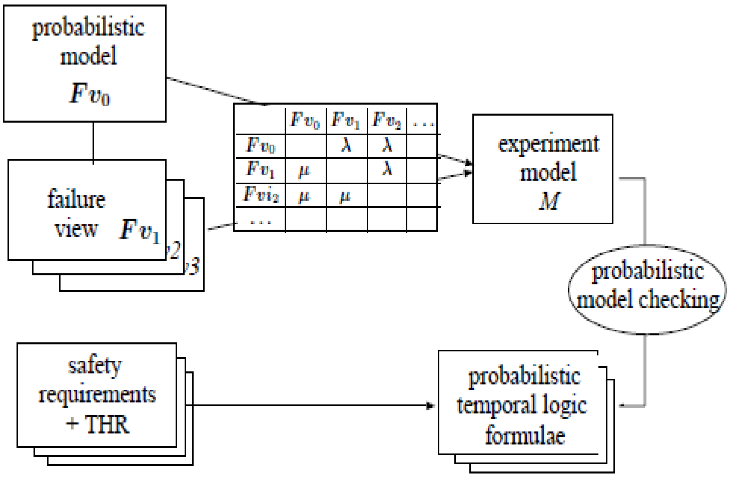

probabilistic model Fv_0
failure view Fv_1
Fv_0 | Fv_1 | Fv_2 | ...
Fv_0 | | λ | λ
Fv_1 | μ | | λ
Fvi_2 | μ | μ
...
experiment model M
probabilistic model checking
safety requirements + THR
probabilistic temporal logic formulae

Fig 1.4 - Probabilistic FMEA (pFMEA) – Approach Overview [18]

As illustrated in Fig. 1.4, first, the user describes the system probabilistic model in a functional vision (normal behavior). Then the user can describe the failure view for each component by injecting in the system specification, their failure modes. After that, the user feeds a matrix that specifies the possible transitions, including their transition rates between the normal operation of system and failure situation. Therefore it is possible to determine quantitatively and formally if a violation happens of the safety requirements stipulated using the Prism probabilistic model checker with the support of temporal expressions. Also, using temporal languages (CSL, PCTL), we can infer the probability that the failures can occur when the system is in a particular failure mode. This advantage makes this approach interesting if comparing the standard FMEA with existing techniques.

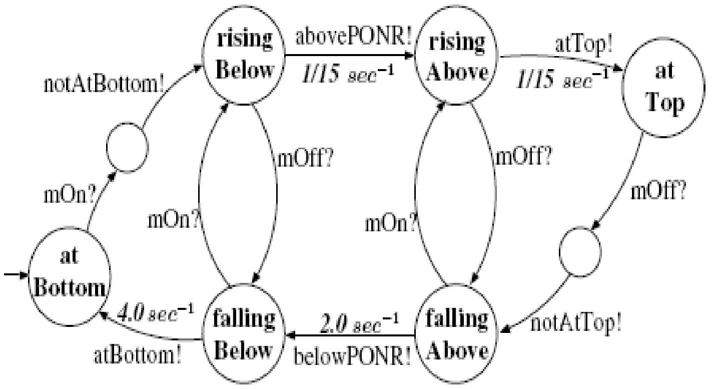


Fig 1.5 - Example of modeling using PFMEA [18]

In one sense, pFMEA performs a more detailed analysis than ours because it considers faulty as well as normal behaviors of a system (see Fig. 1.5). However, this approach does not generate the model systematically, so there is no notion of soundness concerning the model generation, and is more likely to generate state explosion, since it does not present techniques to enable reduction of the Markov model generated.

1.3 Dissertation Organization

This dissertation is organized as follows. Chapter 2 provides the background on the safety assessment process as well as the prominent model-based solution.

Chapter 3 presents an overview of probabilistic models outlining basic concepts of Markov Chains. Furthermore, it presents the Prism formal language with its model checker (Prism).

Then, in Chapter 4 we present the approach developed to achieve formal probabilistic analysis of aircraft systems in a model-based context. All phases of the process are detailed and justified for use in our case study.

The other contribution of this work is given to the development of a case study which is shown in Chapter 5. We describe the application of our strategy in a simple aircraft subsystem.

Finally, Chapter 6 shows our conclusions that discuss the benefits and drawbacks of our strategy. Moreover, we give a brief overview of some future works.

Chapter 2

Safety Assessment Process

In this chapter we introduce the safety assessment process used in the aeronautical industry. This process involves complex phases and activities [2, 4] that are executed during the design of aircraft systems, aiming to minimize the occurrence likelihood of potential hazards of a system.

The process covers several aspects (hardware, software and architecture) of the system, performing qualitative and quantitative analysis, necessary to validate the safety requirements stipulated. Thus, we also describe these analyses during this process as well as the model-based solution, which is the state-of-the-art in the safety assessment process. Finally, we illustrate a model-based scenario with a simple aircraft system that is used in our case study. This process is very detailed and complex and we only explain its essential parts (for a deeper description please refer to [2, 4]).

2.1 Safety Assessment of Aircraft Systems

Safety Assessment is the process used to ensure the adequacy of a system's architecture design with its respective risk of hazard situations, which must be kept at tolerable levels. In short, this process aims to produce a safe aircraft.

This process is driven by aircraft functions that are organized in different stages as can be seen in Fig. 2.1. During this process, hazard analysis is performed in parallel with system design where the failure conditions (potential failures that can affect the aircraft functions) are identified and classified according to its severity. Starting with the functions of the highest level, the assessment goes down gradually to the low-level functions, guided by the system's architecture. As a consequence of this gradual analysis, the derived safety requirements that emerge can be either in qualitative or quantitative form. These new safety requirements are introduced in the top-level and subsystem design. They comprehend the high-level airplane safety goals as well as system safety goals that must be considered in the proposed system architectures.

Industry standards, such as the ARP 4761 [2] for civil aviation, provide the criteria to determine the corresponding criticality of a hazard and which levels are considered acceptable or not. Such standards aim at providing common guidance for engineers and certification authorities on how to address safety and reliability issues throughout the development lifecycle of complex systems.

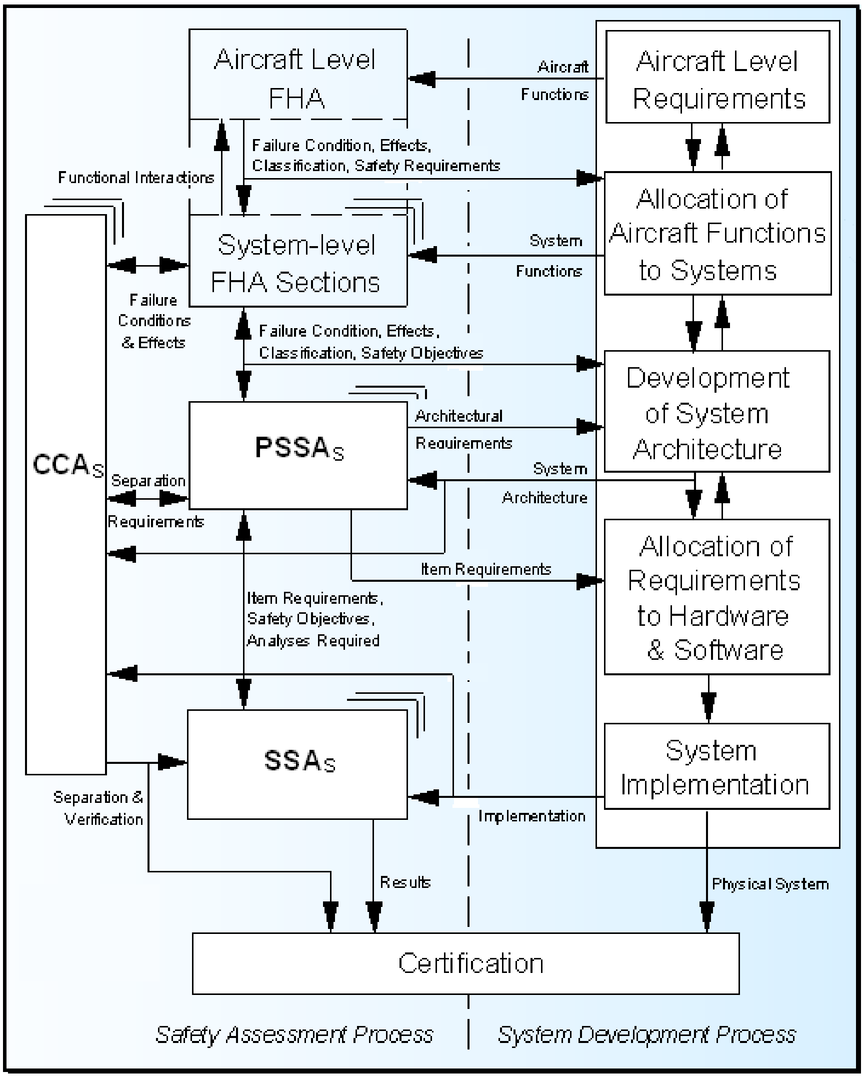
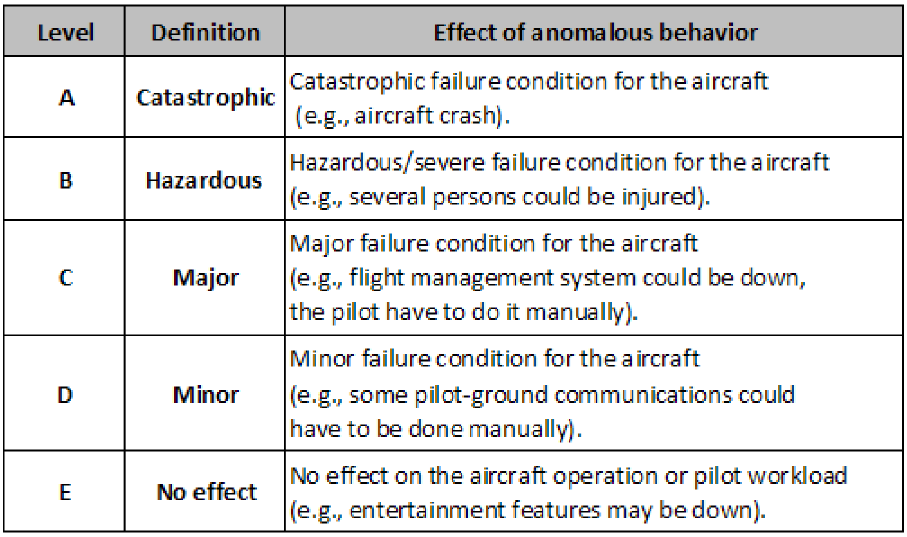

Aircraft Level FHA
Aircraft Level Requirements
Aircraft Functions
Failure Condition, Effects, Classification, Safety Requirements
Functional Interactions
Allocation of Aircraft Functions to Systems
System-level FHA Sections
System Functions
Failure Conditions & Effects
Failure Condition, Effects, Classification, Safety Objectives
Development of System Architecture
PSSAs
Architectural Requirements
System Architecture
CCAs
Separation Requirements
Item Requirements
Allocation of Requirements to Hardware & Software
Item Requirements, Safety Objectives, Analyses Required
SSAs
System Implementation
Implementation
Separation & Verification
Results
Physical System
Certification
Safety Assessment Process
System Development Process

Fig 2.1 - Overview of the safety assessment process

The central element of the safety assessment process is the method FHA (Functional Hazard Analysis). The goal of the method FHA is to identify all possible conditions on which an aircraft function can fail. For example, a failure condition would be "failure of the longitudinal control during the cruise." For each failure condition, a criticality is assigned. The criticality is used to indicate the effect on the aircraft if that failure condition occurs (see Fig. 2.2).

As long as different systems are assigned to a given function, defined on the aircraft level, the effects caused by the loss of this function can spread among them. For instance, the hydraulic system is a system that helps the longitudinal control. If it was defined previously that the loss of longitudinal control is catastrophic, possibly the loss of the hydraulic system is catastrophic as well. In this sense, FHA hierarchically unpacks the systems until the low-level functions are considered and derives the safety requirements based on the defined failure conditions.



Level	Definition	Effect of anomalous behavior
A	Catastrophic	Catastrophic failure condition for the aircraft (e.g., aircraft crash).
B	Hazardous	Hazardous/severe failure condition for the aircraft (e.g., several persons could be injured).
C	Major	Major failure condition for the aircraft (e.g., flight management system could be down, the pilot have to do it manually).
D	Minor	Minor failure condition for the aircraft (e.g., some pilot-ground communications could have to be done manually).
E	No effect	No effect on the aircraft operation or pilot workload (e.g., entertainment features may be down).

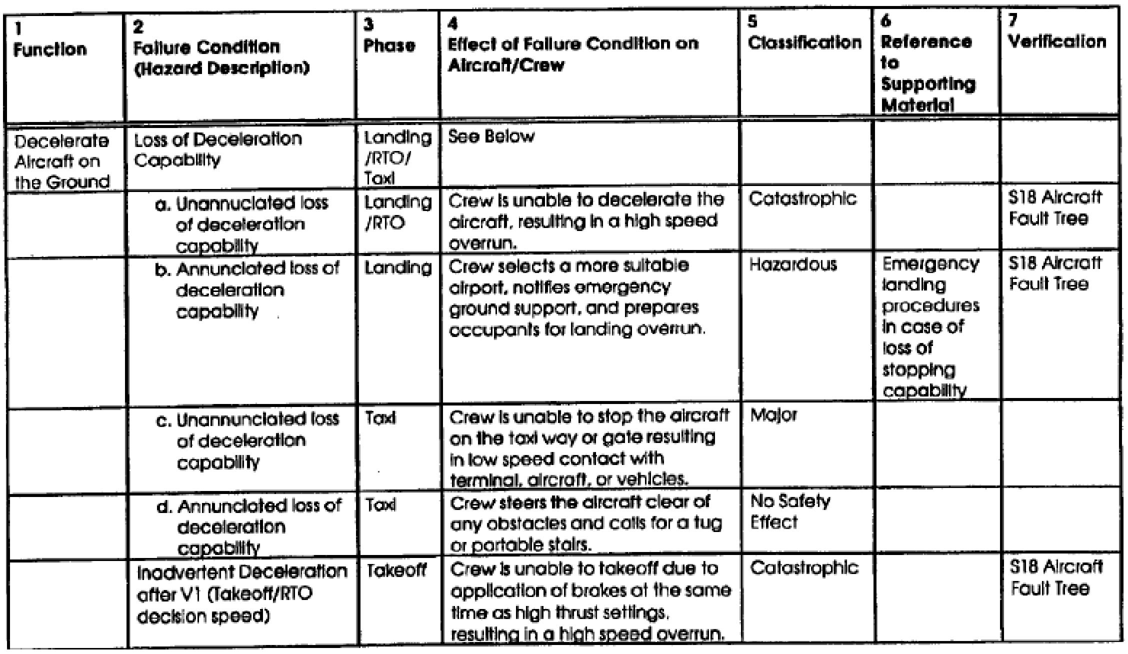
Fig 2.2 - List of criticality levels and its effects

Therefore, FHA is responsible to generate requirements such as “a catastrophic failure condition shall not occur more frequently than 10^{-9} per flight hour” or “No catastrophic failure condition result from a single failure". The former restriction (10^{-9}), associated to the first requirement, corresponds to the allowable quantitative probability, determined by the FAR 25.1309, for the failure conditions whose likelihood of occurrence must be extremely improbable. The latter illustrates a common qualitative requirement.

After identifying the failure conditions in the FHA (Fig. 2.3 illustrates an example of an aircraft FHA table that addresses a failure condition), the engineers employ other techniques to determine which single failures or combinations of failures can exist at the lower levels that might cause each failure condition and verify if the proposed system architecture satisfies the safety objectives defined in the FHA.

They create their own system behavior understanding and perform the safety assessment using a classical technique (Fault Tree Analysis - FTA, Dependence Diagrams - DD, Markov Analysis - MA) to validate the safety requirements as well as demonstrate the design concept. The Preliminary System Safety Assessment (PSSA) usually takes the form of such a technique and also includes the Common Cause Analyses (CCA). A Common Cause Analysis assesses the specific system architecture by evaluating the overall architecture sensitivity to common cause events [2, 4].

During the PSSA stage a systematic examination of the proposed system architecture is performed to determine how failures can cause the functional hazards identified by FHA. PSSA aims to establish the safety requirements of the system in accordance of the safety objectives identified by the FHA.



1 Function	2 Failure Condition (Hazard Description)	3 Phase	4 Effect of Failure Condition on Aircraft/Crew	5 Classification	6 Reference to Supporting Material	7 Verification
Decelerate Aircraft on the Ground	Loss of Deceleration Capability	Landing /RTO/ Taxi	See Below			
	a. Unannuciated loss of deceleration capability	Landing /RTO	Crew is unable to decelerate the aircraft, resulting in a high speed overrun.	Catastrophic		S18 Aircraft Fault Tree
	b. Annunciated loss of deceleration capability	Landing	Crew selects a more suitable airport, notifies emergency ground support, and prepares occupants for landing overrun.	Hazardous	Emergency landing procedures in case of loss of stopping capability	S18 Aircraft Fault Tree
	c. Unannunciated loss of deceleration capability	Taxi	Crew is unable to stop the aircraft on the taxi way or gate resulting in low speed contact with terminal, aircraft, or vehicles.	Major		
	d. Annunciated loss of deceleration capability	Taxi	Crew steers the aircraft clear of any obstacles and calls for a tug or portable stairs.	No Safety Effect		
	Inadvertent Deceleration after V1 (Takeoff/RTO decision speed)	Takeoff	Crew is unable to takeoff due to application of brakes at the same time as high thrust settings, resulting in a high speed overrun.	Catastrophic		S18 Aircraft Fault Tree

Fig 2.3 – Partial Aircraft FHA example that address only "Decelerate Aircraft on the Ground"

The PSSA performs an interactive process associated with the design definition. It includes consideration of the system qualitative issues and consists of analyzing its architecture with focus:

- Required resources for the nominal behavior of each system component: definition of ports and association, control variables and its transfer functions;
- Fail-safe design concept that uses the following design principles or techniques in order to ensure a safe design: designed integrity and quality to ensure intended function and prevent failures: redundancy or backup systems, isolation and/or segregation of systems and components, etc.;
- Possible failure modes and functional mechanisms (monitoring, reconfiguration) elaborated to limit/control their effects: monitor, switches, auxiliary mechanisms;
- Dependencies between system components: power supply and basic components .

Traditionally the PSSA and SSA (System Safety Assessment) stages are based on the FTA, the well-known top-down technique used in industry, with support of the CCA, both described in detail in the ARP 4761 [2]. A fault tree is a graphical model that describes the combination of failure events [1]. It is formed by a top event, intermediary and basic events as well as logic gates ("OR", "AND", etc.) aiming at capturing the relationship between the events whose occurrence, according to the logic captured by the gates, enables a high level event to occur as well (representing the failure condition). Fig. 2.4 illustrates an example of a generic fault tree model.

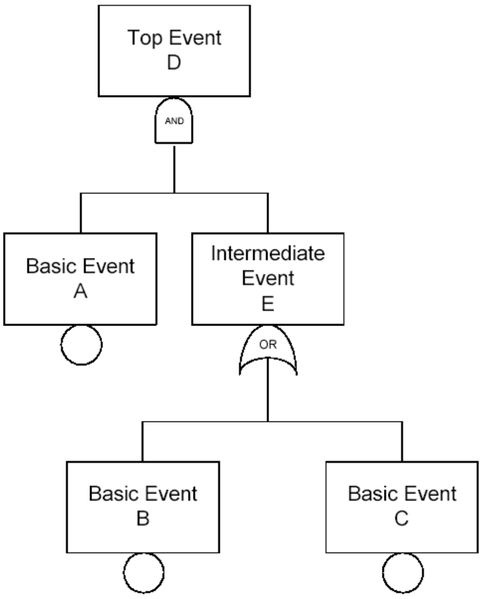


Fig 2.4 Example of a generic fault tree diagram

For each basic event, occurrence probabilities are assigned and the probability of occurrence of a high level event can be calculated from the lower level dependent events. The FTA facilitates subdivision of system level events into lower level events for ease of analysis. At the lowest level, the PSSA determines the safety design requirements of the related systems.

The SSA is responsible to assess each implemented system to show that safety objectives from the FHA and its derived safety requirements from the PSSA are met. The SSA analysis is similar to the PSSA , except that instead of evaluating proposed architectures and deriving system safety requirements, SSA performs a an extensive verification to check if the implemented design meets both the qualitative and quantitative safety requirements as defined in the FHA and PSSA. An assessment to identify and classify failure conditions is necessarily qualitative. On the other hand, an assessment of the probability of a failure condition may be quantitative.

The SSA is commonly derived from the PSSA FTA (DD or MA) and, at component level, it uses the quantitative values obtained from the Failure Modes and Effects Summary (FMES). The FMES is a summary of failures identified by FMEA (Failure Mode Effective Analysis [2]). FMEA is a bottom-up method for assessing the failure modes of a system and determining the effects of the relations among these failures. FMEA is used to evaluate the effects on the system and airplane of each possible element or component failure. When properly formatted, it aids in identifying the possible causes of each failure mode. The system FMEA is summarized into the system FMES to support the failure rates of the failure modes considered in the FTA. Therefore, SSA must verify that all significant effects identified in the FMES are considered for inclusion as basic events in the FTA.

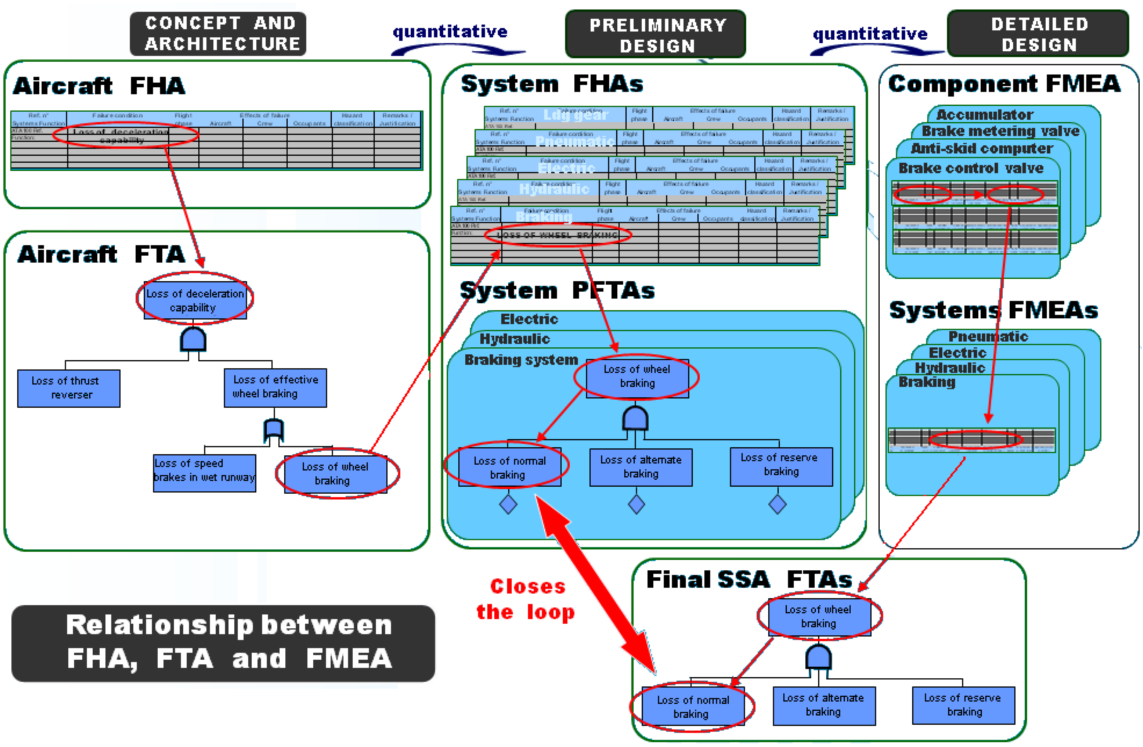


Fig 2.5 - Relationship between FHA, FTA and FMEA

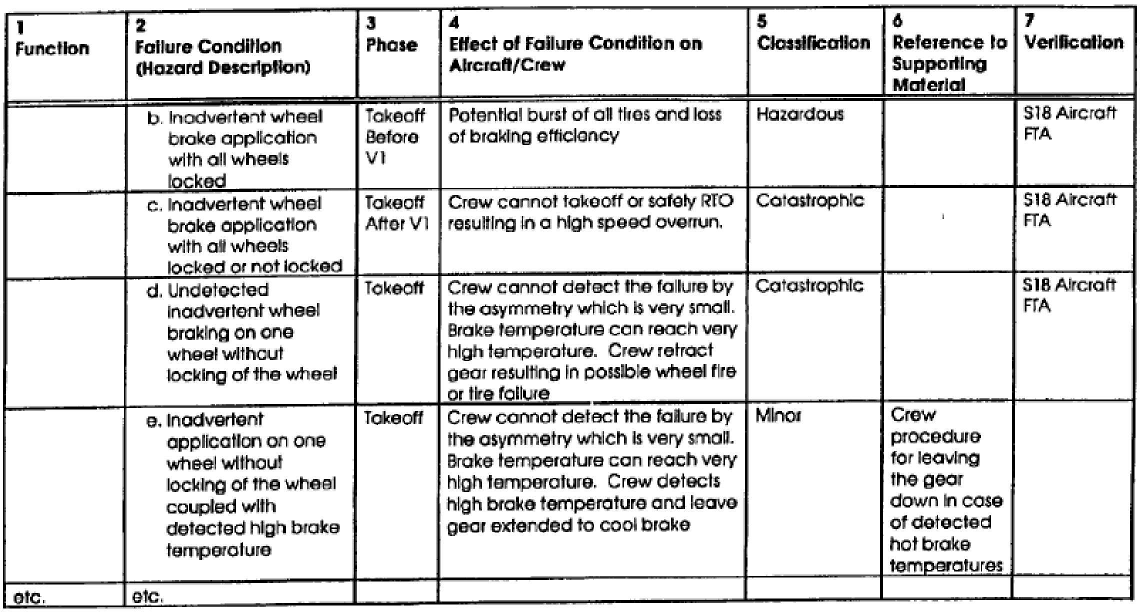
Currently, FTA acts as a logical complement to FHA. The relationship between them can be seen in see Fig. 2.5. That is, fault trees must be created for the aircraft, decomposing top level hazards into their causes, down to single events. These events correspond to system and component failures with associated failure rates. The failure rates of the basic events (failure modes) are determined by reliability prediction methods such as FMEA. Considering the failure conditions identified in the FHA, the PSSA and SSA can be applied mainly to determine:

- Which single failures or combination of failures can exist at the lower levels (basic events) that can cause each failure condition;
- The average probability of occurrence per flight hour for each failure condition.

As result, for each failure condition, it should be determined if the associated safety requirements are met. As an example, Fig 2.6 shows a fragment of System FHA table of the Wheel Brake System (WBS) [2], which is derived from the Aircraft FHA shown in Fig. 2.3. The WBS is used to provide safe retardation of the aircraft during taxiing and landing phases, and in the event of a rejected take-off. The following expressions are a set of significant safety requirements of this system resulted from its FHA analysis:

- *Loss of all wheel braking during landing or RTO shall be less than 5E-7 per flight;*
- *Asymmetrical loss of wheel braking coupled with loss of rudder or nose wheel steering during landing or RTO shall be less than 5e-7 per flight;*

- *Undetected inadvertent wheel braking on one wheel w/o locking during takeoff shall be less than 5e-9 per flight.*



1 Function	2 Failure Condition (Hazard Description)	3 Phase	4 Effect of Failure Condition on Aircraft/Crew	5 Classification	6 Reference to Supporting Material	7 Verification
	b. Inadvertent wheel brake application with all wheels locked	Takeoff Before V1	Potential burst of all tires and loss of braking efficiency	Hazardous		S18 Aircraft FTA
	c. Inadvertent wheel brake application with all wheels locked or not locked	Takeoff After V1	Crew cannot takeoff or safely RTO resulting in a high speed overrun.	Catastrophic		S18 Aircraft FTA
	d. Undetected inadvertent wheel braking on one wheel without locking of the wheel	Takeoff	Crew cannot detect the failure by the asymmetry which is very small. Brake temperature can reach very high temperature. Crew retract gear resulting in possible wheel fire or tire failure	Catastrophic		S18 Aircraft FTA
	e. Inadvertent application on one wheel without locking of the wheel coupled with detected high brake temperature	Takeoff	Crew cannot detect the failure by the asymmetry which is very small. Brake temperature can reach very high temperature. Crew detects high brake temperature and leave gear extended to cool brake	Minor	Crew procedure for leaving the gear down in case of detected hot brake temperatures	
etc.	etc.					

