# A systematic strategy to perform quantitative safety assessment of Simulink diagrams using Prism - Technical Report

Adriano Gomes, Alexandre Mota, Augusto Sampaio, Joabe Jesus

Universidade Federal de Pernambuco, Centro de Informática
P.O.Box 7458 - Zip 50740-540, Recife, Brazil
{ajog, acm, aças, jbjj}@cin.ufpe.br

**Abstract.** Safety assessment is a well-known process for assuring the trustworthiness of critical aeronautic systems. Inside it, quantitative safety assessment aims at providing precise information to show that the safety requirements for the certification of system design are met. In this paper we propose a quantitative model-based safety assessment process, fully automatic. It starts by translating safety information from Simulink diagrams into Prism models and properties. With the Prism model-checker, we can find whether a safety requirement was violated for the whole system as well as identify scenarios of safety maintenance tasks and intervals. We present our work using a representative aircraft case study.

**Keywords:** Quantitative Safety Assessment, Prism, Model Checker, Markov Analysis, Safety Analysis of Aircraft Systems,

## 1 Introduction

Traditionally the Quantitative Safety Assessment of aircraft systems has been based on Fault Tree Analysis (FTA) [FTA Handbook] method, that despite its limitations, meets the ARP 4761 (Aerospace Recommended Practice) [ARP], a guidelines and methods for conducting the safety assessment process, and is followed and referenced by the certifying authority and the industrial applications. In practice, FTA has been widely used during this process mainly because it is conceptually simple and easy to understand [ARP]. However, this process is usually expensive and requires much time and effort to be validated, because it need the application of engineering and management principles, criteria, and techniques to optimize all aspects of safety within the constraints of operational effectiveness, time, and cost throughout all phases of the system life cycle [ARP 4754, McDermid_Towards].

A critical point in this process is the use of the FTA to perform Quantitative analysis for failure conditions (potential failure that can affect some system function). The goal on this analysis, usually performed during the phases of PSSA (Preliminary System Safety Assessment) and SSA (System Safety Assessment) of the process, is to satisfy the quantitative safety requirements (probability and criticality constraints) established for each critical failure condition to avoid or render unlikely the occurrence of each. Currently, the use of the FTA method provides no advantage in terms of cost-effective, because it have to generate the FT for each fault condition to be considered and if the proposed design does not satisfies just one of constraints, it have to be revised and improved to reduce the likelihood of the hazard occurring, restarting the analysis process and therefore causing much rework. A civil or military aircraft design are only allowed to operate whether corresponding certification authorities approve the system and one of the requirements for this is that the degree of safety of the system according to FAR 25.1309 [3], which requires that all safety requirements considered must be satisfied. Considering this scenario, a solution to improve and optimize this analysis is very relevant [Ref?].

In this paper, we propose an alternative to provide a cost-effective quantitative safety assessment based on a model-based approach supported by Prism []. Our solution acts on Quantitative analysis for failure conditions and a Safety related tasks and intervals over the Safety Assessment Process (during the PSSA and SSA stages) using Markov models []. Thus, safety constraints can be analyzed using probabilistic formal models specified in Prism, one can deal with Markov models indirectly and in a high-level representation [15]. This models results from the integration of analysis and information generated by all the steps involved in the safety assessment process (e.g.: FHA, CCA, PSSA, SSA, IF-FMEA [Ref], an extension of FMEA) that can be guided by a model-based solution like HiP-HOPS [X] or through a design tool such as Simulink [Ref?].

The work uses this idea by providing a rule-based mapping from a Simulink diagram, annotated with tabular system failure logic, to a Prism model, augmented with a set of probabilistic temporal logic formulas to analyze the safety aspects of the resulting model. The resulting artifact is a automatic quantitative safety assessment package, where by simply executing the Prism model checker one can check whether the system satisfies its safety constraints or not, without building any fault-tree []. Furthermore, as we are using Markov based formalism, we can also investigate scenarios of different phases and strategy maintenance.

The main contributions of this paper are:

– A quantitative model-based safety assessment process, fully automatic;
– Simulink-based translation rules to create formal models in Prism;
– A cost-effective quantitative analysis to cover all fault conditions considered, allowing the checking of the violation from a single model checking;
– A way to use formal method as a support to verify and validate safety requirements of aircraft systems.

This work is organized as follows. In the next section we present an overview about the Safety Assessment Process and model-based scenarios, in Section 3 we show our main contribution based on the probabilistic model checker Prism, where in Section 3.1 we briefly explain our Quantitative Safety Analysis using Markov models, in Section 3.2 we present our translation rules and in Section 3.3 we discuss the soundness and completeness of our strategy. In Section 4 we show the application of our strategy in a simple aircraft subsystem and in Section 5 we discuss about some related works. Finally, in Section 6 we show our conclusions and future work.

## 2 Quantitative Safety Assessment

Safety Assessment process involves several complex and detailed phases and activities as can be seen in Figure 1. During this process, the safety requirements will decompose in parallel with the system design, and will typically introduces qualitative and quantitative safety requirements for the top-level and subsystem design. It is a systematic and hierarchical method used to define the high-level airplane as well as system safety goals (maximum tolerable probability) that must be considered in the proposed system architectures. Failure Hazard Analysis (FHA) identifies and classifies failure conditions[1], generating requirements such as “show that failure condition X doesn’t shall to occur more frequently than $10^{-9}$ times per flying hour” or “No catastrophic failure condition result from a single failure". Failure rates will be allocated to different components and their failure conditions in such a way that satisfying the component level requirements (SSA) will satisfy the system level requirements (Integration cross check) [1, Towards].

SSA is based on some top-down analysis techniques such FTA, Markov Analysis and Dependence Diagram and uses quantitative values obtained from the Failure Mode and Effects Analysis (FMEA) as well as also include results of the Common Cause Analysis (CCA). The safety analysis role is met these particular requirements and the justification for the design concept. Considering the failure conditions identified in the FHA, these techniques can be applied mainly to determine:

- What single failures or combinations 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.

So, at the certification of an aircraft, for each failure condition, should be determined if the rate targets are met. In accordance with the certification authorities, the proposed system design must assuring that hazardous and catastrophic failure conditions have an average failure probability inferior to $10^{-7}$ and $10^{-9}$ per hour. These kind of failure may be satisfactorily analyzed on a quantitative aspect (in addition to qualitative analysis), because they are more critical. So, the average probability of occurrence per flight hour for each failure condition must be calculated assuming a typical flight of average duration and considering the appropriate exposure time and at risk times [ARP].

[1] Hazards are identified and classified by its severity (the worst credible effects on aircraft operations: *No Effect, Minor, Major, Hazardous, and Catastrophic*).

## 2.1 Model-based Safety Assessment

In the safety-critical systems domain there is an increasing trend towards model-based safety assessment [5, mais…]. It proposes to extend the existing model-based development activities (e.g. simulation, verification, testing and code-generation) that based on a formal model of the system expressed in a notation such as Simulink [X] or SCADE [X]) to incorporate safety analysis. Thus artifacts such as FT, Markov diagrams and flowchart can be automatically generated.

