Universitá degli studi di Cagliari
Universitè de Lorraine

Facoltá di Ingegneria
Institut National Polytechnique de Lorraine

Dipartimento di Ingegneria Elettrica ed Elettronica
École nationale supérieure d'électricité et de mécanique

Corso di Laurea Magistrale in Ingegneria Elettronica
Ingenérie des Systèmes Automatisés

Master thesis

Deterministic and probabilistic dependability assessments of a critical system

Relatore:
Prof. Alessandro Giua

Candidato:
Pinna Bruno

Controrelatore:
Prof.ssa Carla Seatzu

Supervisori esteri:
Ph.D. Génia Babykina
Prof. Nicolae Brînzei
Prof. Jean-François Petin

Anno accademico 2013/2014

Ai miei genitori, Laura e Luigi
che non hanno potuto vedere
il loro figlio diventare ingegnere.

Abstract

The aim of the present work is to build a Coloured Petri net (CPN) model of a critical system and to perform two tasks: an analysis of the system's performance and a formal verification of the model. The case study system used for these purposes is composed of two components functioning in a parallel configuration, thus allowing to improve the whole system's reliability. The physical model of system's behaviour (with failures and reparations) and the logical model of its behaviour are implemented as two different CP nets in a software named CPN Tools. We will see that a single CPN model is not sufficient to perform the two tasks. Thus, two model are developed. The first model is used to carry out the performance analysis and the second CPN model will be used to carry out the formal model verification. While the system's performance analysis can be performed using CPN Tools and MATLAB, the formal verification is performed by three different tools: the CPN Tools integrated state space tool, ASAP tool, and the software ProM. We will see that the formal model verification is quite complicated and none of the three tools perform a reliable, formal, easily implemented analysis. We can however perform a formal model verification, but the improvements of the available documentation (especially for ASAP) are necessary to exploit all the capabilities of these tools.

Lo scopo di questo lavoro di tesi é quello di realizzare un modello di Rete di Petri Colorata (CPN) per un sistema critico con due obiettivi : l'analisi delle prestazioni temporali del sistema e la verifica formale del modello. Il caso di studio del sistema discusso in questo lavoro di tesi é composto da due componenti disposti in configurazione parallela, questo ci permette di migliorare l'affidabilitá complessiva del sistema. Il modello del sistema é descritto da due reti CPN, uno per il comportamento fisico (con i guasti e le riparazioni) ed uno per la parte logica (esempio, la partenza e lo stop delle macchine) il tutto é stato implementato con il software CPN Tools. Abbiamo visto che, un solo modello di sistema con reti CPN non era sufficiente per verificare i nostri due obiettivi. Perciò, sono stati sviluppati due modelli. Il primo medello é usato per l'analisi delle prestazioni temporali e il secondo modello invece per la verifica formale. L'analisi delle prestazioni del sistema sono state fatte usando CPN Tools e MATLAB, mentre la verifica formale é stata effettuata con tre diversi software: lo state space tool integrato in CPN tool, l'estensione ASAP ed il software ProM. Abbiamo visto che la verifica formale é abbastanza complessa e nessuno di questi tre tools ci permette di ottenere risultati affidabili, formali e facili da ottenere. In ogni caso possiamo fare delle prove di verifica formale , ma il miglioramento della documentazione disponibile (sopratutto per il tool ASAP) é una condizione indispensabile per poter sfruttare al massimo le capacitá di questi tools.

Contents

Introduction

Why is dependability important?

For a manufacturer a good component is a component which breaks one day after the warranty is finished, for a user a good component is a component which never breaks. A break of a component is a composition of many factors, thus neither the manufacturer nor the user can know the exact moment of a component's break. The dependability assessment for a single component performed by a manufacturer (and by the user if he were capable of), if the component is a part of a system, is to try to answer one important question: when will the component break? Different approaches are developed to figure that out. Thanks to computer power and to improvement of the manufacturing process, the *reliability*, an aspect of the dependability, of the manufactured products has a fixed degree. We can mention two ways of improving and evaluating the component's dependability: a physical way and a simulation-based way.

The first method is generally implemented for a single component rather than for systems. In the domain of electronics for example, transistors are tested in non-operational conditions in order to define their operational limits, the way they break, etc. By adding some stress (or worsening the operational conditions) we can accelerate their life and estimate the moment of their break-down. This way is not simple, it requires numerous components to test, a stress room with an environmental control, some high-cost equipment, etc.

The second method is based on computer simulations. In general it is applied to systems rather than to single components. Instead of a single component/device a system have a constrains in plus, the *safety*. With this addition is common to talk of a system dependability as Reliability Availability Maintainability Safety acronym. The safety constraints is due to the connection between the devices and they behavior to modify the system. *Reliability, availability and maintenability* properties are generally analysed or proven using *probabilistic* techniques the *safety* properties instead are proven using *deterministic* techniques such as event driven simulation or formal verification. The combination between these techniques are known as Integrated Deterministic and Probabilistic Dependability Analysis. This kind of mixed deterministic and probabilistic analysis is particularly relevant in the context of dynamic reliability where the structure function evolves over the time due to the impact of the physical parameters and device ageing on the dysfunctional behavior and control architecture reconfiguration. Our work lies in this specific domain: the IDPA assessment for a system's model.

System model and method to assess the dependability

The definition of a system is provided by IEEE Guide for Information Technology [1] as *"A collection of interacting components organized to accomplish a specific function or set of functions within a specific environment"*. One may be interested in the continuous response of the system or in the system's characteristics at a specific moment when an event occurs. Since a system is a collection of components, it is reasonable to consider a system as a sequence of different interactions among components. These interactions can be modeled as discrete *events*. Thus, many systems (*e.g.* a manufacturing process or a coffee maker machine) can be modeled as a Discrete Event System (DES).

Case study: an overview

In the case study we consider a subsystem of the secondary circuit of a PWR nuclear power plant. This subsystem is discussed in details in chapter 4. In simple words this system is composed of two parts: a physical model and a logical model. The physical model is composed of two components (named Turbo Pompe Alimentaire (TPA)) which function in parallel. Each of these two components is composed of two subcomponents in series and has two different failure modes. Thus, the whole component is considered broken if a failure of one mode (or of one subcomponent) occurs. The entire system is *down* if both of the two components are broken. The logical model, can be viewed as an observer with sensors and actuators which monitor the system's condition, notify of component failures and give orders to start the reparation process or to stop the entire system. The particularity of our model is that the probability distribution function of failure times of a TPA is *exponential*. For reparation times the *Erlang* distribution is used. This distribution is similar to the exponential one, and is generally used to characterize waiting times in queuing systems. Thus, the basic assumption in our case is that failure occurrence is driven by a memoryless process and the reparation process is similar to a queuing system, implying for example that a reparation may depend on a limited number of specialized technician.

Discrete Event System: what and why

Numerous mathematical models describing a DES are used for dependability assessment. We can mention, for example, Markov Chains, Finite State Automata (FSA), Timed Automata (TA), Stochastic Petri Nets, Coloured Petri Nets. For our case study an *exponential* and an *Erlang* distributions will be used. With only exponential distribution the Markov Chain could be employed as a theoretical framework for modeling, allowing to calculate the dependability assessment parameters by an exact method. Since not only exponential distribution is used in our case, the exact calculations appear to be complex and simulations will be used to obtain the results. Thus, we can choose among Finite State Automata (FSA), Timed Automata (TA), Stochastic Petri Nets and Coloured Petri Nets.

Computer simulations in dependability assessment

An important difference between the methods discussed above is their modularity and their *ability of reduce the modelling time.* For example with a Coloured Petri Net it is possible to use programming languages which can execute functions and perform calculations. With a Timed Automaton we can not do that. This allows us to model a complex system in different ways. Once the model is defined, built and implemented, we should be able to perform *Monte Carlo* *simulations* of its behaviour. The Monte Carlo simulations give us failure and reparation times as well as other performance parameters. Besides these parameters, we are also interested in the comparison of components, *i.e.* in defining if the components have the same characteristics. Concerning a system, we can be interested in what happens if the user exercises one function instead of another. For example, we would check if a system composed of two elements always has at least one element running. If not, we would try to figure out why, to known what are the events responsible for this negative answer. The domain enabling the answers to such questions is referred to as the *Model Checking*.

Formal model verification: *Model Checking*

The *Model Checking* is a formal method of model verification. When the model to verify is built, the *model-checker* (the basic tool of *Model Checking*, we can say its "heart") is responsible of performing all the calculations to answer the question asked, that is to say to check if a particular defined property is verified. If the answer is negative, the *model-checker* produces a counter-example to illustrate in which case the property is not verified. One of the best book in this domain is given in the reference [2].

The used mathematical model should possess three characteristics:

1. The ability to *build a complex model in an easy way.*
2. The ability to perform *Monte Carlo simulations* for the performance analysis.
3. The ability to perform a *Model Checking* analysis.

Among different models, we have chosen the *Coloured Petri Net* (CP-net or CPN) implemented in the software called *CPN Tools*. This choice is motivated by the following reasons:

1. The possibility to *built a complex model in an easy way.* Thanks to the *programming language SML* implemented in *CPN Tools*, we can build functions and perform different algorithms with the aim of reducing the size of our model.
2. The possibility to build a *modular system and to perform Monte Carlo simulations.*
3. The possibility to perform a *state space* analysis. The temporal CTL and LTL logics which are the bases for the *Model Checking* analysis, are also implemented in *CPN Tools.*

The chosen case study takes the advantage of the Coloured Petri Net (CPN). Indeed, it is a modular system with concurrency: a piece of cake for a CP-net.

The objective of the work and the contents of this master thesis

The objective of out work is to carry out the IDPA assessment in a new ways with a CPN Tools and with its state space the safety proprieties using a Model Checking techniques. The system is taken from a real industrial project and has stochastic and deterministic events, namely failures and restorations. This model is used for the performance assessment but will not be appropriate to perform a formal verification. Thus a second model will be defined. This second model is composed only of deterministic events. A series of concurrency and logical problems will be discovered with this deterministic model. To resolve these problems a method taken from an academic *mutual exclusion problem* will be used.

In the first section of this master thesis (chapters 1, 2 and 3) the *theoretical background* for the used methodology is given: the different parameters of the system dependability (MTTFF, MTTR, MTBF, and others) are presented, the Coloured Petri Net is defined, the CPN Tools is introduced, and the basis of *Model Checking* is given using a little example.

In the second part of the thesis we illustrate the case study, present the corresponding CPN models, give an example of the model verification and present the results. In the last chapter (chapter 8) future developments for the work are proposed.

Part I

Theorical background

Anything that can go wrong, will go wrong.

Edward A. Murphy, Jr.

Chapter 1

System dependability

Introduction

In this chapter we explain the basis of dependability and its use. Nowadays a lot of time and money are spent in industrial domains to improve the quality of a product or a system. The dependability is associated with the concept of quality due to some of its aspects, in particular due to the concept of reliability. Reliability improvement is a crucial task when dealing with complex systems. When speaking about a "complex system" we suppose, high structural complexity, high cost, high development duration time, evolution in a dynamic environment, ageeing of components, interaction between components state and environment... In this case the dependability analysis is necessary to minimize the number of errors in every development phase. This chapter is organized as follows. In Section 1.1 we briefly outline the history of dependability, in Section 1.2 the definition of dependability is given following the standard IEC 50 (191). The quantitative and qualitative dependability analysis is presented in Section 1.3. The two probability distribution function used in this work are explain in Section 1.4. A presentation of two methods for the reliability assesments are presented in section 1.5.

1.1 History of dependability

A brief history of dependability is presented below following [3]. The dependability or "the science of failure" was developed in the early 20th century in industries dealing with critical processes. Before the Second World War, the dependability was an empirical notion rather than an exact science. With electronics appeared in 1950, a first indicator of dependability was developed, the **MTBF** (Mean Time Between Failures); it marked the beginning of the analytical studies of dependability. In 1960s, with the conquest of space, some analytical methods arrived from the United States. These methods are sill employed and are the following:

- Fault tree analysis
- Root cause analysis
- Reliability Block Diagram analysis

The nuclear accident of the Three Mile Island that had no human casualties but impacted people's opinions has required a standard of protocol and procedure, the task which the International

Electrotechnical Commission (IEC) fulfilled. This had formalized the definitions of Maintainability, Availability and the associated concepts like: Diagnostic, Testability, Survivability. In the last 20 years numerous tools were developed for the dependability problem, for example the technical fault can be described with Markov chains or Petri Nets. Nowadays, in the microcontroller age, where critical tasks are assigned to it, new restrictive standards and procedures rise up, like SIL (Safety Integrity Level). In all industrial sectors the dependability and its aspects like *Reliability*, *Availability*, *Maintainability* and *Safety* (RAMS) are a common way to improve the quality of a product or a system. The Achilles' heel of the dependability assessment is that the data used to perform a correct modelization are inadequate or nonexistent. This causes little confidence in the produced and used results. New methods to approach dependability still arise (see for example [4]).

1.2 Definitions and aspects of system dependability

A general definition of a system's dependability is given in definition 1.1, a more specific version in the field of computer system, is given in definition 1.2 following [5].