Fig 2.6 - Partial Wheel Brake System FHA (addresses only "Decelerate the Wheels on the Ground)

According to the certification authorities, the proposed system design must assure, for instance, that the probability of catastrophic failure conditions that can occur is extremely improbable. So, this class of failure conditions may be analyzed in a quantitative basis (in addition to qualitative analysis), because these failures are more critical. Consequently, the average probability of occurrence for each failure condition per flight hour must be calculated assuming a typical average flight and considering the appropriate exposure and risk times to check if a certain failure condition is kept at acceptable levels [2, 24].

Considering all these issues, system architecture may be designed and verified in such a way that the system safety requirements at low-level (SSA) satisfy the system high-level requirements As components are integrated into system and systems are integrated into the aircraft, the failure effects are compared with the failure conditions identified in the FHA. This comparison is called an "Integration cross-check" [2].

In summary, the safety assessment process has four basic steps:

1. Clearly identify undesired events, its effects and criticality;
2. Perform a qualitative analysis by constructing a model of the sequence of events leading to an undesired event. This model accurately describes the logic flow of the entire process leading to the event;
3. Perform a reliability prediction for the component elements parts;
4. Perform a quantitative analysis by constructing a mathematical model (a set of equations based on the logic derived from the qualitative model), and calculating the probability of the undesired failure conditions over a certain exposure time.

This process can involve other safety techniques as well but this is out of the scope of this work. For further information about this please refer to ARP 4761 [2]. As was already described, traditionally, steps 2 and 4 are performed using FTA. However this method strongly relies on human intervention and thus it presents limitations directly proportional to the complexity of the airborne systems, mainly because a quantitative analysis of a fault-tree must include the following concerns:

1. Influence of repeated events in the probability of occurrence of top event of fault-trees;
2. Influence of change of flight duration in the value of the probability of occurrence of top event of fault-trees;
3. Adjusts on the procedure to calculate the probability of occurrence of top event given the existence of latent events in fault-trees.

Furthermore, note that there is a critical factor in the application of this method. As a usual complex system has several failure conditions, several different fault trees are constructed, one for each failure condition. These fault trees are constructed to assess the cause and probability of single top event. The key point is that each time a safety requirement is violated, the system architecture must be revised to reduce the likelihood of the hazard occurring and consequently all related fault trees must be constructed again.

2.2 Quantitative Analysis

The relation between the criticality of a failure and its probability of occurrence within the system operational lifetime is commonly the factor that determines the risk of an accident. Hereby, the qualitative aspect concerns the characterization of the behavior of different faults that may result in a top level hazard whereas the quantitative aspect concerns the reliability predictions for the system that may cause or contribute to this hazard. Although the concept of safety itself is not necessarily associated with the concept of reliability, both should be considered simultaneously in order to obtain practical results. Knowing that a system is never free from critical failures, safety analysis methods must consequently consider reliability issues to demonstrate that the likelihood of critical accidents are minimized by using numerical evidences.

Hence, the objective of the quantitative analysis is to ensure an acceptable safety level for systems on the aircraft using numerical evidences. When using quantitative analysis to help determining compliance with the safety requirements, the following descriptions of the probability terms referenced in FAR 25.1309 are mandatory, because they are commonly accepted by the certification authorities. They are expressed in terms of acceptable ranges for the average probability per flight hour:

- ***Probable:*** failure conditions whose average probability occurrence per flight hour is greater than the order of magnitude of 10^{-5};
- ***Remote***: failure conditions whose average probability occurrence per flight hour is less than the order of magnitude of 10^{-5}, but greater than 10^{-7};

- ***Extremely Remote:*** failure conditions whose average probability occurrence per flight hour is less than the order of magnitude of 10^{-7}, but greater than 10^{-9};
- ***Extremely Improbable:*** failure conditions whose average probability occurrence per flight hour is less than the order of magnitude of 10^{-9}.

According to FAR 25.1309, a logical and acceptable inverse relationship must exist between the average probability per flight hour and the severity of failure condition effects, as shown in Figure 2.7:

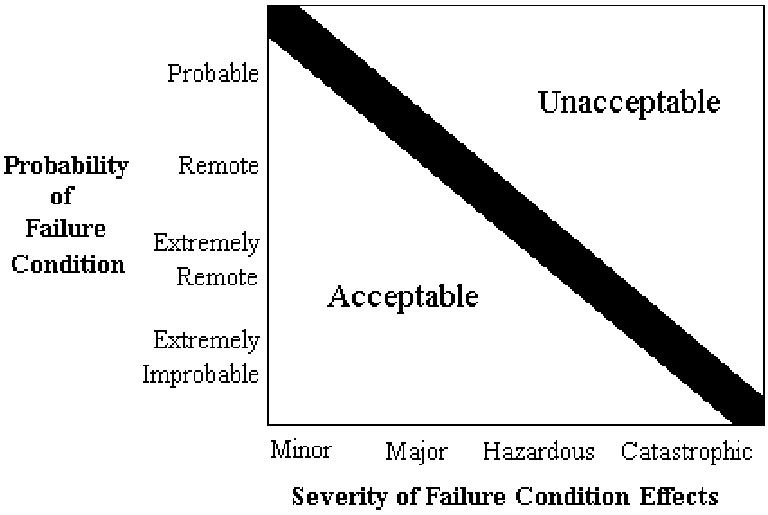
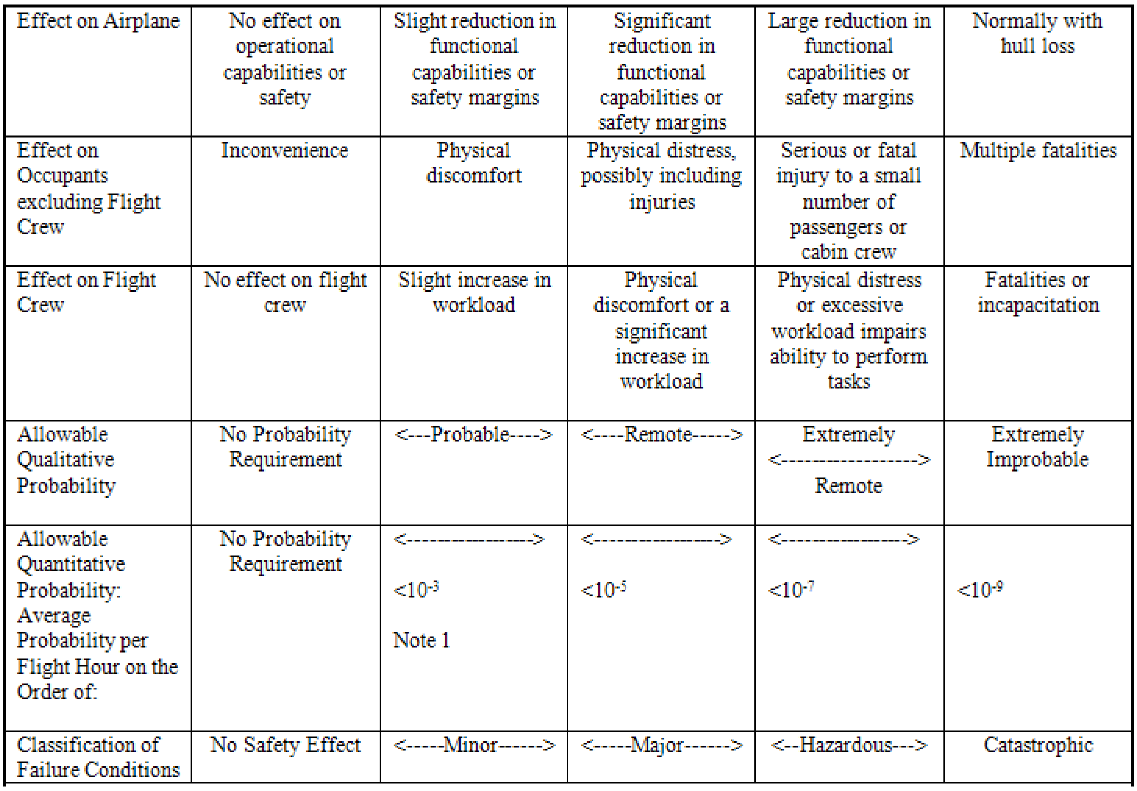


Fig 2.7 - Relationship between Probability and Severity of Failure Condition Effects [24]

It is worth noting that Fig. 2.7 does not exhibit the failure conditions with *No Safety Effect* because they have no numerical probability constraint. For the *Minor* failure conditions, they can even be *Probable*. On the other hand, *Major* failure conditions may be at most *Remote* but not *Probable*; Finally the two most critical failure conditions are *Hazardous*, which may be at most *Extremely Remote*, and *Catastrophic*, which needs to be *Extremely Improbable*. Considering the relationship between the severity of failure conditions effects and their acceptable ranges for the average probability per flight hour, the quantitative requirements associated with failure condition are described in Fig. 2.8.

ARP 4761 also accepts Markov analysis or dependence diagrams as alternatives to perform quantitative analysis during the SSA. The basic information used as input to these methods is failure conditions and failure rates of the primary events. As described in previous sections, failure conditions are identified during the FHA analysis, which considers the severity of the occurrence of each failure condition over the aircraft functions to define the related safety requirement, using an argument (maximum tolerable probability). For example, FHA determines that the probability of occurrence of a catastrophic failure condition must not be greater than 10^{-9} per flight hour. Failure rate is an attribute used to model the likelihood of each basic failure mode (primary and independent failure) of the system. FMEA supplies the failure rates considered in the system.



Effect on Airplane	No effect on operational capabilities or safety	Slight reduction in functional capabilities or safety margins	Significant reduction in functional capabilities or safety margins	Large reduction in functional capabilities or safety margins	Normally with hull loss
Effect on Occupants excluding Flight Crew	Inconvenience	Physical discomfort	Physical distress, possibly including injuries	Serious or fatal injury to a small number of passengers or cabin crew	Multiple fatalities
Effect on Flight Crew	No effect on flight crew	Slight increase in workload	Physical discomfort or a significant increase in workload	Physical distress or excessive workload impairs ability to perform tasks	Fatalities or incapacitation
Allowable Qualitative Probability	No Probability Requirement	<---Probable---->	<----Remote----->	Extremely <------------------> Remote	Extremely Improbable
Allowable Quantitative Probability: Average Probability per Flight Hour on the Order of:	No Probability Requirement	<-----------------> $<10^{-3}$ Note 1	<-----------------> $<10^{-5}$	<-----------------> $<10^{-7}$	$<10^{-9}$
Classification of Failure Conditions	No Safety Effect	<-----Minor------>	<-----Major------>	<--Hazardous--->	Catastrophic

Fig 2.8 - Relationship between probability and severity of failure condition

Independent of the quantitative analysis to be used (FTA, Dependence Diagrams or Markov Analysis), the probabilities are estimated from the failure rates and exposure time of the events. For the purpose of these analyses, the failure rates are commonly constant over time, based on exponential distribution function. They are estimates of mature failure rates after infant mortality and prior to wear-out, as described in Fig 2.9.

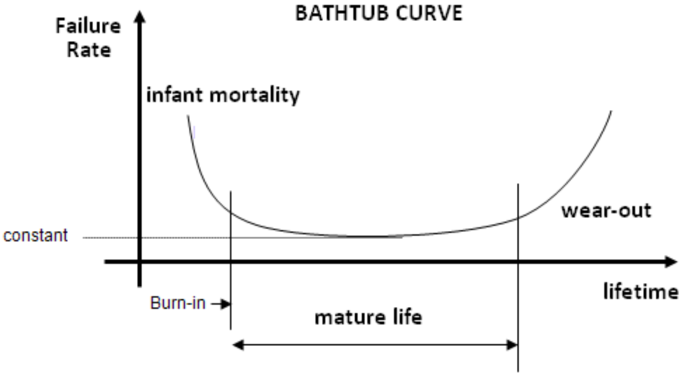


Fig 2.9 - The classic "Bathtub Curve" used to diagram the constant failure rate period in the life of an electronic component

Thus, these analyses disregard the wear-out or infant mortality. When wear-out or infant mortality has to be considered, other distribution functions (such as Weibull) need to be employed. When available, service history of same or similar components in the same or similar environment should be used.

For various reasons, component failure rate data are admittedly not precise enough to enable accurate estimates of the probabilities of failure conditions [4]. This results in some degree of uncertainty, as indicated by the wide line in Fig. 2.7, and by the expression "on the order of" in the descriptions of the quantitative probability terms that are provided previously. When calculating the estimated probability of each failure condition, this uncertainty should be accounted such that it does not compromise safety.

When performing quantitative analysis, consistence must be guaranteed with the maintenance tasks and intervals used by the maintenance program for the aircraft. The following maintenance scenarios can be used to show compliance with FAR 25.1309:

- Evident failures will be corrected before the next flight, or a maximum time period will be established before a maintenance action is required. If the latter is acceptable, the analysis should establish the maximum allowable interval before the maintenance action is required;
- Latent failures will be identified by a scheduled maintenance task (a latent failures is a failure that is not detected and/or annunciated when it occurs). Following its removal and repair, the Mean Time Between Failures (MTBF) of a component should be the basis for checking the interval time.

When one or more failed elements in the system can persist for multiple flights (latent failure), the calculation should consider the relevant exposure times (that is, time intervals between maintenance and operational checks/ inspections). In such cases the probability of the failure condition increases with the number of flights during the latency period.

Hereby, a probabilistic model based on the failure logic of the system is generated aiming to calculate the average probability of such failure conditions per flight hour, assuming the appropriate exposure time of failures and shows if the results are tolerable.

For instance, a Markov Analysis calculates the probability of the system being in various states as a function of time. A transition from one state to another occurs at a given transition rate, which reflects component failure rates and redundancy. A system changes state due to various events such as component failure, completion of repair, reconfiguration after detection of a failure, etc. Each state transition is a random process which is represented by a specific differential equation. The probability of reaching a defined final state can be computed by combinations of the transitions required to reach that state.

2.3 Model-based Safety Assessment

In the safety-critical systems domain there is an increasing trend towards *model-based safety assessment* [11, 13]. It extends the existing model-based development activities (simulation, verification, testing and code generation), which are based on a high-level model of the system (expressed in a notation such as Simulink or Statemate), to incorporate the safety analysis. These new alternatives are interesting because they are simple, compositional, and depend of less engineer's skills to be applied. In addition, they can use formal methods, for instance theorem provers, model-checkers and static-checkers [13, 15], to automate, even if partially, the analysis. Moreover, formal methods are one of the alternative methods proposed in DO-178B [12] for the airborne software certification.

2.3.1 Principles of Model-based Safety Assessment

The Model-based Safety Assessment process consists in building a representative concrete model that can be exercised by dedicated tools to perform assisted safety assessments. The model is created to represent the system architecture and relevant behavior data. Details about each system component can be included considering functional and safety aspects. The modeling environment offers means to represent safety/abstract behaviors of the system components, which describe the relationship between inputs and outputs data in nominal situation as well as its failure events with their occurrence conditions (based on input data and failure mode) and their effects on its outputs. At system level, links between components are created according to the system architecture.

To enhance the readability of the model, a graphic representation is associated to each component so that the model looks like a system architecture diagram (block diagrams). Figure 2.10 illustrates such a system architecture diagram.

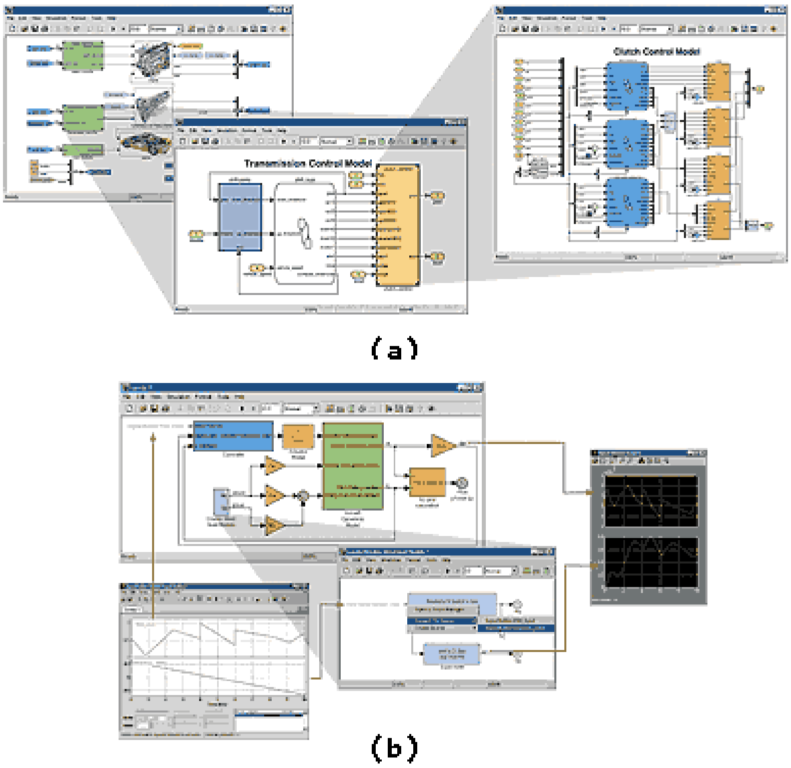


Fig 2.10 (a) Assembling design components to construct a model. (b) Test of multiples scenarios [9]

Furthermore, models are hierarchical: preliminary high level pieces of the system may be further refined into lower level components. Thus the design engineer gets a support to validate the safety analyst system understanding and to reduce the risk of misinterpretation. The model based safety assessment process can be composed in five main steps:

- Engineer interpretation;
- Model creation;
- Validation (components, systems and safety criteria);
- Assisted assessment (safety assessment and simulation results);
- Model update (refining/updating the model).

Fig. 2.11 summarizes a model-based safety assessment process:

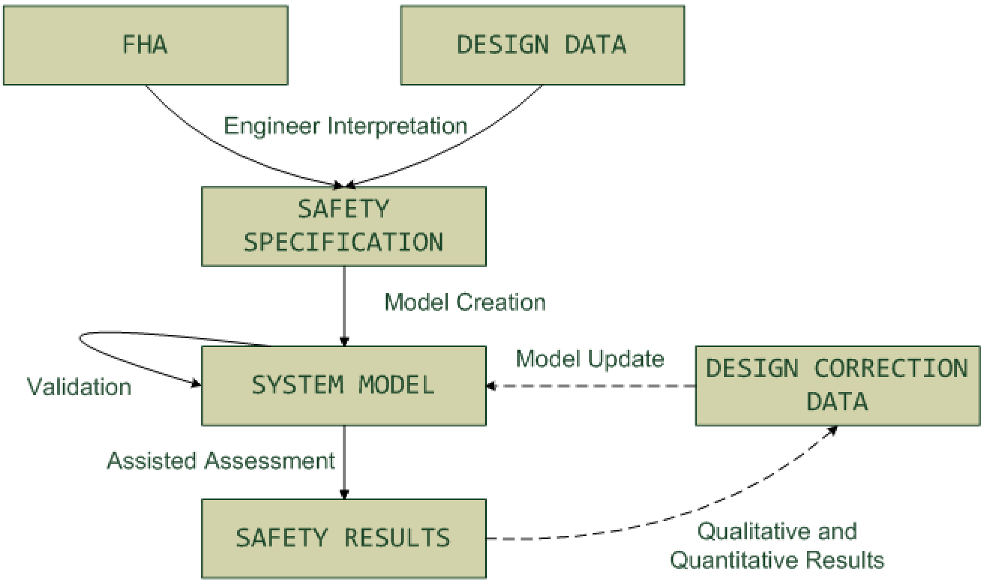


Fig 2.11. A Common Model Based Safety Assessment Process

The first step details the safety description of the system architecture content and its behavior according to its failure conditions. The model serves to analyze several failure conditions impacting a given system architecture and it is performed by the extraction of failure conditions from the FHA. So the engineers can list the relevant data into a safety specification of the system.

Next, the system architecture modeling is done using component library from public or private sources. At component level, the behavior and I/O are summarized in events that are limited to failure or reconfigurations (from FMEA/FMES if available or from previous design). Hence the system safety specification is implemented into the system architecture model.

Syntactic/semantic tool support permits to verify the correctness of the model according to formal language notation. Then the system designer has to validate the model, that is, verify that its behavior is consistent with the system specification or the expected system behavior.

After that, the safety requirements need to be validated, considering the failure conditions of the system. This validation can be done during a review between designers and FHA safety analysts. Hence, the potential advantage to safety analysts, formal methods can ease the validation process by providing mathematical tools to exploit the model. So, qualitative and quantitative results are obtained and used to validate the safety requirements of the system.

When a design correction decision is taken to replace a component or to modify the architecture, the model is refined or updated. The last step consists in refining/updating the model in case of an architectural modification. This occurs primarily when the initial architecture does not fulfill some requirements, or if a technical decision leads to a component replacement.

The assurance that a model representation conforms to the real system can be reinforced by simulating combinations of failures seen on aircraft and checking that the results are coherent.

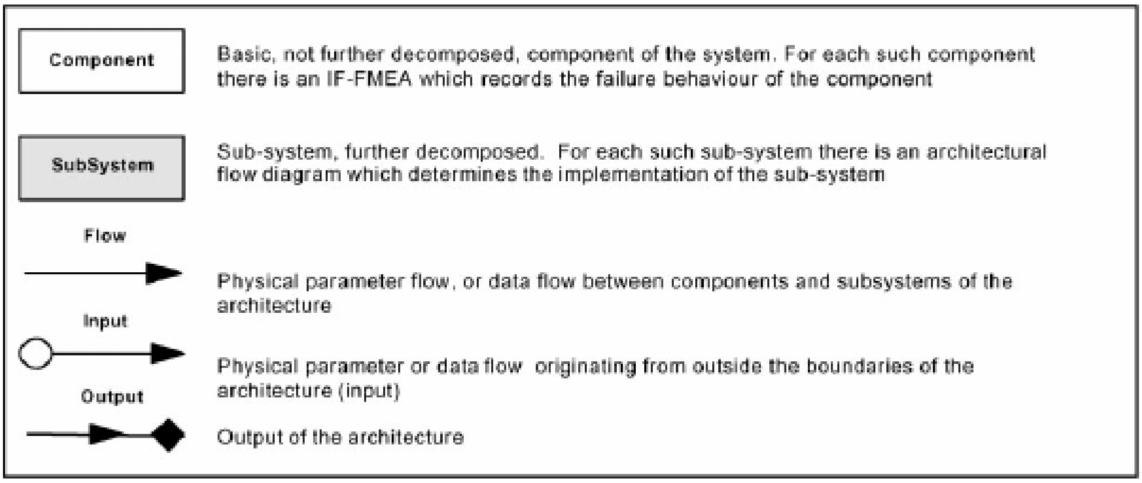
2.3.2 HiP-HOPS

The approach proposed in this work is based on the HiP-HOPS (Hierarchically Performed Hazard Origin and Propagation Studies) [40] method, which is one of the most prominent model-based approaches [10]. It has attracted great interest from the industry and demonstrated a (comparatively) high level maturity.

HiP-HOPS is a systematic method of safety analysis based on the techniques FHA and IF-FMEA (Interface-Focused FMEA) [5, 10]. IF-FMEA is an extension of FMEA inspired by the work described in [41], which defines a graphical notation (called Failure Propagation and Transformation Notation---FTPN) for the representation of the transformation and propagation of failures in a system. It defines a set of equations which characterize the logical relationships between input and output failure events.

HiP-HOPS allows an integrated analysis of a complex system from the high functional level to the low level, represented by the component failure modes. It makes the analysis of the system model in a hierarchical form. HiP-HOPS can assist the development of an appropriate initial architecture for the system, as well as its decomposed sub-systems and basic component elements.

FHA can assist the development of an appropriate initial architecture for the system, as well as its decomposed sub-systems and basic failure elements. Following the FHA analysis, HiP-HOPS allows an integrated analysis of a complex system from the high functional level to the low level, represented by the component failure modes. It makes the analysis of the system model in a hierarchical form that progressively records the increasing details of the implementation of the system. Constraints are used on the modeling notation for describing the levels of design to achieve the consistency in the model. Following the architecture of the system, flow diagrams are used to describe the relations between a system and its subsystems. Fig. 2.12 illustrates the primitive elements of the proposed notation used in HiP-HOPS. This notation is semantically and syntactically linked to the design representation of the system.



Component	Basic, not further decomposed, component of the system. For each such component there is an IF-FMEA which records the failure behaviour of the component
SubSystem	Sub-system, further decomposed. For each such sub-system there is an architectural flow diagram which determines the implementation of the sub-system
Flow	Physical parameter flow, or data flow between components and subsystems of the architecture
Input	Physical parameter or data flow originating from outside the boundaries of the architecture (input)
Output	Output of the architecture

Fig 2.12 – HiP-HOPS modeling notations [5]

The support for a graphical notation enables that complex systems are modeled as hierarchies of architectural diagrams that can be represented either as components or subsystems. When a module is represented as a basic component, its failure behavior is known, and it can be recorded in an IF-FMEA table. Otherwise, the module is rendered as a subsystem, and is further decomposed into architecture of more basic components whose failure behaviors will be also determined using IF-FMEA (see Fig. 2.13).

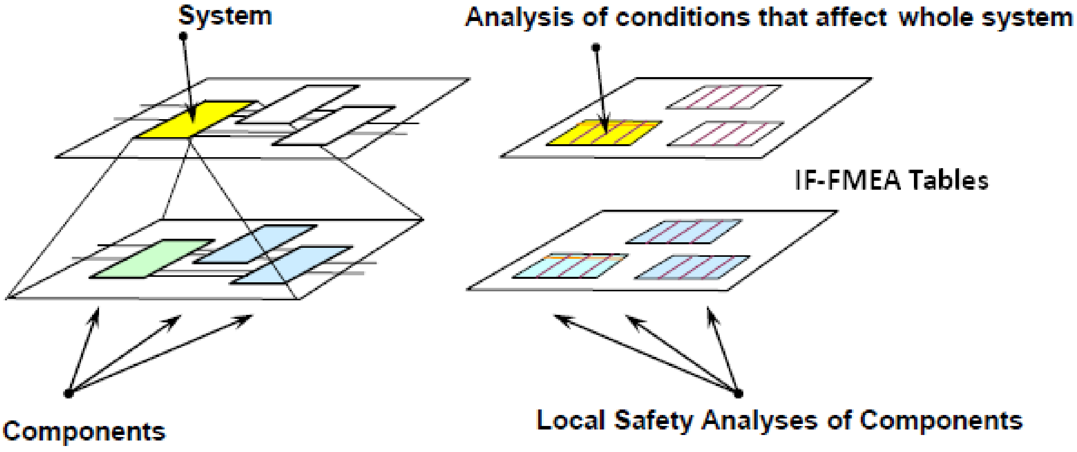
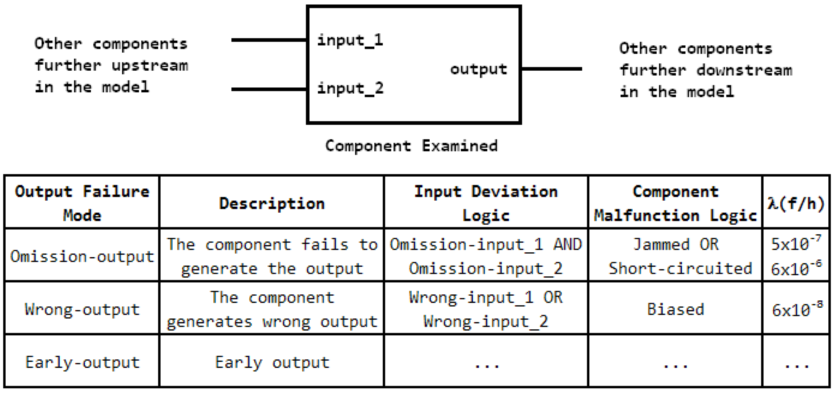

System
Analysis of conditions that affect whole system
IF-FMEA Tables
Components
Local Safety Analyses of Components

Fig 2.13 – Overview of the HiP-HOPS technique

Knowing how the behavior of local (in a lower level) failures of all components is determined, we can determine how the functional failures, which were identified in exploratory analysis of the FHA, arise from combinations of low-level components that have the failure modes identified in IF-FMEA. As illustrated in Fig. 2.14, an IF-FMEA table records how a component reacts to failures generated by other components and set the failure modes of the component itself as well as how the failure spreads to the other components.