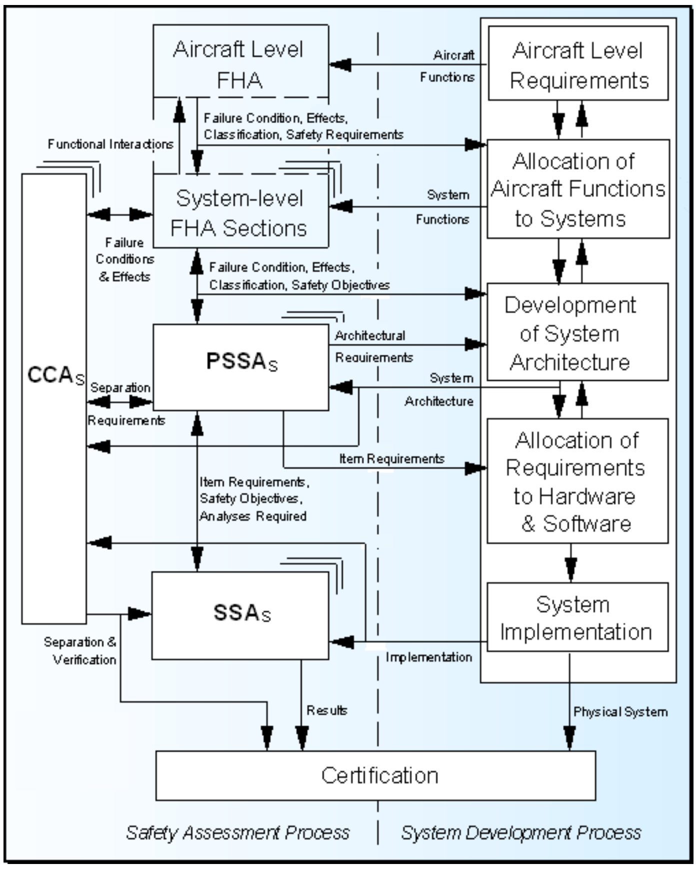
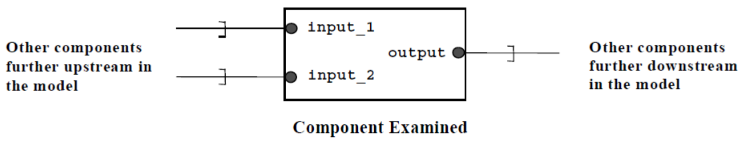

Aircraft Level FHA
Aircraft Functions
Aircraft Level Requirements
Failure Condition, Effects, Classification, Safety Requirements
Functional Interactions
Allocation of Aircraft Functions to Systems
System-level FHA Sections
System Functions
Failure Conditions & Effects
Failure Condition, Effects, Classification, Safety Objectives
Development of System Architecture
Architectural Requirements
PSSAs
CCAs
Separation Requirements
System Architecture
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. 1.** Overview of Safety Assessment Process

These new alternatives are very interesting because they are simple, compositional and do not need complex engineer's skills to be applied. In addition, they may become more powerful if they make use of formal methods [our new NUSMV-ISAAC, Anjali other?], since formal methods provides a set of effective tools (theorem provers, model checkers, static checkers, etc.) in order to automate the most of the analysis while attempting to guarantee the correctness of due process certification. It can act appropriately on support of verification and validation of the safety requirements [ ].

Most of the solutions of model-based safety assessment incorporate the FMEA or IF-FMEA (Interface-Focused FMEA) analysis on its design using a hierarchical tabular structure (see Figure 2) [4, 17]. This is a graphical notation for the representation of the transformation and propagation of failure in a system, allowing that complex systems are modeled as hierarchies of architectural diagrams. This notation is semantically and syntactically linked to the design representation of the system.


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.** IF-FMEA of a hypothetical component system [14]

So considering the structure of the design model expressed in a tool (e.g.: Simulink, SAM), the component failure characterizations (IF-FMEA tables) are overlaid over the system model. This solution creates a failure logic model of the system based on the result of an FHA analysis and can be used to perform a systematic safety analysis.

To illustrate this solution, we consider a hypothetical Actuator Control System showed in Figure 3. Its function is to control the displacement of an electrical actuator (Further details see Section 4).

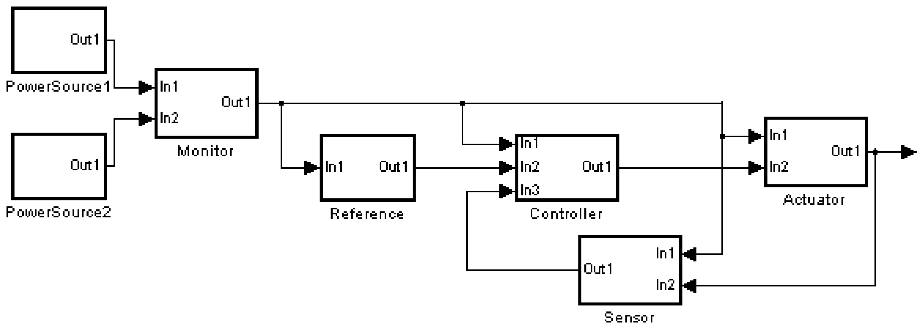


**Fig. 3.** Functional Model of the Actuator Control System

To capture Figure 3 (organization and component interconnections) in tabular form, we use a topology table (see Table 1). For the sake of space, we have omitted the table of Figure 2 (required for each component of the system). Part of the failure conditions of this example is given in Table 2. 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.

**Table 1.** Topology table of the Actuator 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 |
| … | … | … | … |

Following 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). A OmissionSignal deviation can be exhibit in the Reference when a internal (ReferenceDeviceFailure) or external (LowPower via its In1 port) occur. Reference still can exhibit a CorruptedSignal deviation when a ReferenceDeviceDegradation occurs.

**Table 2.** Set of deviation for the Actuator Control System

| Component | Deviation | 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 |
| … | … | … | … |

## 3 Proposed Strategy

In this section we present a methodology that performs quantitative analysis of aircraft systems using probabilistic formal models specified in PRISM. By using the Prism model checker we can detect whether any critical safety requirements all failure conditions is violated without building any fault-tree. This work performs safety analysis based on Markov formal models.

Most of the techniques to create probabilistic formal models of aeronautic systems are highly subjective, because they are dependent on the skill of engineer that specifies the model in an ad hoc fashion [TimKelly, Pfmea]. But instead of creating a Prism specification implicitly via a tool, we follow a systematic strategy by providing formal translation rules that transform a failure logic model of a system into a Prism specification. Thus, the approach captures the failure behavior of components in the formal model, preserving the failure logic, the maintenance and monitoring strategy and the hierarquical system. This is alternative to easily integrate a safety assessment process supported by formal verification to some consolidated model-based tool used to design and simulation of aircraft systems like Matlab-Simulink.

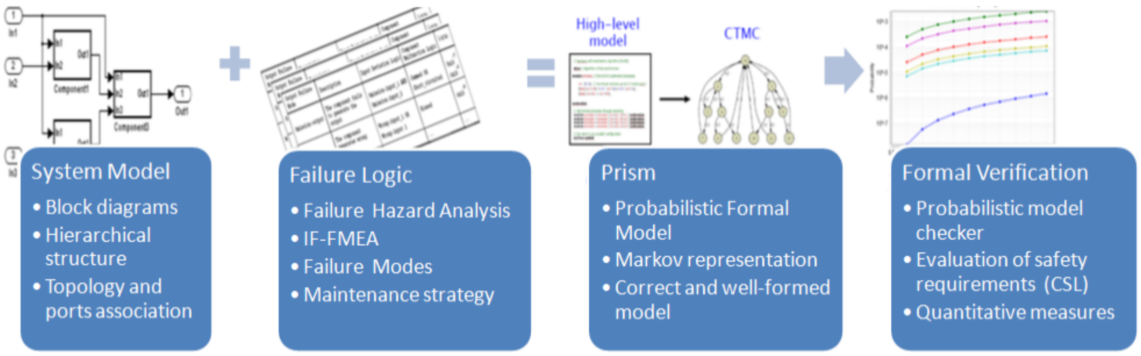


**Fig. 4.** Overview of proposed strategy

Figure 4 presents an overview of our proposed strategy. It starts collecting the system description which contains the system block diagrams and its failure logic model. This model can be constructed from the IF-FMEA technique during the PSSA and SSA stages in an integrated fashion with traditional tools for modeling like Matlab-Simulink. During these steps, the corresponding tabular annotations of system are created and stored in this tool and are commonly accessed to provide some system analysis. Here they are extended and used as input to create a formal model of system.