Definition 1.1 (System dependability - IEC 50 (191))**.** *A system's (devices) dependability is its aptitude to perform one (or more) requested functions in an operational condition.*

Definition 1.2 (Dependability in computer system - IEEE Std 982.1-2005)**.** *Trustworthiness of a computer system such that reliance can be justifiably placed on the service it delivers. Reliability, availability, and maintainability are aspects of dependability.*

Other definitions can be given, depending on the field of study. In this work we refer to the definition 1.1. A series of linked definitions that are commonly used in the sphere of dependability arise.

Definition 1.3 (Entity)**.** *An entity is any element, device, sub-system, equipment or system that can be considered individually.*

Definition 1.4 (Failure)**.** *A failure is the alteration or interruption of an entity's aptitude to perform the required function with the performance defined by specification.*

Definition 1.5 (Fault)**.** *A fault is the interruption of entity's aptitude leading to a failure state.*

Dependability is composed of different elements joined in RAMS acronym. These elements are defined below.

Definition 1.6 (Reliability - IEC 50 (191))**.** *Reliability is the entity's aptitude to perform the requested function in fixed conditions during a fixed time.*

$$R(t_1, t_2) = P[\textit{Entity is not in a fault state in the time interval}[t_1, t_2]] \tag{1.1}$$

Where $P[E]$ it's the probability of the event E and $R(t_1, t_2)$ is the reliability function of a time interval.

Definition 1.7 (Maintainability - IEC 50 (191))**.** *Maintainability is the entity's aptitude to be maintained in or restored to a state in which it can perform a requested function when the maintenance is done in fixed conditions, using with the procedure and resources recommended by standard.*

$$M(t_1, t_2) = P[\textit{Entity's fault in } t = t_1 \textit{and reparation in } t = t_2] \tag{1.2}$$

$M(t_1, t_2)$ is the maintenability function.

Definition 1.8 (Availability). *Availability is the entity's aptitude to perform a requested function in a fixed conditions at a given time moment.*

$$A(t) = P[\textit{Entity is not in a fault state at time } t] \tag{1.3}$$

$A(t)$ is the availability function.

The standard *IEC 50 (191)* doesn't consider the *safety* element, the *safety* is defined in the standard *EN 292.*

Definition 1.9 (Safety - EN 292). *Safety is the machine's aptitude to perform a requested function, to be moved, set, adjusted, maintained, dismantled and discarded in standard conditions without causing injury or damage to health.*

A global vision of dependability elements is shown in figure 1.1. The elements from figure 1.1 not defined in the present section aren't useful for this work, however they are related to the definition of dependability.

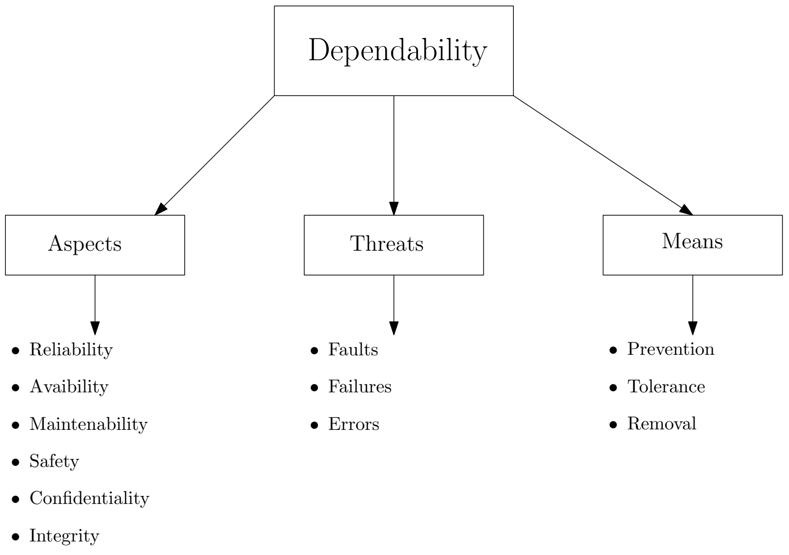


Figure 1.1: Dependability and its elements: schema

1.3 Dependability: temporal parameters

Additionally to probabilistic parameters, other parameters can be measured in dependability analysis such that the time of first failure and other typical time instants. These time instants are randoms variables, linked to the probability of an event. The aim of a typical dependability study is to characterize these variables with mathematical expectations. A list of the mentioned variables is given below and their intuitive representation is illustrated in figure 1.2.

- **MTTFF - Mean Time to First Failure** : expectation of operational time before the first failure.
- **MTTF - Mean Time to Failure** : expectation of operational time before a failure.

- **MTBF - Mean Time Between Failures** : expected time between two consecutive failures of a reparable entity.
- **MUT - Mean Up Time** : expectation of system's availability duration.
- **MDT - Mean Down Time** : expected duration of entity's failure, which include: detection, intervention , reparation , restoration of function durations. In summary, MDT is the system's expectation of unavailability.
- **MTTR - Mean Time To Recovery or Restoration** : expectation of system's reparation duration.

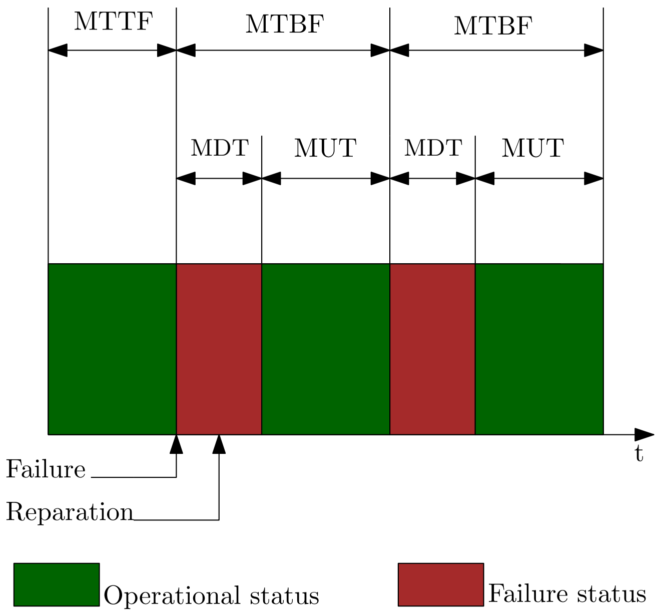


Figure 1.2: Times associated to dependability

An important parameter of an entity in the context of dependability is the failure rate.

Definition 1.10 (Failure rate)**.** *The instantaneous failure rate is the limit, if it exists, of quotient of conditional probability that the instant* T *of entity's fault is between* $]t, t + \Delta t[$, *where* Δt *converges to zero. Knowing that the entity didn't have a failure in the interval* $[0, t]$ *:*

$$\lambda(t) = \lim_{\Delta t \to 0} \frac{1}{\Delta t} P[t < T \leq t + \Delta t / T > t]. \tag{1.4}$$

In other words, $\lambda(t)$ is approximatively the conditional probability of failure occurring in $]t, t + \Delta t]$ knowing that the entity was in an operational state in $[0, t]$. Another mathematical definition is:

$$\lambda(t) = \frac{f(t)}{R(t)} = \frac{\frac{-dR(t)}{dt}}{R(t)} \tag{1.5}$$

Where $f(t)$ is the probability density function of the random variable T which measures the duration of entity's operational status before failure (or lifetime for not reparable entities). Commonly $f(t) = \frac{-dR(t)}{dt}$ is the entity's failure probability density function. $R(t)$ is the entity's reliability function (see definition 1.6).

1.4 Probability distributions used in dependability analysis

A knowledge of the function $f(t)$ allows one to calculate the parameters useful for the reliability assessment of entities or components, precisely the failure rate $\lambda(t)$ and the MTTF. There exists two ways to find this distribution function. The first one is to investigate the entity's failure mode: for example mechanical or electronic. Indeed different distribution functions fit a particular entity's failure mode, and the knowledge of the failure mode can give us the way to find our distribution function. The second way is to perform accelerated life time experiment to extract an experimental distribution of $f(t)$, this will be discussed later in this section, or another method is made different simulations. A combination of this two methodologies is the widely used way to analyze the reliability of an entity.

Theoretical study

There exists various probability distributions potentially useful for dependability analysis. Exponential and Erlang distribution will be used in this work. A Weibull distribution will be presented for completeness, due to its adaptability to different entity's lifetimes.

Definition 1.11 (Exponential distribution)**.** *The exponential distribution describes the time between events in a Poisson process in which events occur continuously and independently at a constant average rate λ (λ is a constant). It is principally used for electronic components and for memorylessness systems where the entity's failures are independent from each other. In this case the parameters of the entity are:*

$$\lambda(t) = \frac{-dR(t)}{dt} \tag{1.6}$$

$$R(t) = e^{-\lambda t} \tag{1.7}$$

$$f(t) = \frac{-dR(t)}{dt} = \lambda e^{-\lambda t} \tag{1.8}$$

$$MTTF = \int_0^\infty R(t)dt = \frac{1}{\lambda} \tag{1.9}$$

Definition 1.12 (Weibull distribution)**.** *The first application of this distribution was carried out in 1933 to describe the size of particles. It is widely used in reliability engineering and elsewhere due to its versatility and relative simplicity. The Weibull distribution is characterized by the following relations.*

$$\lambda(t) = \frac{\beta(t-\gamma)^{\beta-1}}{\alpha^\beta} \tag{1.10}$$

$$R(t) = e^{-(\frac{t-\gamma}{\alpha})^\beta} \tag{1.11}$$

$$f(t) = \frac{\beta(t-\gamma)^{\beta-1}}{\alpha^\beta} e^{-(\frac{t-\gamma}{\alpha})^\beta} \tag{1.12}$$

, where α is a scale parameter, β is the shape parameter and γ is the location parameter.

An important property of Weibull distribution is that it can fit the failure rate function to the bathub curve (see figure 1.3).

Analyzing the bathtub curve is beyond the purpose of this work, but it describes how the failure rate is influenced by the age of the entity. In this work we assume that all the components are in the central part of the lifetime, i.e. the failures are independent and the failure rate is constant. We take the Erlang distribution for the reparations.

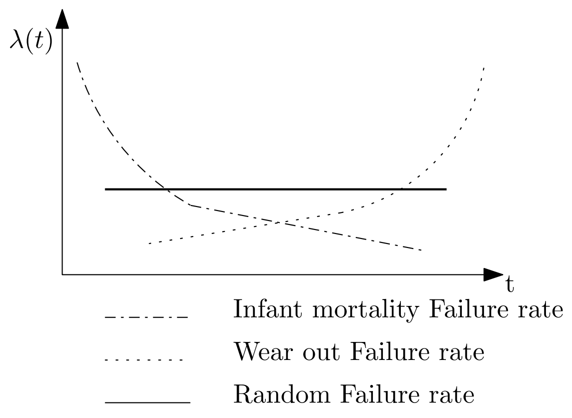


Figure 1.3: Bathtub curve

Definition 1.13 (Erlang distribution). *The Erlang distribution is developed by A.K. Erlang to examine the number of telephone calls which can be made at the same time to the operator of the switching stations. This distribution is used, in general, to characterize waiting times in queueing systems. The following parameters characterize this distribution:*

$$\lambda(t) = \frac{f(t)}{R(t)} \tag{1.13}$$

$$R(t) = \sum_{n=0}^{k-1} \frac{1}{n!} e^{-\lambda t} (\lambda t)^n \tag{1.14}$$

$$f(t) = \frac{\lambda^k t^{k-1} e^{-\lambda t}}{(k-1)!} \tag{1.15}$$

$$MTTF = \frac{k}{\lambda} \tag{1.16}$$

where k is the shape parameter, if $k = 1$ the Erlang distribution is simply the exponential distribution of parameter λ.

Empirical study

Empirically the distribution function is obtained by measuring the time moments of an entity's fault. Plotted in a semi-logharitmic chart, with time in a logaritmic form in abscissa and the cumulative distribution function $F(t) = 1 - R(t)$ in the ordinate axe, a Weibull distribution can be extracted (see figure 1.4 for presentation). An accelerated lifetime test can also be perfomed to analyze the failure times. For a high cost component/system where the failure data are hardly obtained empirically, modelisation is used to produce data.

1.5 Reliability assessment methods: Fault Tree Analysis and Reliability Block Diagram

Fault tree analysis (FTA) is a top down method developed in military and aeronautic fields in 60s, the objectives of this method are :

- Creating a list of entity's faults that lead to the critical event.

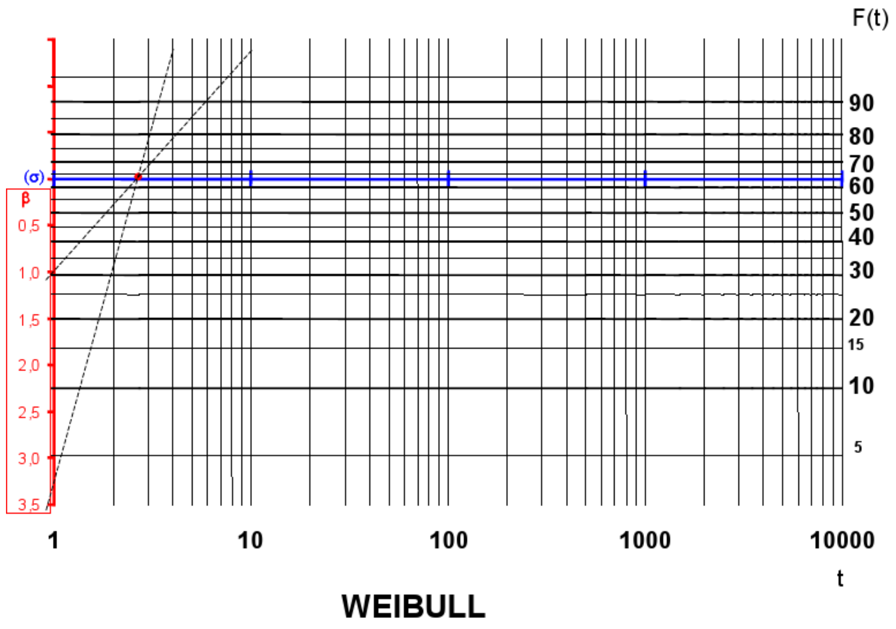


Figure 1.4: Weibull semi-logharitmic chart

- Performing *reliability*, *availability* and *safety* analysis.