Other components further upstream in the model
input_1
input_2
output
Other components further downstream in the model
Component Examined

Output Failure Mode	Description	Input Deviation Logic	Component Malfunction Logic	λ(f/h)
Omission-output	The component fails to generate the output	Omission-input_1 AND Omission-input_2	Jammed OR Short-circuited	$5x10^{-7}$ $6x10^{-6}$
Wrong-output	The component generates wrong output	Wrong-input_1 OR Wrong-input_2	Biased	$6x10^{-8}$
Early-output	Early output	...	...	...

Fig 2.14 - IF-FMEA of a hypothetical component system

The table shown in Fig. 2.14 records five columns of failure based information and a descriptive field: (i) the possible failure modes of a component; (ii) the description of failure (iii) the dependency of such failure modes with respect to the identified failures via its input ports; (iv) what happens upon a certain failure mode occurrence and (v) a failure rate.

The application of this method results in a consistent semi-automatic constructed qualitative safety analysis that determines the origins and global propagation of failure in the system.

2.3.3 Applying HiP-HOPS to an Aircraft System

In this section we show some details on how HiP-HOPS is applied to an aircraft system, which generates a qualitative model of the system. To explain the analysis using this model-based solution, we use a pilot system designed by Embraer that represents a hypothetical Elevator Control System (ECS). Its function is to control the displacement of an electro-hydraulic actuator, according to the longitudinal orientation desired by the pilot.

The implementation of the system is done using the Matlab/Simulink design tool. In this framework, the system is modeled using graphical and block diagrams representations. Also, Matlab's environment variables are accessed to create matrix structures, which are used to represent the tabular annotations of the system failure model. This last resource is not available via a user-friendly graphical interface as default from the Matlab/Simulink tools, however, this tool allows that plug-ins and scripts to be programmed to support and incorporate these features.

System Description

In most aircrafts, the pitching movement (the up-and-down motion of the aircraft’s nose) is controlled by elevator surfaces at the rear of the aircraft as shown in Fig. 2.15.

These surfaces are driven by electrical-hydraulic actuators of the ECS, controlled by the pilot intent. This system is part of the Flight Control Systems (FCS), which commands all flight control surfaces (elevators, ailerons, rudders, etc.) [42].

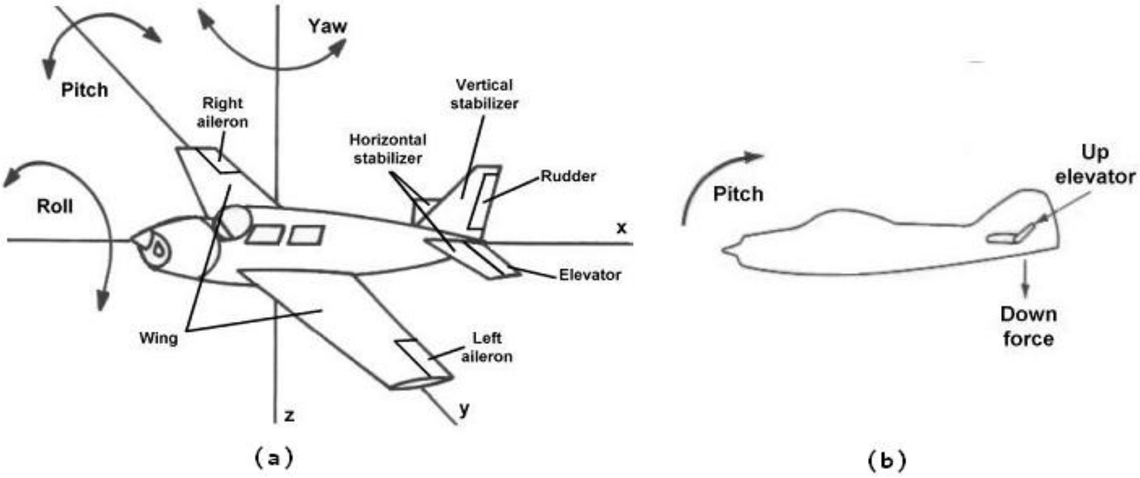


Fig 2.15 - (a) Aircraft control surfaces and axes of motion. (b) Pitch motion.

The ECS has a reference unit (Reference), a device commanded by the user to generate the reference signal which that represents the desired displacement; and a sensor component (Sensor) that converts the displacement of an actuator in an electric signal. The Actuator Control Electronics (ACE) device (Controller) processes three signals: from the reference, from the sensor, and a third one that computes the actuation signal. All of these components have an input of electrical power supply. The power supply is provided by two power sources (PowerSource1 and PowerSource2) and a monitor box (Monitor). The ACE receives the pilot command (control column), compares this computed command to the actuator output to define the current servo command. Fig. 2.16 shows the block diagram of the ECS expressed in Simulink.

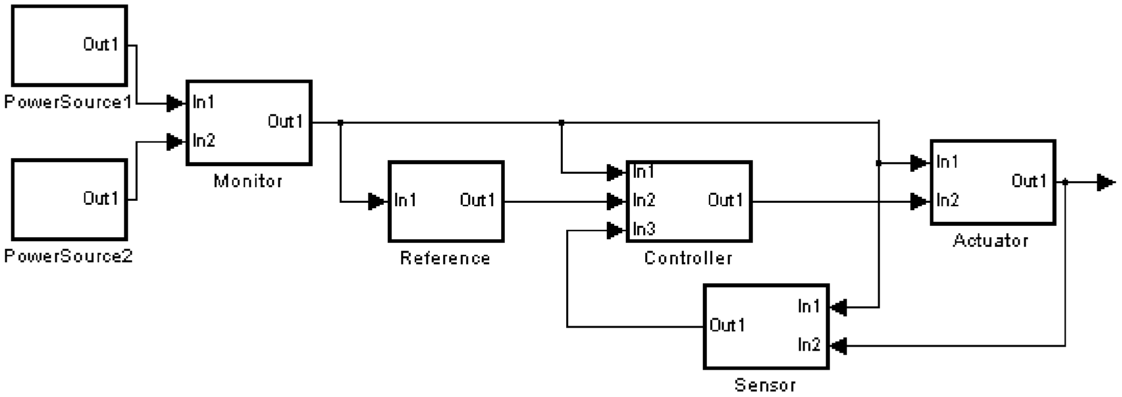


Fig 2.16 - Blocks diagram of the Elevator Control System

After the FHA Analysis of the entire aircraft is performed, the following expressions are a set of significant safety requirements of this system extracted from its FHA analysis:

Omission of speed of the Elevator Control System shall be less than 3.10^{-3} *per flight;*

- *A Wrong Position signal from the Elevator Control System shall be less than* 5.10^{-3} *per flight;*
- *Commission of speed of the Elevator Control System shall be less than* 3.10^{-3} *per flight.*

In normal operation the monitor box receives electrical power from both power sources and makes it available to the other components. In a case of internal failure of one power source (a short circuit for example), the monitor is capable of switching to receive power from the remaining power source. We consider here that the loads do not affect the power sources, to simplify the example. This simplification may be done whenever the effects of the loads over the sources can be neglected. The electrical actuator has an input of power supply, an input of control signal that comes from controller and an output of mechanical displacement. The controller has an internal component that processes the signal from reference (Component1 in Fig. 2.17), another that processes the signal from the sensor (Component2), and a third one that computes the actuation signal (Component3).

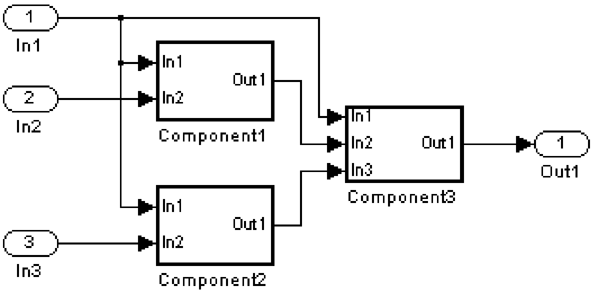


Fig 2.17 - Details of controller subsystem

Performing System Analysis

Since the operation of the system and the function of each of its components are known, it is possible to analyze the failure behavior by performing the IF-FMEA. According to the notation, the module of the architecture can be represented either as components or subsystems. If the failure behavior of a module is known, and it can be recorded in an IF-FMEA table, then the module is represented as a basic component. In the opposite case, the element is rendered as a subsystem, and is further decomposed into an architecture of more basic components, where failure behavior of each can be determined using IF-FMEA. For instance, the controller module is represented as a subsystem, because it contains internal components that have a failure behavior and must be taking into account for the analysis. Next we describe the failure logic of the Monitor.

The monitor has two inputs and one output. Each input of this component receives power from a power source, while the output provides power to the other components. For each input of the monitor, the only important deviation (failures that that can be displayed at the output port of a component/system and propagated to other components/systems) is a low level of power, insufficient for operation of the components. Consequently, in the output the only relevant deviation would be the low level of power. Considering the function of the monitor (detect a low level from one of the inputs) and switching to the remaining power source, and assuming that its detection mechanism does not fail, the only relevant failure that may occur internally is the failure of the monitor to switch the source. In this case, the monitor would not be able to supply power to the other components in a level above the minimum required. From the behavior of this component and the simplifications assumed, it can be concluded that a low level of power at the output of the monitor would occur when at least one of the following events below occur:

1. Both inputs have low level of power simultaneously;
2. One input has low level of power and the monitor fails to switch to the other input.

This analysis results in the following boolean equation:

$$\textit{LowPower-Monitor.Out1} = (\textit{SwitchFailure AND (LowPower-Monitor.In1 OR LowPower-Monitor.In2)) OR (LowPower-Monitor.In1 AND LowPower-Monitor.In2)} \quad (1)$$

Note that the term *SwitchFailure* is the failure mode of the Monitor, where its failure rate is 5.10^{-4} (that is, Median Time Between Failures - MTBF = 2,000 flying hours). The failure rates associated with each failure mode can be defined by manufacturers' specification, historical data, similar components or even specialist judgment [2]. The resulting IF-FMEA table of the component Monitor is described in Table 2.1.

Table 2.1 IF-FMEA Table of the monitor component

Output Failure Mode	Description	Input Deviation Logic	Component Malfunction Logic	λ (f/h)
LowPower-Monitor.Out1	The component is not able to supply power	LowPower-Monitor.In1 OR LowPower-Monitor.In2	SwitchFailure	5.10^{-4}
LowPower-Monitor.Out1	The component is not able to supply power	LowPower-Monitor.In1 AND LowPower-Monitor.In2	-	-

An electrical actuator usually has an electric motor, an electronic driver that controls the motor, and may also have a mechanism or transmission between the motor shaft and the output of displacement. In the power source input it can be considered as a deviation a low level of power. In the signal input, the deviations that could occur are the omission of control signal and an erroneous control signal. In the actuator output the deviations are the omission of speed (movement), erroneous position of the output displacement, and a non-commanded (by the user) speed. A level of power in the supply input below the minimum required for operation would render the motor inoperative. The same effect occurs when the control signal is omitted. Both deviations would cause an omission in the output of the actuator.

Other possible causes for the omission of output could be the failure of the electronic driver, the failure of the electric motor and the jamming of the mechanism/transmission. A wrong output displacement could occur due to an erroneous control signal, or an internal degradation of the driver, or also due to a worn mechanism/transmission. A non-commanded movement of the actuator could occur due to a degradation of the driver, or a non-commanded input signal. This non commanded behavior is called here as "commission". The following equations define the failure logic of the actuator and its corresponding tabular structure is described in Table 2.2.

$$\textit{OmissionSpeed-Actuator.Out1} = \textit{LossOfDriver OR LossOfMotor OR MechanismJamming OR LowPower-Actuator.In1 OR OmissionSignal-Actuator.In2} \tag{1}$$

$$\textit{WrongPosition-Actuator.Out1} = \textit{MechanismDegradation OR DriverDegradation OR CorruptedSignal-Actuator.In2} \tag{2}$$

$$\textit{CommissionSpeed-Actuator.Out1} = \textit{DriverDegradation OR CommissionSignal-Actuator.In2} \tag{3}$$

Table 2.2 IF-FMEA table of the actuator component

Output Failure Mode	Description	Input Deviation Logic	Component Malfunction Logic	λ (f/h)
OmissionSpeed-Actuator.Out1	The component fails to generate speed signal	LowPower-Actuator.In1 OR OmissionSignal-Actuator.In2	LossOfDriver OR LossOfMotor OR MechanismJamming	1.10^{-4} 1.10^{-3} 1.10^{-3}
WrongPosition-Actuator.Out1	The component generate a wrong signal	CorruptedSignal-Actuator.In2	MechanismDegradation OR DriverDegradation	1.10^{-3} 1.10^{-4}
CommissionSpeed-Actuator.Out1	The component is not able to supply the speed signal	CommissionSignal-Actuator.In2	DriverDegradation	1.10^{-4}

This table records the synthesis of the deviations present in each component. It contains the logic of failures propagation established in terms of input-output connections between components. For conciseness, only the monitor and actuator examination is shown here. The other components are relatively straightforward (follow the same principle). Table 2.3 lists the respective equations for all the components of the system and the tolerable probability of each deviation which needs to be evaluated.

Table 2.3. Set of deviation for the Elevator Control System

Component	Deviation	Criticality	Port	Annotation
PowerSource	LowPower		Out_1	PowerSourceFailure
Monitor	LowPower		Out_1	(SwitchFailure and (LowPower-In1 or LowPower-In2)) or (LowPower-In1 and LowPower-In2)
Reference	Omission Signal		Out_1	ReferenceDeviceFailure or LowPower-In1
	Corrupted Signal		Out_1	ReferenceDeviceDegradation
Component1	OmissionSignal		Out_1	LossOfComponent1 OR LowPower-In1OR OmissionSignal-In2
	CorruptedSignal		Out_1	Component1Degradation OR CorruptedSignal-In2
Component2	OmissionSignal		Out_1	LossOfComponent2 OR LowPower-In1 OR OmissionSignal-In2
	CorruptedSignal		Out_1	Component2Degradation OR CorruptedSignal-In2
Component3	OmissionSignal		Out_1	LossOfComponent3 OR LowPower-In1 OR OmissionSignal-In2 OR OmissionSignal-In3
	CorruptedSignal		Out_1	Component3Degradation OR CorruptedSignal-In2 OR CorruptedSignal-In3
	ComissionSignal		Out_1	Component3Degradation
Sensor	OmissionSignal		Out_1	SensorFailure OR LowPower-In1
	CorruptedSignal		Out_1	SensorDegradation
Actuator	OmissionSpeed	3.10^{-3}	Out_1	LossOfDriver OR LossOfMotor OR MechanismJamming OR LowPower-In1 OR OmissionSignal-In2
	WrongPosition	5.10^{-3}	Out_1	MechanismDegradation OR DriverDegradation OR CorruptedSignal-In2
	ComissionSpeed	3.10^{-3}	Out_1	DriverDegradation OR ComissionSignal-In2

This table states that a PowerSource can exhibit a LowPower deviation via its Out1 port when a PowerSourceFailure (a boolean condition) occurs. A more complicated situation occurs in the Monitor. A LowPower can also occur but its origin can be internal (SwitchFailure and one of the connected power sources also failed) or external (both power sources have failed). An OmissionSignal deviation can be exhibited in the Reference when an internal (ReferenceDeviceFailure) or external (LowPower via its In1 port) failure occur. Reference still can exhibit a CorruptedSignal deviation when a ReferenceDeviceDegradation occurs. The controller annotation is not set because it contains subcomponents and consequently only its subcomponents are described. Finally, the Elevator can exhibit the three deviations, whose annotations have already been shown. These last deviations are special because they represent the failure conditions associated for this system. Based on its severity, a tolerable probability is considered. Also, to capture the organization and component interconnections shown in Fig. 2.15 and Fig. 2.16 in tabular form, we use a topology table (see Table 2.4).

Table 2.4. Topology table of the Elevator Control System

Component	Hierarquical Division	Port	Connected or Associated Port
Monitor	No	In_1	PowerSource1-Out_1
		In_2	PowerSource2-Out_1
Reference	No	In_1	Monitor-Out_1
Controller	Yes	In_1	Monitor-Out_1
		In_2	Reference-Out_1
		In_3	Sensor-Out_1
		Out_1	Controller/Component3-Out1
Sensor	No	In_1	Monitor-Out_1
		In_2	Actuator-Out_1
Actuator	No	In_1	Monitor-Out_1
		In_1	Controller-Out_1
Component1	No	In_1	Controller-In_1
		In_2	Controller-In_2
Component2	No	In_1	Controller-In_1
		In_2	Controller-In_3
Component3	No	In_1	Controller-In_1
		In_2	Component1-Out_1
		In_3	Component2-Out_1

Once all the system components are analyzed, their failure behavior are completely and correctly annotated, the architectural information is registered, and the failure conditions are chosen, it is possible to generate a mathematical model referenced in ARP 4671 (such as Markov Analysis or FTA) to evaluate a quantitative analysis in the system.

2.3.4 Support for Quantitative Analysis

The main advantage of a model-based approach like HiP-HOPS is its ability to generate and analyze a qualitative model of a system using a design tool such as Simulink. The reason to use this tool as a design environment is that Simulink models are already used in practice and exert a helpful role in the design of programmable systems during all the system development process (see Fig. 2.1). In the early stages of the design, for example, such models help to define and validate through simulation the functional structure of the system. In later stages, they serve as a basis for modeling non-functional (timing behavior, for example) aspects and for the automatic generation of code, which can be usefully employed in the effective implementation of the system.

Like HiP-HOPS, most model-based proposals are mainly based on FHA and FMEA (and in particular in its extended version, IF-FMEA). IF-FMEA is of particular interest because its tabular structure (see Fig. 2.12) is very useful to capture the transformation and propagation of failures in a system, allowing that complex systems are modeled in a compositional way.

The component failure characterizations (IF-FMEA tables) can be overlaid over the system model as well as the failure conditions and its tolerable rates, identified during the FHA analysis. They can also be included in a tabular structure [2, 5] and is easily incorporated into the system model, using annotations, as described in Fig. 2.18.

Although a qualitative model can address the aspects that refer to the characterization of the behavior of different failures (that may result in a top level hazard), it is also necessary to perform reliability predictions for system components that may cause or contribute to this hazard. This analysis is essential to quantitatively validate the safety constraints of the proposed model. This can be supplied by constructing a mathematical model (a set of equations based on the logic derived from the qualitative model), and calculating the probability of the undesired event over an exposure time.

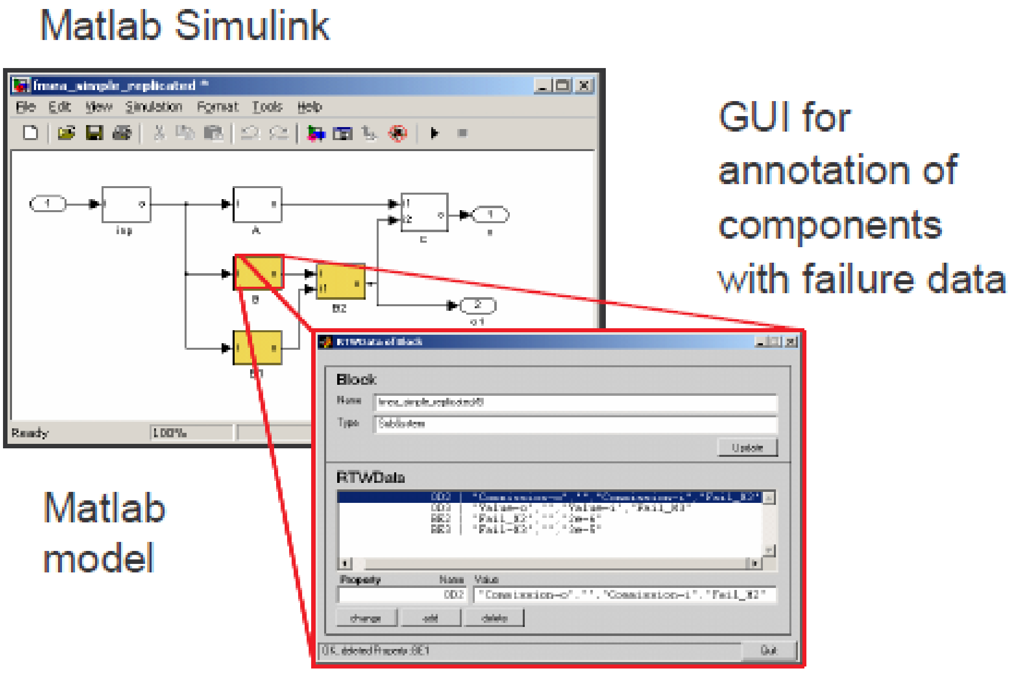


Fig 2.18 – Simulink diagram and a GUI for annotation of components with failure data [70]

As a result, this qualitative model, which represents the failure logic model of the system, recommends some methods of the ARP 4671 (FTA, Markov Analysis, DD) to be applied to provide a quantitative analysis of the model. This type of analysis can be applied iteratively during design, allowing a consistent and continuous assessment of the models as they naturally evolve in the course of the design life-cycle. For instance, HiP-HOPS allows the automatic synthesis of system fault trees from the qualitative model (annotated with appropriate descriptions of component failures and their local effects) of the system (see Fig. 2.19). The Safety Argument Manager (SAM) is one of the tools that support this technique [5].

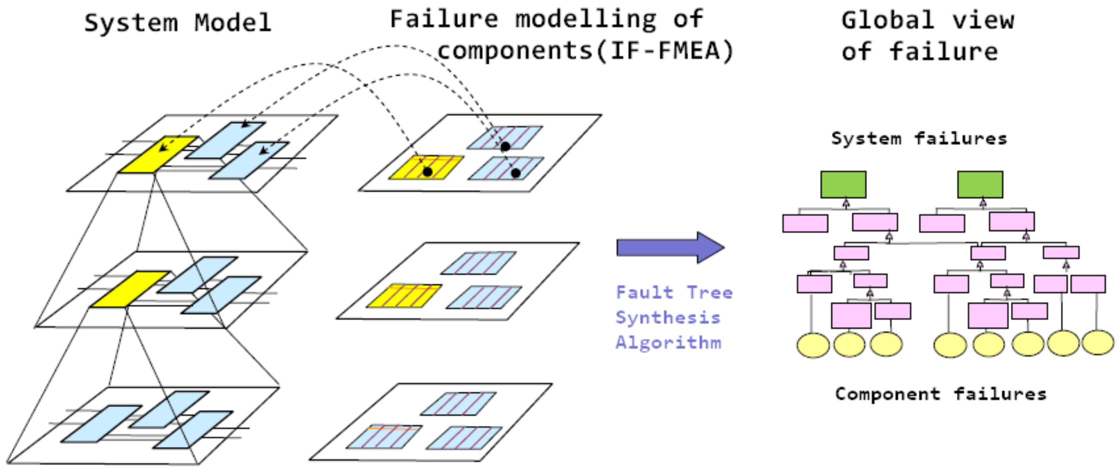

System Model
Failure modelling of components(IF-FMEA)
Global view of failure
System failures
Fault Tree Synthesis Algorithm
Component failures

Fig 2.19 – A overview of model-based synthesis of fault trees using HiP-HOPS

Chapter 3

Probabilistic Model Checking and Prism

Model checking [19] is a well-established and widely used automatic technique for verifying the properties (requirements) of systems. Recently, model checking has become increasingly present in the industry, with large companies such as NASA and Airbus investing considerable resources in this area.

This technique requires two inputs: a description of the system in some high-level modeling formalism (such as process algebra or a Petri net [44]), and the specification of the desired properties of that system, usually in temporal logic such as CTL (Computation Tree Logic) or LTL (Linear-time Temporal Logic).

From the formal description of the system (known as the formal specification), a precise mathematical model is constructed, which typically defines the set of all possible states of the system and the transitions that can occur between these states. And with the formal properties (requirements stated in temporal logics for instance), a model checker can automatically determines whether or not each property is satisfied via a systematic and exhaustive exploration of the model. In the case of a violation, a counterexample is often generated: an explicit trace (sequence of states and corresponding transitions) of the system's behavior that illustrates why such a property was not satisfied.

Probabilistic model checking is a variant which permits automatic formal verification of systems involving stochastic behavior. Several systems can be analyzed by probabilistic model checking, particularly those involving unreliable or unpredictable processes, such as fault-tolerant systems or communication networks, and randomized algorithms [46].

As in the non-probabilistic case, this technique involves constructing, from a description in some high-level formalism, a finite-state model of a real-life system. But in probabilistic model checking, the models are augmented with quantitative information regarding the likelihood that transitions occur as well as the time to perform a transition. Models can be also endowed with labels in states and transitions, for example to describe propositional characteristics of the state or expected costs. In practice, these models are typically Markov chains or Markov decision processes (MDP). In this chapter, we focus on continuous-time Markov chains (CTMC), in which transitions between states are assigned (positive, real-valued) rates, interpreted as the rates of negative exponential distributions.

Properties of probabilistic modeled systems are now quantitative in nature and stated in a variant of temporal logic able to describe such a quantitative aspect of a system. These probabilistic extensions allow specification of properties such as: "the system eventually shuts down with probability at most 0.001”; or “what is the long-run probability that an adequate number of sensors are operational?”. Probabilistic specification formalisms include PCTL [47], a probabilistic extension of the temporal logic CTL applicable for MDP, and the logic CSL [48, 49], a temporal language based on CTL and PCTL that is used for CTMC models.

In particular, CSL includes the means to express both transient and steady-state performance measures of CTMC. Transient properties describe the system at a fixed real-valued time instant t, whereas steady-state properties refer to the behavior of a system in the “stationary state”.

A probabilistic model checker applies algorithmic techniques to analyze the state space of the probabilistic model and determine whether such properties are satisfied. A typical probabilistic model checker uses operations based on graph-based analysis and solution of linear equation systems or linear optimization problems.

In the remainder of this chapter we present an introduction to CTMC and the Prism language and model checker, which provides support for probabilistic model checking of CTMC models using CSL. Furthermore, we also describe how to specify an aircraft system using the Prism specification. For further details about the probabilistic model checking and Prism, please refer to [8, 46, 50].

3.1 Probabilistic (Stochastic) Models

In some scenarios, it is impossible to describe a system by deterministic models. However, there are theoretical results that allow modeling such systems by means of stochastic processes.

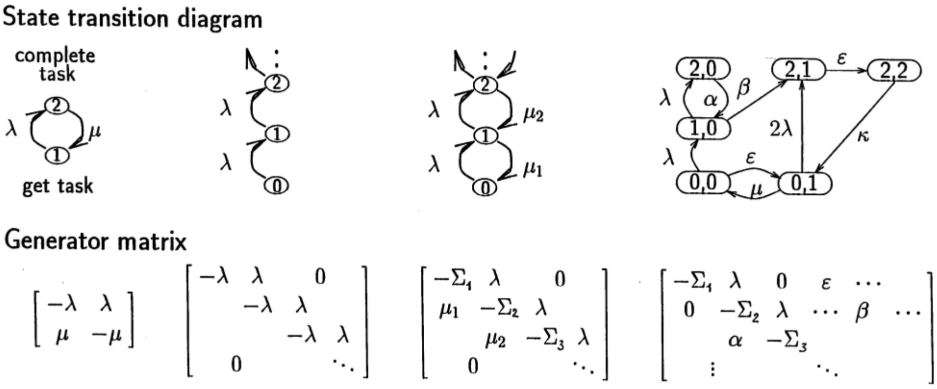
The dynamic behavior of the possible failures of a system, for example, can be modeled by some fundamental concepts of statistics and probability theory. The uncertain outcome of an event is captured by a random variable. Random variables are characterized and distinguished by their distribution function. Furthermore, a stochastic process allows one to describe a sequence of related events. The class of Continuous Markov processes is of special interest here. All its concepts are summarized as follows and, for more details see [51].

3.1.1 Continuous-Time-Markov Chains

A stochastic process is considered a Markovian process if the conditional probability of any future event, depends only of the present state, regardless of past events. This type of stochastic process is also called a *memoryless process*, because the past is ignored. This is a feature naturally present in electro-electronic components in the case of functions that represent its operational performance over its lifetime [51].

Moreover, a Markovian process is considered a Markov chain only if the random variables are defined in terms of a space of discrete states. When time is discrete, the Markov chain is called a Discrete Time Markov Chain (DTMC). In the continuous time we have the Continuous Time Markov Chain (CTMC) which is characterized by discrete states and exponential distribution time that determines the rate of the transition for each state.