Appling our translation rules, we produce a formal model of system in a correct Prism specification. This model has a Markov representation and captures the semantics of failure logic model of system. And from the Prism specification, one can automatically perform quantitative analysis as well as check whether there is any failure condition violating its criticality level.

## 3.1 Quantitative Safety Analysis using Markov Models

Considering the verification means used for aircraft certification, we can calculate the average failure condition rate during finite time period, applying a steady-state or transient analysis over Markov models [Ref?]. Transient analysis represents the average instantaneous failure rate over a single period T and can be conducted on either closed-loop or open-loop models, i.e., models with or without repairs, whereas the steady-state analysis approximates the long-term average failure rate over multiple intervals of duration T and can be only conducted on closed-loop models. See the solutions illustrated in Figure 5.

The choice over these type of analysis depends on how are treated the system repairs. The close-loop solutions consider repair transitions as if they occur at constant rates, which can be conservatively represented by a constant repair rate of 1/T per hour, where T is the inspection time.

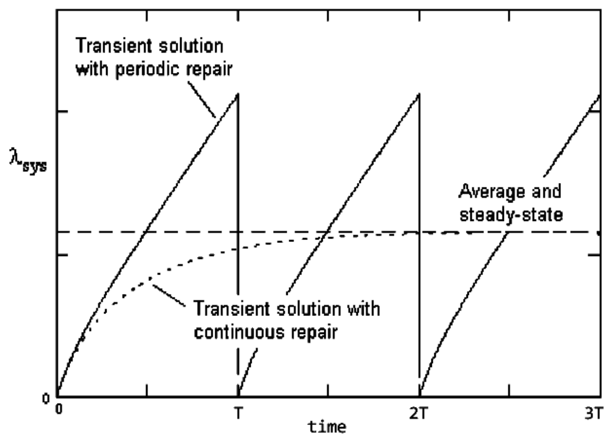


**Fig. 5.** Graph plotting the common behavior of different Markov analysis.

Our Markov chains are a close-loop models (i.e., models with repairs), composed of a set of discrete states, each of them is the representation of the state (faulty, operational and degraded) of each component failure mode. The transition occurs over constant rates and represent which state changes are possible and that often they occur. Briefly, these models consist of representations of chains of events, i.e. transitions within the system that under the safety assessment, match the sequence of failures and repairs. This charge requires the use of exponential probability distributions for modeling of failure rates and repair.

Considering the verification means used for system validation, we may calculate the average failure condition rate, applying a steady-state analysis over Markov models. It provides adequate accuracy for most practical purposes, because knowing that critical systems are modeled to deal with latency, almost all components affecting the functionality of a critical system are monitored or inspected at fairly short intervals of time, and repaired or replaced if they are found failed. Therefore the system is repeatedly restored to its operational state, and our main interest is in determining the long-term average failure condition rate over many such maintenance cycles. As a result, the steady-state solution of the closed loop model is usually what the analyst needs to determine [Falta referencia!].

### 3.1.1 Prism as Support to modeling and analyzing

Prism is a high-level abstract language that can be used to model the behavior of fault-tolerant system. Prism can be an alternative to compression of models and generates and analyzes, by a hidden way, the diagrams of transitions between states (Markov chains) [Ref].

Prism supports CSL [Ref], a temporal language used to generate the PRISM property specification to verify the system requirements on PRISM Model Checker. Using operators of Prism, such as P (transient) and S (steady-state), our strategy are able to perform transient or steady-state analysis over the model, allowing to reason about the probability of executions. The operator S checks the system behavior in the steady-state (long term). With the formula

$$\mathbf{S<=}\ \boldsymbol{10^{-9}}\ \mathbf{[\ "Failure\ Condition"\ ]}$$

We can check if in the long term, the probability that a failure condition occurs could be less than $10^{-9}$. Note that such an expression answers "yes" or "no", based on quantitative analysis (the result value is implicit: average failure rate). We can also check the probability itself by using a slightly different question to Prism:

$$\mathbf{P\ =?\ [\ true\ U^{<=3600}\ "SystemFailure"\ ]}$$

This will return the instantaneous probability of the system will fully-failed within 3600 time units. Therefore, Prism can support both Markov analysis solutions (steady-state or transient analysis) offering quantitative measures to safety requirements validation. Also experiments can be done to allow the user to investigate scenarios of different phases and strategy maintenance using graphs.

Moreover, the used model checking has two advantages if compared to other formal verification methods. First it is fully automatic, and second is that the generated model in Prism covers all fault conditions considered, allowing the checking of the violation from a single model checking analysis[2].

The primary limitation of 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.

### 3.1.2 Input Data Model

Recall from Section 2.1 that the failure logic model of a system can be captured by hierarchical tabular structures (Figure 3, Table 1 and 2). Actually, these tables are a concrete representation of system failure model with utilizes some notation to modeling each component failure characterization and propagation. Despite all these information are very consistent and integrated with relation to system failure mode and propagation over its components hierarchy, it is not sufficient to create a

[2] As we want an automatic way of analyzing the entire system at once, we can apply the Prism model checker in batch mode to check it each one of the formula corresponding to a failure condition being analyzed gradually.

probabilistic formal model and performs a quantitative analysis using Markov models. For represent aeronautical systems consistently and in accordance with the current scenario, we need we need more information to model the not monitored failures of the system. This involves whether the components can fail in a latent or evident way, if a component is monitored or not and how often repair for each one. Thus, we extend this modeling notation with the addition of such information.

The first information to be incorporated was the classification of each basic component of the system about the monitoring of its faults. Some components are checked before each flight to confirm that it is working, and repaired if necessary. So, this type of component is called self-monitored because we need to know if it is working at the start of each flight. But, some aircraft systems include components which are not inspected every flight. Failures of this type of component are called latent because they are not detected unless another failure occurs or a scheduled maintenance. For this last type of component we must consider two situations: Component with external monitoring and unmonitored components. The first one type of components is monitored continuously by an independent monitor. If it fails and the monitor is working, the component can be repaired before the next dispatch. If the monitor is not working, the component can fail latently. The last one type represents all components that are not monitored. So, thus failure of these components can obviously be latent too. Its faults only are checked at its periodic maintenance interval. In short, we need to distinguish between a monitored failure and unmonitored failure of the component because the implications of unmonitored failure are likely to be more severe in safety analysis.

Based on reliability and safety factors (dispatchability, MTBF, severity, redundancy, and other several reasons) the periodic inspection/repairs intervals for each component are defined. This is the second information that we incorporated to the input model. Next, we present a summary of this additional information:

**Table 3.** Definition of the additional information

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

Considering this assumptions, the tabular annotation was expanded to store this data. Table 1 shows its new additional information.