The general algorithm of the FTA is the following:

- Identify the undesirable events.
- Find all the possible event combinations that allow to avoid the undesirable event.
- Represent graphically a tree with the system's faults in the top of the FTA) and elementary event in the "root" of the FTA.

A representation of event symbol for a FTA is given in figure 1.5, typically used events are as follows:

- Elementary event - failure or error of a system's component or element.
- External event - an event indipendent from the system's state (e.g. function of an environment).
- Undeveloped event - an event described insufficiently, or event with no consequence.
- Intermediate event - combination of result of other events linked by a logic operator.

Definition 1.14 (Critical event)**.** *The critical (dangerous) event in reliability analysis is a system's fault, for safety analysis it is the dangerous event, for availability analysis it is the fact that the system is not available.*

Another method to show how components contributes to the success or failure of a complex system is the *Reliability Block Diagram (RBD)*. An RBD is drawn as a series of blocks connected

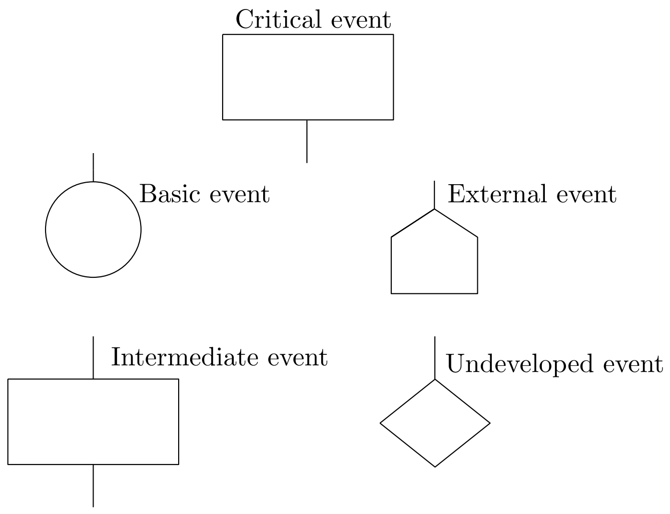


Figure 1.5: Graphical symbols of FTA

in parallel or series configuration. Each block represents a component of the system with a failure rate or its reliability function. Parallel paths are redundant, meaning that all of the parallel parts must fail for the parallel network to fail. In contrast, any failure along a series path causes the entire series path to fail. A RBD can be translated in a success tree by replacing series paths with *and* gates and parallel paths with *or* gates.

Example 1.1. *Figure 1.6 illustrates an electrical schema of a system composed of two lamps, a switch and the power line.*

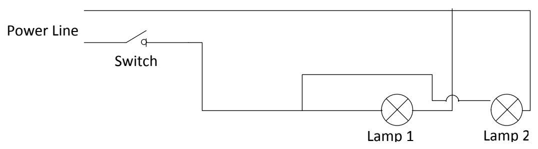


Figure 1.6: Electrical schema of two lights in a room

In this example 1.1 the failure of the light system implies the failure of the power source or the failure of the two lamps. A fault tree of the schema 1.6 is given in figure 1.7.

The next step is to rise the FTA until we will arrive to the critical event. In this case the events are:

- Lights Broken (LB) = Old.L1 . Old.L2
- Lights without power (LNP) = PW.off + SW.ko
- No Light (Fault) = LB + LNP = PW.off + SW.ko + Old.L1 . Old.L2

where, PW is the acronyms of Power Line, SW of Switch, L1 is the Lamp1 and *ko, off, old* are failure modes. A definition of the cut set is necessary to complete the analysis.

Definition 1.15 (Cut set)**.** *A cut set is a set of events which allow the critical event.*

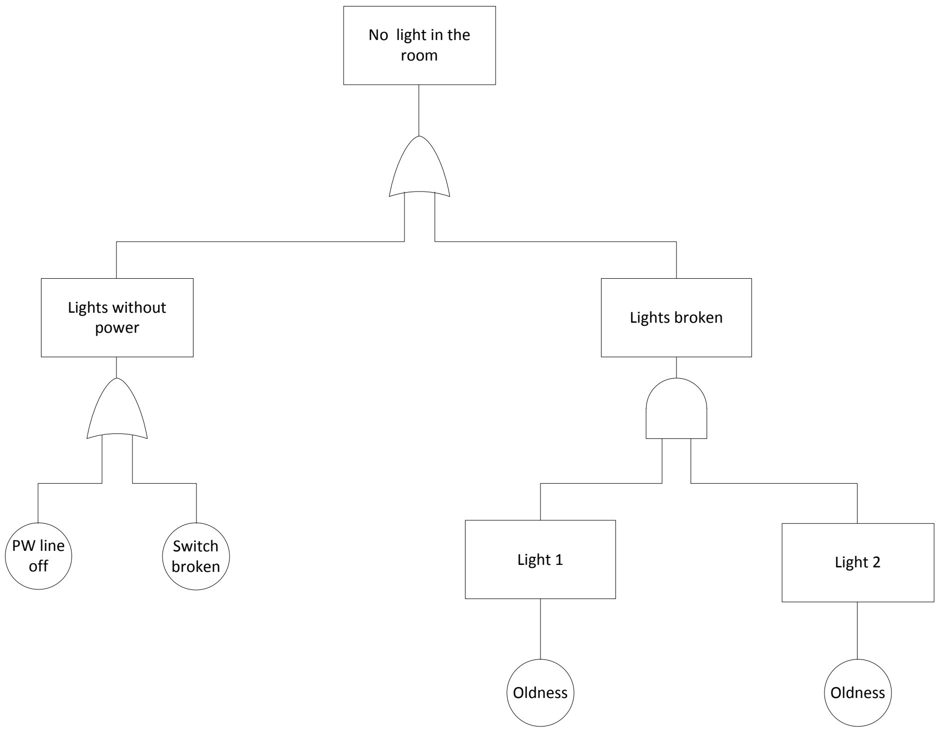


Figure 1.7: Fault tree of Figure 1.6

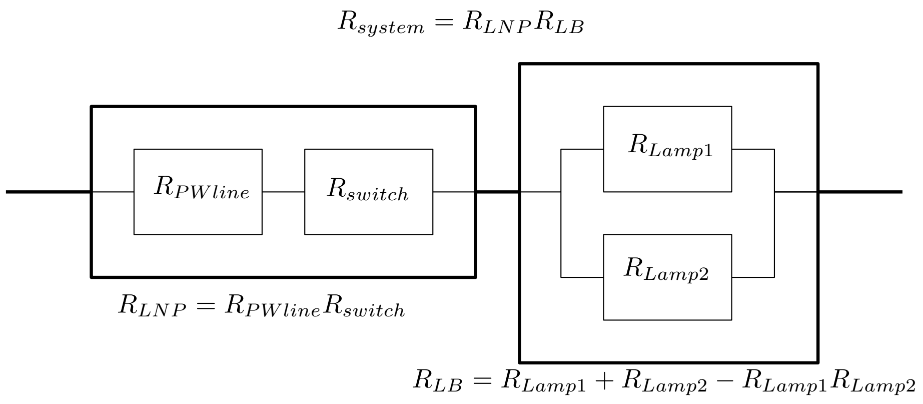

$R_{system} = R_{LNP}R_{LB}$
R_{PWline}
R_{switch}
R_{Lamp1}
R_{Lamp2}
$R_{LNP} = R_{PWline}R_{switch}$
$R_{LB} = R_{Lamp1} + R_{Lamp2} - R_{Lamp1}R_{Lamp2}$

Figure 1.8: Reliability Block Diagram of Figure 1.6

Definition 1.16 (Minimal cut set)**.** *A minimal cut set is a cut set which doesn't include other subset's which are also cut set.*

Definition 1.17 (Order of cut set)**.** *The order of a cut set is the number of event in a cut set. If the order is low the cut set is critical, because if the cut set is verified the system fails.*

With these definitions, in our example the cut sets are:

- Order 1 : PW.off, SW.ko

- Order 2 : Old.L1 . Old.L2

If the powerline goes offline the light system fails, whereas if a single lamp is broken there is still a light in the room due to lamp 2. Then the Cut set of first order is the critical for this system, if on of this occurs, the system fail.

Thanks to the cut set it is possible to calculate the probability of the critical event, using eq. 1.17. Equation 1.17 allow to calculate the recurrence of the critical event when a minimal cut set appears. Using the Poincaré theorem is possible to calculate the probability of the critical event is calculated as

$$P(Fault) = \sum_{j=1}^{m} P(C_j) - \sum_{j=2}^{m}\sum_{i=1}^{j-1} P(C_i \cap C_j) + \ldots + (-1)^m P(C_1 \cap C_2 \cap \ldots \cap C_m) \tag{1.17}$$

where C_i is a minimal cut set and m is the number of minimal cut sets. The calculus is very long, an approximation can be performed if the probabilities are reliable.

$$\sum_{j=1}^{m} P(C_j) - \sum_{j=2}^{m}\sum_{i=1}^{j-1} P(C_i \cap C_j) \leq P(Fault) \leq \sum_{j=1}^{m} P(C_j). \tag{1.18}$$

The probability of a cut set $P(C_j)$ is the product of the probabilities of its basic events.

$$P(C_j) = P(E_1 \cap E_2 \cap \ldots \cap E_{C_j}) = \prod_{i=1}^{C_j} P(E_i). \tag{1.19}$$

Thus the probability of a cut set is:

$$\sum_{j=1}^{m}\prod_{i=1}^{C_j} P(E_i). \tag{1.20}$$

Considering the example 1.1 with the following probabilities:

$$P(\text{Old.L1}) = P(\text{Old.L2}) = 1 \times 10^{-2}, \tag{1.21}$$

$$P(\text{W.off}) = 1 \times 10^{-3}, \tag{1.22}$$

$$P(\text{SW.ko}) = 1 \times 10^{-5}, \tag{1.23}$$

we calculate of the critical event probability as follows:

$$\begin{aligned} P(\text{Lights broken}) &= P(\text{Old.L1}) \times P(\text{Old.L2}) \\ &= 1 \times 10^{-4} \end{aligned} \tag{1.24}$$

$$\begin{aligned} P(\text{system fault}) &= P(\text{Lights broken}) + P(\text{NoPower}) - P(\text{Lights broken}) \times P(\text{NoPower}) \\ &= 1 \times 10^{-4} + 1 \times 10^{-3} - 1 \times 10^{-4} \times 1 \times 10^{-3} \\ &\approx 1 \times 10^{-3} \end{aligned} \tag{1.25}$$

$$\begin{aligned} P(\text{NoPower}) &= P(\text{W.off}) + P(\text{SW.ko}) - P(\text{W.off}) \times P(\text{SW.ko}) \\ &= 1 \times 10^{-3} + 1 \times 10^{-5} - 1 \times 10^{-8} \approx 1 \times 10^{-3} \end{aligned} \tag{1.26}$$

Fault Tree Analysis and Reliability Block Diagram, limits

The limits of the Fault Tree Analysis and the Reliability Block Diagram are the incapacity to model a reparable system. In fact neither FTA or RBD allow loops of failure or reparations. For this is necessary to use another method which allows it. Automata and Petri Net are best suited for doing this task. This will be discussed in the next chapter.

. . . automaton is the latinization of the Greek $\alpha\dot{\upsilon}\tau\acute{o}\mu\alpha\tau o\nu$, automaton, (neuter) "acting of one's own will".

Wikipedia

Chapter 2

Petri Nets

Introduction

The discrete events systems are modeled with different model languages for example automata, timed automata, Markov chains and Petri Nets. Unlike the others approaches, which are used to describe global changes in the states of a system, Petri Nets focus on local events (corresponding to transitions), local conditions (corresponding to places), and local links between events and conditions. Therefore, one can give a more adequate simulation of distributed asynchronous systems using Petri nets rather than using automata. Petri Nets are widely used in industry's cases, due to its easy translation to UML (Unified Model Language), for e.g. a method is explain in [6]. Traditionally Petri Nets are divided into *low-level Petri Nets* and *high-level Petri Nets.* Low level Petri Nets (such as Place/Transition Nets explained in section 2.1) are primarily used as a theoretical model for concurrency, although certain classes of low-level Petri Nets are often applied to modelling and verification of hardware systems. A high-level Petri nets are suitable for practical use, in particular because they allow the construction of compact and parameterised models. In this work the Coloured Petri Nets (CP-nets) are chosen. CP-net is a high-level Petri net characterised by the combination of a Petri Nets and programming language (Stantard ML, in our case). A hierarchical and a non-hierarchical CP-net are described in section 2.2 in a formal way, following [7].