The Markov chain standard representation is given by a state transition diagram, suitable for graphical representation, or a transition matrix, used for calculations. The transition states diagram shows the number of possible states and transition rates between them. Figure 3.1 shows some examples of state transitions diagrams and its correspondent generator matrix.



$$\begin{bmatrix} -\lambda & \lambda \\ \mu & -\mu \end{bmatrix} \quad \begin{bmatrix} -\lambda & \lambda & & 0 \\ & -\lambda & \lambda & \\ & & -\lambda & \lambda \\ 0 & & & \ddots \end{bmatrix} \quad \begin{bmatrix} -\Sigma_1 & \lambda & & 0 \\ \mu_1 & -\Sigma_2 & \lambda & \\ & \mu_2 & -\Sigma_3 & \lambda \\ 0 & & & \ddots \end{bmatrix} \quad \begin{bmatrix} -\Sigma_1 & \lambda & 0 & \varepsilon & \cdots & & \\ 0 & -\Sigma_2 & \lambda & \cdots & \beta & \cdots & \\ & \alpha & -\Sigma_3 & & & & \\ \vdots & & & \ddots & & & \end{bmatrix}$$

Fig 3.1 – Examples of Markov state transition diagrams and its correspondent generator matrix

Formally, a CTMC is defined by a finite set of states S and a transition rate matrix $\boldsymbol{R}: S \times S \rightarrow \mathbb{R}_{\geq 0}$, where a positive rate $\lambda = \boldsymbol{R}(s, s')$ between two states s and s' denotes that the probability of moving from s to s' is described as a negative exponential distribution $(1 - e^{-\lambda . t}$, where t is the exposed time), with the rate used as the parameter. Typically, in a state s, there is more than one state s' for which $\lambda > 0$. This is known as a race condition, because a race between the outgoing transitions from s occurs and the first transition to be triggered determines the next state. So, the probability of moving from state s to s' in a single step is the probability that this transition is enabled first (that is, the delay of this transition finishes before the delays of all other transitions leaving s). In the context of reliability, the transition rates represent parameters such as failure rates and repair rates of the system.

To illustrate the operation of Markov chains, we consider, for example, the simplest case of a system with two possible states: operating system (available) and failure (system unavailable). In this case, transitions between these states could represent the failure and repair processes to which the system is subjected. Therefore, the dynamic behavior of the system can be regarded as a sequence of states of the system as time evolves. Thus, in Fig. 3.2, we can see a system consisting of two components in parallel, which is operational when at least one of these components is working.

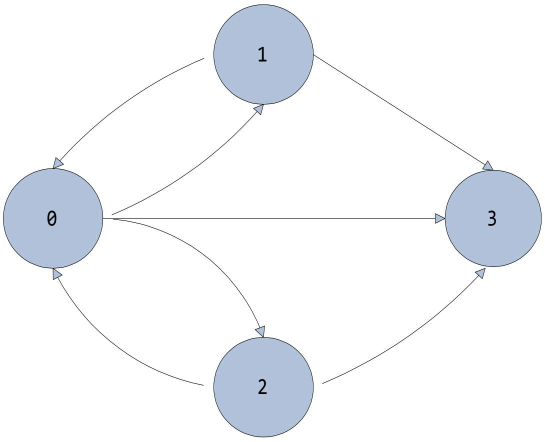


Fig 3.2 - Markov diagram of two components in parallel

As each component has two states, the system in parallel has four possible states, as shown in Table 3.1:

Table 3.1. Topology table of the Elevator Control System

System state	State of the component 1	State of the component 2
0	operational	operational
1	failure	operational
2	operational	Failure
3	failure	Failure

Markov chains are not limited to sequential structures. As shown in Fig. 3.2, multiple transitions can occur from a state. The model enables a direct transition from the state 0 to state 3. Within the context of reliability, this transition could represent the simultaneous failure of two components (due to a common cause failure of components), resulting in immediate unavailability of the system. Thus, there is the possibility of characterizing both independent failures and dependent failures while the system is in state 0.

The representation of behavior of Markovian processes is captured by the system called the Chapman-Kolmogorov equation for the probability of transition. The solution of the equation gives the probability of the unconditional state (determining the probability of a state without depending of the probability of others). This temporary solution is very significant when the system under investigation must be evaluated with respect to its behavior in short term.

Assuming a long term, however, it can be shown that the state probabilities often converge to constant values. These stationary state or equations of equilibrium can be derived from the system of differential equations that expresses the appearance and disappearance of a state ***s*** relative to other states, through a statistical equilibrium [16, 51].

CTMC can be analyzed using two traditional properties: transient behavior, which considers the state of the model at a particular time instant; and steady-state behavior, which describes the state of the CTMC in the long-run. The transient probability $\pi_{s,t}(s')$ is defined as the probability, having started in state s, of being in state s' at time instant t. The steady-state probability $\pi_s(s')$ is the probability of, having started in state s, being in state s' in the long-run, that is, in the equilibrium state of the system. The steady-state probability distribution, that is, the values $\pi_s(s')$ for all $s' \in \boldsymbol{S}$, can be used to infer the percentage of time, in the long-run, that the CTMC spends in each state.

3.2 Prism

Prism [52, 53] is a formal probabilistic analysis tool developed by the University of Birmingham. It accepts probabilistic models described in a simple, high-level modeling language. Prism enables the analysis of Markov models specified in discrete time (DTMC), continuous (CTMC), and Markov decision processes (MDP). The verification of the specified properties in the model is made with the aid of the temporal logic language PCTL for DTMC and MDP models and CSL for CTMC.

The choice of model to be specified (DTMC, CTMC and MDP) will depend on the nature of the system to work. DTMC provides a relatively simple model for systems where the exact probability of different behaviors for a sample of discrete time is known. MDP contains DTMC and adds support for non-determinism, which can be used to model competition between processes running in parallel or for specifications whose exact values of some system parameters are unknown. CTMC extends DTMC by allowing transitions to occur in real-time (using exponential distributions), rather than only in discrete steps.

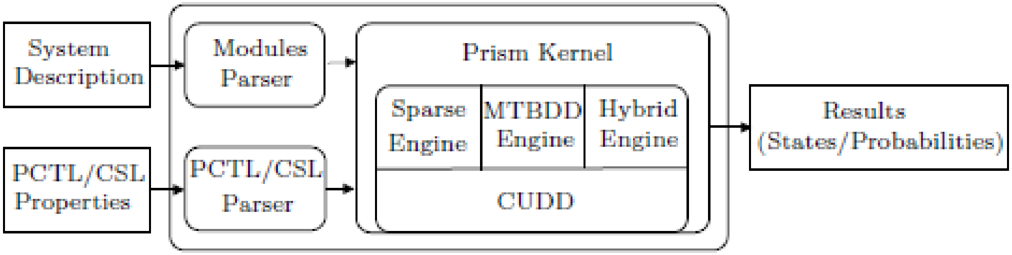
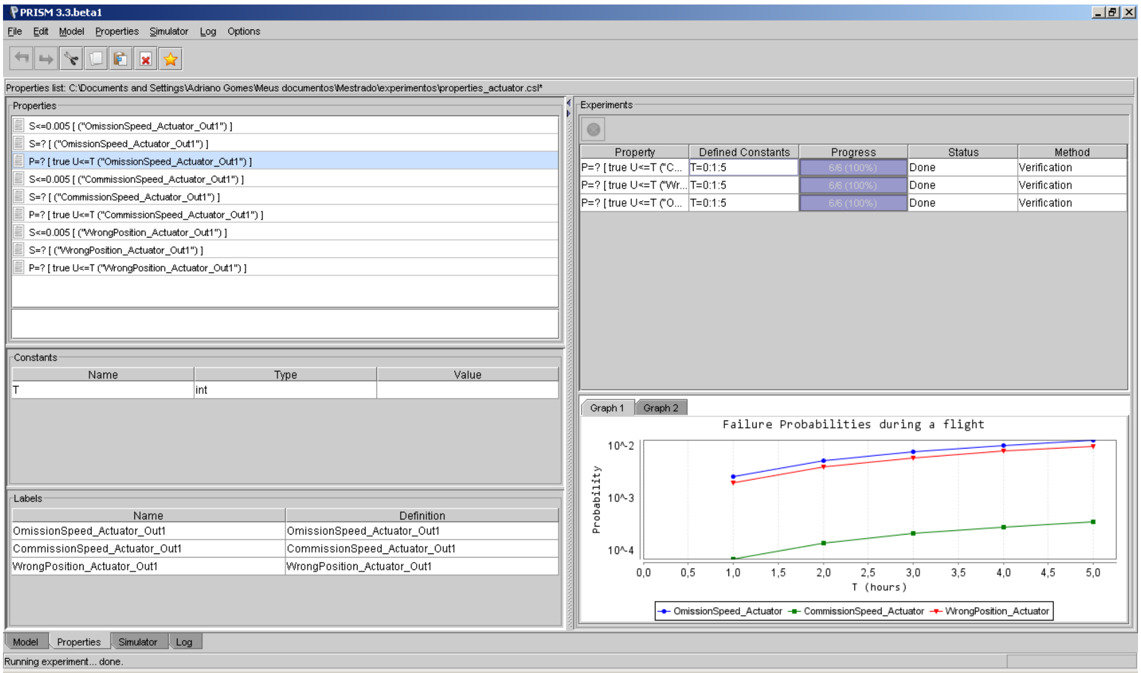


Fig 3.3 – The structure of Prism [52]

Fig. 3.3 illustrates how this tool acts: first, it reads and analyzes a system's description written in Prism, then builds the corresponding representation in DTMC, CTMC or MDP, calculates the set of all reachable states, and identifies any deadlock states (that is, absorbing states). If necessary, the transition matrix from the constructed probabilistic model can be exported for use in another tool such as Matlab [9] or MRMC [37]. Then Prism analyzes one or more properties in PCTL or CSL determining if the model satisfies each property.

The underlying data structures in Prism are BDD (Binary Decision Diagram) [50] and MTBDD (Multi-Terminal Binary Decision Diagram) [62]. However the tool provides three different engines that can be used for numerical computation (a conventional explicit version using sparse matrices, a pure MTBDD-based implementation, and hybrid approach of both).

The tool is implemented in a combination of Java and C++. The high-level parts of the tool, such as the user interface and parsers, are written in Java. The low-level libraries are written in C++ and the CUDD package [61], which is written in C, enabling the use of BDD and MTBDD. Prism is a free, open source application that can be downloaded from its website [53]. It is available from either a command-line or a graphical user interface. The graphical user interface provides a built-in text-editor for the Prism language, an editor for Prism properties, tools for plotting of graphs and a simulator tool for exploring and debugging Prism models (see Fig. 3.4).



Properties list: C:\Documents and Settings\Adriano Gomes\Meus documentos\Mestrado\experimentos\properties_actuator.csl*

```
S<=0.005 [ ("OmissionSpeed_Actuator_Out1") ]
S=? [ ("OmissionSpeed_Actuator_Out1") ]
P=? [ true U<=T ("OmissionSpeed_Actuator_Out1") ]
S<=0.005 [ ("CommissionSpeed_Actuator_Out1") ]
S=? [ ("CommissionSpeed_Actuator_Out1") ]
P=? [ true U<=T ("CommissionSpeed_Actuator_Out1") ]
S<=0.005 [ ("WrongPosition_Actuator_Out1") ]
S=? [ ("WrongPosition_Actuator_Out1") ]
P=? [ true U<=T ("WrongPosition_Actuator_Out1") ]
```

Name	Type	Value
T	int	

Name	Definition
OmissionSpeed_Actuator_Out1	OmissionSpeed_Actuator_Out1
CommissionSpeed_Actuator_Out1	CommissionSpeed_Actuator_Out1
WrongPosition_Actuator_Out1	WrongPosition_Actuator_Out1

Property	Defined Constants	Progress	Status	Method
P=? [true U<=T ("C...	T=0:1:5	6/6 (100%)	Done	Verification
P=? [true U<=T ("Wr...	T=0:1:5	6/6 (100%)	Done	Verification
P=? [true U<=T ("O...	T=0:1:5	6/6 (100%)	Done	Verification

Running experiment... done.

Fig 3.4 - Screenshots of the PRISM tool running

3.2.1 Prism Modeling Language

A large range of formalisms have been proposed for specifying probabilistic models. These include stochastic variants of process algebras (such as Probabilistic CSP [54], PMaude [55]), Petri nets [44, 56], stochastic activity networks [57] and many others. Nowadays, Prism is one of the most prominent formalism, because it provides a simple, textual modeling language, based on the concept of reactive modules formalism defined by Alur and Henzinger [45]. It is the only formalism that specifies and analyzes all these variations using efficient and viable techniques of representation of states that allows modeling larger systems than the other formalisms.

In this section, we present a brief introduction to the Prism language. It offers a solid way of describing all model types (DTMC, CTMC and MDP) supported by the tool. For further details about the Prism language and its semantics, see the Prism documentation and case study at [53].

Modules, variables and commands

Modules and variables are the basic components of this language and the system is built as a parallel composition of the declared modules. Its datatypes include: integers, reals and booleans and can be declared local or globally. Modules can interact with each other (synchronization) and contain a number of variables that reflect their possible states. Its behavior (the changes between states via quantified transitions) is determined by a list of guarded commands. For a CTMC, a command uses the following syntax:

$$[action]\ \langle guard \rangle \rightarrow rate : \langle update \rangle;$$

Each command (initiated by a [], possibly with an action label inside) is formed of a guard (boolean expression before the symbol →, which is a predicate over the model variables) followed by a rate (an non-negative real-valued expression, where 1 means 100%) and the *update* expression gives the new values of the variables in the module by the following form:

$$(v_1' = u_1)\ \&\ (v_2' = u_2)\ \&\ \ldots\ \&\ (v_k' = u_k)$$

where $v_1, v_2, \ldots, v_k$ are local variables of the module and $u_1, u_2, \ldots, u_k$ are expressions over all variables. A module can access all the variables of the model, but it can only update its own local variables. The transitions represent which state changes are possible and how often they occur. A simple command for a module with one variable *sensor_sensorfailure* might be:

[](!sensor_sensorfailure) -> (5E-4) : (sensor_sensorfailure' = true);

which states that whether *sensor_sensorfailure* is false, it is changed by one (the *sensor_sensorfailure'* denotes the new value of the variable). In this case, the update of the variable occurs with rate *5E-4* (that is, the delay before this transition is completed is sampled from a negative exponential distribution with parameter *5E-4*).

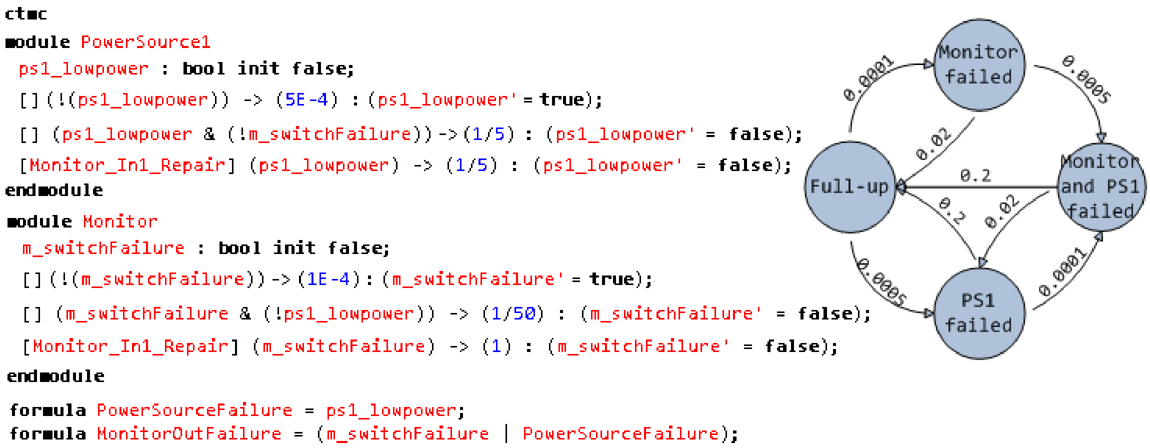
Composing modules

The modules are integrated typically using the standard CSP [59] parallel composition (that is, modules synchronize over all their common actions). Prism also supports other CSP process-algebraic operators (alphabetized parallel, interleaving, etc) that can specify more precisely the synchronization between the modules.

A command (belonging to any of the modules) is enabled in a global state of the probabilistic model whether the actual state satisfies the predicate guard. If a command is enabled, a transition that updates the module's variables can occur with rate. For

CTMC, the choice between which command is performed (that is, the scheduling) depends on the race condition.

The multi-way synchronization provides interactions between multiple modules, that is, simultaneous changes in their states. It is modeled by augmenting guarded commands with action labels that are placed inside the square brackets. We illustrate this with a simple example derived from the Elevator control system described in Chapter 2 (see Fig. 3.5). It implements a `PowerSource` and a `Monitor` unit, whose two of the repair update situations are defined by synchronized commands. For example, the last command of both modules is labeled with `Monitor_In1_Repair` name, because if both components fail, they must be repaired simultaneously, because the `PowerSource` is monitored by the `Monitor`.



```
ctmc
module PowerSource1
 ps1_lowpower : bool init false;
 [](!(ps1_lowpower)) -> (5E-4) : (ps1_lowpower' = true);
 [] (ps1_lowpower & (!m_switchFailure))->(1/5) : (ps1_lowpower' = false);
 [Monitor_In1_Repair] (ps1_lowpower) -> (1/5) : (ps1_lowpower' = false);
endmodule
module Monitor
 m_switchFailure : bool init false;
 [](!(m_switchFailure))->(1E-4):(m_switchFailure' = true);
 [] (m_switchFailure & (!ps1_lowpower)) -> (1/50) : (m_switchFailure' = false);
 [Monitor_In1_Repair] (m_switchFailure) -> (1) : (m_switchFailure' = false);
endmodule
formula PowerSourceFailure = ps1_lowpower;
formula MonitorOutFailure = (m_switchFailure | PowerSourceFailure);
```

Fig 3.5. System representation using Prism

This `Monitor_In1_Repair` action is used to force two modules to make transitions simultaneously. For example, in the state (`ps1_lowpower` = true and `m_swithcFailure` = true), the composed model can move to state (`ps1_lowpower` = false and `m_swithcFailure` = false), synchronizing over the `Monitor_In1_Repair` action. The rate of a synchronous transition is defined as the product of the individual rates. In this example, there is only one initial state, but Prism allows the specification of a set of initial states. Therefore, we can define formulas that can be used as shorthand for the expressions.

3.2.2 Property Expressions

In the Prism Model Checking, verification properties are interpreted in a similar way to non-probabilistic case, in which a formula containing temporal expressions can typically returns all executions that satisfy a certain property, or that there is an execution that satisfies it. In this section, we use the temporal logic CSL (Continuous Stochastic Logic) which is designed for specifying properties of CTMC specifications.

The two principal operators in the Prism property specification language are the P (probabilistic) and S (steady-state) operators. P allows one to reason about the probability that executions of the system satisfy some property. For instance, the formula $P \leq 5e^{-4}$ *[* $F^{[t;t]}$ *!MonitorOutFailure]* checks if the probability of the instantaneous availability of the system is 0.0005 or less, the, that is, the probability that it is operational at time instant t. Moreover, the formula $P \geq 1e^{-3}$ *[true* $U^{<=200}$ *“PowerSourceFailure”]* indicates that, with probability 0,001 or greater, the power source component will fail within 200 time units. The operator S deals with the system behavior in the stationary state (long term). The formula $S < 1e^{-3}$ *[“MonitorOutFailure”]* for instance, says that in the long term, the probability that the output port of the monitor does not exhibit a failure is less than 0.001.

Hence, the satisfaction of a property (that is whether it is true or false) is defined for a single state of a model. When analyzing a property, Prism considers it to be true if it is satisfied in all states of the model, and false otherwise.

In Prism, we can also directly specify properties that evaluate to a numerical value. This is achieved by replacing the probability bounds from the P and S operators with =?. Thus, we can write an expression of the form *P =? [*$F^{[0;600]}$ *!PowerSourceFailure]*, for which the model checker will return a real probability that the system ends. This formula checks the probability that power source component fails within 600 time units. Moreover, the formula *S =? [num sensors* $\geq$ *min sensors]* checks what is the long-run probability that an acceptable number of sensors are operational.

In many cases, the most useful form of analysis is to compute such values for a range of models or properties. For example, one might determine *P = ? [true U<=T “Monitor_Failure”]* for a range of values of T in order to gain insight into the likelihood of the system terminating as time progresses.

In addition, other properties can be analyzed. Prism models can be augmented by introducing the notion of costs and rewards. The properties state some characterisation about the expected value of these costs/rewards. These are specified using the R operator, which works in a very similar fashion to the P and S operators [7, 8].

3.2.3 Prism Model Checker

Prism is a symbolic model checker and incorporates a range of model analysis techniques [50]. The model construction and reachability are implemented using MTBDD and BDD respectively. The Prism implementation use data structures based on BDD. This offers an important feature for this model checker, because it provides compact representations and efficient manipulation of large probabilistic models to exploit their structure and regularity. Prism also uses MTBDD in combination with a conventional explicit storage scheme such as sparse matrices and arrays in order to store numerical values. The model size capacity of Prism is nearly to 10^7 for CTMC and higher for other types of models.

Reachability analysis using BDD forms the basis of non-probabilistic symbolic model checking. For both PCTL and CSL, model checking generally reduces to a combination of reachability-based computation and the solution of linear equation systems. More specifically, the underlying computation in Prism involves a combination of:

- Graph-theoretical algorithms, for qualitative probabilistic model checking and conventional temporal-logic model checking.
- Numerical computation, for quantitative probabilistic model checking, to provide solution of linear equation systems (for DTMC and CTMC) and linear optimization problems for (MDP).

Graph-theoretical algorithms are always performed in Prism using BDD. They are comparable to a non-probabilistic model checker. For numerical computation, Prism uses iterative methods rather than direct methods due to the size of the models that need to be handled. For transient analysis of CTMC, Prism incorporates an iterative numerical method known as uniformisation or Jensen's method. For solving linear equation systems, it supports a range of well-known techniques, including Jacobi, Gauss-Seidel and SOR (successive over-relaxation) methods. Finally, for the linear optimization problems which arise in the analysis of MDP, Prism uses dynamic programming techniques, in particular, value iteration.

In the case of numerical computation, Prism actually provides three distinct numerical engines. The first is implemented purely in MTBDD; the second uses more conventional data structures for numerical analysis: sparse matrices and full vectors; and the third is a hybrid, using a combination of the two. Typically the sparse engine provides faster numerical computation than its MTBDD counterpart, but it requires more memory. Sometimes, MTBDD can also exploit the models' structure and represent them far more compactly than a sparse matrix. Moreover, in cases where high regularity occurs, MTTB is able to perform quantitative analysis for models extensively larger than those used in a sparse matrix form. Thus, the performance of the tool may vary depending on the choice of the engine. The hybrid engine stores models in a MTBDD structure which is adapted so that numerical computation can be performed in combination with a full vector. It aims to use less memory than sparse matrices, but providing a faster computation than pure MTBDD. By default, PRISM uses the hybrid engine.

3.2.4 Modeling a Simple System using Prism

The system shown in Fig. 3.6 consists of a primary component (Comp1) with continuous failure monitoring, a backup component (Comp2) with no self-monitoring, and an external monitoring component (Monitor) whose function is to monitor the health of the backup component.

The failure rate of Comp1 is $\lambda_1 = 5x10^{-5}$ per hour. The self-monitoring strategy of this component enables its functionality to be verified prior to every flight. (The median time duration of each flight is assumed to be 5 hours). Thus, the repair rate of Comp1 is $\mu_1 = 1/5$ per hour. If this component is failed or inoperative, it is repaired before the next dispatch. The failure rate of Comp2 is $\lambda_2 = 2.5x10^{-5}$ per hour. The backup component has no self-monitoring, but it is monitored continuously by an independent monitor. If the backup system fails and the monitor is working, the backup is repaired before the next dispatch. If the monitor is not working, the backup component can fail latently, because the backup component is checked only every 10 flights (50 hours - $\mu_2 = 1/50$ per hour). If the backup component is failed at one of these scheduled inspections, with no indication of a failure informed from the monitor, it is assumed that the monitor is also failed, so both are repaired prior to the next flight.

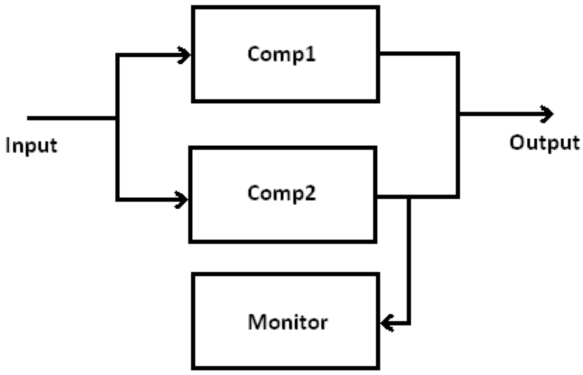


Fig 3.6 - Diagram of a system with an component and backup with an independent monitor

The monitor has a failure rate of $\lambda_3 = 2.5x10^{-5}$ per hour. Whether the monitor is failed, it can be repaired in two different situations. First, as noted above, if the backup component is failed at its periodic 50-hour inspection and there was no monitor indication of a component failure, then the monitor is repaired along with the component before the next dispatch. Second, the monitor is checked periodically every 100 flights (500 hours - $\mu_3 = 1/500$ per hour), and if the monitor is failed, it is repaired prior to the next flight.

As described in this chapter, Prism can be used to analyze the behavior of fault-tolerant systems. Also, it offers an interesting language specification to abstract the mathematical representations of the system. Considering this specification, the method that we use for modeling repairable systems is traditionally called as components approach, because we consider the components individually. In this method it is necessary to know the density functions of failure and repair probability for each component and how they are connected.

The failure model of a repairable system usually includes the reliability of components, system architecture, the physical layout of operation, as well as aspects related to availability, maintainability and maintenance practices used. In aeronautical context, all such information results from the safety analysis are performed in such a system. To generate a stochastic process of a repairable system, the random variables of interest are the median time between failures (MTBF) and median time to repair (MTBR).

Once we perform the analysis of random variables on time between failures and repair times and if it is observed the adequacy of the exponential distribution for both variables, the system can be modeled using Markov models.