**Table 4.** 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 |

…  …  …  …

By the information showed on previous section, is possible to create systematically a formal specification in Prism to represent the system failure model. Applying a set of translation rules we can generate the probabilistic formal model. Before details the model construction, is necessary to represent the tabular template in a logical schema. So, we need to capture these tables as mathematical elements and defined its syntax. Just as the Simulink tool creates n-dimensional arrays to represent these values, here we define several typed structures - shown in Figure 5 - to better represent all information. This consists of set of structures, representing kinds of things of significance in the domain, and relationships assertions about associations between pairs of types.

```
System              ::=  System_Name  X Seq(Subsystem)
Subsystem           ::=  System | Module
Module              ::=  Module_Name X Type X Seq(Port)
                          X Seq(Deviation) X Seq(Malfunction)
                          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 AssociatedPort
MS_Type             ::=  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         ::= ℝ  | empty
InspectionTime      ::= ℝ
Rate                ::= ℝ
```

**Fig. 6.** Defined types based on tabular annotations

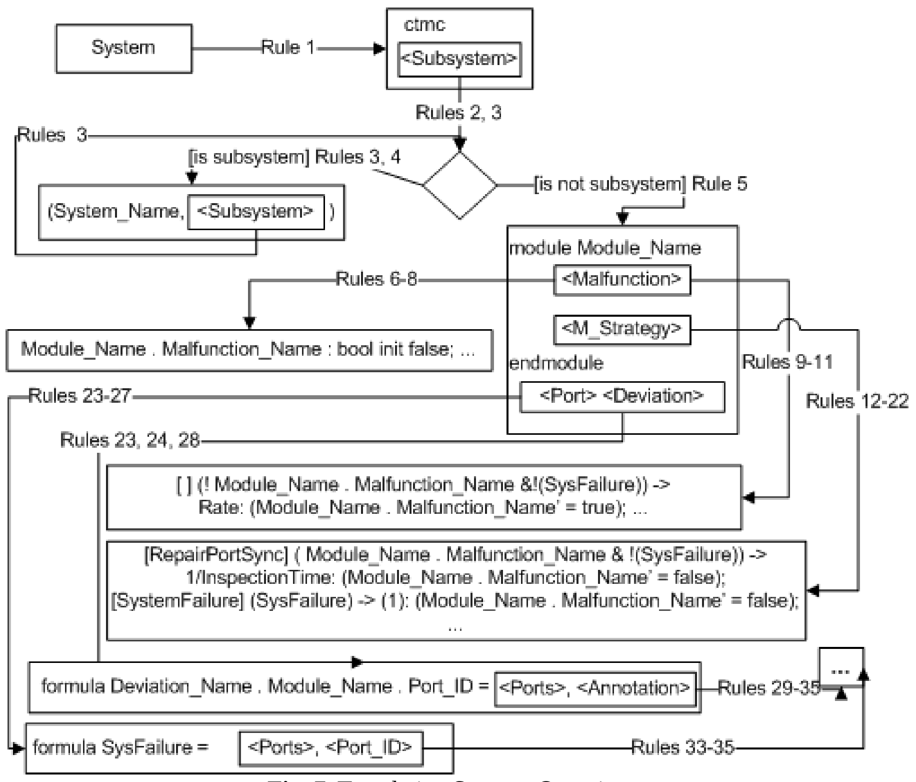
We start by considering a system (`System`) as a structure that contains a name (`System_Name`), a list of subsystems (`Seq(Subsystem)`). Each subsystem can be another system or a module; because components can also be systems. A module (`Module`) represents the lower level component that contains a 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` (it represents the identifiers of input/ output port of components) and an `AssociatedPort` (it stores the connected port of others components). `Annotation` is a boolean expression that represents the failure logic of failure conditions. Its definition considers And/ Or operators and its 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` also represents real numbers used to the rate of occurrence of some malfunction and repair, respectively.

## 3.2 Rules

In this section we present our rules. Our strategy applies a set of translation rules which are based on the data structures of Figure 4. To ease the overall understanding about their applicability we also provide the expected sequence of their application in Figure 7. Here, we will describe the main concept and description of these rules. The translation strategy is divided into the following steps:

- **Parsing the model:** The model is read from its textual and tabular representation of the design model expressed in Simulink. Irrelevant information about the graphics of the model is discarded and all already described input information is extracted.
- **Data Manipulation**: The data collected are organized following the syntax described in the previous section, allowing the translation rules can be applied to generate the Prism specification.
- **Model Generation**: In this step, the generated structure from previous step is processed and their respective Prism specification is generated as output according to the defined semantic rules of translation.


System
Rule 1
ctmc
<Subsystem>
Rules 2, 3
Rules 3
[is subsystem] Rules 3, 4
(System_Name, <Subsystem> )
[is not subsystem] Rule 5
module Module_Name
<Malfunction>
Rules 6-8
<M_Strategy>
endmodule
Module_Name . Malfunction_Name : bool init false; ...
Rules 9-11
Rules 12-22
Rules 23-27
<Port> <Deviation>
Rules 23, 24, 28
[ ] (! Module_Name . Malfunction_Name &!(SysFailure)) -> Rate: (Module_Name . Malfunction_Name' = true); ...
[RepairPortSync] ( Module_Name . Malfunction_Name & !(SysFailure)) -> 1/InspectionTime: (Module_Name . Malfunction_Name' = false); [SystemFailure] (SysFailure) -> (1): (Module_Name . Malfunction_Name' = false); ...
formula Deviation_Name . Module_Name . Port_ID = <Ports>, <Annotation>
Rules 29-35
...
formula SysFailure = <Ports>, <Port_ID>
Rules 33-35
**Fig. 7.** Translation Strategy Overview

The strategy always starts by applying Rule 1. Rule 1 states that we are dealing with a CTMC Markov model and calls 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 repairs 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). Note that some rules are missing because they are very similar to others presented. For instance, Rules 6 and 7 are missing because they are equivalent to Rules 2 and 3.

### 3.2.3 Compound Systems and subsystems

Now we present our rules in detail. We start by Rule 1 which takes 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 themselves. They access each component of this system and call the function *module* recursively for each component (Rules 4 and 5). For the Actuator Control System, the implementation of these rules creates 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 }| $^{subsystem}$

...

Step9: |{<Term>}| $^{subsystem}$ -> |{ Term }| $^{module}$

### 3.2.4 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 5 takes as input a tuple containing the module name, its type, its set of ports, its set of deviation logics, its malfunctions, maintenance strategy and its inspection time. The module name (`MName`) is used to name the Prism module (note the keywords ***module*** and ***endmodule***). Inside the module, the function *declars* is called to create the declaration part and 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 patterns matching the Rule 5 will be used in its translation that is shown below:

module Monitor

|{ Monitor, [Malfunction1] }| *declars*
|{ Monitor, [Port1, Port2, Port3], [Malfunction1] }| *actions*
endmodule
|{ Monitor, [Port1, Port2, Port3], [Malfunction1] }|formulas

For the Controller, which is a subsystem of the actuator and comprises three components, the Rule 4 will apply:

|{ [Component1, Component2, Component3] }| subsystem

### 3.2.5 Declarations

Malfunctions are representations of possible failures within a component. To capture this feature in Prism, for each component malfunction, boolean local variables with initial values equal to *false* are defined. The 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. The rule 8 uses each component malfunction to generate the declaration of its respective local variable inside the module block.

**Rule 6** |{ Module_Name, <M>: tail }| declars ->
|{ Module_Name, M }|*declar*
|{ Module_Name, tail }|*declars*,

**Rule 7** |{ Module_Name, <M> }| declars -> |{ Module_Name, M }|*declar*

**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 Name (`MName`) and Malfunction Name (`MfName`) are used to create the local variable name.