2.1 Petri Nets: definitions and proprieties

As defined in [8] the Petri Nets are divided into three different levels:

1. the first level concerns of net systems whose places are marked by at most one unstructured token; i.e. of net systems whose places represent "conditions";
2. the second level concerns net systems whose places are marked by several unstructured token; i.e. net systems whose places represent "counters";
3. the third level concerns net systems whose places are marked by structured tokens; i.e. net systems, often denoted, generally, as "high-level nets", where the information is attached to tokens.

A general definition of basic Petri Net is given below:

Definition 2.1 (Basic Petri Nets)**.** *A basic Petri nets is a triple* $N = (S, T, F)$*:*

1. $S \cap T = \varnothing$,
2. $S \cup T \neq \varnothing$,
3. $F \subseteq (S \times T) \cup (T \times S)$,
4. $\operatorname{dom} F \cup \operatorname{cod} F = S \cup T$,

The elements of S are called S-elements, the elements of T are called T-elements, F is the flow relation and its elements are called arcs The set $X = S \cup T$ is the set of elements of the net, $\operatorname{dom} F$ *and* $\operatorname{cod} F$ *are respectively the domain and the codomain of F.*

Given a net N=(S, T, F) and $x \in X$, we define

Definition 2.2 (Pre-set)**.** ${}^{\bullet}x = y \in X|(y, x) \in F$
${}^{\bullet}x$ *is called a pre-set of x and its elements are called pre-elements of x.*

Definition 2.3 (Post-set)**.** $x^{\bullet} = y \in X|(x, y) \in F$
$x^{\bullet}$ *is called a post-set of x and its elements are called post-elements of x.*

The most widely used Petri nets model is the Place/Transition(P/T) Petri Net, it is a second level Petri Net. The places can be marked by one or more unstructured tokens and represent counters. Places are characterized by a capacity, which expresses the maximum of tokens each place can contain; arcs are characterized by a weight, which expresses how many tokens flow through them at each occurrence of the involved transition.

A definition of a general P/T system taken from [9] is given below:

Definition 2.4 (P/T system)**.** *A sixtuple $\Sigma = (S, T, F, K, W, M_0)$ is called a Place/Transition system iff*

(a) (S, T, F) is a net where the S-elements are called places and the T-elements are called transitions.

(b) $K : S \to N^+ \cup \infty$ is a capacity function.

(c) $W : F \to N^+$ is a weight function.

(d) $M_0 : S \to N$ is an initial marking function which satisfies: $\forall s \in S : M_0(s) \leq K(s)$.

A P/T system such that: $\forall s \to S : K(s) = \infty$ and $\forall f \in F : W(f) = 1$ can be simply denoted by $\Sigma = (S, T, F, M_0)$ and is often called an *ordinary* Petri Net or simply a *Petri Net.*

Definition 2.5 (P/T Petri Nets)**.** *A Place Transition net (or simply a Petri Net) is a tuple (P, T, F, W), where:*

- *(P, T, F) is a basic Petri Net,*
- *$W \in F \to \mathbb{N} \backslash \{0\}$ is an arc's weight function.*

An example of a Petri Net is given in Figure 2.1.

Example 2.1. *The Petri Net illustrated in Figure 2.1 is characterized by :*

- *Places : $P = \{p_1, p_2\}$*
- *Transitions : $T = \{t_1, t_2\}$*

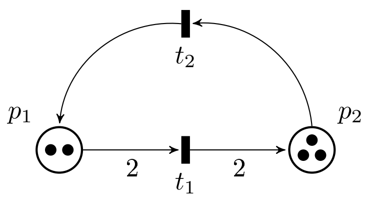


Figure 2.1: Example of a Petri Net

- *Arcs :* $F = \{(p_1, t_1), (t_1, p_2), (p_2, t_2), (t_2, p_1)\}$
- *Weights :*
  $W(p_1, t_1) = 2$
  $W(t_1, p_2) = 2$
  $W(p_2, t_2) = 1$
  $W(t_2, p_1) = 1.$

We will further need the definitions given below.

Definition 2.6 (Multi-set)**.** *Let A be a set of tokens. $\mathbb{B}(A) = A \to \mathbb{N}$ is defined as the set of multi-sets (bags) over A, i.e., $X \in \mathbb{B}(A)$ is a multi-set where for each $a \in A : X(a)$ denotes the numbers of times a is included in the multi-set.*

In other words the multi-set is a generalisation of the notion of set in which elements are allowed to appear more than once. This definition is used periodically in Petri Net, in this case the element are the token which has the ability to be in different places.

Definition 2.7 (Marking)**.** *Let $N = (P, T, F, W)$ be a Petri net. A marking M of N is a multi-set over P, i.e. $M \in \mathbb{B}(P)$*

From the example 2.1, where M is a marking of the places:

- $M(p_1) = 2$
- $M(p_2) = 3$

Definition 2.8 (Firing rule)**.** *Let $N = (P, T, F, W)$ be a Petri net and $M \in \mathbb{B}(P)$ be its marking.*

- *An enabled transition $t \in T$ is noted as $(N, M)[t >$; if and only if, $M \geq^\bullet t$.*
- *An enabled transition t can fire while changing the state M to M', it is noted as $[(N, M)[t > (N, M')]$, if and only if, $M' = (M -^\bullet t) + t^\bullet$.*

An important tool for Petri Net analysis is its *reachability* graph. The reachability graph is an oriented graph that describes the *state space* of the system, in other words, the possible system's states. Following Murata [10] the reachability is defined as: *Reachability is a fundamental basis for studying the dynamic properties of any system. The firing of an enabled transition will change the token distribution (marking) in a net according to the transition rule. A sequence of firings will result in a sequence of markings. A marking M, is said to be reachable from a marking M_0 if there exists a sequence of firings that transforms $M_0 \to M$.*

Definition 2.9 (Reachability graph)**.** *Let $N = (P, T, F, W)$ be a Petri net and $M \in \mathbb{B}(P)$ be a marking. The reachability graph of (N, M) is the graph (V, E) with as vertices $V = R(N, M)$ the set of all reachable markings and as edges $E = \{(M', t, M'') \in V \times T \times V | ((N, M')[t > (N, M''))\}$ the set of all possible state changes. Note that $(M', t, M'') \in E$ denotes that M'' is reachable from M' by firing t.*

In general, the reachability graph may be infinite, if there is no bound on the number of tokens on some at its places. This situation is called the *state space explosion problem*, a problem still existing in the Petri Nets methodologies which limits the power of the state space analysis. Another graph can also be defined in the Petri Nets analysis : the *coverability graph.* The coverability graph is used when the number tokens in a place infinitely grows. The coverability graph is a finite graph but it computes an *overapproximation* of the reachable markings. In contrast a reachability graph gives an *exact* information about the reachable markings. For this work a finite reachability graph is used, the coverability graph (and tree) will not be used, thus is not defined. A reachability graph, coverability graph and tree are illustrated in Figure 2.3, for the definition or the construction algorithm of coverability and reachability graphs the reader can refer to [10]. It is further necessary to define the ω-markings, used in the coverability graph and in automata with infinite words.

Definition 2.10 (ω-marking)**.** *Let $N = (P, T, F, W)$ be a Petri net with initial marking M'.*
A ω-marking M of N is an extended multi-set over P, i.e. $M \in A \rightarrow (\mathbb{N} \cup \{\omega\})$.
If $M(p) = \omega$, then place $p \in P$ is said to be unbounded in M.
If $M(p) \neq \omega$ for all $p \in P$, then M is said to be ω-free.
M is a reachable ω-marking of (N, M') if and only if it appears in the coverability graph of (N, M').

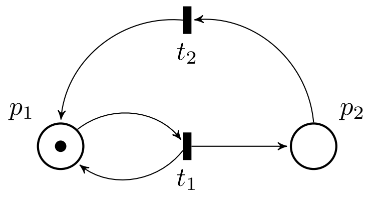


Figure 2.2: Petri Nets

A series of properties of Petri Nets are given below:

Definition 2.11 (Basic Properties)**.** *Let $N = (P, T, F, W)$ be a Petri net and $M \in \mathbb{B}(P)$ be its marking. Then*

- *(N, M) is* ***terminating*** *(or dead) iff there exists a $k \in \mathbb{N}$ such that $|\sigma| \leq k$ for any firing sequence σ (i.e. $(N, M)[\sigma >$).*
- *(N, M) is* ***deadlock-free*** *iff for any $M' \in R(N, M)$ there exists a transition t such that $(N, M')[t >$.*
- *(N, M) is* ***live*** *if and only if for any $t \in T$ and any $M' \in R(N, M)$ there exists a marking $M'' \in R(N, M')$ such that $(N, M'')[t >$.* ***N*** *is live it all of its transitions are live.*
- *(N, M) is* ***bounded*** *iff there is a $k \in \mathbb{N}$ such that for any $M' \in R(N, M)$ and any $p \in P$: $M'(p) \leq k$.* ***N*** *is bounded if and only if all of its places are bounded.*

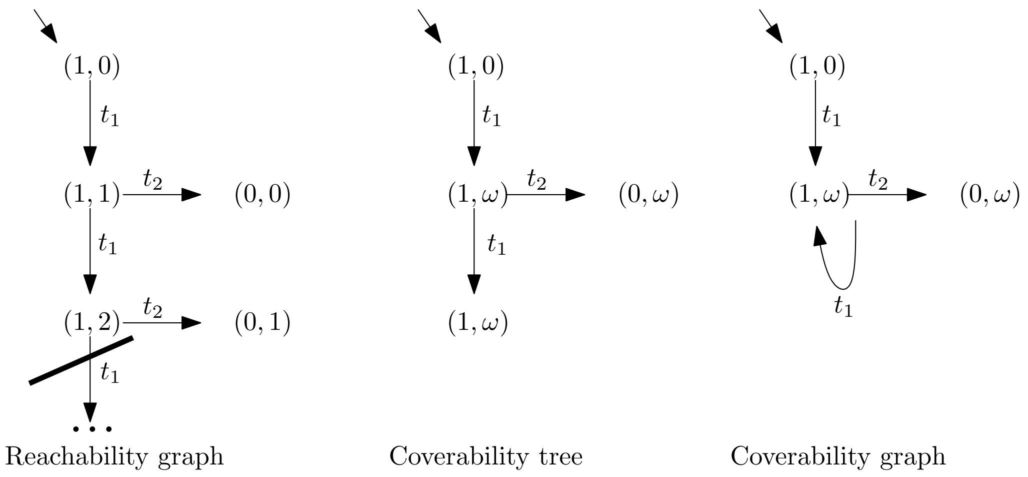


Figure 2.3: Reachability graph and coverability tree/graph of Figure 2.2

- *(N, M) is **safe** iff for any $M' \in R(N, M)$ and $p \in P : M'(p) \leq 1$. **N** is safe if and only if all of its place are safe*
- *M' is a **home marking** if it is reachable from any reachable marking, i.e. for any $M'' \in R(N, M) : M' \in R(N, M'')$.*
- *(N, M) is **reversible** iff for any $M' \in R(N, M) : M \in R(N, M')$. In other words (N, M) is reversible iff **M** is a home marking.*

Supervision in a Petri Nets

The supervision techniques are essential in the transition between probabilistic and deterministic automata, in the chapter 5 is provide the application of this principles. In this section is given an approach learned during my studies at the University of Cagliari with Prof.Giua, the *GMEC approach* in [11]. This approach is taken as begin of an idea to resolve a mutual exclusion problem between two CPN Tools.
The idea of a Generalized Mutual Exclusion Constraints is to assure a condition that limits the weighted sum of tokens in a set of places. To achieve that is used a place called *monitor*.
In our case, the expected control places (monitor) should enforce a set of linear constraints of place markings that maintain the reachable markings in the only behavior that is enabled by the stochastic net. The set of these linear constraints may be expressed by the following equation:

$$Lm \leq k$$

where m is the marking vector of stochastic Petri net, L is a matrix of coefficients and k is a vector of constants.

In the net of the Fig. 2.4, the making constraints are:

(a) $m(P1) + m(P2) + m(P4) + m(P5) \leq 2$

(b) $m(P1) + m(P2) + m(P3) + m(P6) \leq 2$

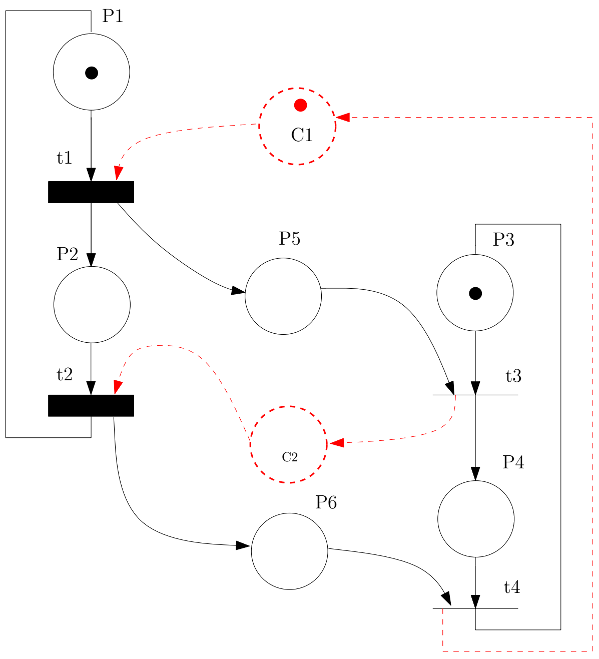


Figure 2.4: Example for the determinisation problem

(a) means that in the marking $(0,1,1,0,1,0)$, t_3 must be fired before t_2 while (b) means that in the marking $(1,0,0,1,0,1)$, t_4 must be fired before t_1. These inequalities can be represented by:

$$L = \begin{pmatrix} 1 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix} k = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

Applying [11], the incidence matrix W_c of the supervisor that enforces the previous constraints and its initial marking are given by:

$$W_c = -LW$$

$$m_{c0} = k - Lm_0$$

where W is the incidence matrix and m_0 is the initial marking of the original SPN. The deterministic untimed net has the following incidence matrix and initial marking:

$$W_{detPN} = [W^T, W_c^T]^T$$

$$m_{detPN0} = [m_0^T, W_{c0}^T]^T$$

For Fig. 2.4 net, this algorithm leads to add two places C1 and C2 with an initial making $m(C1) = 1$ and $m(C2) = 0$.
This operation provides a marking graph that could be analysed using model checking techniques such as explain in the next chapters.