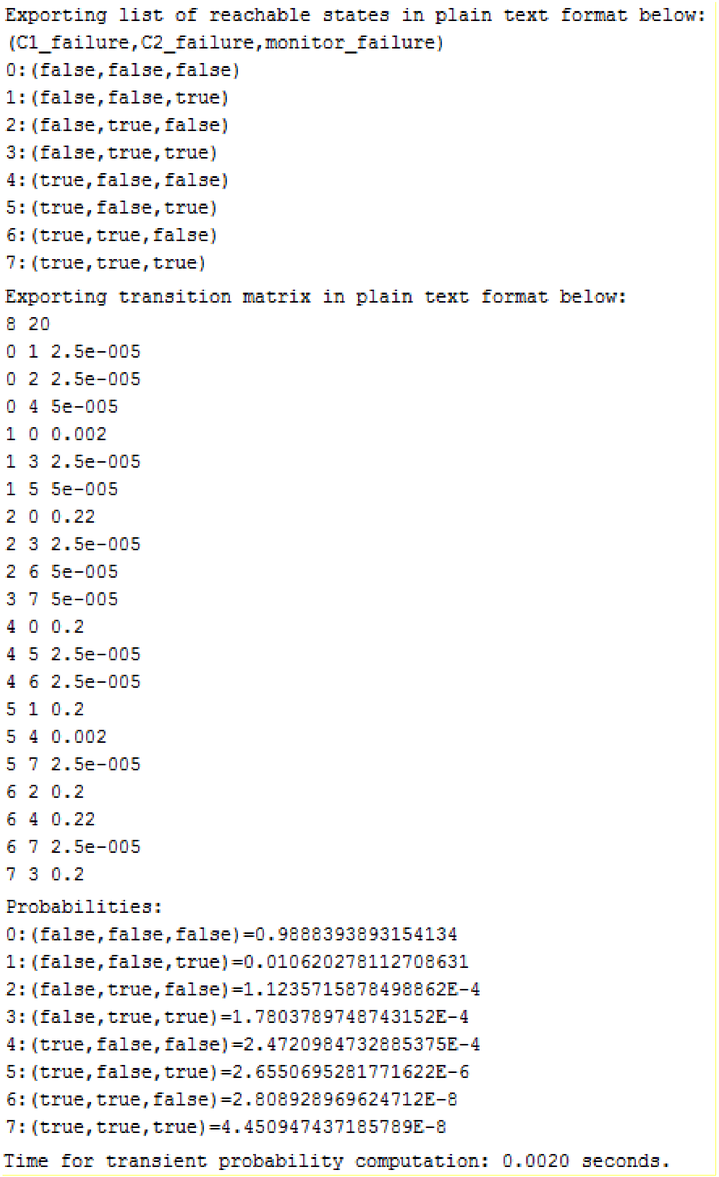
Fig.3.7 illustrates a Prism specification of this system. The first module, `Comp1`, specifies an abstract failure behavior of the Comp1. The variable `c1_failure` represents its single failure mode. The first transition captures one of the possible changes in the failure mode: from an operational state it can fail with a rate of 5e^{-5} (failure/hour). The next command represents a repair transition. Comp2 is the second module, which has different repair transitions command. One of them is synchronized (the labels inside [and] state the synchronization points) with the module Monitor. They work similarly to the first transition of this module, except that they need to synchronize with the corresponding labels of the module Monitor, allowing them to be triggered. The module Monitor also uses a single variable: `monitor_failure`. Its first command states a failure transition command whereas the second represents the capability of its single failure mode being repaired with a rate of 1/500 (repair/hour). The last command represents repair transitions corresponding to the repair transitions of the `Comp2`.

```
ctmc

module Comp_1

  C1_failure : bool init false;

  [](!C1_failure ) -> (5e-5):(C1_failure'=true);

  [](C1_failure ) -> (1/5): (C1_failure'=false);

endmodule

module Comp_2

  C2_failure : bool init false;

  [](!C2_failure ) -> (2.5e-5):(C2_failure'=true);

  [](C2_failure & !monitor_failure ) -> (1/5) : (C2_failure' = false);

  [Monitor_Repair] (C2_failure) -> (1/50):  (C2_failure' = false);

endmodule

module Monitor

  monitor_failure : bool init false;

  [](!monitor_failure ) -> (2.5e-5):(monitor_failure'=true);

  [] (monitor_failure & !(C2_failure) ) -> (1/500) : (monitor_failure' = false);

  [Monitor_Repair] (!monitor_failure ) -> (1) : (monitor_failure' = monitor_failure);

endmodule

formula SystemFailure =  C2_failure & C1_failure;
```

Fig 3.7- Prism specification of a small system

The first line of this specification states that we are considering a continuous time Markov chain that is composed of a set of discrete states, where each of them is the representation of the state (operational, degraded and faulty) of each failure mode (local variables) of a component. This chain of events requires the use of exponential probability distributions for modeling failure mode rates and repairs (this is why we use the CTMC model). Therefore, the model is basically composed of modules, internal variables and instruction of transition states that can be synchronized or not. Also, we use the “formula” operator, which is used to represent the logic of propagation of failure and acts as a variable in the observation on the stage of model verification.

Fig. 3.8 shows the set of states and the transition matrix that represents the respective Markov model of the system as well as the transient probabilities of each state considering an exposition time of 1000 hours. The states 6 and 7 represent the situation of the system failure.



```
Exporting list of reachable states in plain text format below:
(C1_failure,C2_failure,monitor_failure)
0:(false,false,false)
1:(false,false,true)
2:(false,true,false)
3:(false,true,true)
4:(true,false,false)
5:(true,false,true)
6:(true,true,false)
7:(true,true,true)
Exporting transition matrix in plain text format below:
8 20
0 1 2.5e-005
0 2 2.5e-005
0 4 5e-005
1 0 0.002
1 3 2.5e-005
1 5 5e-005
2 0 0.22
2 3 2.5e-005
2 6 5e-005
3 7 5e-005
4 0 0.2
4 5 2.5e-005
4 6 2.5e-005
5 1 0.2
5 4 0.002
5 7 2.5e-005
6 2 0.2
6 4 0.22
6 7 2.5e-005
7 3 0.2
Probabilities:
0:(false,false,false)=0.9888393893154134
1:(false,false,true)=0.01062027811270863l
2:(false,true,false)=1.1235715878498862E-4
3:(false,true,true)=1.7803789748743152E-4
4:(true,false,false)=2.4720984732885375E-4
5:(true,false,true)=2.6550695281771622E-6
6:(true,true,false)=2.808928969624712E-8
7:(true,true,true)=4.450947437185789E-8
Time for transient probability computation: 0.0020 seconds.
```

Fig 3.8 – States, matrix transitions and steady-state probabilities of the small system specification

Chapter 4

Proposed Strategy

This chapter presents a strategy to perform quantitative safety assessment of aircraft systems using probabilistic model checking. The main objective is to use formal models as support for verification and validation of the system safety requirements. The used formal notation is a textual representation of Markov chains (it is called Prism) and is systematically generated from a Simulink diagram, annotated with failure logic, by applying translation rules. Hence in this chapter, we also describe the rules responsible for the systematic formal model generation. An overview of the strategy is described in Section 4.1. Section 4.2 presents the extended tabular notation used in our strategy. The details about collecting and processing the input data for the formal model generation are outlined in Section 4.3. Afterwards, Section 4.4 discusses how to generate the formal model by applying the set of proposed translation rules. Finally, Section 4.5 shows how to perform the quantitative analysis from a Prism model.

4.1 Strategy Overview

Our strategy aims to perform a quantitative analysis over an aircraft system, which is designed using a well-established model-based approach. The quantitative analysis is based on the use of probabilistic formal models. The formal model is specified using the Prism language which is later on verified by model checking. In the Prism analysis, time and probability queries are dealt with in the model checker using the CSL temporal languages. By using the Prism model checker we can detect whether any criticality level condition is violated without building any fault-tree.

Most of the techniques to create probabilistic formal models of aeronautic systems are highly subjective, because they are dependent on the skill of engineers that specify the formal model in a non-systematic ad hoc fashion [6, 18]. But instead of creating a Prism specification implicitly via a tool, we follow a systematic strategy by providing formal translation rules that transform a high-level system description (Simulink diagrams) into a Prism specification. The input information necessary for this strategy comes from a qualitative model constructed during a common safety assessment process, enabling to mechanize the strategy.

Fig. 4.1 presents an overview of our strategy. It starts by collecting the system description, which contains the system block diagrams and a failure logic model. With this information, we apply our translation rules to create a Prism specification and the associated CSL formulas to analyze the safety requirements of the system. Then, the

Prism model-checker is invoked to check all formulas and only when one of them is not satisfied, this is reported to the user.

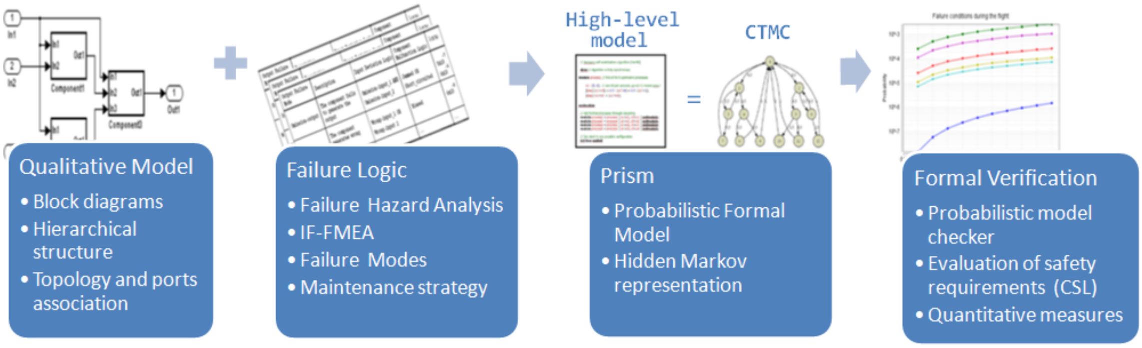

High-level model
CTMC
Qualitative Model
• Block diagrams
• Hierarchical structure
• Topology and ports association
Failure Logic
• Failure Hazard Analysis
• IF-FMEA
• Failure Modes
• Maintenance strategy
Prism
• Probabilistic Formal Model
• Hidden Markov representation
Formal Verification
• Probabilistic model checker
• Evaluation of safety requirements (CSL)
• Quantitative measures

Fig 4.1 Overview of proposed strategy

The generated Prism model has a Continuous-Time Markov Chain (CTMC) representation and captures the failure logic about the system. Using this Prism model, the probabilistic model checker can automatically perform quantitative analysis that can answer several kinds of questions about the system. Hence, using a notation of ease understanding instead of working directly with their Markovian representation, we provide a more user-friendly notation to engineers.

The key idea is to incorporate the support of formal analysis in the process of safety assessment to provide time and probability characteristics, enabling a more dynamic and efficient safety analysis. As result, we describe the system in a high-level specification, capable of providing an efficient quantitative analysis, considering the architectural issues in order to maintain integrity with the usual solution. The translation strategy is divided into the following steps:

- **Extending the tabular notation**: Recall from Sections 2.3.2 and 2.3.3 that we used tabular annotation (IF-FMEA tables) to describe the failure model of a system. In this step these tabular structures are extended to add information about system component repairs and its failures monitoring.
- **Collecting and processing the input data:** The model is generated from its textual and tabular Simulink representation. We organize the data following an abstract syntax, allowing that the translation rules can be applied to generate the Prism specification.
- **Translation Rules**: In this step, the generated structure from the previous step is processed and its respective Prism specification is generated as output according to the semantics given by the translation rules.
- **Quantitative Analysis**: Finally, we show how to analyze the generated probabilistic model using the Prism model checker which is supported by verification of formulas expressed in CSL language.

We describe these steps in the following sections. The details are applied in practice using the case study presented in Chapter 5.

4.2 Extending the tabular notation

Although all tabular structures presented in Section 2.3.3 are consistent and integrated with respect to the system failure mode and propagation, they are not sufficient to create a probabilistic formal model to perform a quantitative analysis using Markov models in a systematic way. To represent aeronautical systems consistently and according to the ARP 4761 and FAR 25.1309, we need more information to model the non-monitored failures of the system. This involves knowing about component's latency (if a component is monitored or not) and how often a repair takes place (Mean Time To Repair [MTTR]). According to the FAR 25.13.09 [24]:

"If one or more failed elements in the system can persist for multiple flights (latent, dormant, or hidden failures), the calculation should consider the relevant exposure times (e.g. time intervals between maintenance and operational checks/ inspections). In such cases the probability of the Failure Condition increases with the number of flights during the latency period"

Thus, we extend this modeling notation (tabular structures) with the addition of such information. As result, a new tabular structure is defined.

The first information to be incorporated is the classification of each basic component of the system about its failures' monitoring. In the aeronautic context, some components are checked before each flight to confirm that they are working, and repaired if necessary. So, this type of component can be called as *self-monitored*, because we need to know if it is working before of each flight. But some aircraft systems include components that are not inspected before and during every flight. Failures in such components are called latents because they are not detected unless another combined failure occurs and compromise a function that needs such components or during scheduled maintenance (generally, after some flights). For this last type of component we must consider two situations: *externally monitored* and *non-monitored* components. The first type of components is monitored continuously by an independent monitor. If the component fails and the monitor is working, the component can be repaired before the next dispatch. If the monitor is not working, latency reappears. The type *monitor* is a particular component responsible for monitoring relevant components. The latter type represents all components that are *not monitored* and naturally they have latent failures. Their faults are only checked in regular periods of maintenance. In short, we need to distinguish between a monitored and non-monitored failure of a component because non-monitored failures are more severe in safety analysis.

Based on reliability predictions and safety factors (dispatchability, MTBF, severity, redundancy, and other several reasons) the periodic inspection/repair intervals for each component is also defined. This is the second information that we added to the input model. Table 4.1 presents a summary of this additional information.

Table 4.1. Definition of the additional information

Maintenance strategy	Inspection Time
Self-monitored Monitored Non-monitored Monitor	It is the maximum exposure time which a component is submitted without inspection or repair. Ex.: 50 hours, 10 flights.

Considering these assumptions, the tabular structure of Section 2.3.3 is extended to store this data. Table 4.2 shows only the additional information.

Table 4.2. Additional information using a tabular notation

Component	Maintenance strategy	External Component	Inspection Time
PoweSource_1	Monitored	Monitor-In1	50 hours
PoweSource_2	Monitored	Monitor-In2	50 hours
Monitor	Monitor		100 hours
Reference	Self-monitored		5 hours
Controller	Self-monitored		5 hours
Sensor	Self-monitored		5 hours
Actuator	Self-monitored		5 hours

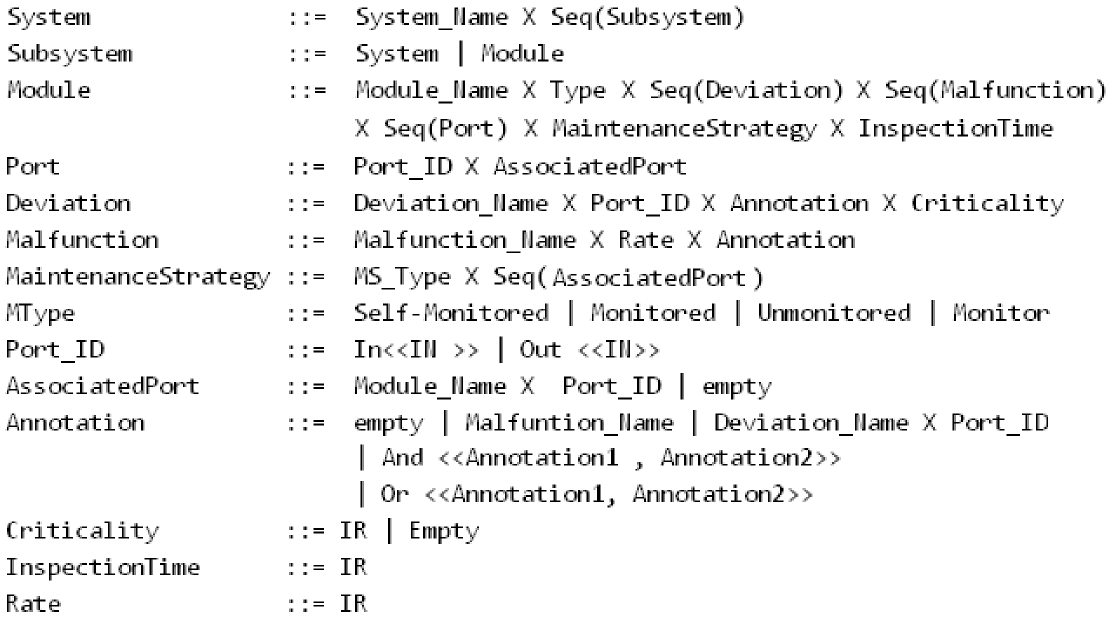
4.3 Collecting and Processing the Input Data

Recall from Section 2.3.3 that the component failure characterizations can be captured by hierarchical tabular structures (Table 2.1 through Table 2.4). These tables, also considering the additional notation of the previous section, are a concrete representation of the system failure model. Although the system model is illustrated in a graphical and diagrammatic view, its failure model is commonly stored in this tabular structure. This facilitates the data extraction and processing as well as model transformation [5, 13].

In Matlab/Simulink, for instance, matrix structures (tabular notation) are created using Matlab environment variables to store the failure model [9]. These structures store user data related to each component in a Simulink model. The matrix structures can be accessed from the variable *UserData* calling the function *get_param*. Hence, all information required for parsing the model is read from the *UserData* variable, considering the structure defined. Irrelevant information about the graphics of the model is discarded, extracting only the relevant information. Matlab/Simulink also allows accessing these structures via a single text file.

Although the annotations that we collect in the Simulink diagram are similar to the tabular structures presented during the safety analysis, first we need to process the input data in the tabular format to systematically generate the Prism specification. Currently, our translation rules are stated in terms of the abstract syntax presented in Fig. 4.2.

These data structures are an abstract representation of all the information introduced previously (see Section 2).



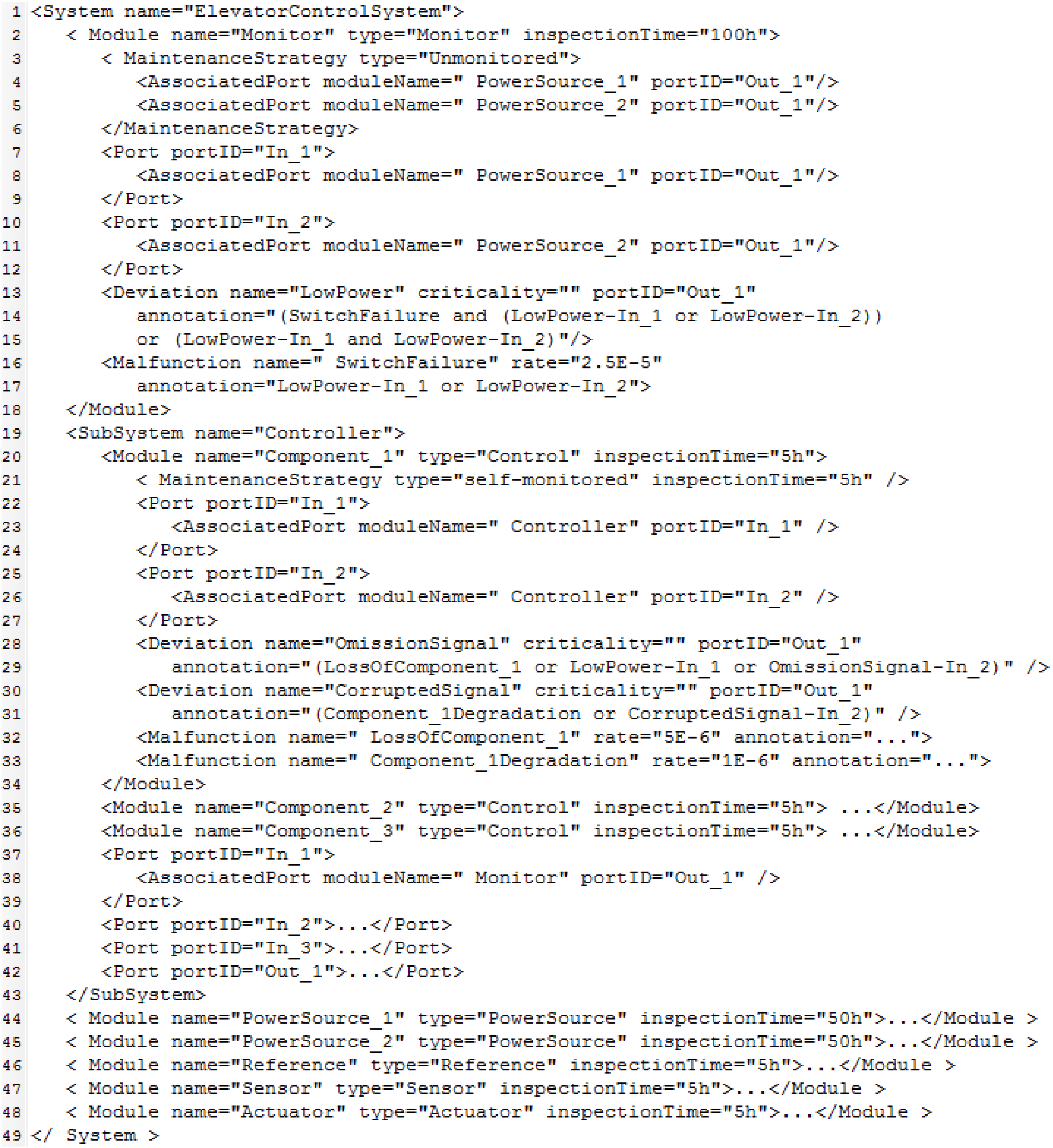
```
System               ::=  System_Name X Seq(Subsystem)
Subsystem            ::=  System | Module
Module               ::=  Module_Name X Type X Seq(Deviation) X Seq(Malfunction)
                          X Seq(Port) X MaintenanceStrategy X InspectionTime
Port                 ::=  Port_ID X AssociatedPort
Deviation            ::=  Deviation_Name X Port_ID X Annotation X Criticality
Malfunction          ::=  Malfunction_Name X Rate X Annotation
MaintenanceStrategy  ::=  MS_Type X Seq( AssociatedPort )
MType                ::=  Self-Monitored | Monitored | Unmonitored | Monitor
Port_ID              ::=  In<<IN >> | Out <<IN>>
AssociatedPort       ::=  Module_Name X  Port_ID | empty
Annotation           ::=  empty | Malfuntion_Name | Deviation_Name X Port_ID
                          | And <<Annotation1 , Annotation2>>
                          | Or <<Annotation1, Annotation2>>
Criticality          ::= IR | Empty
InspectionTime       ::= IR
Rate                 ::= IR
```

Fig 4.2 - Defined types based on tabular annotations

We start by considering a system (System) as a structure that contains a name (System_Name) and a list of subsystems (Seq(Subsystem)). Each subsystem can be another system or a module; because components can also be systems. A module (Module) represents the lower level component that contains a name, a list of ports (Seq(Ports)), a list of deviations (Seq(Deviation)), a list of malfunctions (Seq(Malfunction)), the maintenance strategy info and the inspection time. All these types (Port, Deviation, Malfunction, MaintenanceStrategy and InspectionTime) are associated with the tabular structures used to store all system information about its architecture, hierarchy, failure conditions, failure modes, repairs and the characteristics of monitoring and propagation of component failures. Port is a structure that contains a Port_ID (representing the identifiers of input/output ports) and an AssociatedPort (which stores the connected port of other components).

Annotation is a boolean expression that represents the failure logic of deviations. Its definition considers And/Or operators and their terminal terms can be malfunction names or deviations from any port. Criticality represents a real number ($\mathbb{R}$) used to quantify the tolerable probability associated with a failure condition (expressed via a deviation). Finally, InspectionTime and Rate are also real numbers used to represent the rate of occurrence of a malfunction and of a repair, respectively.

To exemplify this abstract syntax, we describe below the equivalent representation of the Elevator Control System described in Section 2.3.3:



```
1  <System name="ElevatorControlSystem">
2     < Module name="Monitor" type="Monitor" inspectionTime="100h">
3        < MaintenanceStrategy type="Unmonitored">
4           <AssociatedPort moduleName=" PowerSource_1" portID="Out_1"/>
5           <AssociatedPort moduleName=" PowerSource_2" portID="Out_1"/>
6        </MaintenanceStrategy>
7        <Port portID="In_1">
8           <AssociatedPort moduleName=" PowerSource_1" portID="Out_1"/>
9        </Port>
10       <Port portID="In_2">
11          <AssociatedPort moduleName=" PowerSource_2" portID="Out_1"/>
12       </Port>
13       <Deviation name="LowPower" criticality="" portID="Out_1"
14          annotation="(SwitchFailure and (LowPower-In_1 or LowPower-In_2))
15          or (LowPower-In_1 and LowPower-In_2)"/>
16       <Malfunction name=" SwitchFailure" rate="2.5E-5"
17          annotation="LowPower-In_1 or LowPower-In_2">
18    </Module>
19    <SubSystem name="Controller">
20       <Module name="Component_1" type="Control" inspectionTime="5h">
21          < MaintenanceStrategy type="self-monitored" inspectionTime="5h" />
22          <Port portID="In_1">
23             <AssociatedPort moduleName=" Controller" portID="In_1" />
24          </Port>
25          <Port portID="In_2">
26             <AssociatedPort moduleName=" Controller" portID="In_2" />
27          </Port>
28          <Deviation name="OmissionSignal" criticality="" portID="Out_1"
29             annotation="(LossOfComponent_1 or LowPower-In_1 or OmissionSignal-In_2)" />
30          <Deviation name="CorruptedSignal" criticality="" portID="Out_1"
31             annotation="(Component_1Degradation or CorruptedSignal-In_2)" />
32          <Malfunction name=" LossOfComponent_1" rate="5E-6" annotation="...">
33          <Malfunction name=" Component_1Degradation" rate="1E-6" annotation="...">
34       </Module>
35       <Module name="Component_2" type="Control" inspectionTime="5h"> ...</Module>
36       <Module name="Component_3" type="Control" inspectionTime="5h"> ...</Module>
37       <Port portID="In_1">
38          <AssociatedPort moduleName=" Monitor" portID="Out_1" />
39       </Port>
40       <Port portID="In_2">...</Port>
41       <Port portID="In_3">...</Port>
42       <Port portID="Out_1">...</Port>
43    </SubSystem>
44    < Module name="PowerSource_1" type="PowerSource" inspectionTime="50h">...</Module >
45    < Module name="PowerSource_2" type="PowerSource" inspectionTime="50h">...</Module >
46    < Module name="Reference" type="Reference" inspectionTime="5h">...</Module >
47    < Module name="Sensor" type="Sensor" inspectionTime="5h">...</Module >
48    < Module name="Actuator" type="Actuator" inspectionTime="5h">...</Module >
49 </ System >
```

Lines 2 through 18 define the `monitor` component, whose maintenance strategy attributes are defined in line 3. Association between its ports is described in lines 7 to 12. Lines 13 to 15 describe the deviations and its failure logic expression and lines 16 and 17 relates the attributes of component failure modes. Next, lines 19 through 43 describe the `controller` which is treated as a subsystem. So inside its correspondent tags, this structure defines the three subcomponents that compose this subsystem (`Component_1`, `Component_2`, `Component_3`) and the input/output port associations (lines 37 through 42). For conciseness, only the `Component_1` is described (lines 20 through 34). Finally, the remaining system components are listed in lines 44 to 49, whose data structures are similar to the `monitor` and `Component_1`.

4.4 Translation Rules

In this section, our strategy applies a set of translation rules which are based on the abstract syntax of Fig. 4.2 to generate the Prism specification. To ease the overall understanding about their applicability we also provide the typical sequence of their application in Fig. 4.3. Also, we will describe meticulously the main concept and description of these rules.

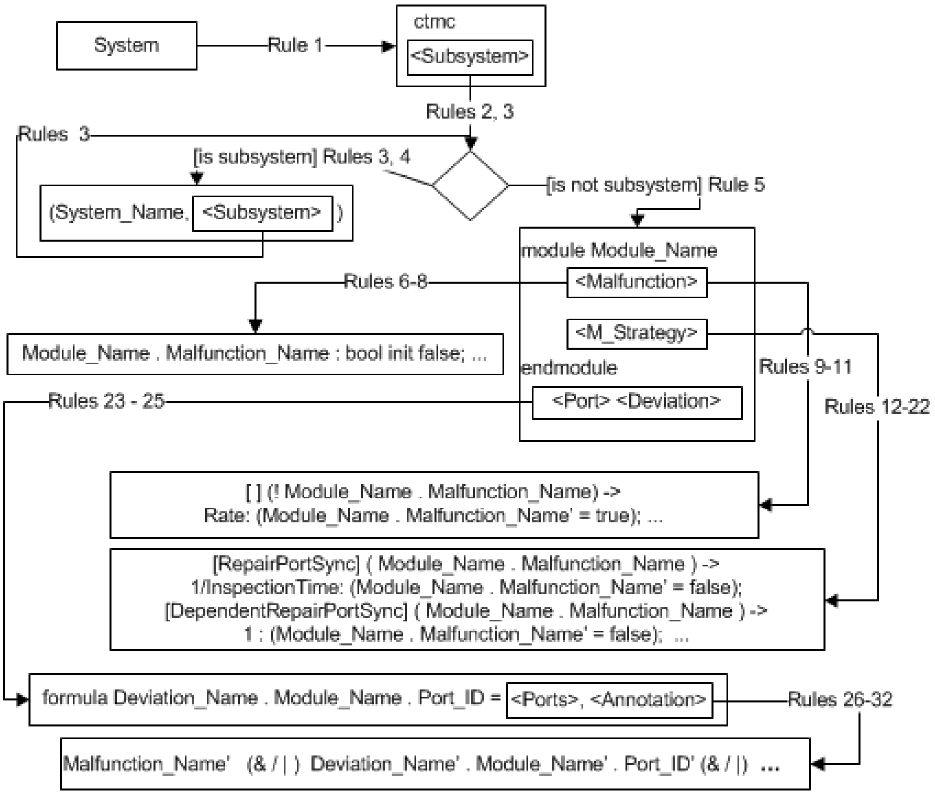

System
Rule 1
ctmc
<Subsystem>
Rules 2, 3
Rules 3
[is subsystem] Rules 3, 4
[is not subsystem] Rule 5
(System_Name, <Subsystem>)
module Module_Name
<Malfunction>
<M_Strategy>
endmodule
<Port> <Deviation>
Rules 6-8
Module_Name . Malfunction_Name : bool init false; ...
Rules 9-11
Rules 12-22
Rules 23 - 25
[ ] (! Module_Name . Malfunction_Name) ->
Rate: (Module_Name . Malfunction_Name' = true); ...
[RepairPortSync] (Module_Name . Malfunction_Name) ->
1/InspectionTime: (Module_Name . Malfunction_Name' = false);
[DependentRepairPortSync] (Module_Name . Malfunction_Name) ->
1 : (Module_Name . Malfunction_Name' = false); ...
formula Deviation_Name . Module_Name . Port_ID = <Ports>, <Annotation>
Rules 26-32
Malfunction_Name' (& / |) Deviation_Name' . Module_Name' . Port_ID' (& / |) ...