### 3.2.6 Failure Transition Command

PRISM transition commands are responsible to update the state of module local variables. We can translate rates and logic expression present in malfunction table to PRISM transition commands able to updates the malfunction state based on its failure rate. Thus, for each component malfunction, a state transition command is created. 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** |{ Module_Name, Ports, <M>: tail }| *commands* ->
|{ Module_Name, Ports, M }| *command*
|{ Module_Name, Ports, tail }| *commands*

**Rule 10** |{ Module_Name, Ports, <M> }|*commands* -> |{ Module_Name, Ports, M }|*command*

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)` }|$^{command}$ $\Rightarrow$

**`[] (!(MName .MfName) & !(SysFailure)) -> Rate: (MName .MfName'=`*`true`*`);`**

### 3.2.1 Repairs Transitions 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 monitoring of its faults. The difference between Rules 12, 13 and 14 lies in the treatment of the type of maintenance strategy. Rule 12 considers two types: `Self-monitored` and `Unmonitored` (Note the **provided** clause), whereas Rules 13 and 14 tackle the other cases: `Monitored` and `Monitor`, respectively. A same command is created for all cases whose always deactivate the module malfunctions if the system is fully failed.

**Rule 12** |{ `MName,Ports,Malfuncs,(MSType,AssocPort),IT`}| $^{repairCommands}$ $\Rightarrow$

**`[] ((`** |{ `MName, Malfuncs` }|$^{orLogic}$**`) & !(SysFailure)) -> (1/IT):`**

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

**`[SystemFailure] (SysFailure) -> (1):`** |{ `MName,Malfuncs` }|$^{update}$**`;`**

**provided** `MSType = Self-Monitored` or `MSType = Unmonitored`

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). However, if the component is `Monitored`, its repair commands must be synchronized with the external component that is monitoring it (function *monitoredRCommmand*).

**Rule 13** |{ `MName,Ports,Malfuncs,(MSType,AssocPort),IT` }|$^{repairCommands}$ $\Rightarrow$

|**{** `Malfuncs, AssocPort, IT` **}**|$^{\text{monitoredRCommand}}$

**`[SystemFailure] (SysFailure) -> (1):`** |**{** MName, Malfuncs**}**|$^{\text{update}}$**`;`**

**provided** `MSType = Monitored` and `AssocPort ≠ empty`

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 14** |{ `MName,Ports,Malfuncs,(MSType,AssocPort),IT` }|$^{repairCommands}$ $\Rightarrow$

**`[] ((`**|**{** `MName, Malfuncs` **}**|$^{\text{orLogic}}$**`) & !(SysFailure)) -> (1/IT):`**

|**{** `MName,Malfuncs`**}**|$^{\text{update}}$**`;`** |**{** `MName,Malfuncs,Ports,IT`**}**|$^{\text{sincronizedRCommand}}$

**`[SystemFailure] (SysFailure) -> (1):`** |**{** `MName, Malfuncs` **}**|$^{\text{update}}$**`;`**

**provided** `MSType = Monitor`

Rules 15 through 18 are used to define the synchronized repair commands between the monitored (Rule 15) and the monitoring component (Rule 18). Rules 16 and 17 do not produce Prism code. They work similarly to Rules 2 and 3.

**Rule 15** |{ `Malfuncs, (MName, PortID), IT` }| $^{\text{monitoredRCommand}}$ ⇒

```
[MName . PortID . DependentRepair] (( |{MName, Malfuncs}|^orLogic ) &
    !(SysFailure)) -> (1/IT): |{MName, Malfuncs}|^update ;
[MName . PortID . Repair] ( |{MName, Malfuncs}|^orLogic ) -> (1):
     |{Malfuncs}|^update ;
```

**Rule 16** |{MName, Malfuctions, <P> : tail , IT}| $^{\text{monitorRepairCommand}}$ ->
|{ MName, Malfuctions, P, IT}| $^{\text{sincronizedRepairCommand}}$
|{ MName, Malfuctions, tail , IT}| $^{\text{monitorRepairCommand}}$

**Rule 17** |{MName, Malfuctions, <P> , IT}| $^{\text{monitorRepairCommand}}$ ->
|{ MName, Malfuctions, <P>, IT}| $^{\text{sincronizedRepairCommmand}}$

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

```
[MName . PortID . Repair] (( |{ MName,Malfuncs}|^OrLogic ) &
    !(SysFailure)) -> (1/IT): |{ MName,Malfuncs}|^update ;
[MName . PortID . DependentRepair] (( |{ MName,Malfuncs}|^orLogic )
    -> (1): |{MName,Malfuncs}|^update ;
```

**Rule 19** |{ MName, <(Malfunction_Name, Rate, Annotation)> : tail}| $^{\text{OrExpression}}$ ->
MName **.** Malfunction_Name | |{ tail }| $^{\text{OrExpression}}$

**Rule 20** |{ MName, <(Malfunction_Name, Rate, Annotation )> }| $^{\text{OrExpression}}$ -> MName **.** Malfunction_Name

**Rule 21** |{ MName, <(Malfunction_Name, Rate, Annotation)> : tail}| $^{\text{Update}}$ ->
(MName **.** Malfunction_Name' = false) ***&*** |{ tail }| $^{\text{Update}}$

**Rule 22** |{ MName, <(Malfunction_Name, Rate, Annotation )> }| $^{\text{Update}}$ -> (MName **.** Malfunction_Name' = false)

The function |{ }| $^{\text{failureExpression}}$ takes a malfunction annotation and the list of component ports to translate the annotation logic expression to a prism boolean expression. The list of component ports is used to replace the input port references on logic expression to its respective associated output port.

### 3.2.7 Formulas

Each annotation (logic expression) that can represent the possible system failure conditions (deviations) is transformed into a PRISM formula. As these expressions, the formula is written in compositional form. I.e., it is formed from formulas already established based on the local variables of each component representing the malfunctions. Once again, the Rules 13 and 14 act in the same style of rules 2 and 3 and is used to access each component deviation by your list.

**Rule 23** |{ Module_Name, Ports, <D> : tail, boolValue }| $^{\text{formulas}}$ ->
|{ Ports, <D> : tail, boolValue }| $^{\text{systemFormula}}$
|{ Module_Name, Ports, D }| $^{\text{formula}}$
|{ Module_Name, Ports, tail, false}| $^{\text{formulas}}$

**Rule 24** |{ Module_Name, Ports, <D> }| $^{\text{formulas}}$ -> |{ Module_Name, Ports, D }| $^{\text{formula}}$

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

**Rule 25** |{ Ports, <D> : tail, boolValue ) }| $^{\text{systemFormula}}$ ->
**formula** SysFailure = |{Ports, D}|$^{\text{term}}$ | |{ Ports, tail, boolValue ) }| $^{\text{systemFormula}}$
**provided** boolValue = true

**Rule 26** |{ Ports, <D>, boolValue ) }| $^{\text{systemFormula}}$ -> |{Ports, D}|$^{\text{term}}$
**provided** boolValue = true

**Rule 27** |{ Ports, (Deviation_Name, Crit, Port_ID, Annotation) }|$^{\text{term}}$ ->
|{ Port_ID, Ports }|$^{\text{Associated}}$
**provided** Crit ≠ empty

**Rule 28** |{ Module_Name, Ports, (DeviatioName, Crit, Port_ID, Annotation) }| $^{\text{formula}}$ ->
**formula** Deviation_Name . Module_Name . Port_ID = |{ Ports, Annotation }| $^{\text{failureExpression}}$

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

**Rule 29** |{ Ports, And( Annotation$_1$ , Annotation$_2$) }|$^{\text{failureExpression}}$ ->
**(** |{ Ports, Annotation$_1$ }| $^{\text{failureExpression}}$, **) and (** |{ Ports, Annotation$_2$ }| $^{\text{failureExpression}}$**)**

**Rule 30** |{ Ports, Or ( Annotation$_1$ , Annotation$_2$) }| $^{\text{failureExpression}}$ ->
**(** |{ Ports, Annotation $_1$}| $^{\text{failureExpression}}$**) or (** |{ Ports, Annotation$_2$ }| $^{\text{failureExpression}}$**)**

At this point, is necessary to identify the terminal terms of logic expression. As we can see in annotation type definition, there are two kinds of terminal terms.