2.2 Coloured Petri Nets

An informal definition of Coloured Petri Nets following [7] is given below.

The Coloured Petri Nets (CP-nets or CPNs) is a graphical language for constructing models of concurrent systems and analysing theirs properties. CP-nets is a discrete-event modelling language combining the capabilities of Petri nets with the capabilities of a high-level programming language. Petri nets provide the foundation of the graphical notation and the basic primitives for modelling concurrency, communication, and synchronisation. The CPN ML programming language, which is based on the functional programming language Standard ML, provides the primitives for the definition of data types, for describing data manipulation, and for creating compact and parameterisable models.

The main difference between a P/T Petri Net and a CPN is that in CPN the token can have different "colors", the "colors" represent different data types in a CP-nets (e.g. integer, string, etc). Verification of CPN models and system properties is supported by the *state space method.* The basic idea underlying the state spaces method is to compute all reachable states and state changes of the CPN model and to represent them as a directed graph, where nodes represent states and arcs represent occurring events. From a constructed state space, it is possible to answer a large set of verification questions concerning the behaviour of the system, such as absence of deadlocks, the possibility of always being able to reach a given state, and the guaranteed delivery of a given service. These proprieties are explain in definition 2.11 for a P/T Petri Nets but are the same for a CP-nets. An important version of CP-nets are the *Hierarchical CP-nets.* In this case the non-hierarchical CP-nets are used as a module of a large Hierarchical CP-nets, allowing the construction of a complex system. A formal definition of CP-nets is follows.

Definition 2.12 (Non-hierarchical Coloured Petri Net)**.** *A non-hierarchical Coloured Petri Net is a nine-tuple $CPN = (P, T, A, \Sigma, V, C, G, E, I)$, where:*

1. *P is a finite set of* ***places****.*
2. *T is a finite set of* ***transitions*** *such that $P \cap T = \emptyset$.*
3. *$A \subseteq P \times T \cup T \times P$ is a set of directed* ***arcs****.*
4. *Σ is a finite set of non-empty* ***colour sets****.*
5. *V is a finite set of* ***typed variables*** *such that Type[ν] $\in \Sigma$ for all variables $\nu \in V$.*
6. *$C : P \to \Sigma$ is a* ***colour set function*** *that assigns a colour set to each place.*
7. *$G : T \to EXPR_V$ is a* ***guard function*** *that assigns a guard condition to each transition t such that Type[G(t)] = Bool, bool standing for boolean datatype.*
8. *$E : A \to EXPR_V$ is an* ***arc expression function*** *that assigns an arc expression to each arc a such that Type[E(a)] $= C(p)_{MS}$, where p is the place connected to the arc a.*
9. *$I : P \to EXPR_{\emptyset}$ is an* ***initialisation function*** *that assigns an initialisation expression to each place p such that Type[I(p)] $= C(p)_{MS}$.*

An hierarchical version of the CP-net should be defined, since it is necessary to model complex systems. We will first define the module hierarchy.

Definition 2.13 (Module hierarchy)**.** *The module hierarchy for a hierarchical Coloured Petri Net $CPN_H = (S, SM, PS, FS)$ is a directed graph $MH = (N_{MH}, A_{MH})$, where*

1. *$N_{MH} = S$ is the set of* ***nodes****.*
2. *$A_{MH} = \{(s_1, t, s_2) \in N_{MH} \times T_{sub} \times N_{MH} | t \in T_{sub}^{s_1} \wedge s_2 = SM(t)\}$ is the set of* ***arcs****.*

Definition 2.14 (Hierarchical Coloured Petri Net)**.** *An hierarchical Coloured Petri Net is a four-tuple $CPN_H = (S, SM, PS, FS)$ where:*

1. *S is a finite set of modules. Each module is a **Coloured Petri Net Module** $s = ((P^S, T^S, A^S, \Sigma^S, V^S, C^S, G^S, E^S, I^S), T^S_{sub}, P^S_{sub}, PT^S)$. It is required that $(P^{S_1} \cup T^{S_1}) \cap (P^{S_2} \cup T^{S_2}) = \emptyset$ for all $s_1, s_2 \in S$ such that $s_1 \neq s_2$.*

2. *$SM : T_{sub} \to S$ is a **submodule function** that assigns a **submodule** to each substitution transition. It is required that the module hierarchy is acyclic.*

3. *PS is a **port-socket relation function** that assigns a **port-socket relation** $PS(t) \subseteq P_{sock}(t) \times P^{SM(t)}_{port}$ to each substitution transition t. It is required that $ST(p) = PT(p'), C(p) = C(p')$, and $I(p)\langle\rangle$ for all $(p, p') \in PS(t)$ and all $t \in T_{sub}$.*

4. *$FS \subseteq 2^P$ is a set of non-empty **fusion sets** such that $C(p) = C(p')$ and $I(p)\langle\rangle = I(p')\langle\rangle$ for all $p, p' \in fs$ and all $fs \in FS$.*

The hierarchical CP-net are used in this work, their implementation is shown in chapter 5. For an easier understanding of a CP-nets, an academic example of concurrency is presented below (see figure 2.5 for illustration).

Example 2.2 (The Dining Philosophers)**.** *Five silent philosophers sit at a table around a bowl of spaghetti. A fork is placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. Eating is not limited by the amount of spaghetti left: an infinite supply is assumed. However, a philosopher can only eat while holding both the fork from the left and the fork from the right (an alternative problem formulation uses rice and chopsticks instead of spaghetti and forks). Each philosopher can pick up an adjacent fork, when available, and put it down, when holding it. These are separate actions: forks must be picked up and put down one by one.*

To model this problem we use a CP-Net, well-suited for concurrency problems. The modelisation and analysis of this CP-net is performed by CPN-Tool, developed by Aarhus University and now maintained by AIS Group in Eindhoven University of Technology (CPN Tools website).

CPN Tools

CPN Tools is the software used for modeling the CP-nets. It is composed of a functional language (Standard ML) and a graphical representation thanks to arcs, places and transitions. We define a marking in CPN following the further definition.

Definition 2.15 (Untimed marking in CPN Tools)**.** *A marking in CPN Tools is represented as:*

```
n`tokenA ++ k`tokenB;
```

where n and k are numbers, n is of tokenA or of tokenB present in the place and $++$ means that after the first token definition the other one is defined.

The marking can be a timed.

Definition 2.16 (Timed marking in CPN Tools)**.** *A timed marking in CPN Tools is represented as:*

```
n`tokenA@int ++ k`tokenB@int2;
```

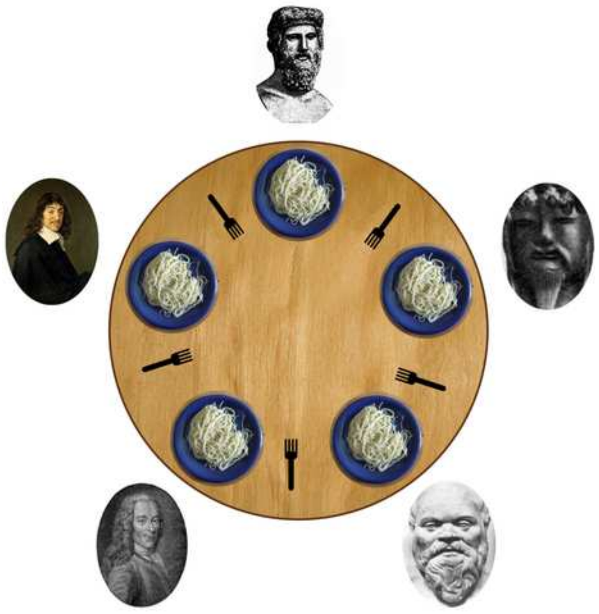


Figure 2.5: The Dining Philosophers problem illustration

where n, k, tokenA, tokenB and ++ are the same of the definition 2.15 and the value after @ is an integer which means that the token can be fired ***only*** *after the moment when the simulation clock come to @int value. In this manner, the token is not always in the place but appears only at time @int.*

An implementation of the Dining Philosophers problem in CPN Tools is illustrated below.

In Figure 2.6 a CP-nets, where there are five philosophers and five chopsticks is illustrated. In Figure 2.7 we can see the initial marking for the CP-net. The place *Think* have five tokens of type PH (that means philosophers), the place *Unused Chopsticks* have five tokens of type CH (that means Chopsticks). We can see the variable *p* in the arc from *Think* by transition Take Chopsticks and the function Chopstick(p) from the place *Unused Chopsticks*. The function *Chopsticks(p)* is written in Standard ML, with the following code:

```
fun Chopsticks(ph(i)) =
1`cs(i) ++ 1`cs(if i=n then 1 else i+1);
```

The code means that when a philosopher *i* wants to eat, he takes his chopstick (if we assume a clockwise numeration, the right chopstick is taken) and the other chopstick near him (on his left side).

The selection of which philosopher starts to eat is random. This first transition is represented in figure 2.8.
A reachability graph can be drawn in CPN Tools (see figure 2.9) or exported and drawn with another software, for example Graphviz (see figure 2.10).

Monitors

A useful tool to inspect a CP-Nets developed in CPN Tools is the monitor system, its definition is given below. For detail refer to [12].

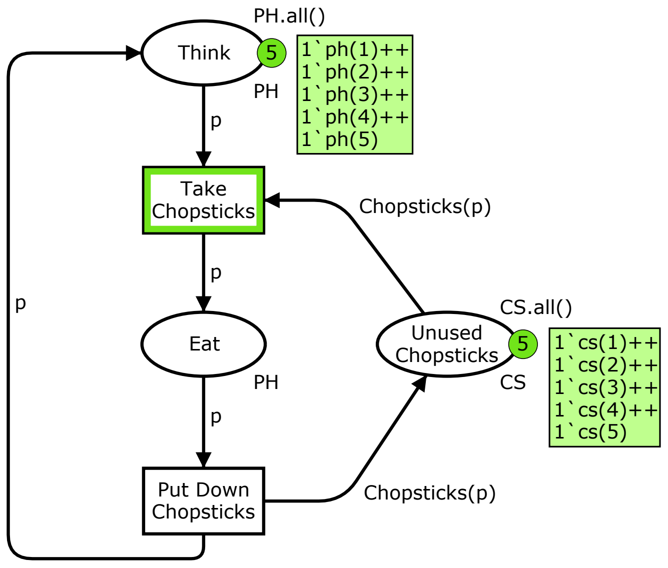


Figure 2.6: The Dining Philosophers problem with a CP-Nets: the model.

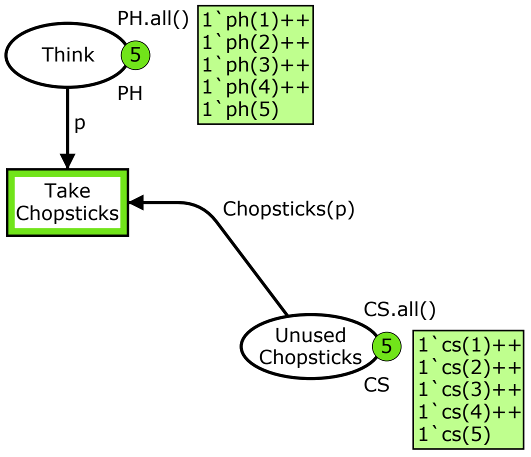


Figure 2.7: The Dining Philosphers problem with a CPN: the initial markings and the first fireable transition.

Definition 2.17 (CPN Tools: Monitors)**.** *A monitor is a mechanism in CPN Tools that is used to observe, inspect, control, or modify a simulation of a CP-net. Many different monitors can be defined for a given net. Monitors can inspect both the markings of places and the occurring binding elements during a simulation, they can take appropriate actions based on the observations. Monitors can be used for each of the activities mentioned above.*

The monitors are used to carry out a performance analysis of a CP-nets. There exists different kinds of monitors, in this work we have used the following ones:

- *Breakpoint monitors* are used to stop a simulation.