Fig 4.3 - Translation Strategy Overview

The strategy always starts by applying Rule 1, which state that we are dealing with a CTMC Markov model and applies other rules to create the several Prism modules from the system components (Rules 2–4). The body of a module is effectively created by Rule 5. After that, basic declaration instructions (Rules 6-8), commands (Rules 9-11) and repair transitions (12-22) are created. To complete the translation strategy, formula expressions are created (Rules 23-28) using a set of rules that decomposes all logic expressions (Rules 29-35).

4.4.1 Compound Systems and Subsystems

Our rules are inductively defined on the structure of a Prism system. We start with Rule 1 that takes as argument a pair where the first element has the name of a system (`SName`) and the second element a list of its subsystems (`SubSys`).

Rule 1 |{ (SName, SubSys) }|^{system} ⇒ ***ctmc*** |{ SubSys }|^{subsystem}

Following Rule 1, the resulting Prism code is basically the directive ctmc (instructing Prism to perform a CTMC interpretation), and a call to the function *subsystem*. This function is defined by Rules 2 (base case) and 3 (recursive case).

Rule 2 |{ <S> }|^{subsystem} ⇒ |{ S }|^{module}
Rule 3 |{ S: tail }|^{subsystem} ⇒ |{ S }|^{module} |{ tail }|^{subsystem}

Rules 2 and 3 do not produce Prism code. They access each component of this system and call the function *module* recursively for each component (Rules 4 and 5). For instance, applying these rules on the ECS, we obtain the following situation:

Step1: |{ PowerSource_1: tail }|^{subsystem} ⇒
|{ PowerSource_1 }|^{module} |{ tail }|^{subsystem}
Step2: |{ PowerSource_2: tail }|^{subsystem} ⇒
|{ PowerSource_2 }|^{module} |{ tail }|^{subsystem}
Step3: |{ Monitor : tail}|^{subsystem} ⇒
|{ Monitor }|^{module} |{ tail }|^{subsystem}
...
Step7: |{<Actuator>}|^{subsystem} -> |{ Actuator }|^{module}

4.4.2 Module

As modules can be subsystems as well, we translate modules by using two rules: Rule 4 (which calls function *subsystem*) and Rule 5 (which starts the creation of a Prism module).

Rule 4 |{ (SName, SubSys) }|^{module} ⇒ |{ SubSys }|^{subsystem}

Rule 4 can be distinguished from Rule 5 by pattern matching. One of them will be applied depending on the type that they are dealing. For instance, the Controller has internal components, that is it is a subsystem. So this type matches with the Rule 4.

|{ Controller, Component1:tail}| ^{module} ⇒ |{Component1: tail}|^{subsystem}

Rule 5 takes as input a tuple containing the module elements: name, type, set of ports, set of deviation logics, malfunctions, maintenance strategy and inspection time. The module name (MName) is used to name the Prism module (between the keywords ***module*** and ***endmodule***). Inside the module, the function *declars* is called to create the declaration part, and the function *commands* the behavioral part. Finally, the function *formulas* is called to create the set of Prism formulas outside the module.

Rule 5 |{ (MName,Type,Ports,Deviations,Malfuncs,MStrategy,IT) }|^{module}⇒
module MName
|{ MName, Malfuncs }|^{declars}
|{ MName, Ports, Malfuncs }|^{failureCommands}
|{ MName, Ports, Malfuncs, MStrategy, IT }|^{repairCommands}
endmodule
|{ MName, Ports, Deviations, true }|^{formulas}

For example, the Monitor is a lower level component, and then by pattern matching the Rule 5 will be used in its translation that is shown below:

```
|{ (Monitor, Monitor, (In1, (PS1, Out1)): tail, <LowPower>,
<switchFailure>, Monitor, 50) }|module ⇒

module Monitor
    |{ Monitor, <switchFailure> }|declars
    |{ Monitor, (In1, (PS1, Out1)): tail, <switchFailure> }| failureCommands
    |{ Monitor, (In1, (PS1, Out1)): tail, <switchFailure>, Monitor, 50
     }|repairCommands
endmodule

|{ Monitor, (In1, (PS1, Out1)): tail, <LowPower>, true }|formulas
```

4.4.3 Declarations

Rules 6 and 7 act in the same style of rules 2 and 3 and is used to access each component malfunction by your list.

Rule 6 `|{ MName, Malfuncs: tail }|`^{declarations} ⇒

`|{ MName, Malfuncs }|`^{declar}

`|{ MName, tail }|`^{declarations}

Rule 7 `|{ MName, <Malfuncs> }|`^{declarations} ⇒ `|{ MName, Malfuncs }|`^{declar}

Malfunctions are representations of possible failures within a component. To capture this feature in Prism, for each component malfunction, local boolean variables initialized with *false* are defined.

Rule 8 `|{ MName, (MfName, Rate, Annot) }|`^{declar} ⇒

`MName . _ . MfName: bool init false;`

Rule 8 uses each component malfunction to generate the declaration of its respective local variable inside the module block. Module's name (`MName`) and malfunction's name (`MfName`) are used to create the local variable name. For instance, the translation of sensor malfunctions using Rules 6, 7 and 8 generates the following Prism:

```
module Sensor
  |{ sensor, (sensorfailure, 1e-4, Annot): tail }|declarations ⇒
      sensor_sensorfailure: bool init false;
      sensor_sensordegradation: bool init false;
...
endmodule
```

4.4.4 Failure Transition Commands

Prism transition commands are responsible to update the state of the local variables. We translate malfunction structure (rates and logic expression) into failure transition commands which updates the malfunction to a failure state based on its failure rate. The Rules 9 and 10 act in the same style of rules 2 and 3 and is used to access each component malfunction by your list.

Rule 9 |{ MName, Ports, Malfuncs: tail }|^{failureCommands} ⇒

|{ MName, Ports, Malfuncs }|^{fCommand}

|{ Name, Ports, tail }|^{failureCommands}

Rule 10 |{ MName, Ports, < Malfuncs > }|^{failureCommands} ⇒

|{ MName, Ports, Malfuncs }|^{fCommand}

Rule 11 translates each malfunction into a Prism command. It always assumes the guard as a logical conjunction between the negation of a malfunction (this comes from Rule 8) and the negation of the fully failed system situation (a term defined by a Prism formula). If such a guard is valid then, with a rate given by Rate, this malfunction is activated.

Rule 11 |{ MName, Ports, (MfName, Rate, Annot) }|^{fCommand} ⇒
[] (!(MName .MfName**)) ->** Rate: **(**MName .MfName’=***true*****);**

As an example, we present the translation of the Sensor malfunctions in Prism commands using Rules 9, 10 and 11.

```
module Sensor
   sensor.sensorfailure: bool init false;
   sensor. sensordegradation: bool init false;
      |{sensor, (In1, (PS1, Out1)): tail, (sensorfailure, 1e-4, Annot): tail
       }|failureCommands ⇒
      [](!sensor_sensorfailure) -> (5E^-4) : (sensor_sensorfailure' = true);
      [](!sensor_sensordegradation) -> (5e^-4) :
              (sensor_sensordegradation' = true);
…
endmodule
```

4.4.5 Repair Transition Commands

Rules 12 through 17 translate the maintenance strategy (defined for each component) into Prism repair commands. This is performed according to the classification of each basic component of the system with respect to the treatment of the type of monitoring of its faults. Rule 12 considers two types: Self-monitored and Non-monitored (note the **provided** clause), whereas Rules 13 and 14 tackle the other cases: Monitored and Monitor, respectively.

In Rule 12, if the corresponding guard is valid, then, with a rate (1/Inspection Time), all component malfunctions are deactivated. Function *orLogic* takes a logical disjunction between all malfunctions (this comes from Rule 8) and function *Update* deactivates all malfunctions (set the value *false* to each malfunction).

Rule 12 |{ MName,Ports,Malfuncs,(MSType, AssocPorts:tail),IT}|^{repairCommands} ⇒

[] (|{ MName, Malfuncs }|^{orLogic}) -> (1/IT): |{ MName, Malfuncs }|^{update} ;

provided MSType = Self-Monitored or MSType = Non-monitored

However, if the component is Monitored, its repair commands must be synchronized with the Monitor component (function *monitoredRCommmand*).

Rule 13 |{ MName,Ports,Malfuncs,(MSType, AssocPorts:tail),IT }|^{repairCommands} ⇒

|**{** Malfuncs, AssocPorts, IT **}**|^{monitoredRCommand}

|{ MName,Ports,Malfuncs,(MSType, tail),IT }|^{repairCommands}

provided MSType = Monitored

Rule 14 |{ MName,Ports,Malfuncs,(MSType, <AssocPorts>),IT }|^{repairCommands} ⇒

|**{** Malfuncs, AssocPorts, IT **}**|^{monitoredRCommand}

provided MSType = Monitored

If a component is a Monitor, instead of the synchronized repair commands corresponding to the monitored component (function *sincronizedRCommand*), another repair command is created to represent the single repair of this component.

Rule 15 |{ MName,Ports,Malfuncs,(MSType, AssocPorts:tail),IT }|^{repairCommands} ⇒

|{ MName,Malfuncs, AssocPorts, IT}|^{sincronizedRCommand}

provided MSType = Monitor

Rule 16 |{ MName,Ports,Malfuncs,(MSType, <AssocPorts>),IT }|^{repairCommands} ⇒

|{ MName,Malfuncs, AssocPorts, IT}|^{sincronizedRCommand}

[] (|{ MName, Malfuncs }|^{orLogic}) -> (1/IT): |{ MName,Malfuncs}|^{update} ;

provided MSType = Monitor

Rules 17 and 18 are used to define the synchronized repair commands between the monitored (Rule 17) and the monitoring component (Rule 18).

Rule 17 |{ Malfuncs, (MName, PortID), IT }| ^{monitoredRCommand} ⇒

[MName._.PortID._.**DependentRepair**] (|{ MName, Malfuncs }|^{orLogic}) ->
(1/IT): |{ MName, Malfuncs }|^{update} ;

[MName._.PortID._.**Repair**] (|{ MName, Malfuncs }|^{orLogic}) -> (1) :
|{ MName,Malfuncs }|^{update};

Rule 18 `|{ MName,Malfuncs, (MName, PortID), IT }|`$^{\text{sincronizedRCommand}}$ ⇒

`[MName._.PortID._.Repair] ( |{ MName,Malfuncs }|`$^{\text{OrLogic}}$ `) -> (1/IT):`

`|{  MName,Malfuncs }|`$^{\text{update}}$ `;`

`[MName._.PortID._.DependentRepair] (( |{  MName,Malfuncs }|`$^{\text{orLogic}}$ `)`

`-> (1): |{ MName,Malfuncs }|`$^{\text{update}}$ `;`

Rules 19 and 20 generate a logical expression used as guard of the module repair commands. The guard assumes a logical disjunction between the component malfunctions.

Rule 19 `|{ MName, (MfName, Rate, Annot): tail}|`$^{\text{orLogic}}$ ⇒

`MName._.MfName   |   |{ tail }|`$^{\text{orLogic}}$

Rule 20 `|{ MName, <(MfName, Rate, Annotation )> }|`$^{\text{orLogic}}$ ⇒ `MName._.MfName`

Rules 21 and 22 create assignment commands that are part of repair command and are responsible for deactivate each malfunction defined for a module.

Rule 21 `|{ MName, (MfName, Rate, Annotation): tail}|`^{update} ⇒

`(MName._.MfName’ = false) & |{ tail }|`^{update}

Rule 22 `|{ MName, <(MfName, Rate, Annotation )> }|`^{update} ⇒

`(MName._.MfName’ = false)`

For instance, the repair transition commands of the sensor module are generated applying rules 12, 19, 20, 21 and 22 that translate the sensor malfunction information in the following Prism code:

`|{sensor, (In1, (PS1, Out1)): tail, (sensorfailure, 1e-4, Annot): tail,`

`(Self-monitored,””),5}|`^{repairCommands} ⇒

`[] ( |{ sensor, (sensorfailure, 1e-4, Annot): tail }|`^{orLogic} `) -> (1/5):`

`|{ MName, Malfuncs }|`^{update} `;`

`|{ sensor, (sensorfailure, 1e-4, Annot): tail }|`$^{\text{orLogic}}$ ⇒

`sensor_sensorfailure | sensor_sensordegradation`

`|{ sensor, (sensorfailure, 1e-4, Annot): tail }|`$^{\text{update}}$ ⇒

`(sensor_sensorfailure’ = false) & (sensor_sensordegradation’ = false)`

4.4.6 Formulas

The final elements we address are Prism formulas. They correspond to the failure logic expressions annotated in Simulink diagrams. Each expression that represents the possible system failure conditions (deviations) is transformed into a Prism formula. As these expressions, the formula is written in compositional form. That is, it is formed from basic formulas that are based on the local variables of each module (representing the malfunctions). Once again, the Rules 23 and 24 act in the same style of rules 2 and 3 and is used to access each component deviation by your list.

Rule 23 `|{ MName, Ports, Deviation : tail, boolValue }|` $^{\texttt{formulas}} \Rightarrow$

`|{ MName, Ports, Deviation }|` $^{\texttt{formula}}$

`|{ MName, Ports, tail, false}|` $^{\texttt{formulas}}$

Rule 24 `|{ MName, Ports, <Deviation>, boolValue }|` $^{\texttt{formulas}} \Rightarrow$

`|{ MName, Ports, Deviation }|` $^{\texttt{formula}}$

At this point, we are able to translate the failure logic expressions. Formulas are labeled considering the deviation name, module name and output port id.

Rule 25 creates the component deviation formulas compounding a name for the formula based on the deviation name (DName), followed by the module name (MName) and the identifier of the port (PortID). The formula's body is a boolean expression resulting from function *fExpression*.

Rule 25 `|{ MName, Ports,(DName, PortID, Annot, Crit) }|` $^{\textit{formula}} \Rightarrow$

formula `DName._.MName._.PortID = |{ Ports, Annot }|` $^{\textit{fExpression}}$

The function |{ }|$^{\text{fExpression}}$ takes a deviation annotation and the list of component ports to translate the annotation logic expression to a prism boolean expression. Next rules (26 and 27) are responsible for this.

Rule 26 `|{ Ports, And( Annot`$_1$` , Annot`$_2$`)  }|` $^{\textit{fExpression}} \Rightarrow$

(`|{ Ports, Annot`$_1$` }|` $^{\textit{fExpression}}{}_{,}$**) & (** `|{ Ports, Annot`$_2$` }|` $^{\textit{fExpression}}$**)**

Rule 27 `|{ Ports, Or ( Annot`$_1$` , Annot`$_2$`) }|` $^{\textit{fExpression}} \Rightarrow$

(`|{ Ports, Annot`$_1$`}|` $^{\textit{fExpression}}$**) | (** `|{ Ports, Annot`$_2$` }|` $^{\textit{fExpression}}$**)**

To complement the expression formation, it is necessary to identify the terminal terms of the logic expression. As we can see in the annotation type definition, there are two kinds of terminal terms. The first is the component malfunction name (Rule 28) and the other is the input port deviation name (Rule 29).

Rule 28 `|{ Ports, MfName }|` $^{\textit{fExpression}} \Rightarrow$ **(**`MfName`**)**

Rule 29 `|{ Ports, (DName, Port_ID) }|` $^{\textit{fExpression}} \Rightarrow$

(`DName . |{ Port_ID, Ports }|` $^{\texttt{Associated}}$ **)**

Finally, to express the formulas on compositional form, is need to change the input port deviation name to its associated port deviation. So an input deviation is replaced by its respective formula that describes the associated output port deviation.

Rule 30 `|{ Port_ID, (Port_ID’, AssocPort): tail }|` $^{\texttt{Associated}} \Rightarrow$

`|{ Port_ID, tail }|` $^{\texttt{Associated}}$

Rule 31 `|{ Port_ID,  < (Port_ID, AssocPort) > }|`$^{\text{Associated}}$ ⇒

`|{ AssocPort }|`$^{\text{AssociatedName}}$

Rule 32 `|{ (MName, Port_ID) }|`$^{\text{AssociatedName}}$ ⇒ `( MName . Port_ID)`

For instance, in order to generate the failure logic expressions about the Sensor deviations, we have to apply Rules 23 through 32. This translation results in the following Prism code:

`|{sensor, (In1, (PS1, Out1)): tail, (OmissionSignal, Out1, Annot, “”): tail, true}|`^{repairCommands} ⇒

```
formula OmissionSignal_Sensor_Out1 = sensor_sensorfailure |
    LowPower_Monitor_Out1 | OmissionSpeed_Actuator_Out1;
formula CorruptedSignal_Sensor_Out1 = sensor_sensordegradation;
```

Using these translation rules, we generate a valid formal failure model retaining the semantics of diagrams and the system hierarchical model.

4.4.7 Generation of system verification expressions

In this step we create the set of expressions in the CSL language to analyze the failures conditions of the system. The failure conditions are represented as deviations of the system associated with a criticality. They are selected to be evaluated based on the FHA analysis that also specifies their tolerable probability in the tabular structures. Thus, for each *Failure Condition* to be evaluated, the following verification expressions are created.

P = ? [true U<=T "Failure Condition"]
((P=? [true U<=T " Failure Condition"]) / T)
(((P=? [true U<=T " Failure Condition"]) / T) <= Crit)

where Crit is the tolerable probability of the failure condition

Next, we present the translation rules used to generate the above mentioned verification expressions. This translation strategy follows the same principle of the strategy for the generation of the formal Prism specification. Rule 33 declares a variable of type Double to be used as a time argument in the verification expression.

Rule 33 `|{ (SName ,  Subsystems) }|`$^{\text{system}}$ ⇒ **`const double T;`**

`|{ Subsystems }|`$^{\text{subsystem}}$

Rules 34, 35 and 36 are similar to Rules 1, 2, 3 and 4 defined previously, except that instead of calling the function *module*, they call the function *expressions*.

Rule 34 `|{ <S> }|`^{subsystem} ⇒ `|{ S }|`^{expressions}

Rule 35 `|{ S: tail }|`^{subsystem} ⇒ `|{ S }|`^{expressions} `|{ tail }|`^{subsystem}

Rule 36 `|{ (SName, SubSys) }|`^{expressions} ⇒ `|{ SubSys }|`^{subsystem}

Rules 37 and 38 are used to access each component deviation in the respective list.

Rule 37 `|{ (MName, Type , Ports, Deviation: tail, Mfuncs) }|`$^{\text{expressions}}$ ⇒

`|{ MName, Deviation }|`$^{\text{expr}}$

`|{ MName, Type , Ports, tail, Mfuncs }|`$^{\text{expressions}}$

Rule 38 `|{ (MName, Type , Ports, <Deviation>, Mfuncs) }|`$^{\text{expressions}}$ ⇒

`|{ MName, Deviation }|`$^{\text{expr}}$

Rule 39 calls the function responsible for the creation of the verification expression. A deviation is considered a failure condition if Crit ≠ empty. Firstly the rule creates the label that will compose the argument of the verification expressions. After, it creates two different temporal expressions.

Rule 39 `|{ MName, (DName, Crit, Port_ID, Annot) }|`$^{\text{expres}}$ `->`

```
Label "DName._.MName._.Port_ID" = DName._.MName._.Port_ID
P =? [true U <= T " DName._.MName._.Port_ID "]
((P =? [true U <= T " DName._.MName._.Port_ID "]) / T)
(((P =? [true U <= T " DName._.MName._.Port_ID "]) / T) <= Crit)
provided Crit ≠ empty
```

4.4.8 Model Considerations

Our solution still does not consider bidirectional data flows (such as the propagation of failure as short-circuit). However, such features can be added by considering new translation rules. Our strategy is sound with respect to the following assumptions:

- Component failures are detected in flight only and repaired during ground maintenance or before the next flight (description level), but the failures and repairs occur at constant rates (model level).
- The system is assumed with perfect failure coverage and can to reconfigure to a degradable mode within no time.

In terms of completeness, our rules are complete in the sense that they can translate any Simulink diagram annotated with failure logic in the IF-FMEA style [5]. Besides, this approach is not limited to just using the Simulink diagram as input. Actually, the necessary input data, which contains information from the qualitative model and the respective failure logic and propagation, is obtained from the tabular structures, which are user defined. Simulink diagrams work implicitly with these structures [10].

Our strategy follows a systematic process that has proved viable and of little impact in practice, since the tabular structures are generated by traditional methods and analysis used by the aircraft industry during the qualitative safety assessment (FHA, FMEA, IF-FMEA, CCA). So, adding a plug-in to some usual design tool, it is possible to automate our systematic approach.

The primary limitation of a stochastic model-checking is the size of the reachable state space, though recent breakthroughs allow very large (> 10^7 reachable states) state spaces to be explored in reasonable time.

4.5 Quantitative Analysis

In this section we analyze the generated model specified in Prism. Basically, from this Prism model we propose analytical expressions that allow calculating the average failure rate of the possible failure conditions of the system from the analysis of the temporal evolution of its possible states, whose behavior is defined by transition rates, parameterized using the mode failure rates and repair time of the model.

At first, our strategy focuses on identifying situations of violation of safety requirements of the system. Thus, it is possible to examine whether the probability of occurrence of certain failure conditions violates the standard safety limit ($\leq 10^{-9}$ in catastrophic failure condition, for instance).

Considering our context, to analyze the failure behavior of these systems, we can use, depending on the purpose, a steady-state or transient analysis [7, 16]. Transient analysis represents the instantaneous failure rate over a single period T whereas the steady-state analysis approximates the long-term average failure rate over multiple time intervals T, as illustrated in Fig. 4.4. The choice over these types of analyses depends on how system repairs are handled. Transient analysis can be performed in either closed-loop (models with repairs) or open-loop models (models without repairs), whereas the steady-state analysis can be performed only on closed-loop models.

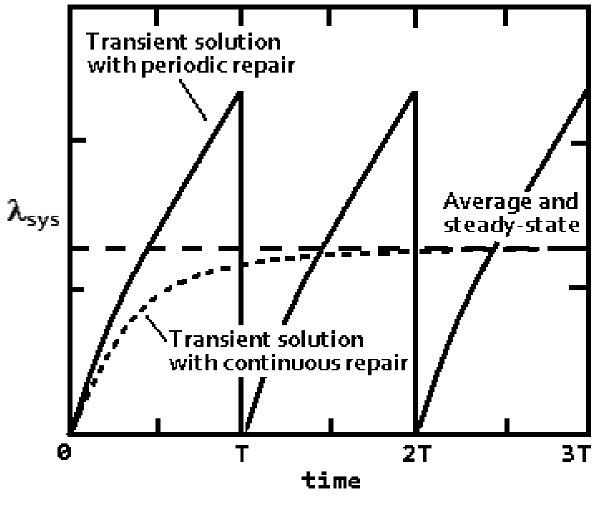


Fig 4.4. Graph plotting the common behavior of different Markov analysis.

Our proposed strategy creates models that consider repair transitions as if they occurred at constant rates. Thus they are typical closed-loop models and both analyses can be performed. We calculate the average rate of a failure condition applying the transient analysis.

Particularly, the transient analysis with continuous repair provides adequate accuracy on their results for our purposes, since (see examples in Fig. 3.5 and 3.6) most critical systems are modeled in such a way that they can deal with latency. In this scenario, several components affecting the system functionality must be monitored, maintained at regular intervals and repaired if they are faulty and the transient analysis with continuous repair is more representative in this situation. On the other hand, the transient analysis without repairs it applies strictly to just a single interval T, as if this was the entire life of the system, whereas most critical systems have maintenance cycles, where they are periodically restored to the full-up condition. Hence, the more representative analysis for this scenario is when the period T usually represents a repetitive repair interval rather than a life limit [16].

Fig. 4.4 shows that a transient analysis on the open-loop model represents a repair interval as a discrete limit, because it applies strictly to just a single interval T, repeating the interval until the entire life of the system, if necessary. Its entire plot in the figure (a sawtooth function) represents a situation in which the period T usually stands for a repetitive repair interval rather than a life limit (performed by several transient results). The mean value of the sawtooth function is almost equivalent to the continuous value. However, calculating the mean value of this function can generate extra work. This task is typically adopted in the traditional aeronautical approaches that use FTA to evaluate the average probability of the system failure conditions [4, 24].

Comparing with the steady-state analysis, the transient behavior during the first several hours is insignificant, requiring more care for the engineers to perform the analysis appropriately. But the instantaneous rate of the transient analysis generally has already come close the asymptotic steady-state rate in few hours and can be explored in a lot of instants rather than steady-state that only analyses the long-run situation. Moreover, a transient analysis can determine the contour of the instantaneous failure rate as a function of time, showing the system sensitivity. A steady-state analysis does not provide this information.

Therefore, to perform the quantitative safety analysis, we use the CSL language [8]. The operators P (transient) and S (steady-state) of Prism can be used to reason about the tolerable probabilities of all system failure conditions. For example, with the formula:

$$S \leq 10^{-9} \text{ [“Failure Condition”] .} \qquad \textbf{(1)}$$

we can check if, in the long run, the probability that a certain “Failure Condition” can occur is less than or equal to 10^{-9}. The satisfaction of a property (“true” or “false”) is defined for a single state of a model. When analyzing a property, PRISM considers it to be true if it is satisfied in all states of the model, and false otherwise. We can also use the following formula to obtain this probability value in the long term:

$$S = ? \text{ [“Failure Condition”] .} \qquad (\textbf{2})$$

We can also check the exact probability itself by using other CSL formula:

$$P = ? \text{ [true } U \leq T \text{ “Failure Condition”].} \qquad (\textbf{3})$$

This yields the instantaneous probability of occurrence of a certain “Failure Condition” at time instant T. We can also perform such an analysis for a range of values of T in order to gain insight into the likelihood of the system as time progresses. Therefore, Prism can support both analysis solutions (steady-state or transient analysis).

Moreover, as the steady-state analysis value is considered to a limit situation (equilibrium state), to calculate the average probability of a failure condition on the situation where the equilibrium state is not achieved during the lifetime of the system, we can applying another formula in Prism using the transient operator normalized with a specific time T:

$$((P = ?\ [\ \text{true}\ U \leq T\ \text{“Failure Condition”}\ ])./T) \qquad \textbf{(4)}$$

Following this principle, we also can check if the probability that a certain “Failure Condition” can occur is less than or equal to 10-9.using the transient operator:

$$((P = ?\ [\ \text{true}\ U \leq T\ \text{“Failure Condition”}\ ])./T) <= 10^{-9} \qquad \textbf{(5)}$$

Whereas we reported in this section, the formulas 3, 4 and 5 are more appropriated to analyze our models.

Chapter 5

Case Study

In this section we illustrate our strategy using a very simple example to ease understanding all elements of the strategy. We demonstrate our proposed strategy using a feedback control system—the Elevator Control System (ECS), which is responsible for commanding an electro-hydraulic actuator, according to the longitudinal orientation desired by the pilot. This case study was already introduced in Section 2.3. It is presented graphically in Figure 2. Although it is a simple example, it is representative in the aeronautics context in the sense that it has dependent and independent failures, a hierarchical architecture, latency, evident, repeated and developed events [1, 2].

5.1 System Description

This system acts in one type (elevator) of the several flight control surfaces, which are designed to allow pilots to change the forces and moments acting on the aircraft. Fig. 5.1 shows the main components of the ECS: the reference unit (Reference) captures commands from the pilot and it is usually a side-stick (or yoke) providing longitudinal deflections in degrees, the controller (Controller) is an Actuator Control Electronics device (ACE) responsible to process the reference signal and the elevator position provided by the sensor component (Sensor) to generate the correct commands to the associated power control unit (PCU or Actuator). Moreover, this system is powered by two power sources (PowerSource) which are monitored by a monitor (Monitor). The further details about this system already introduced in Section 2.3.4. Next we will describe the maintenance strategy of this system that as related in Section 4 is necessary to generate the formal model.