The first is the component malfunction name and the other is the input port deviation name.

**Rule 31** |{ Ports, (Deviation_Name, Port_ID) }| $^{\text{failureExpression}}$ **->**
**(** Deviation_Name . |{ Port_ID, Ports }|$^{\text{Associated}}$ **)**

**Rule 32**|{ Ports, Malfunction_Name }| $^{\text{failureExpression}}$ **-> (**Malfunction_Name**)**

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

**Rule 33** |{ Port_ID, <(Port_ID', AssociatedPort)> : tail }|$^{\text{Associated}}$ **->** |{ Port_ID, tail }|$^{\text{Associated}}$

**Rule 34** |{ Port_ID, < (Port_ID, AssociatedPort) > }|$^{\text{Associated}}$ **->** |{ AssociatedPort }|$^{\text{AssociatedName}}$

**Rule 35** |{ (Module_Name, Port_ID) }|$^{\text{AssociatedName}}$ **->** ( Module_Name . Port_ID)

## 3.3 3.4 Generation of system verification expressions

In this step it is created the set of expressions using CSL language about the failures conditions to be analyzed, considering the required safety requirements. Thus, the FHA should be consulted in order to identify the most critical and important parameters for evaluation.

Generally, the most relevant property to be checked is the probability of a failure condition considered catastrophic. According to the regulatory bodies, this probability should not be greater than $10^{-9}$ / average flight time of aircraft. Thus, for each failure condition to be evaluated can be created the following verification expression:

*P=? [ true U<=T*3600 "Failure Condition" ], when T = flight time.*

In CSL, this expression has the following meaning: *The probability of failure occur in T hours*.

For instance, to evaluate the monitor failure condition, considering the flight time equals to 1h, one of evaluation expressions of system is:

*P=? [ true U<=1*3600 "LowPower_Monitor_Out1" ]*

Next, we present the translation rules used for generation of mentioned verification expressions. This translation strategy follows the same principle of the strategy for generation of formal specification. Its also uses the rules 2, 3 and 4 defined previously.

**Rule 36** |{ (System_Name , Subsystems) }| $^{\text{system}}$ **-> const double T;**

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

**Rules 2, 3 and 4**

**Rule 37** |{ (Module_Name, Type , Ports, Deviations, Malfunctions) }|$^{module}$ **->**
|{ Module_Name, Deviations }|$^{labels}$
|{M odule_Name, Deviations }|$^{expressions}$

**Rule 38** |{ Module_Name, <D> : tail }| $^{labels}$ **->**
|{ Module_Name, D }| $^{label}$
|{ Module_Name, tail }| $^{labels}$

**Rule 39** |{ Module_Name, <D> }| $^{labels}$ **->** |{ Module_Name, D }| $^{label}$

**Rule 40** |{ Module_Name, (Deviation_Name, Crit, Port_ID, Annotation) }|$^{label}$ **->**
***Label*** **"**Deviation_Name . Module_Name . Port_ID**"** **=** Deviation_Name.Module_Name . Port_ID

**Rule 41** |{ Module_Name, <D> : tail }| $^{expressions}$ **->**
|{ Module_Name, D }| $^{expression}$
|{ Module_Name, tail }| $^{expressions}$

**Rule 42** |{ Module_Name, <D> }|$^{expressions}$ **->** |{ Module_Name, D }| $^{expression}$

**Rule 43** |{ Module_Name, (Deviation_Name, Crit, Port_ID, Annotation) }| $^{expression}$ **->**
***P =? [true U <= T*3600 "*** Deviation_Name.Module_Name . Port_ID ***"]***
|{ MName, DName, Crit, PortID }|$^{assertions}$

Rules 44 and 45 create two different temporal expressions. They are used like assertions about one property:

**Rule 44** |{ MName, DName, Crit, PortID }|$^{assertions}$ **->**
***S*** **<=** Crit ***[ "*** DName . MName . PortID ***"]***

Using this proposed translation strategy, we can generated a valid formal failure model retaining the semantics of diagrams and the system hierarchical model. These concepts will be illustrated more appropriated by an example at Section 4.

## 3.4 Soundness and Completeness

It is important to highlight that we are using basic system safety assumptions. These assumptions are commonly used for safety assessment of aircraft system [1]:

- Element failure rates are constant
- System components fail in flight only.
- Component failures are detected in flight only and repaired during ground maintenance or before the next flight (description level), but the repairs occurs at constant rates (model level).

- The system is assumed with perfect failure coverage and can to reconfigure to a degradable mode within no time.
- The inspections are perfect and essentially 100% of the failures are detected and fixed.
- No new failures are introduced as a result of the maintenance.

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].

The system behavior after a fault occurs is defined and there could be several decisions to be made (fault detected, fault isolated, system reconfigured, system repaired and system failure due to near coincident fault. FEHM modeling can describe all these scenarios [1]. Based on our model description and its safety assumptions, we are using the FORM modeling. So the failed components of the aircraft systems may not be handled in-flight and the system behavior is not described. This solution is provided for scheduled tasks on the system condition monitoring, and unscheduled tasks on repair of the failed system components according to the maintenance schedule established for an aircraft type. In detailed level of SSA, the state flow of each system, which is based on its controlled variables and functions are also modeled in quantitative view. Matlab/ Simulink is a widely to tool to model and simulate these behavioral aspects. The pFMEA approach [1] is an alternative to model this characteristic in PRISM. Even so, we will show in Section 5, it also provides limitations and problems.

In our analysis we have considered repair transitions as if they occur at constant rates. But, in practice, system repairs do not typically occur at a constant rate. Instead, repairs occur only at discrete intervals therefore these considerations are not “Markovian” in the strict sense. However, the repairs can be conservatively represented by a constant repair rate of 1/T per hour, where T is the inspection time. Admittedly it might be repaired more quickly, but in the absence of any data to substantiate the quicker repairs, it is usually best to just assume that the operator waits the full T hours before making the repair. So, we can always conservatively represent such repairs by a transition with a constant rate of 1/T per hour, because the time from failure to repair can never exceed T hours. The immediate repair of the system failed state was also accurately represented by a constant-rate transition with infinite rate, so we stipulated that the failure of the system that is under review (total failure state) will be repaired immediately. We can set the repair rate on the total failure state to infinity because the actual repair rate for this state does not affect the hazard rate.

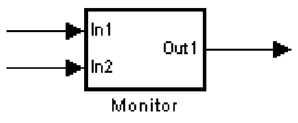
## 4 Case Study

Our case study was already introduced. It is the Actuator Control System presented graphically in Figure 2. Although it is a simple example, it is representative in the sense that it has dependent and independent failures, a hierarchical architecture, latency, evident, repeated and developed events [2, Serra].

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 fuselage. These surfaces are driven by electrical-hydraulic actuators controlled according to the pilot intent. Figure 5 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 de_ections in degrees, the controller (Controller) is an Actuator Control Electronics (ACE) responsible to process the reference signal and the sensed position (ActuatorSensor) of the actuator raw to generate the correct commands to the associated power control unit (PCU or Actuator). Moreover, this system is powered by a single power source (PowerSource).