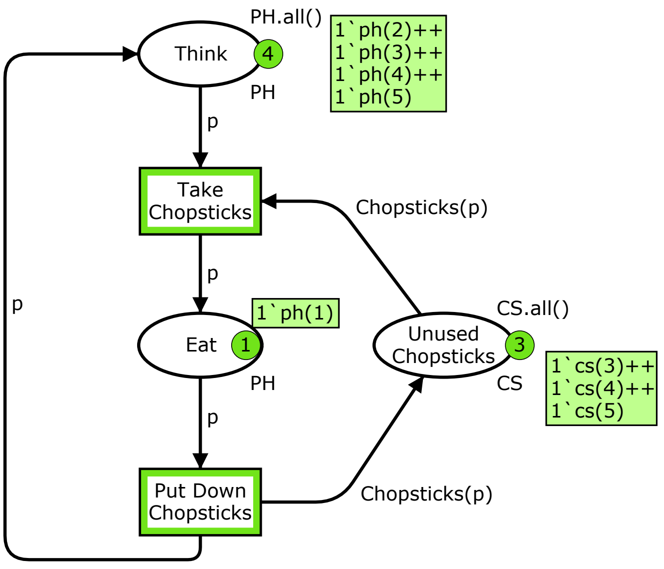


Figure 2.8: The CP-nets after the first transition

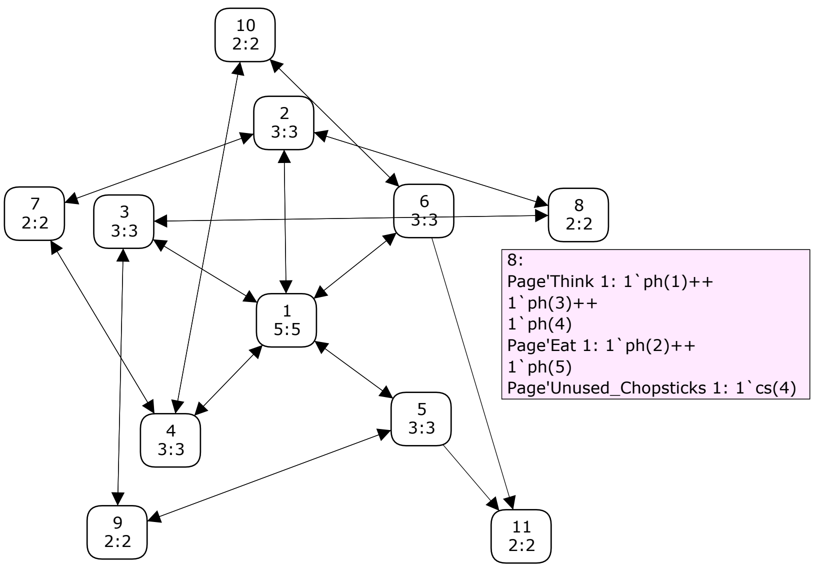


Figure 2.9: Reachability graph of Dining philosophers problem using CPN Tools.

- *Data collector monitors* are used to extract numerical data from a net. The numerical data are then used to perform a statistical analysis of results and the data can be saved in log files. The log files can then be post-processed, e.g. by importing them into spreadsheet programs or plotting.

The application of this monitors in our CP-nets is presented in chapter 6.

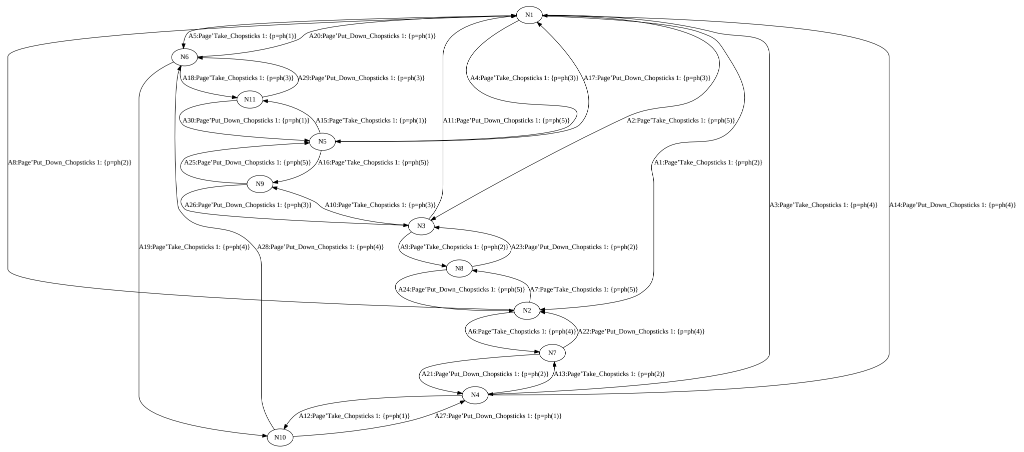

N1
A5:Page'Take_Chopsticks 1: {p=ph(1)}
A20:Page'Put_Down_Chopsticks 1: {p=ph(1)}
N6
A18:Page'Take_Chopsticks 1: {p=ph(3)}
A29:Page'Put_Down_Chopsticks 1: {p=ph(3)}
A4:Page'Take_Chopsticks 1: {p=ph(3)}
A17:Page'Put_Down_Chopsticks 1: {p=ph(3)}
N11
A30:Page'Put_Down_Chopsticks 1: {p=ph(1)}
A15:Page'Take_Chopsticks 1: {p=ph(1)}
A11:Page'Put_Down_Chopsticks 1: {p=ph(5)}
A2:Page'Take_Chopsticks 1: {p=ph(5)}
N5
A8:Page'Put_Down_Chopsticks 1: {p=ph(2)}
A25:Page'Put_Down_Chopsticks 1: {p=ph(5)}
A16:Page'Take_Chopsticks 1: {p=ph(5)}
A1:Page'Take_Chopsticks 1: {p=ph(2)}
N9
A26:Page'Put_Down_Chopsticks 1: {p=ph(3)}
A10:Page'Take_Chopsticks 1: {p=ph(3)}
A3:Page'Take_Chopsticks 1: {p=ph(4)}
A14:Page'Put_Down_Chopsticks 1: {p=ph(4)}
N3
A19:Page'Take_Chopsticks 1: {p=ph(4)}
A28:Page'Put_Down_Chopsticks 1: {p=ph(4)}
A9:Page'Take_Chopsticks 1: {p=ph(2)}
A23:Page'Put_Down_Chopsticks 1: {p=ph(2)}
N8
A24:Page'Put_Down_Chopsticks 1: {p=ph(5)}
A7:Page'Take_Chopsticks 1: {p=ph(5)}
N2
A6:Page'Take_Chopsticks 1: {p=ph(4)}
A22:Page'Put_Down_Chopsticks 1: {p=ph(4)}
N7
A21:Page'Put_Down_Chopsticks 1: {p=ph(2)}
A13:Page'Take_Chopsticks 1: {p=ph(2)}
N4
A12:Page'Take_Chopsticks 1: {p=ph(1)}
A27:Page'Put_Down_Chopsticks 1: {p=ph(1)}
N10

Figure 2.10: Reachability graph of Dining philosophers problem using Graphviz.

SML, Standard ML

A characteristic to the CP-nets are the integration with the functional programming language named *Standard ML*. Like all functional programming languages, the *Standard ML* key feature is the function and his modular system. There are Three main syntactic constructs comprise the SML module system: structures, signatures, and functors. We will define.

Definition 2.18 (Structure in SML)**.** *A structure is a module; it consists of a collection of types, exceptions, values and other structures packaged together into a logical unit*

Definition 2.19 (Signature in SML)**.** *A signature is an interface, usually thought of as a type for a structure: it specifies the names of all the entities provided by the structure as well as the arities of type components, the types of value components, and signatures for substructures.*

Definition 2.20 (Functors in SML)**.** *A functor is a function from structures to structures; that is, a functor accepts one or more arguments, which are usually structures of a given signature, and produces a structure as its result. Functors are used to implement generic data structures and algorithms. It is very important that the input structure of a functors respect the input signature of its.*

For all our needed we have used the book [13] free available online.

Houston, we've had a problem here.

Jack Swigert - Apollo 13 crew member

Chapter 3

Model checking in automata theory

Introduction

The principal idea of verifying the proprieties of a DES is to use a formal method to analyze its state space; for this purpose a temporal logic is used. Temporal logics are a formal methods used to describe a temporal proposition with various symbolisms and writing rules. Temporal propositions are traditionally defined in terms of Kripke structures. We first need to define the atomic proposition.

Definition 3.1 (Atomic proposition). *An atomic proposition (AP) is a type of sentence which is either true or false and which cannot be broken down into other simpler sentences. For example "The dog ran" is an atomic sentence in natural language. In case of automata theory under the AP we understand a particular system's state from the reachability graph which can be used to verified with true or false question. For example if a particular state condition is always true or not.*

Definition 3.2 (Kripke structure). *Let p be a non-empty set of atomic propositions. A Kripke structure is a four-tuple $M = (S, s_0, R, L)$, where*

- *S is a finite set of states,*
- *$s_0 \in S$ is an initial state,*
- *$R \subseteq S \times S$ is a transition relation, for which it holds that $\forall s \in S : \exists s' \in S : (s, s') \in R$,*
- *$L : S \to 2^{AP}$ is labeling of a function which labels each state with the atomic propositions which hold in that state.*

A Kripke structure is basically a graph having the reachable states of the system as nodes and state transitions of the system as edges. It also contains labeling of system's states of with properties that hold in each state. The method of translation from Kripke structure to an automaton is given in section 3.2. The model checking problem can now be defined.

Definition 3.3 (Model Checking problem). *Let a Kripke structure K and an LTL formula φ. The model checking problem consists in to verifying that for all path π of the atomic proposition $p \in AP$ the following relation is satisfied*

$$K \models \varphi \tag{3.1}$$

If equation 3.1 is satisfied, the result of model checking is a true value else the result is a counter-example indicating that φ is not satisfied.

An illustration of this problem is given in figure 3.1, presenting an overview of LTL model checking algorithm. The field of Model Checking is in a continuous development, thus numerous

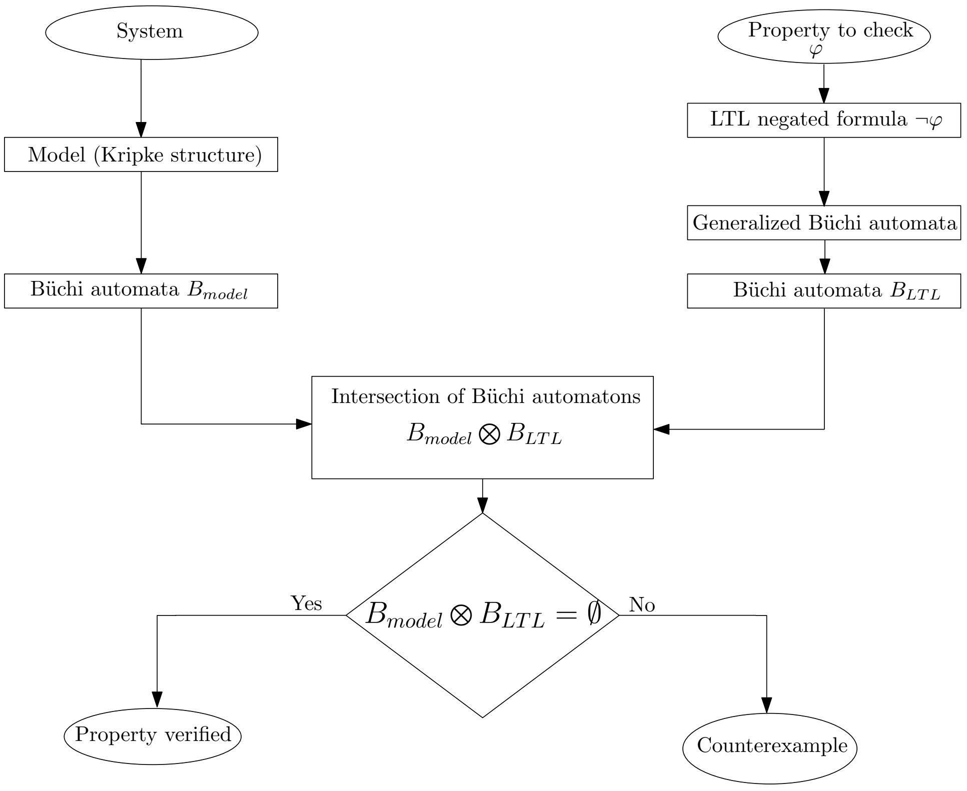

System
Model (Kripke structure)
Büchi automata B_{model}
Property to check φ
LTL negated formula $\neg\varphi$
Generalized Büchi automata
Büchi automata B_{LTL}
Intersection of Büchi automatons $B_{model} \otimes B_{LTL}$
$B_{model} \otimes B_{LTL} = \emptyset$
Yes
No
Property verified
Counterexample

Figure 3.1: An overview of the LTL Model Checking

techniques have been proposed to address this problem. Only one of those will be approached in this work. Researchers particularity focus on improvement of the translation of LTL formula to an automaton. Others approaches to optimize the solution of the model checking problem are explained in [14], [15] and [16]. In section 3.5 a presentation of the *On-The-Fly* model checking procedure is given. This procedure is aimed at an efficient state space analysis. On-the-fly method is used for model checking in the software that we use for case study.