The main details about the system description and its failure behavior were already described in Section 2.3.3, where a qualitative model of this system was generated using tabular structures. Now, we will explain the maintenance strategy applied for the system that will be useful to create the additional tabular structures also necessary to the strategy application.

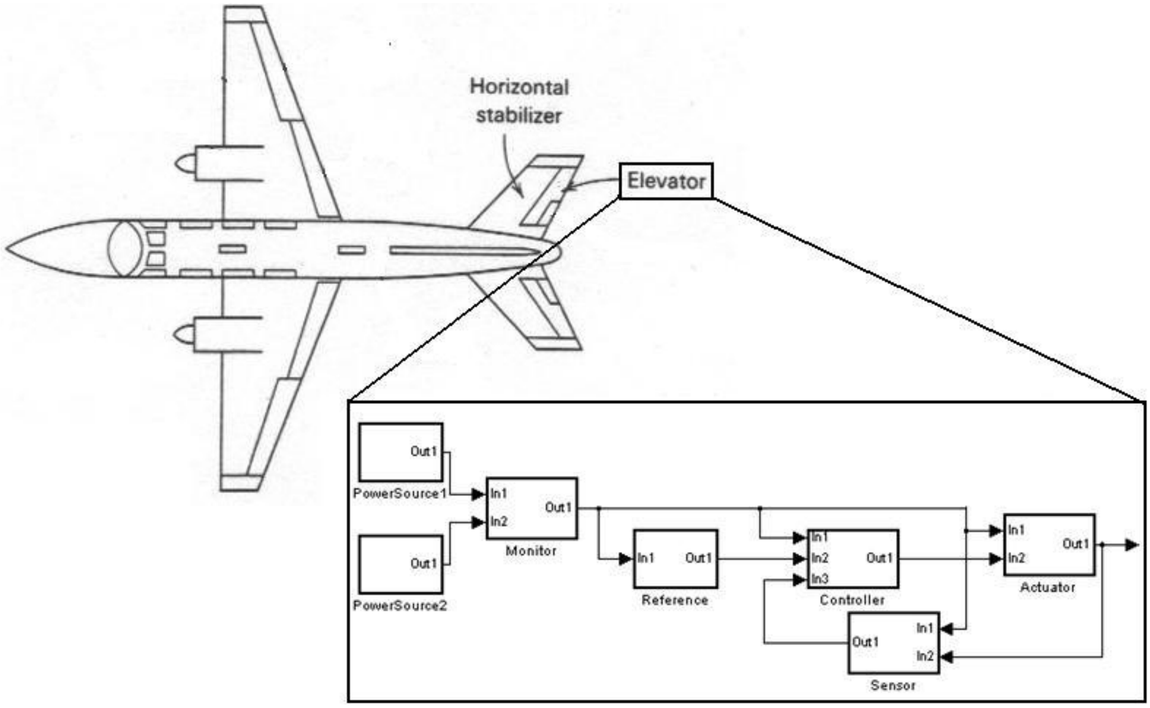


Fig 5.1. Elevator Control System

The maintenance strategy for this system is as follows: The PowerSource 1 and 2 have no self-monitoring, but are monitored continuously by a monitor. If some PowerSource fails and the monitor is working, the PowerSource is repaired before the next dispatch. If the monitor is not working, the PowerSources can fail latently, but it is checked every 10 flights (we are considering that the median time of flight is 5 hours, so 10 flights = 50 hours) and if some has failed during one of these periodic checks, it is repaired at that time. If some PowerSource unit is found faulty at one of these 50-hour checks, with no indication of this failure from the monitor, it is assumed that the monitor system has also failed, so all units are repaired prior to the next flight. The monitor can be repaired from two ways. First, as noted above, if some PowerSource unit has failed at its periodic 50-hour inspection and there was no monitor indication of this failure, then the monitor is repaired along with the PowerSource unit prior to the next flight. Second, a periodic check of the monitor is performed every 100 flights (500 hours), and if the monitor has failed, it is repaired prior to the next flight. The Reference is a self-monitored component, hence it is inspected and repaired if necessary before of every dispatch. The maintenance strategy of the remaining modules (Controller, Sensor and Actuator) are similar to the Reference and were omitted for conciseness.

5.2 Applying the Strategy

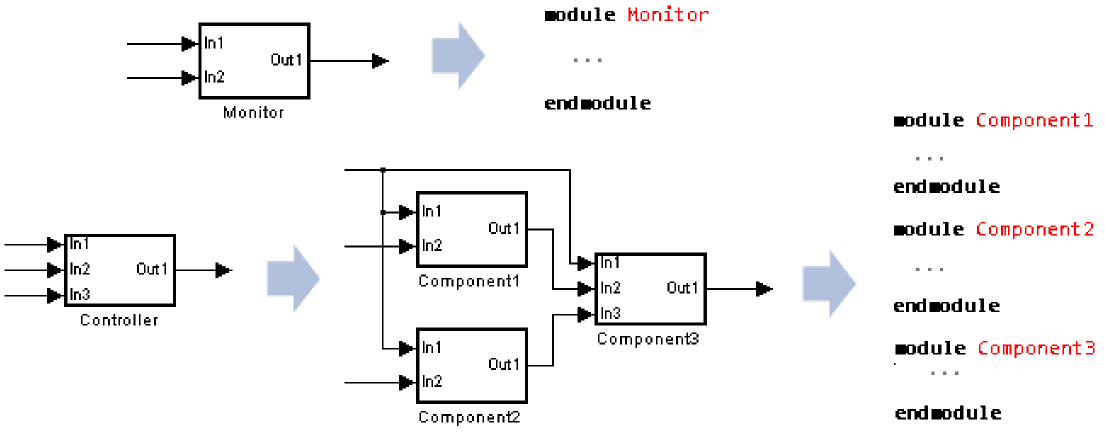
Considering the highlighted Simulink diagram in Fig. 5.1, the failure analysis of the system is performed following the model-based system safety assessment process explained in Section 2.3.3, where all tabular structures of the system resulting from this process was described in Table 2.1, Table 2.2, Table 2.3 and Table 2.4. Since all components are analyzed, describing their failure behaviors and registering its information about the topology, it is now possible to apply the proposed strategy to generate the formal specification in Prism and perform a quantitative analysis over this system using the probabilistic model checking. We implement our strategy following the five steps defined in Section 6.

As described in Section 2.3.3, the extended tabular information is user defined using Simulink. So, when we include these maintenance strategy and inspection time defined for each component, the resulting tabular information about this system can be depicted (see Table 4.1). Subsequently, the failure model of this system is stored in matrix structures kept in the Simulink environment variables and these data are extracted accessing a text file provided by the tool. Therefore, all data is processed and organized following the abstract syntax defined in Section 4.3. The resulting data structure is shown in Fig. 4.2. We create a script program to implement this last two steps and we intend to incorporate this program into an automation tool for future work.

5.2.1 Model Generation

Considering the resulting data information about these components and including the appropriate repair scheduled, the system failure model is ready to be used to generate the formal specification. To illustrate this, we simply apply the transformation rules presented in Section 4.4 on the system step by step.

Firstly, each system component is represented by a module in the specification. If a component is also a system, this component is discarded and its subcomponents will be represented by a module.

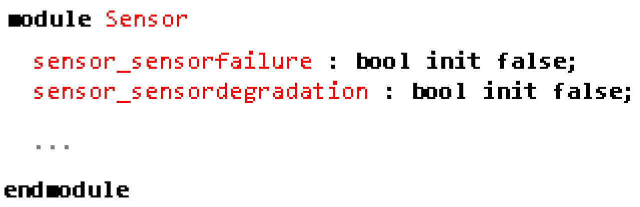


```
module Monitor
  ...
endmodule
```

```
module Component1
  ...
endmodule
module Component2
  ...
endmodule
module Component3
  ...
endmodule
```

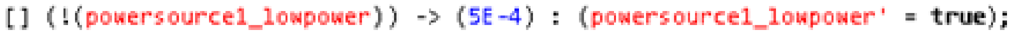
Fig 5.2. Step that demonstrate the module creation

The next part describes the declaration instructions. For each component malfunction (failure modes), local boolean variables initialized with *false* must be defined to represent the failure state for each malfunction associated with the module.



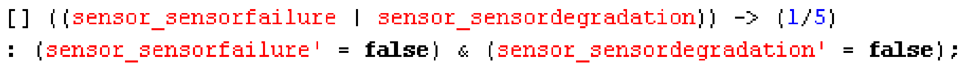
```
module Sensor
  sensor_sensorfailure : bool init false;
  sensor_sensordegradation : bool init false;

  ...

endmodule
```

Now we create a set of failure transition commands into each module. For each local variable in the module, a state transition command is created. Their guard expressions are stated as a conjunction of the negations of the local failure as well as the system's failure. The update commands of the local variable value are based on the corresponding failure rate. These commands represent transitions to a failure state associated with the malfunction represented by the local variable.



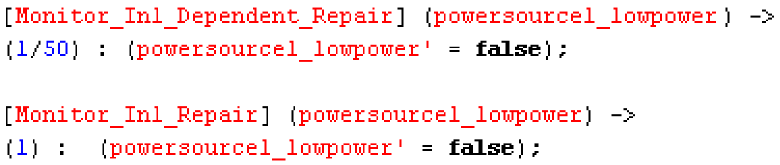
```
[] (!(powersource1_lowpower)) -> (5E-4) : (powersource1_lowpower' = true);
```

Depending on the component maintenance strategy, different set of repair transition commands are created into each module. If the component is self-monitored (Sensor, for instance) or non-monitored, just one state transition command is created. This command has no synchronization and its guard expression is assigned with the local variables. The command updates all local variable value to an operational situation based on its repair rate (the used value is the inverse of T, where T is the inspection time[2]. For self-monitored components, T = MedianTimeOfFlight.



```
[] ((sensor_sensorfailure | sensor_sensordegradation)) -> (1/5)
: (sensor_sensorfailure' = false) & (sensor_sensordegradation' = false);
```

In the situations where the component is externally monitored (PowerSource), instead of the previous command, two synchronized transition commands are created, and these commands are synchronized with the repair command of the stateful component. The first command occurs when both components fail (to represent repair of latent failure). The last occurs when the monitor detects that a monitored component fails. The transition rate of this last command is always 1 (it’s a Prism best practice used to quantify synchronized transitions: just one command controls the transition rate).



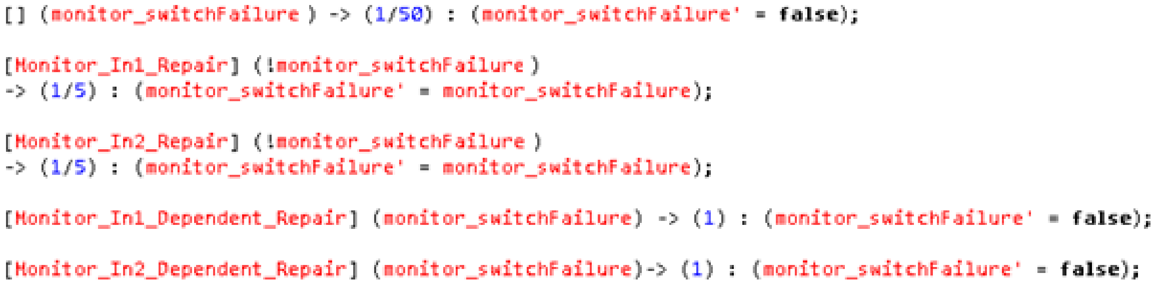
```
[Monitor_In1_Dependent_Repair] (powersource1_lowpower) ->
(1/50) : (powersource1_lowpower' = false);

[Monitor_In1_Repair] (powersource1_lowpower) ->
(1) :  (powersource1_lowpower' = false);
```

The last case covers the monitor type. In addition to adding the non-synchronized transition (because it is an non-monitored component), we have to create repair transition commands synchronized with all monitored components. Note that this is a complement to the previous item and allows us to represent the possible cases: 1) the monitor is repaired without failure occurred in the monitored components, 2) the monitor is repaired together with the components

---

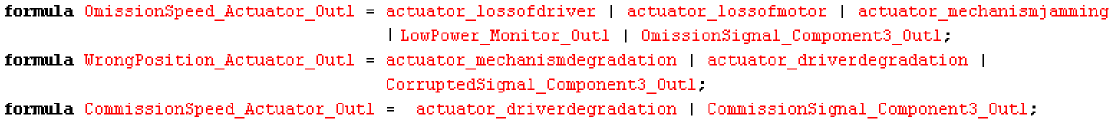
[2] A continuous transition can represent a periodic inspection/repair using a rate that gives the same mean time between a component failure and repair. To provide a conservative representation, the appropriate value of this time must be in the range from T/2 to T.

monitored. See also that the guard expression of no synchronized transitions is assigned with the negation of input deviation logic of the monitor failure mode (that is this kind of repair only occurs if no fails was detected from the monitored components).



```
[] (monitor_switchFailure ) -> (1/50) : (monitor_switchFailure' = false);

[Monitor_In1_Repair] (!monitor_switchFailure )
-> (1/5) : (monitor_switchFailure' = monitor_switchFailure);

[Monitor_In2_Repair] (!monitor_switchFailure )
-> (1/5) : (monitor_switchFailure' = monitor_switchFailure);

[Monitor_In1_Dependent_Repair] (monitor_switchFailure) -> (1) : (monitor_switchFailure' = false);

[Monitor_In2_Dependent_Repair] (monitor_switchFailure)-> (1) : (monitor_switchFailure' = false);
```

The last part of the generation creates a set of formulas. Each failure logic expression that can compose the failure conditions of the system is transformed into a PRISM formula. Like the expressions, the formulas are also written in compositional form. That is, they are formed from formulas already established, which are based on the local variables of each component representing their malfunctions. The complete system failure state is transformed into a single PRISM formula too. This formula is composed by an AND logic with its failure conditions. The negation of this formula is put into all guard expression of the modules using a AND operator.

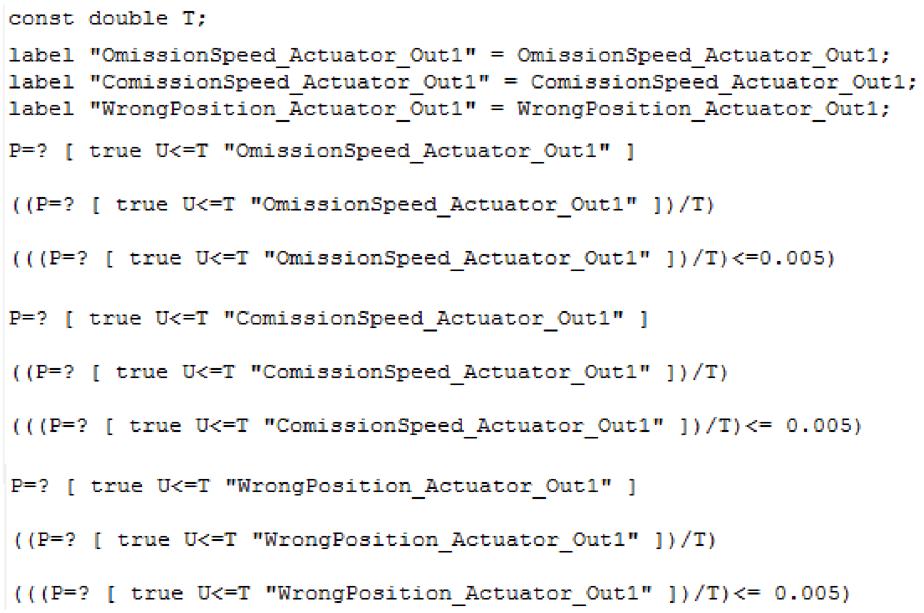


```
formula OmissionSpeed_Actuator_Outl = actuator_lossofdriver | actuator_lossofmotor | actuator_mechanismjamming
                                      |LowPower_Monitor_Outl | OmissionSignal_Component3_Outl;
formula WrongPosition_Actuator_Outl = actuator_mechanismdegradation | actuator_driverdegradation |
                                      CorruptedSignal_Component3_Outl;
formula CommissionSpeed_Actuator_Outl =  actuator_driverdegradation | CommissionSignal_Component3_Outl;
```

After applying the translation rules, we obtain the formal specification of the ECS which is depicted in Appendix A.

5.2.2 Quantitative Analysis

The next step we use the Prism model-checker to check whether any critical failure condition probability violates the permitted limit. To accomplish this, we execute the Prism model checker using the expressions in the CSL language (see Fig. 5.3) obtained after applying the rules defined in Section 4.4:



```
const double T;
label "OmissionSpeed_Actuator_Out1" = OmissionSpeed_Actuator_Out1;
label "ComissionSpeed_Actuator_Out1" = ComissionSpeed_Actuator_Out1;
label "WrongPosition_Actuator_Out1" = WrongPosition_Actuator_Out1;
P=? [ true U<=T "OmissionSpeed_Actuator_Out1" ]

((P=? [ true U<=T "OmissionSpeed_Actuator_Out1" ])/T)

(((P=? [ true U<=T "OmissionSpeed_Actuator_Out1" ])/T)<=0.005)

P=? [ true U<=T "ComissionSpeed_Actuator_Out1" ]

((P=? [ true U<=T "ComissionSpeed_Actuator_Out1" ])/T)

(((P=? [ true U<=T "ComissionSpeed_Actuator_Out1" ])/T)<= 0.005)

P=? [ true U<=T "WrongPosition_Actuator_Out1" ]

((P=? [ true U<=T "WrongPosition_Actuator_Out1" ])/T)

(((P=? [ true U<=T "WrongPosition_Actuator_Out1" ])/T)<= 0.005)
```

Fig 5.3. Generated expressions in CSL

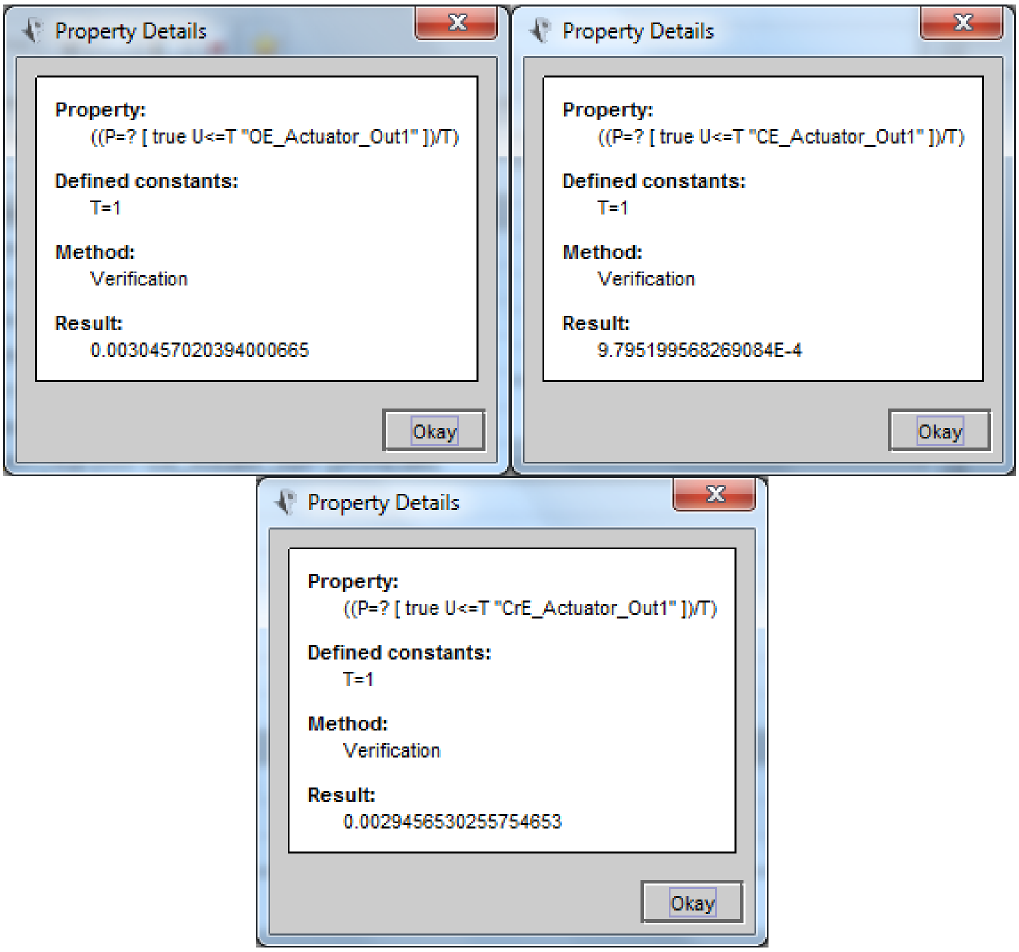
Considering the tabular information of the ECS (see Table 2.3), our strategy creates probabilistic temporal formulas to check the following failure conditions:

Omission of speed at Actuator output port shall be less than 3.10^{-3} per flight;
Commission of speed at Actuator output port shall be less than 5.10^{-3} per flight;
Wrong position signal at Actuator output port shall be less than 3.10^{-3} per flight.

Following ARP 4761, only catastrophic, hazardous, and major failures are analyzed quantitatively. In principle, considering only the tolerable values of these failure conditions we could mistakenly conclude that none of them need a quantitative analysis. However, as we describe in Section 2, the safety assessment process is hierarchical and based on levels, where the high-level safety requirements are decomposed into smaller. Consequently the tolerable rates of some potential hazard are also decomposed to the extent that the aircraft systems are broken down into other subsystems. The task responsible for check the tolerable probabilities of each subsystem and evaluate if the high-level safety requirements are really preserved in the entire hierarchy is called integration of cross-checking [2]. Therefore, knowing that the ECS is a subsystem which composes a high-level system of an aircraft [42], the proposed values for these system failure conditions are consistent with the context. We verify if some failure condition violates theses safety requirements using the formula shown in **(4)**:

((P=? [true U<=T "OmissionSpeed_Actuator_Out1"])/T)
((P=? [true U<=T "ComissionSpeed_Actuator_Out1"])/T)
((P=? [true U<=T "WrongPosition_Actuator_Out1"])/T)

By asking the model checker, we obtain the results shown in Fig. 5.4.



```
Property:
    ((P=? [ true U<=T "OE_Actuator_Out1" ])/T)

Defined constants:
    T=1

Method:
    Verification

Result:
    0.0030457020394000665
```

```
Property:
    ((P=? [ true U<=T "CE_Actuator_Out1" ])/T)

Defined constants:
    T=1

Method:
    Verification

Result:
    9.795199568269084E-4
```

```
Property:
    ((P=? [ true U<=T "CrE_Actuator_Out1" ])/T)

Defined constants:
    T=1

Method:
    Verification

Result:
    0.0029456530255754653
```

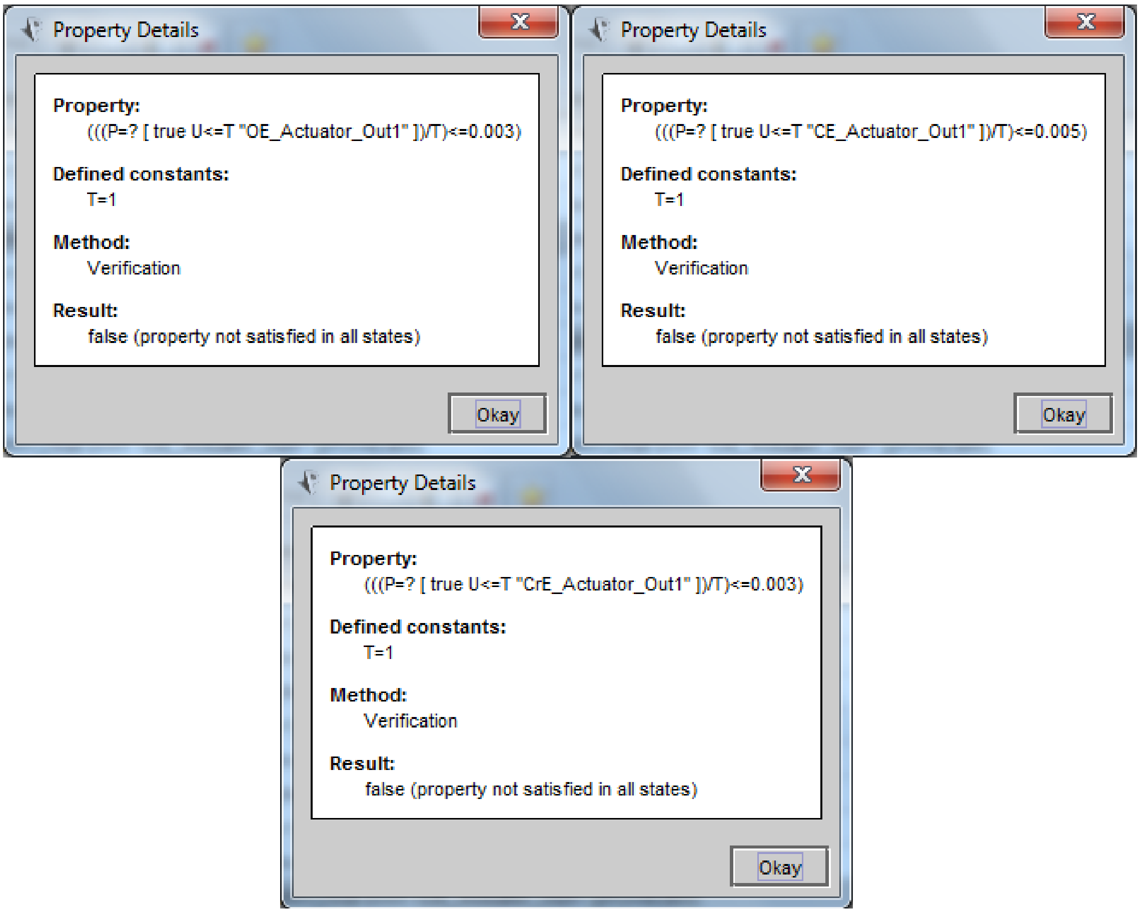
Fig 5.4. Results of expression verification (numerical value)

5.3 Quantitative Results

After checking these formulas, the model checker shows that only the first formula was not satisfied. Because the exact value of the average probability obtained via transient analysis for this situation was $3.04e^{-3}$. So the Prism result indicated that this failure condition was violated. Furthermore, we can check using the formula shown in **(4)**:

(((P=? [true U<=T "OmissionSpeed_Actuator_Out1"])/T)<=0.003)
(((P=? [true U<=T "OmissionSpeed_Actuator_Out1"])/T)<=0.005)
(((P=? [true U<=T "OmissionSpeed_Actuator_Out1"])/T)<=0.003)

if some failure condition satisfies these safety requirements considering all states of the model. Figure 5.5 shown that the Prism result was ***false***, indicating that all failure condition is not satisfied for, at least, one state of the model.



Property:
(((P=? [true U<=T "OE_Actuator_Out1"])/T)<=0.003)

Defined constants:
T=1

Method:
Verification

Result:
false (property not satisfied in all states)

Property:
(((P=? [true U<=T "CE_Actuator_Out1"])/T)<=0.005)

Defined constants:
T=1

Method:
Verification

Result:
false (property not satisfied in all states)

Property:
(((P=? [true U<=T "CrE_Actuator_Out1"])/T)<=0.003)

Defined constants:
T=1

Method:
Verification

Result:
false (property not satisfied in all states)

Fig 5.5. Results of expression verification (satisfaction of a property)

As we have said previously, this strategy can be performed in a hidden way by instructing the Prism model-checker to check each formula automatically; in such a way that only when a formula is violated this result can be sent back to engineers using Simulink plug-ins, for example. Thus the complete quantitative safety analysis can be hidden from the engineers.

So, from such reports, control engineers must adjust the system design by inserting more fault-tolerance features to avoid such failure violations. When all safety requirements are satisfied, the current system design (including its failure and repair rates) is acceptable. To show this analysis to certification authorities, the Markov model can be extracted from Prism by using certified tools like SHARPE or HARP [20].

Furthermore, one can also investigate scenarios of different phases and maintenance strategies using graphs of the instantaneous probabilities during a certain time interval. For instance, Fig. 5.5 is the result of evaluating the following formulas defined in (3), setting the T parameter from 0 to 100 hours.