Considering the resultant tabular information about these components and including its appropriated 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 as depicted in Section X over the system step by step.

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



```
module Monitor

...

endmodule
```

This part describes the declaration instructions. For each component malfunction (failure modes), boolean local variables with initial value *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 transitions commands into each module. For each local variable in the module, a state transition command is created. Their guard expression is assigned with the local variable negation and negation of system failure expression (will be explained below). The command updates this local variable value based on its failure rate. These commands refer to a transition to a failure state associated with the malfunction represented by the local variable.

```
[] (!(monitor_switchFailure) & !(SystemFailure)) -> (2.5E-5) : (monitor_switchFailure' = true);
```

Depending of component maintenance strategy, different set of repair transitions commands are created into each module. If the component is self-monitored (Sensor for instance) or unmonitored, 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[3]. For self-monitored, T = MedianTimeOfFlight).

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

Case the component is external 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 are failed (to represent repair of latent failure). The last occurs when the monitor detects that 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 & !(SystemFailure)) ->
(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 no synchronized transition (because it is an unmonitored component), we have to create repair transition commands synchronized with all monitored components. Note that this is a supplement 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 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 (i.e. this kind of repair only occurs if no fails was detected from the monitored components).

3 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.

```
[] (monitor_switchFailure & !(LowPower_PowerSource1_Out1 | LowPower_PowerSource2_Out1)
& !(SystemFailure)) -> (1/500) : (monitor_switchFailure' = false);

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

   [Monitor_In2_Repair] (!monitor_switchFailure & !(SystemFailure))
   -> (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 generation creates a set of formulas. Each failure logic expression that can represent the considered component failure conditions is transformed into a PRISM formula. As these expressions, the formula is written in compositional form. I.e., it is formed from formulas already established based on the local variables of each component representing the malfunctions. The system failure conditions (deviations over the output of the higher system) are transformed into a single PRISM formula too. This formula is composed by an OR logic with these failure conditions. The negation of this formula is putted into all guard expression using a AND operator.

```
formula LowPower_PowerSource1_Out1 = powersource1_lowpower;
formula LowPower_PowerSource2_Out1 = powersource2_lowpower;
formula LowPower_Monitor_Out1 = (monitor_switchFailure &
                               (LowPower_PowerSource1_Out1 | LowPower_PowerSource2_Out1)) |
                               (LowPower_PowerSource1_Out1 & LowPower_PowerSource2_Out1);
formula SystemFailure = LowPower_Monitor_Out1;
```

After applying the translation rules, we obtain the Prism specification showed in bellow.

```
ctmc

module PowerSource1

          powersource1_lowpower : bool init false;

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

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

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

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

formula LowPower_PowerSource1_Out1 =
powersource1_lowpower;

module PowerSource2

      powersource2_lowpower : bool init false;

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

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

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

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

formula LowPower_PowerSource2_Out1 =
powersource2_lowpower;


module Monitor

      monitor_switchFailure : bool init false;

      [] (!(monitor_switchFailure) & !(SystemFailure)) ->
(1E-4) : (monitor_switchFailure' = true);

      [] (monitor_switchFailure & !(SystemFailure)) ->
(1/50) : (monitor_switchFailure' = false);

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

```
      [Monitor_In2_Repair] (!monitor_switchFailure &
!(SystemFailure))
      -> (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);

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

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

      [SystemFailure] (SystemFailure) -> (1) :
(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 & !(SystemFailure) ) ->
(5E-4) : (sensor_sensorfailure' = true);
        [](!sensor_sensordegradation & !(SystemFailure) )
-> (5e-4) : (sensor_sensordegradation' = true);

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

      [SystemFailure] (SystemFailure) -> (1) :
(sensor_sensorfailure'=false) &
(sensor_sensordegradation'=false);

endmodule

formula OmissionSignal_Sensor_Out1 = sensor_sensorfailure
| LowPower_Monitor_Out1 | OmissionSpeed_Actuador_Out1;
formula CorruptedSignal_Sensor_Out1 =
sensor_sensordegradation;


module Component1

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

      [](!component1_lossofcomponent1 & !(SystemFailure)
) -> (9E-5) : (component1_lossofcomponent1' = true);
            [](!component1_component1degradation &
!(SystemFailure) ) -> (9e-5) :
(component1_component1degradation' = true);
```

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

      [SystemFailure] (SystemFailure) -> (1) :
(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 & !(SystemFailure))
-> (1E-4) : (component2_lossofcomponent2' = true);
            [](!component2_component2degradation &
!(SystemFailure)) -> (1e-4) :
(component2_component2degradation' = true);

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

      [SystemFailure] (SystemFailure) -> (1) :
(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 & !(SystemFailure)
) -> (6E-5) : (component3_lossofcomponent3' = true);
            [](!component3_component3degradation &
!(SystemFailure) ) -> (6e-5) :
(component3_component3degradation' = true);

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

      [SystemFailure] (SystemFailure) -> (1) :
(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 Actuador

      actuador_lossofdriver : bool init false;
            actuador_lossofmotor : bool init false;
            actuador_mechanismjamming : bool init false;
            actuador_mechanismdegradation : bool init
false;
```

```
            actuador_driverdegradation : bool init false;

      [](!actuador_lossofdriver & !(SystemFailure) ) ->
(1E-4) : (actuador_lossofdriver' = true);
            [](!actuador_lossofmotor & !(SystemFailure) )
-> (1E-3) : (actuador_lossofmotor' = true);
      [](!actuador_mechanismjamming & !(SystemFailure) )
-> (1E-3) : (actuador_mechanismjamming' = true);
      [](!actuador_mechanismdegradation &
!(SystemFailure) ) -> (1.5E-3) :
(actuador_mechanismdegradation' = true);
      [](!actuador_driverdegradation & !(SystemFailure) )
-> (8E-5) : (actuador_driverdegradation' = true);

      [] ((actuador_lossofdriver | actuador_lossofmotor |
actuador_mechanismjamming | actuador_mechanismdegradation
| actuador_driverdegradation)  & !(SystemFailure) ) ->
(1/5)
      :  (actuador_lossofdriver' = false) &
(actuador_lossofmotor' = false) &
(actuador_mechanismjamming' = false) &
(actuador_mechanismdegradation' = false) &
(actuador_driverdegradation' = false);

      [SystemFailure] (SystemFailure) -> (1) :
(actuador_lossofdriver' = false) & (actuador_lossofmotor'
= false) & (actuador_mechanismjamming' = false) &
(actuador_mechanismdegradation' = false) &
(actuador_driverdegradation' = false);

endmodule

formula OmissionSpeed_Actuador_Out1 =
actuador_lossofdriver | actuador_lossofmotor |
actuador_mechanismjamming | LowPower_Monitor_Out1 |
OmissionSignal_Component3_Out1;
formula WrongPosition_Actuador_Out1 =
actuador_mechanismdegradation |
actuador_driverdegradation |
CorruptedSignal_Component3_Out1;
formula CommissionSpeed_Actuador_Out1 =
actuador_driverdegradation |
CommissionSignal_Component3_Out1;

formula SystemFailure = OmissionSpeed_Actuador_Out1 |
WrongPosition_Actuador_Out1 |
CommissionSpeed_Actuador_Out1;
```

Considering the Simulink diagram of Figure 6, annotated with the corresponding failure logic, we can generate the formal specification (see [20] for the complete failure logic of the system), which is depicted in Figure 7. The next step is using the Prism model-checker to check whether any critical failure condition probability violates its permitted limit. For instance, we can verify the following formula:

$$\mathtt{S <= 10^{-3}\ [\ ("OmissionSpeed\_Actuator\_Out1")\ ]}$$

After checking this formula, where the exact value of the average probability obtained via steady-state analysis for this situation is *2.54e*$^{-3}$, Prism returns a false, indicating that this failure condition was violated. 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 repairs rates) is acceptable[4]. To show this analysis to certification authorities, the Markov model can be extracted from Prism by using tools like SHARPE, or HARP [21].

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, Figure 8 is the result of evaluating the following formula in Prism, setting the T parameter from 0 to 100 hours.