3.1 Temporal logic

Temporal logic is used to describe various types of systems with rules and symbolism for their representation and reasoning about them. The propositions in temporal logic are formulated in terms of time. For example we can express statements like "In a typical week in Lorraine there is *always* sun", "In a typical week in Lorraine there is *eventually* sun" or "In a typical week in Lorraine there is sun *until* it starts raining". An important application of temporal logic is formal verification, where it is used to state requirements of hardware or software systems. There exist different types of temporal logics: CTL*, CTL, LTL. CTL* (Computational tree logic *) is a superset of Computation tree logic (CTL) and Linear temporal logic (LTL). The CTL and LTL have different power of expression. Indeed, CTL is a modal branching temporal logic and includes two path quantifiers. LTL is a modal linear temporal logic referring to time, and it has only one path, a linear temporal path. CTL and LTL are not equivalent but have a common subset, which is a proper subset of both CTL and LTL. Thus, some formulae exist in CTL but not in LTL and viceversa. For example :

- LTL formula $G(Fp)$ can't be defined in CTL (G stand for the *GLOBALLY* operator and F for *FUTURE* operator, the meaning are explain in the section 3.3).
- CTL formula $AG(p \rightarrow (EXq \wedge EX\neg q))$ can't be defined in LTL. In this case we have the operator X that stand for *NEXT*, $\wedge$ for the intersection of the two subformula and $\rightarrow$ that means implies, like above, this operators are explain in 3.3. The operators E and A which stands for *EVENTUALLY* and *ALWAYS* are not explained. Their are defined in CTL as *path operators*. A AP expressed with *ALWAYS* should be verified for every path, instead with *EVENTUALLY* is sufficient only one path.

The LTL logic used further for our study will be peresented in detailes in section 3.3. There exists different dialects of CTL and LTL, like ACTL, QLTL, LTL-X, the only difference between these dialects is lies in extensions of the general rules, the symbolism remaining the same. An important property of the LTL logic is that an LTL formula can be translated into an automaton. Thanks to this property the model checking problem is solved by the analysis of the product between the LTL automaton (in general it is a Labeled Büchi Automaton (LBA) defined in section 3.2) and the system automaton (another LBA). Next section deals with translation of an system's model represented as a Kripke structure to a LBA, used in *model checking*.

3.2 From Kripke structure to Labeled Büchi Automata

Various phases of a typical model checking analysis with LTL are illustrated in figure 3.1. The model system should be a Kripke structure. A finite automata over finite or infinite words (also called ω-automaton) is a Kripke structure. Most concurrent systems are designed not to halt during normal execution, that is why the common choice is to use an automaton over infinite sequences of states, which is a ω-automata. The simplest automata over infinite words are the Labeled Büchi Automaton. A formal definition of Labeled Büchi Automaton is given below:

Definition 3.4. *An LBA is a six-tuple:*

$$LBA = \langle S, A, \Delta, s_0, F, L \rangle, \tag{3.2}$$

where

- S *is a finite number of states*
- $A = 2^{AP}$
- $\Delta \subseteq S \times S$ *is a transition relation*
- $s_0 \in S$ *is an initial state*
- $F \subseteq S$ *is the acceptance condition.*
- $L : S \rightarrow 2^{AP}$ *the labelling function*

LBA accept a runs in which at least one of the infinitely often occurring states is in F.

A Labeled Büchi Automaton has the same components as an automaton over finite words. However, F is called the set of accepting states, rather than final states. The transformation from a k to LBA is very easy. A Kripke structure corresponds to an LBA, where all the states are accepted. An example of a Kripke structure transformed to LBA is given in figure 3.2. Sometimes it is convenient to work with a variation of Labeled Büchi Automaton named *Labeled*

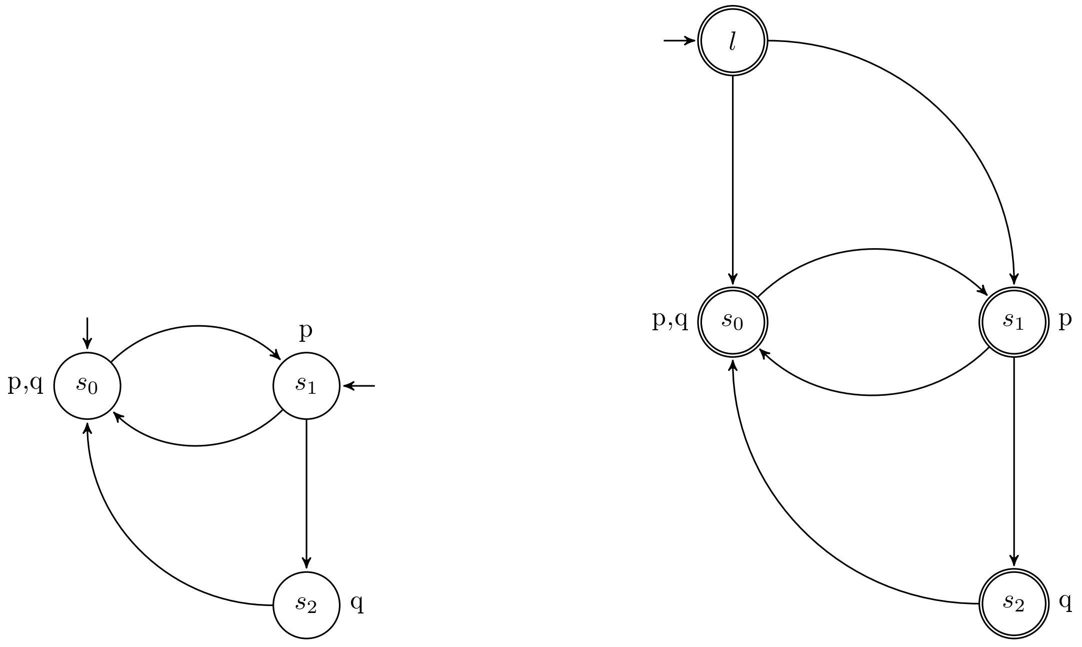


Figure 3.2: Trasforming a Kripke structure (left) into a Labeled Büchi Automaton (right). p, q are atomic propositions.

Generalized Büchi Automaton with several accepting sets, although this does not extend the set of languages that can be expressed. The Labeled Generalized Büchi Automaton (LGBA) is defined as follows:

Definition 3.5 (Labeled Generalized Büchi Automaton). *A LGBA is a six-tuple:*

$$LGBA = \langle S, A, \Delta, s_0, F, L \rangle, \tag{3.3}$$

where

- *S is a finite number of states*
- *$A = 2^{AP}$ is the set of all the atomic propositions*
- *$\Delta \subseteq S \times S$ is a transition relation*
- *$s_0 \in S$ is an initial state*
- *$F_1, F_2, \ldots, F_n$ is a set of acceptance condition*
- *$L : S \to 2^{AP}$ the labelling function*

LGBA accepts only runs in which the set of infinitely often occurring states contains at least a state from each $F_1, F_2, \ldots, F_n$.

Multiple sets of states in acceptance condition can be translated into one set of states by an automaton construction. The translation from a LGBA to a LBA is explained below.

Definition 3.6. *Let LGBA $= \langle S, A, \Delta, s_0, F_0, \ldots, F_n, L\rangle$ where $F_0, \ldots, F_n$ are sets of accepting states, the equivalent Labeled Büchi Automaton is LBA $= \langle S', A', \Delta', q_0', F', L'\rangle$ where:*

- $S' = S \times i | 0 < i \leq k$
- $A' = A$
- $s_0' = s_0 \times i$ *for any* $0 < i \leq k$
- $(s, i) \rightarrow (s', i) \iff s \rightarrow s'$ *and* $s \notin F_i$
  $(s, i) \rightarrow (s', (i \mod k) + 1) \iff s \rightarrow s'$ *and* $s \in F_i$
- $F' = F_i \times i$ *for any* $0 < i \leq k$
- $L'(s, i) = L(s)$.

Example 3.1. *Consider the following Labeled Generalized Büchi Automaton:*

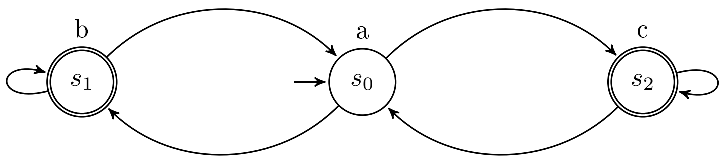


Figure 3.3: An example of Labeled Generalized Büchi Automaton

It has two acceptance sets $F_1 = s_1$ and $F_2 = s_2$. The states of the corresponding simple LGBA are $\{s_0, s_1, s_2\} \times \{1, 2\}$.
The following transitions are possible (only some examples are given) :

- $(s_0, 1) \rightarrow (s_1, 1)$ *since* $s_0 \rightarrow s_1$ *and* $s_0 \notin F_1$
- $(s_0, 1) \rightarrow (s_2, 1)$ *since* $s_0 \rightarrow s_2$ *and* $s_0 \notin F_1$
- $(s_1, 1) \rightarrow (s_0, 2)$ *since* $s_1 \rightarrow s_0$ *and* $s_1 \in F_1$
- $(s_1, 2) \rightarrow (s_1, 2)$ *since* $s_1 \rightarrow s_1$ *and* $s_1 \notin F_2$
- $(s_2, 2) \rightarrow (s_2, 1)$ *since* $s_2 \rightarrow s_2$ *and* $s_2 \in F_2$

A possible LBA will have the set of initial states $(s_0, 1)$ and the set of accepting states $(s_2, 2)$. Any accepting run of the LBA must visit $(s_2, 2)$ infinitely often ($s_2 \in F_2$). In order to do so it also has to visit a state labelled with $s_1 \in F_1$ infinitely often. Thus, an accepting run of the resulting LBA visits some state of F_1 and some state of F_2 infinitely often.

Once the system represented as a Labeled Büchi Automaton, the property to be verified is also to be translated to Labeled Büchi Automaton in order to perform the model checking analysis. Translation of a property formulated in LTL to Labeled Büchi Automaton is detailed in the next section.

3.3 From LTL formula to Labeled Büchi Automaton

After the definition of the Labeled Büchi Automaton we can be more specific about the *model checking problem*. In automata theory the definition 3.3 can be translated in terms of automata language:

$$L(M) \subseteq L(S), \tag{3.4}$$

where M is the system's automaton model and S the specification automaton. Using LTL, the specification automaton can be translated to a LBA, introduced in this section. It is indeed that the model presented as are LBA. The expression 3.4 can be rewritten in the following way.
Let $L(\bar{S})$ be the language $A^{\omega} - L(S)$ then:

$$L(M) \cap L(\bar{S}) = \emptyset \tag{3.5}$$

where the $L(\bar{S})$ is the complement of the language of S automaton. The expression 3.5 is explained in section 3.4.

LTL - Linear Temporal Logic

The LTL logic is chosen for the purposes of this work. Indeed its translation into a Labeled Büchi Automaton is simple compared to other temporal logics.

Definition 3.7 (Linear temporal logic)**.** *Let $p \in AP$ an atomic proposition and $\varphi aproperty$, the LTL formulae is defined as follows:*

$$\varphi = p|\varphi \vee \varphi|\varphi \wedge \varphi|\neg\varphi|\Box\varphi|\Diamond\varphi| \bigcirc \varphi|\varphi U \varphi|\varphi R \varphi \tag{3.6}$$

where $\vee, \wedge, \neg, \bigcirc, \Diamond, \Box, U, R$ are operators defined further in tables 3.1, 3.2, 3.3 and in table 3.4 some useful expansion rules.

An extended lists of operators are reported in the next tables.

Symbol	Explanation
$\varphi \vee \varphi$	φ union φ
$\varphi \wedge \varphi$	φ intersection φ
$\neg\varphi$	NOT φ
$\rightarrow \varphi$	implies φ
$true$	true value
$false$	false value

Table 3.1: LTL logical operators

Textual	Symbol	Explanation
$X\varphi$	$\bigcirc\varphi$	ne$X$t : φ as to hold at the next state
$\psi U \varphi$	$\psi U \varphi$	Until : ψ has to hold at least until φ

Table 3.2: LTL temporal operators

Others derived operators can be defined from these basic operators but for this work the derived operators are not used. A little example of an LTL formula with a natural human language is given below.

Textual	Symbol	Relation	Explanation
$F\varphi$	$\Diamond\varphi$	$trueU\varphi$	Future :φ as to hold at almost one path
$G\varphi$	$\Box\varphi$	$FR\varphi$	Globally : φ as to hold at all path
$\psi R\varphi$	$\psi R\varphi$	$\neg(\neg\psi U\neg\varphi)$	Release : φ has to true until ψ is verified

Table 3.3: LTL operators

$$G\varphi \approx \varphi \wedge XG\varphi$$
$$F\varphi \approx \varphi \vee XF\varphi$$
$$\varphi U\psi \approx \psi \vee (\varphi \wedge X(\varphi U\psi))$$

Table 3.4: LTL expansion rules

Example 3.2. *For this example we set $AP = Sun$, and use some LTL formula where the model **M** is a week. Figure 3.4 illustrates the situation where AP holds in at least 1 day in a week*

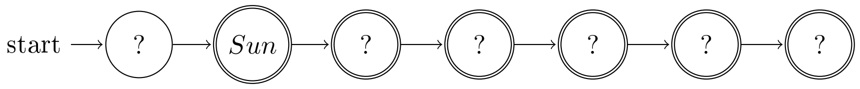


Figure 3.4: $M, s \models \mathbf{F}AP$, AP is holds at least one day in a week

Figure 3.5 illustrates the situation where all days in a week are sunny.