P=? [true U<=T ("OmissionSpeed_Actuator_Out1")]
P=? [true U<=T ("CommissionSpeed_Actuator_Out1")]
P =? [true U<=T ("WrongPosition_Actuator_Out1")]

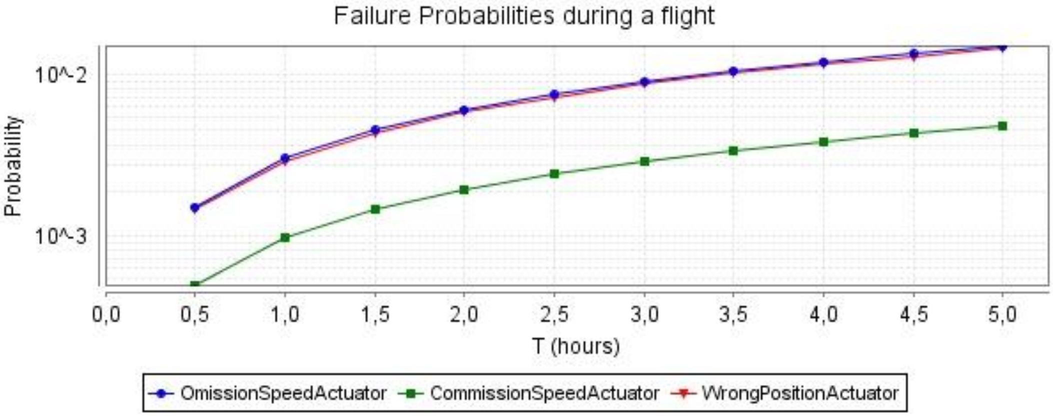


Fig 5.6. Instantaneous probability during a period of time

With respect to this quantitative analysis, the main advantage is that the Prism models basically use booleans and thus they are not so complex. To give an idea of the probabilistic model checking complexity, the effort to analyze the ECS design required 262,144 states and 3,858,432 transitions. But only a few seconds were necessary to analyze them using Prism 3.3 beta 1 in an Intel Core 2 Duo of 1.8 GHz, 2GB RAM, HD 160GB, Windows 7 Professional. It is worth noting that Prism supports models of more than 10^7 reachable states.

Chapter 6

Conclusion

The new generation of aircraft systems brings more advanced control systems to the context where size and complexity challenges the current verification and validation approaches. On the positive side, the recent adoption of model-based development tools, such as Simulink, by the aerospace industry, is making it feasible to use formal methods as verification solutions.

In the same way Simulink interacts with Matlab to provide the desired solution, Simulink can also interact with several different formal method solutions to tackle a wide variety of problems that the control system engineers have to confront in practice. Furthermore, the current tendency is to hide these formal method solutions completely in terms of Simulink interactions to ease the use by engineers to avoid any decrease in development productivity. For instance, Airbus [64] reports that formal methods were used and, with an equivalent effort of a usual test campaign, were able of finding problems (bugs) not detected by testing.

In this work we propose a systematic strategy to perform quantitative safety assessment of critical systems. Our approach generates a Prism specification from a Simulink diagram, annotated with failure logic. The strategy also creates CSL formulas that allow us to mechanically check whether all safety requirements are satisfied.

There are several potential benefits associated with the systematic approach we propose: an alternative to represent and analyze probabilistic models, understanding the context of the system, and the validation of required properties are a few examples.

Another potential benefit emerges from the safety assessment process. In the traditional fault-tree technique, several fault-trees are explicitly built even if all safety requirements are met. However, if a problem is detected in one of such fault-trees, the system architecture may be changed and several fault-trees (in some cases a considerable sub fault-trees which corresponds to low level systems) associated to that problem must be rebuilt. With our Prism based approach, no fault-tree is built. It only reports a safety violation, if one exists, indicating the failure mode [2, 10]. With Markov chains, for instance those created via Prism, it is possible to represent all failure conditions of a system with a single model. Also, checking the CSL formulas can be more efficient than creating several fault-trees. We consider this as a distinguishing feature of our approach when contrasted with the traditional fault-tree analysis technique.

Prism specifications are also interesting because they allow the creation and analysis of Markov chains in a more user-friendly and condensed way. They also ease the exploration of aspects such as latent and evident failure, monitoring and repair scheduling, which are essential to aeronautical systems. Furthermore, engineers can use the Prism specification (Markov chains) to investigate dynamic aspects of a system: experiments to check existing failure scenarios can be performed by simply changing the values of local variables of the model. Maintenance scheduling experiments can be created to determine the Minimum Equipment List, and Phased Mission and reconfiguration triggers based on synchronization with failure events [2, 8].

However, the current version of the Prism tool also has some limitations. Particularly, the tool has no facility to generate counter-examples when some property is violated. Fortunately, recent researches are already identifying counter-examples of stationary models, allowing a better traceability of the basic failures and facilitating the cycle of checking and validating the system design [21]. Unfortunately, this solution is not available in Prism yet. On the other hand, our research group proposes a work based on this dissertation, which explores quantitative analysis using CSP [63], and is able to generate traces and hierarchical fault-trees.

Nowadays, the low incidence of tools and methods that provide the development of trusted systems within the goals of dispatchability, safety requirements and costs is still a major challenge [6, 10]. Therefore, we think that the development of a model-based strategy for analyzing the safety and reliability of aircraft systems using a formal language is of great value.

6.1 Future Work

As future work we intend to mechanize the translation strategy and incorporate it as a plug-in in the Matlab/Simulink software. This allows immediate use of our work. From this, we will collect some metrics, check how much the strategy scales, and identify practical advantages/disadvantages of the strategy.

An obvious improvement to the current work is to capture the behavior of the components through its defined state machine, which can also be obtained from the Simulink tool. Also, considering the system reconfiguration and failure covering aspects will provide a more detailed fault tolerant modeling, which can capture the dynamic information in the same way as the static information.

Another concern we intend to tackle in the future is the size and complexity of the Markov chains generated by Prism. This can make it difficult for our proposal to scale in practice. Therefore we plan to investigate the use of abstraction strategies to reduce the Markov chains, such as State Aggregation and Model Truncation, as well as compositional verification.

When any system requirement is not satisfied then the current system design must be revised and improved to reduce the likelihood of a hazard occurring, and ensure the correct execution of its functions. Thus another direction is to study refinement relations that allow obtaining an improved design from a previous version while preserving the original characteristics concerning functionality. We see that fault-tolerance patterns and analysis of model evolution as feasible alternatives to achieve this goal. The initial steps to realize this was also reported in [43], where we propose a methodology to assess the entire process.

Moreover, we also intend to study the stochastic behavior of a system, considering an open-loop model to represent a specific aircraft mission, to evaluate the defined maintenance strategy using a more detailed transient analysis supported by CSL formulas. In some scenarios, a transient analysis of the open-loop model (without repairs) is useful. For instance, when determining the minimum acceptable system configuration for a dispatch and the length of time allowed for such a dispatch, we could know the worst case of instantaneous failure rate as a function of time, for a given configuration. We could also know the sensitivity of the worst-case instantaneous failure rate to variations in the dispatchability interval, to account for in-service waivers, and so on. Using this analysis we can determine the "kind" of the instantaneous failure rate as a function of time, enabling us to assess its sensitivity.

Finally, we intend to improve the model to allow other types of analysis can be performed such as Fussell-Vesely, that investigates about how much influence a component on a failure condition; analysis of uncertainty propagation to evaluate the propagation of uncertainty about the availability of the system and assess, for instance, the uncertainty distribution of MTTF or MTBF; traceability of failure: Markov is not causal and loses traceability. Thus, we will investigate the use of Bayesian Networks as a superset of Markov.

Bibliography

1. M. Stamatelatos, W. Vesely, J. Dugan, J. Fragola, J. Minarick III, J. Railsback. Fault Tree Handbook with Aerospace Applications. NASA Office of Safety and Mission Assurance, Washington, DC. Aug. 2002.
2. ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems, SAE Inc, Nov. 1996.
3. MIL-STD-882D – Standard Practice for System Safety. February 2000.
4. P. R. Serra, Safety Assessment of aircraft systems. 2º Edition. 2008.
5. Y. Papadopoulos, J. McDermid, R. Sasse, and G. Heiner. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. Reliability Engineering & System Safety, 71 (3):229-247, 2001.
6. R. D. Alexander and T. P. Kelly. Escaping the non-quantitative trap. 27th International System Safety Conference, pages 69-95, 2009.
7. M. Kwiatkowska, G. Norman and D. Parker. PRISM: Probabilistic Model Checking for Performance and Reliability Analysis. ACM SIGMETRICS Performance Evaluation Review, 36(4), pages 40-45.March 2009.
8. M. Kwiatkowska, G. Norman and D. Parker. Quantitative analysis with the Probabilistic Model Checker PRISM. Electronic Notes in Theoretical Computer Science, 153(2), pages 5-31, Elsevier. May 2005.
9. The MathWorks Inc. Simulink User's Guide, 2008.
10. J. A. McDermid, O. Lisagor, D. J. Pumfrey. Towards a Practicable Process for Automated Safety Analysis. 24th Int. System Safety Conference, 596-607, 2006
11. A. Joshi and M. P. Heimdahl. Model-Based Safety Analysis of Simulink Models Using SCADE Design Verifier. In SAFECOMP, volume 3688 of LNCS, pages 122–135. Springer-Verlag, Sept 2005.
12. Software Considerations in Airborne Systems and Equipment Certification. DO-178B, RTCA Inc., Washington D.C., December 1992.
13. O. A. Kerlund, P. Bieber, E. Boede, M. Bozzano, M. Bretschneider, C. Castel, A. Cavallo, M. Cifaldi, J. Gauthier, A. Griffault, O. Lisagor, A. Lüdtke, S. Metge, C. Papadopoulos, T. Peikenkamp, L. Sagaspe, C. Seguin, H. Trivedi, L. Valacca. ISAAC, A Framework for Integrated Safety Analysis of Functional, Geometrical and Human Aspects. In ERTS, 2006.
14. Y.Papadopoulos, M. Maruhn, Model-based Synthesis of Fault trees from Matlab-Simulink Models, Inter. Conference on Dependable Systems and Networks, 2001.

15. M. Bozzano and A. Villafiorita. Improving system reliability via model checking: The FSAP/NuSMV-SA safety analysis platform. In Proceedings of SAFECOMP 2003, LNCS 2788, Edimburgh, Scotland, UK, pages 49-62. Springer, 2003.
16. B. R. Haverkort. Markovian Models for Performance and Dependability Evaluation, volume 2090, of Lectures on Formal Methods and Performance Analysis, pages 38-83. Springer Berlin/ Heidelberg, 2001.
17. Saglimbene, Mark S. Reliability analysis techniques: How they relate to aircraft certification. Reliability and Maintainability Symposium, p. 218-222, 2009.
18. L. Grunske, R. Colvin, K. Winter. pFMEA: Probabilistic Model-Checking Support for FMEA. 4th Int. Conference on the QEST, 2007.
19. Clarke, O. Grumberg, and D. Peled. Model Checking. The MIT Press, 1999.
20. D. Siewiorek, R. Swarz. Reliable Computer System: Design and Evaluation, 3th edition, 1998.
21. H. Aljazzar, M. Fischer, L. Grunske, M. Kuntz, F. Leitner, S. Leue. Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counter Examples. p. 299-308, In QEST, 2009.
22. M. Bozzano, A. Cimatti, J. Katoen, V. Y. Nguyen, T. Noll, M. Roveri. The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems, Proceedings of the 28th Int. Conference on Computer Safety, Reliability and Security, September 15-18, 2009.
23. A. D. Dominguez-Garcia, J. G. Kassakianb, J. E. Schindallb, J. J. Zinchukc. An Integrated Methodology for the Dynamic Performance and Reliability Evaluation of Fault-tolerant Systems. Reliability Engineering & System Safety. Volume 93, Issue 11, November 2008, Pages 1628-1649.
24. Federal Aviation Regulations FAR part 25.1309: System Design and Analysis. Advisory Circular, FAA, USA.
25. P. Bieber, C. Castel, and C. Seguin. Combination of Fault Tree Analysis and Model Checking for Safety Assessment of Complex System. In Proc. 4th European Dependable Computing Conference, Volume 2485 of LNCS, pages 19-31, Springer-Verlag, 2002.
26. P. Bieber, C. Bougnol, C. Castel, J. P. Heckmann, C. Kehren, S. Metge, and C. Seguin. Safety Assessment with Altarica - Lessons Learnt Based on Two Aircraft System Studies. In 18th IFIP World Computer Congress, Topical Day on New Methods for Avionics Certification, Toulouse France, 26 - 26 August 2004. IFIP.
27. Bozzano, M. Villafiorita, A. The FSAP/NuSMV-SA Safety Analysis Platform. International Journal on Software Tools for Technology Transfer. Volume 9, Pages 5-24, June, 2006

28. Moura, Márcio José das Chagas; Droguett, E. A. L. A Continuous-Time Semi-Markov Bayesian Belief Network Model for Availability Measure Estimation of Fault Tolerant Systems. Pesquisa Operacional, v. 28, p. 353-373, 2008.
29. Marseguerra, M., Zio, E., Devooght, J., Labeau, P.E.: A concept paper on dynamic reliability via Monte Carlo simulation. Math. Comput. Simulat. 47, 371–382 (1998)
30. Smidts, C., Devooght, J.: Probabilistic reactor dynamics II. A Monte-Carlo study of a fast reactor transient. Nucl.Sci.Eng. 111(3), 241–256 (1992)
31. G.G. Infante-Lopez, H. Hermanns, and J.-P. Katoen. Beyond memoryless distributions: Model checking semi-Markov chains. In Process Algebra and Probabilistic Methods, LNCS 2165: 57–70, Springer-Verlag, 2001.
32. Papazoglou, I.A.: Markovian reliability analysis of dynamic systems. In:Aldemir, T., Siu, N.O., Mosleh, A., Cacciabue, P.C., Göktepe, B.G. (eds.) Reliability and Safety Assessment of Dynamic Process Systems, vol. 120 of NATO ASI Series F, pp. 24–43. Springer, Berlin Heidelberg New York (1994)
33. Siu, N.O.: Risk assessment for dynamic systems: an overview. Reliab. Eng. Syst. Safe. 43, 43–74 (1994)
34. J. B. Dugan, S. Bavuso, and M. Boyd. Dynamic fault-tree models for fault-tolerant computer systems. IEEE Transactions on Reliability, 41(3):363–77, 1992.
35. M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. Modelling with Generalized Stochastic Petri Nets. Wiley series in parallel computing. Wiley, New York, 1995.
36. Herbstritt, Marc; B, Eckard; Adelaide, Michael; Johr, Sven. AVACS Analysis of Large Safety-Critical Systems : A quantitative Approach.
37. Joost-Pieter Katoen, Ivan S. Zapreev, Ernst Moritz Hahn, Holger Hermanns, and David N. Jansen. The ins and outs of the probabilistic model checker MRMC. In Quantitative Evaluation of Systems (QEST), pages 167-176. IEEE Computer Society, 2009.
38. Marc Herbstritt, Ralf Wimmer, Thomas Peikenkamp, Eckard Böde, Michael Adelaide, Sven Johr, Holger Hermanns, and Bernd Becker. Analysis of Large Safety-Critical Systems: A quantitative Approach. AVACS Technical Report No. 8, SFB/TR 14 AVACS, Feb 2006.
39. Anthony Hall. Realising the Benefits of Formal Methods, Formal Methods and Software Engineering, Lecture Notes in Computer Science, LNCS 3785, Springer, p.1 – 4, 2005.
40. Y. Papadopoulos, J. A. McDermid, HiP-HOPS: Hierarchically Performed Hazard Origin and Propagation Studies. In SAFECOMP '99, Toulouse, LNCS 1698, pages 139-152, Sept. 1999.

41. Fenelon P., McDermid J.A., Nicholson M. and Pumfrey D.J, Towards Integrated Safety Analysis and Design, ACM Applied Computing Review, 2(1):21-32, 1994.
42. J. B. J. Jesus. Designing and formal verification of fly-by-wire flight control systems. Master's thesis, Federal University of Pernambuco, 2009.
43. A. Mota, A. Gomes, J. Jesus, F.Ferri and E. Watanabe. Evolving a Safe System Design Iteratively. Accepted for publication in Proceedings of SAFECOMP (2010).
44. Kurt Jensen. Coloured Petri Nets: A high level language for system analysis and design. In G. Rozenberg, editor, Advances in Petri Nets 1990, volume 483. Springer-Verlag, 1991. Also a technical report from the CS Dept, Aarhus University, DAIMI PB-338, Nov. 1990.
45. R. Alur and T. Henzinger. Reactive Modules. Formal Methods in System Design, 15:7-48, 1999.
46. M. Kwiatkowska. Quantitative Verification: Models, Techniques and Tools. In Proc. 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), pages 449-458, ACM Press. September 2007.
47. H. Hansson and B. Jonsson. A logic for reasoning about time and reliability. Formal Aspects of Computing, 6(5):512–535, 1994.
48. A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time Markov chains. In Proc. CAV'96, volume 1102 of LNCS, pages 269–276. Springer, 1996.
49. C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In Proc. CONCUR'99, volume 1664 of LNCS, pages 146–161. Springer, 1999.
50. M. Kwiatkowska, G. Norman and D. Parker. Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. International Journal on Software Tools for Technology Transfer (STTT), 6(2), pages 128-142. September 2004.
51. Ross, S.M. Introduction to Probability Models. 8ª Edition. 2003.
52. M. Kwiatkowska, G. Norman and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In T. Field, P. Harrison, J. Bradley and U. Harder (editors) Proc. TOOLS 2002, volume 2324 of Lecture Notes in Computer Science, pages 200-204, Springer. April 2002.
53. Oxford University Computing Laboratory. PRISM web site. Available in: http://www.prismmodelchecker.org/
54. Sorensen, E.V.; Nordahl, J.; Hansen, N.H.; From CSP models to Markov models. Software Engineering, IEEE Transactions, Volume: 19 Issue:6, page(s): 554 - 570, ISSN: 0098-5589, 1993.
55. Gul Agha, José Meseguer and Koushik Sen. PMaude: Rewrite-based Specification Language for Probabilistic Object Systems. Proceedings of the Third Workshop on

Quantitative Aspects of Programming Languages (QAPL 2005), Volume 153, Pages 213-239, May 2006.

56. Benveniste, A.; Fabre, E.; Haar, S.; Markov nets: probabilistic models for distributed and concurrent systems. Automatic Control, IEEE Transactions . Volume: 48, Issue:11, page(s): 1936 - 1950, ISSN: 0018-9286, Nov. 2003
57. Baier, C.; Ciesinski, F.; Grosser, M.; PROBMELA: a modeling language for communicating probabilistic processes. Formal Methods and Models for Co-Design, Proceedings. Second ACM and IEEE International, ISBN: 0-7803-8509-8, page(s): 57 - 66, Germany, 2004
58. Peikenkamp,T., Böede, E.,Brückner, I., Spenke, H.,Bretschneider, M., Holberg, H.-J.: Model-based safety analysis of a flap control system. In: Proceedings of the International Symposium INCOSE 2004 (2004)
59. C. A. R. Hoare. Communicating sequential processes. Commun. ACM, 21(8):666-677, 1978.
60. J. Woodcock and J. Davies. Using Z: Specification, Refinement, and Proof. Prentice Hall, 1996.
61. F. Somenzi. CUDD: CU Decision Diagram package. Public software, Colorado University, Boulder, 1997.
62. E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multiterminal binary decision diagrams: An efficient data structure for matrix representation. In Proc. International Workshop on Logic Synthesis (IWLS'93), pages 1–15, 1993. Also available in Formal Methods in System Design, 10(2/3):149–169, 1997.
63. A. Mota, Quantitative Analysis with CSP. Submitted to the Information Processing Letters (IPL), (2010).
64. Third International Conference on Software Testing, Verification and Validation (ICST 2010), http://vps.it-sudparis.eu/icst2010/
65. A. Gomes, A. Mota, A. Sampaio, F. Ferri, J. Buzzi. Systematic Model-Based Safety Assessment via Probabilistic Model Checking. Accepted for publication in Proceedings of ISOLA (2010).
66. Lars Grunske. Specification patterns for probabilistic quality properties. InWilhelm Schafer, Matthew B. Dwyer, and Volker Gruhn, editors, ICSE, pages 31–40. ACM, 2008.
67. A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: a new symbolic model checker. International Journal on Software Tools for Technology Transfer, 2(4):410–425, 2000.
68. Matthias Güdemann and Frank Ortmeier.Probabilistic Model-Based Safety Analysis, EPTCS 28, pp. 114-128, volume 1006.5101, 2010.

69. Elmqvist, J.; Nadjm-Tehrani, S.; Formal Support for Quantitative Analysis of Residual Risks in Safety-Critical Systems. High Assurance Systems Engineering Symposium, HASE 2008. 11th IEEE, ISSN: 1530-2059, page(s): 154 - 164, 2008.
70. Y. Papadopoulos, D. Parker, C Grante; A Method and Tool Support for Model-based Semi-automated Failure Modes and Effects Analysis of Engineering Designs. Proceedings of the 9th Australian workshop on Safety critical systems and software - Volume 47, Pages: 89 - 95, 2004

Appendix A

Elevator Control System.sm

```
ctmc

module PowerSource1

   powersource1_lowpower : bool init false;

   [] (!(powersource1_lowpower)) -> (5E-4) :  (powersource1_lowpower' = true);

   [Monitor_In1_Dependent_Repair] (powersource1_lowpower)
   -> (1/5) : (powersource1_lowpower' = false);

   [Monitor_In1_Repair] (powersource1_lowpower) -> (1) :
   (powersource1_lowpower' = false);

endmodule

formula LowPower_PowerSource1_Out1 = powersource1_lowpower;

module PowerSource2

   powersource2_lowpower : bool init false;

    [] (!(powersource2_lowpower)) -> (5E-4) : (powersource2_lowpower' = true);

   [Monitor_In2_Dependent_Repair] (powersource2_lowpower)
   -> (1/5) : (powersource2_lowpower' = false);

    [Monitor_In2_Repair] (powersource2_lowpower) -> (1):
   (powersource2_lowpower' = false);

endmodule

formula LowPower_PowerSource2_Out1 = powersource2_lowpower;

module Monitor

   monitor_switchFailure : bool init false;

   [] (!(monitor_switchFailure)) -> (1E-4) :  (monitor_switchFailure' = true);
   [] (monitor_switchFailure) -> (2/50) : (monitor_switchFailure' = false);

   [Monitor_In1_Repair] (!monitor_switchFailure)
   -> (2/5) : (monitor_switchFailure' = monitor_switchFailure);

   [Monitor_In2_Repair] (!monitor_switchFailure)
   -> (2/5) : (monitor_switchFailure' = monitor_switchFailure);

   [Monitor_In1_Dependent_Repair] (monitor_switchFailure) -> (1) :
   (monitor_switchFailure' = false);

   [Monitor_In2_Dependent_Repair] (monitor_switchFailure)-> (1) :
   (monitor_switchFailure' = false);

endmodule

formula LowPower_Monitor_Out1 = (monitor_switchFailure &
             (LowPower_PowerSource1_Out1 | LowPower_PowerSource2_Out1))
             | (LowPower_PowerSource1_Out1 & LowPower_PowerSource2_Out1);

module Reference

   reference_devicefailure : bool init false;
   reference_devicedegradation : bool init false;

   [](!reference_devicefailure) -> 2E-4 : (reference_devicefailure' = true);

   [](!reference_devicedegradation) -> (2E-4) :
   (reference_devicedegradation' = true);

   [] ((reference_devicefailure | reference_devicedegradation) &
   !(SystemFailure) ) -> (1/5) : (reference_devicefailure' = false) &
   (reference_devicedegradation' = false);

endmodule

formula OmissionSignal_Reference_Out1 = reference_devicefailure |
                                        LowPower_Monitor_Out1;
formula CorruptedSignal_Reference_Out1 = reference_devicedegradation;

module Sensor

   sensor_sensorfailure : bool init false;
   sensor_sensordegradation : bool init false;

   [](!sensor_sensorfailure ) -> (5E-4): (sensor_sensorfailure' = true);

   [](!sensor_sensordegradation) -> 5e-4: (sensor_sensordegradation' = true);

   [] ((sensor_sensorfailure | sensor_sensordegradation))  -> (1/5) :
(sensor_sensorfailure' = false)
   & (sensor_sensordegradation' = false);
```

```
endmodule

formula OmissionSignal_Sensor_Out1 = sensor_sensorfailure |
                   LowPower_Monitor_Out1 | OmissionSpeed_Actuator_Out1;

formula CorruptedSignal_Sensor_Out1 = sensor_sensordegradation;

module Component1

   component1_lossofcomponent1 : bool init false;
   component1_component1degradation : bool init false;

   [](!component1_lossofcomponent1) -> (9E-5) :
   (component1_lossofcomponent1' = true);

   [](!component1_component1degradation) -> (9e-5) :
   (component1_component1degradation' = true);

   [] ((component1_lossofcomponent1 | component1_component1degradation)) ->
(1/5) : (component1_lossofcomponent1' = false) &
   (component1_component1degradation' = false);

endmodule

formula OmissionSignal_Component1_Out1 = component1_lossofcomponent1 |
                   LowPower_Monitor_Out1 | OmissionSignal_Reference_Out1;
formula CorruptedSignal_Component1_Out1 = component1_component1degradation |
                         CorruptedSignal_Reference_Out1;

module Component2

   component2_lossofcomponent2 : bool init false;
    component2_component2degradation : bool init false;

   [](!component2_lossofcomponent2) -> (1E-4) :
   (component2_lossofcomponent2' = true);

   [](!component2_component2degradation) -> (1E-4) :
   (component2_component2degradation' = true);

   [] ((component2_lossofcomponent2 | component2_component2degradation)) ->
(1/5) : (component2_lossofcomponent2' = false) &
   (component2_component2degradation' = false);

endmodule

formula OmissionSignal_Component2_Out1 = component2_lossofcomponent2 |
                                        LowPower_Monitor_Out1 ;
formula CorruptedSignal_Component2_Out1 = component2_component2degradation |
                                         CorruptedSignal_Sensor_Out1;
```

```
module Component3

   component3_lossofcomponent3 : bool init false;
   component3_component3degradation : bool init false;

   [](!component3_lossofcomponent3) -> (6E-5) :
   (component3_lossofcomponent3' = true);

   [](!component3_component3degradation) -> (6e-5) :
   (component3_component3degradation' = true);

   [] ((component3_lossofcomponent3 | component3_component3degradation)) ->
(1/5) : (component3_lossofcomponent3' = false) &
   (component3_component3degradation' = false);

endmodule

formula OmissionSignal_Component3_Out1 = component3_lossofcomponent3 |
                   LowPower_Monitor_Out1 | OmissionSignal_Component1_Out1 |
                   OmissionSignal_Component2_Out1;
formula CorruptedSignal_Component3_Out1 = component3_component3degradation |
         CorruptedSignal_Component1_Out1 | CorruptedSignal_Component2_Out1;

formula CommissionSignal_Component3_Out1 = component3_component3degradation;

module Actuator

   actuator_lossofdriver : bool init false;
   actuator_lossofmotor : bool init false;
   actuator_mechanismjamming : bool init false;
   actuator_mechanismdegradation : bool init false;
   actuator_driverdegradation : bool init false;

   [](!actuator_lossofdriver) -> (1E-4) : (actuator_lossofdriver' = true);

   [](!actuator_lossofmotor) -> (1E-3) : (actuator_lossofmotor' = true);

   [](!actuator_mechanismjamming) -> (1E-3) :
 (actuator_mechanismjamming' = true);

   [](!actuator_mechanismdegradation) -> (1E-3) :
   (actuator_mechanismdegradation' = true);

   [](!actuator_driverdegradation) -> (1E-5) :
   (actuator_driverdegradation' = true);

   [] ((actuator_lossofdriver | actuator_lossofmotor |
   actuator_mechanismjamming | actuator_mechanismdegradation |
   actuator_driverdegradation)) -> (1/5):
   (actuator_lossofdriver' = false) & (actuator_lossofmotor' = false) &
   (actuator_mechanismjamming' = false) & (actuator_mechanismdegradation' =
   false) & (actuator_driverdegradation' = false);
```

```
endmodule

formula OmissionSpeed_Actuator_Out1 = actuator_lossofdriver |
actuator_lossofmotor | actuator_mechanismjamming | LowPower_Monitor_Out1 |
OmissionSignal_Component3_Out1;

formula WrongPosition_Actuator_Out1 = actuator_mechanismdegradation |
actuator_driverdegradation | CorruptedSignal_Component3_Out1;

formula CommissionSpeed_Actuator_Out1 =  actuator_driverdegradation |
                                CommissionSignal_Component3_Out1;

formula SystemFailure = OmissionSpeed_Actuator_Out1 &
             WrongPosition_Actuator_Out1 & CommissionSpeed_Actuator_Out1;
```