```
P =? [true U<=T ("WrongPosition_Actuador_Out1")]
```

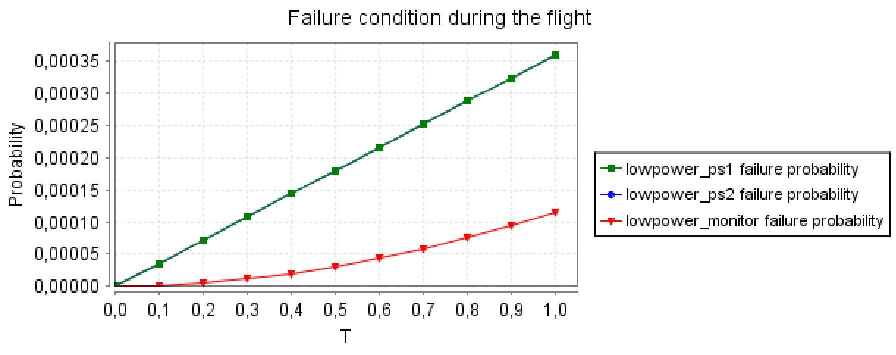


**Fig. 8.** Probabilities of failure condition during the flight

[4] In the entire safety assessment process, the design satisfaction must also to consider optimization of all aspects of safety within the constraints of operational effectiveness, time, and cost throughout all phases of the system life cycle [3].

## 5 Related Work

The work reported in [19] (which proposes pFMEA or Probabilistic FMEA) also uses the PRISM model checker to support the FMEA process. In one sense, pFMEA performs a more detailed analysis than ours because it considers faulty as well as normal behaviors of a system. However, pFMEA does not generate the model systematically and has a greater potential to the state explosion problem. Furthermore, pFMEA applies transient analysis over closed-loop models to determine whether some safety requirement is violated. Although the transient analysis is "exact", it is applied strictly to take just an instantaneous probability of a single interval T to validate the safety requirement (the estimation must considering every flight mission during the entire lifetime of an aircraft). Therefore, depending on the interval T, the obtained value cannot be sufficient to validate a tolerable average failure probability per hour (average failure rate), because it considers only a sample of failure rate.

We found another work [25] that also proposes a model-based quantitative safety assessment using Prism. It provides an effective approach over the initial steps of the safety analysis, using SCADE and Esterel Studio tools, to perform compositional reasoning about the fault tolerance in the system. However, its resulting Prism model (quantitative model) is created in a non-systematic way, and thus it can contain errors. With the Prism model, the work applies transient analysis over open-loop models, to analyze an average failure condition rate. However, it does not provide effective results on the validation of safety requirements under certification, because, in this case, only the mean value of an open-loop function (the sawtooth function) can be virtually indistinguishable from an steady-state value. This can be considered a worst-case analysis if the interval T is large.

## 6 Conclusion

In this paper we propose a systematic strategy to provide a formal quantitative analysis of aircraft systems. Our approach generates a Prism specification from the system failure model described in a tabular notation and can, in a simple and direct way, verify the violation of any safety requirements using the Prism model checker over a computed Markov representation.

Markov models can be quite complex to handle, due to the necessary treatment of space-time characteristics. Although there are several tools that enable the creation and analysis of Markov models from graphical interface useful [21], it can be extremely difficult to create the failure model of a system from an ad-hoc approach using Markov chains directly to obtain a reliable result of the analysis, especially when considering aspects of latent and evident failure, monitoring and repair schedule, which are essential in the context of aircraft. Moreover, if we consider that the traditional fault-tree model is constructed to assess cause and probability of single undesirable failure condition, is a fact see that the effort and the number of models generated to allow the analysis of each failure condition are extremely large making the process very expensive [2, 10].

In Prism, as already seen, each modeled system computes a Markov representation and contrary to what occurs traditionally, our strategy offers an alternative to the specialists to create the system failures model in Markov systematically. In addition, the generated model can check all undesirable failure conditions about its criticality applying the Prism Model Checker. Furthermore, engineers can extract and develop the generated formal model to investigate dynamic aspects of system: experiments can be done changing the initial values of local variables to check existing failure scenarios, maintenance scheduled to account for the Minimum Equipment List, Phased Mission, reconfiguration triggers based on sync with failure events [2, 8].

Therefore, recent researches are advancing to identify counter-examples of stationary models, allowing a better traceability of the basic failures and facilitating the cycle of checking and validating of the system design [22].

As future works 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 try to identify practical advantages/disadvantages of the strategy. Another direction is to consider dynamic faulty behavior, capturing the dynamic information in the same way as static information.

## References

1. M. Stamatelatos et al.. Fault Tree Handbook with Aerospace Applications. NASA Office of Safety and Mission Assurance, Washington, DC. Version 1.1. Aug. 2002.
2. ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems, SAE Inc, Nov. 1996.
3. ARP 4754: Certification Considerations for Highly Integrated or Complex Aircraft, SAE Inc, Dec. 1996.
4. O. Lisagor, J. McDermid, D. Pumfrey. Towards Safety Analysis of Highly Integrated Technologically Heterogeneous Systems– A Domain-Based Approach for Modelling System Failure Logic, 24th Inter. System Safety Conference, 2006.
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. A. Joshi and M. Heimdahl, Behavioral Fault Modeling for Model-based Safety Analysis, 199-208, 10th High Assurance Systems Engineering Symposium, 2007
13. O. A. Kerlund et al. ISAAC, A framework for integrated safety analysis of functional, geometrical and human aspects. European Congress on Embedded Real Time Software (ERTS 2006), 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. M. Kwiatkowska, G. Norman, and D. Parker. Controller dependability analysis by probabilistic model checking. Control Engineering Practice, 15(11):1427–1434, 2006.
18. P. R. Serra, Safety Assessment of aircraft systems. 2º Edition. 2008.
19. L. Grunske, R. Colvin, K. Winter. pFMEA: Probabilistic Model-Checking Support for FMEA. 4th Int. Conference on the QEST, 2007.
20. Technical Report. Available on: www.cin.ufpe.br/~ajog/technical_report.pdf
21. D. Siewiorek, R. Swarz. Reliable Computer System: Design and Evaluation, 3th edition, 1998.
22. H. Aljazzar, et al. Safety Analysis of an Airbag System Using Probabilistic FMEA and Probabilistic Counterexamples. pp.299-308, 6th International Conference on the Quantitative Evaluation of Systems, 2009.
23. C. Baier, B. Haverkort, H. Hermanns and J. Katoen. Automated Performance and Dependability Evaluation Using Model Checking. Lecture Notes In Computer Science; Vol. 2459, Pages: 261 - 289, ISBN:3-540-44252-9, London, UK, 2002.
24. Software considerations in airborne systems and equipment certification. DO-178B, RTCA Inc., Washington D.C., December 1992.
25. Elmqvist, J. Nadjm-Tehrani, S. Formal Support for Quantitative Analysis of Residual Risks in Safety-Critical Systems. 11th High Assurance Systems Engineering Symposium, p. 154-164, Nanjing, 2008.