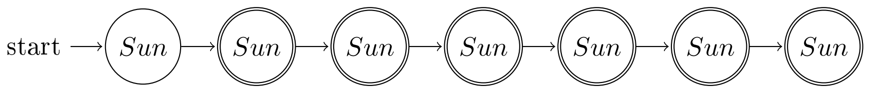


Figure 3.5: $M, s \models \mathbf{G}AP$, AP is hold in all days in the week

Translation algorithm

To translate an LTL formula there exists various techniques, such as tableau construction [17], very weak alternating automata [18], reduction rules [19]. The research in this filed is still very active. In this subsection the tableau algorithm [17] will be presented. In order to apply the translation procedure using the algorithm, the LTL formula φ should be put into *negation normal form*, in which negation is only applied to the atomic proposition. The rules for the negation form are given below:

- $\neg\neg\varphi = \varphi$
- $\neg\Box\varphi = \Diamond\neg\varphi$
- $\neg\Diamond\varphi = \Box\neg\varphi$
- $\neg(\varphi U\psi) = (\neg\psi)R(\neg\varphi)$
- $\neg(\varphi R\psi) = (\neg\psi)U(\neg\varphi)$

The negated formula should have only the operators given in tables 3.1, 3.2 plus the R operator. R operators is dual of U and is used to avoid an exponential blow up in the size of translated formula [16].
To understand the translation process the [20] tableau method is chosen among various techniques. This is a tableau based method where two tables are defined to classify a formula and the result of its translation which is a $LGBA$. Expansion rules are shown in table 3.4, they are necessary for the formula closure.

Definition 3.8 (Labeled Generalized Büchi Automaton). *A Labeled Generalized Büchi Automaton is a six-tuple:*

$$LGBA = \langle S, A, L, \Delta, s_0, F \rangle, \qquad (3.7)$$

where

- *S is a finite number of states*
- *A is a finite set of labels*
- *$L :\rightarrow 2^A$ is a state labelling function*
- *$\Delta \subseteq S \times S$ is a transition relation*
- *$s_0 \in S$ is an initial state*
- *$F \subseteq 2^S$ is a set of accepting states.*

The tableau construction is composed of a few steps (an overview are illustrated in figure 3.6) :

a. Form the closure Φ_φ of the LTL formula φ .

b. Form atoms of given LTL formula φ, using the definition 3.11. Atoms will play the role of states in the resulting generalized Labeled Büchi Automaton.

c. Add transition from atoms A to B.

d. Make atom A initial if A contains φ and a generalised Labeled Büchi Automaton acceptance.

Tableau method

a. We define the tableau method with an example. We take the following formula for explain the method, $\varphi : Fp$.

Definition 3.9 (Formula closure). *Let Φ_φ (formula closure of φ) be the smallest set of formulae satisfying:*

- $\varphi \in \Phi_\varphi$
- $\forall p \in \Phi_\varphi$ *and subformula q of* $\{p, q\} \in \Phi_\varphi$
- $\forall p \in \Phi_\varphi$, $\neg p \in \Phi_\varphi (\neg\neg p \equiv p)$
- $\forall \psi \in Gp, Fp, pUq$, *if* $\psi \in \Phi_\varphi$, *then* $X\psi \in \Phi_\varphi$

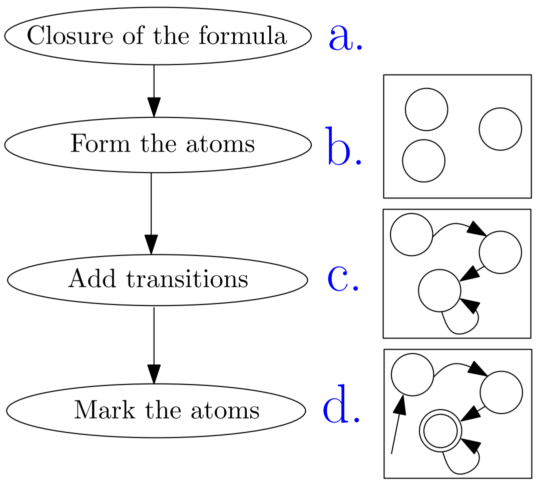


Figure 3.6: Overview of the tableau methods

b. A φ-atom is a set of all the subformula $\in \varphi$. A subformula is a combination of LTL operators. For our formula the subformula are Fp (subformula of φ). The tableau method is composed by the definition of two tables for two different type of subformula α and β formula, this it will explain further. The tables are reported in 3.5. At this moment we can give a formal definition

α	$k(\alpha)$
$p \wedge q$	p, q
Gp	p, XGp

β	$k_1(\beta)$	$k_2(\beta)$
$p \vee q$	p	XFp
Fp	p	p, XFp
pUq	q	$p, X(pUq)$

Table 3.5: α and β tables

of atoms.

Definition 3.10 (Atom definition)**.** *A φ-atom is a subset $A \subseteq \Phi_\varphi$ satisfying:*

- R_{sat} *: the conjunction of all local formulae in A is satisfiable*
- $R_\neg$ *: for every $p \in \Phi_\varphi$, $p \in A$ iff $\neg p \notin A$ (example, for every $p \in \Phi_\varphi$, a φ-atom must contain either p or $\neg p$)*
- R_α *: for every α-formula $\alpha \in \Phi_\varphi$, $\alpha \in A$ if and only if $k(\alpha) \subseteq A$ (example, $Gp \in A$ iff both $p \in A$ and $XGp \in A$)*
- R_β *: for every β-formula $\beta \in \Phi_\varphi$, $\beta \in A$ iff either $k_1(\beta) \in A$ or $k_2(\beta) \subseteq A$ (or both)*

With this definition we can made atoms. For the formula $\varphi : Fp$ we have defined its subformula Fp. From the tables 3.5 we can see that the Fp is a β-subformula. For the definition 3.10 a β formula to have a true value should respect the k_1 or both of the k_2 formulae. An example is reported below.

Example 3.3. *Let $\varphi : Fp$*

$$A_1 = \{\varphi, p, XFp\}$$
$$A_2 = \{\varphi, \neg p, XFp\}$$

A_1 is an atom, A_2 is not (R_β is violated), because with $\neg p$ the Fp should be false. Thus the φ should be $\neg\varphi$.

In other words the property which an atoms should satisfied should be explain as the property showed below.

Definition 3.11 (Property of a formulae)**.** *A α-formula holds at position j if and only if all of $k(\alpha)$-formula hold at j.*
A β-formula that hold at position j if and only if either the $k_1(\beta)$-formula holds at j or all $k_2(\beta)$-formulae hold at j (or both).

A definition of a basic formulae is necessary to check if an atom have all the closure subformula.

Definition 3.12 (Basic formulae)**.** *Basic formulae are propositions or formulae of the form Xp. The presence or absence of basic formulae in an atom A determine the presence or absence of all other closure formulae in A.*

c. An example of this technique is given below.

Example 3.4. *Let $\varphi : Fp$ we can form the closure with the subformula Fp and the 3.5. We obtain $Fp \to p, XFp$ then :*

$$\Phi^+_\varphi : \{\varphi, p, XFp\} \tag{3.8}$$

$$\Phi^-_\varphi : \{\neg\varphi, \neg p, \neg XFp\} \tag{3.9}$$

$$\Phi_\varphi : \{\neg\varphi, \neg p, \neg XFp, \varphi, p, XFp\} \tag{3.10}$$

For the first atom we supposed that the formula is true and both, p and XFp are true. For the second atom, the formula is true but p isn't true, for the third atom XFp isn't true, in the last all subformula aren't true then the $\varphi \notin A_4$ but $\neg\varphi \in A_4$.

$$\begin{aligned} A_0 &= \{\varphi, p, XFp\} \\ A_1 &= \{\neg\varphi, \neg p, XFp\} \\ A_2 &= \{\varphi, p, \neg XFp\} \\ A_3 &= \{\neg\varphi, \neg p, \neg XFp\} \end{aligned} \tag{3.11}$$

After the definition of the atoms we can drawn a graph T_φ of the formula. The nodes of T_φ are the atoms of φ and there exist an edge from an atom A to an atom B if for every $Xp \in \Phi_\varphi$, $Xp \in A$ if and only if $p \in B$. T_φ is the tableau of φ. The atoms A_i correspond to the states s_i in the LGBA illustrated in figure 3.7. For draw correctly the LGBA we should find the initial states and the accepted state.

d. *The initial states are the atoms with the **TRUE** value of φ, in our example A_0 and A_2. The acceptance condition is given by the subformula which is composed our LTL formula. In our case we have a Fp formula then we should look for two acceptance condition.*

1. *Fp: atoms with p or $\neg Fp$*

2. *$\neg Gp \cong F\neg p$: atoms with $\neg p$ of $\neg Gp$*

3. $\neg G\neg p \cong Fp$: atoms with p or $G\neg p$

4. pUq: atoms with q or $\neg(pUq)$

In this way the accepting states are: A_0, A_1, A_2, A_3. This example was very simple, indeed the expansion of the Fp was really unnecessary, without other subformula the four atoms represented can be reduced to only two. This because the atoms A_0 and A_2 can be reduced to $A_{0,2} = \{\varphi, Fp\}$ and the other atoms A_1, A_3 to $A_{1,3} = \{\neg\varphi, \neg Fp\}$. The presence of two acceptance condition, is given by the Fp. Thus $F_0 = \{A_0, A_2\}$ when p is true, $F_1 = \{A_1, A_3\}$ where $\neg Fp$ is true.

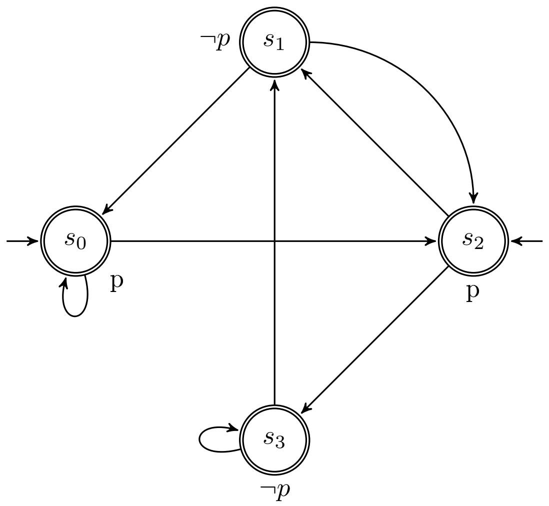


Figure 3.7: The Labeled Generalized Büchi Automaton of $\varphi : Fp$

Labeled Generalized Büchi Automata to Labeled Büchi Automata

The equivalence between LGBA and LBA is explained in the definition ??. In this case we have two acceptance condition F_0 and F_1. We are interest in the evaluation of φ. $\varphi \to p$ because if p is TRUE the formula is satisfied. Thus we choose F_0 as acceptance condition. Afterwards we can build a LBA which accepts an infinite word of p. To remind, a LBA is a ω-language automata, the formula is accepted only if we have a ω-word (an infinitive word). The Labeled Büchi Automaton is illustrated below: It means that if the preposition is true a ω-word should be produced. For example, considering the automata illustrated in figure 3.8 if we start from s_2 only a word is produced: $p(\neg p)^\omega$, the only p can't be accepted from a LBA.

3.4 Intersection automata and checking emptiness

Intersection automata

According to the introduction of chapter 3, the final step in model checking is checking if the product between the LBA of the model and the LBA of the LTL formula is empty. Before doing this a little remark is necessary. There are two ways to realize an LBA, one way is its labeled version with the AP in state, the other way is with the AP in arcs. Later in this section we will present a method to make the product between two LBA with AP in arcs, therefore it is

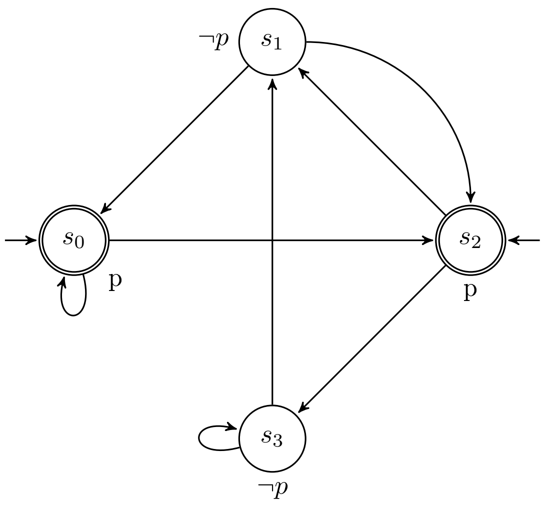


Figure 3.8: The Labeled Büchi Automaton of $\varphi : Fp$

necessary to put the AP from state to arcs. It is an easy operation, the input arcs of a place s_1 should carry on the AP of s_1, this procedure is repeated for all the other states. The definition of an intersection is as follows (we refer to [2]).

Definition 3.13 (Intersection of Labeled Büchi Automaton)**.** *Let $B_1 = \langle S_1, A, \Delta_1, s_1^0, F_1\rangle$ and $B_2 = \langle S_2, A, \Delta_2, s_2^0, F_2\rangle$ the two LBA. We can build an automaton that accepts $L(B_1) \cap L(B_2)$ (where L is the language of automata, consisting of all the accepted words) as follows: $B_1 \cap B_2 = \langle A, S_1 \times S_2 \times 0, 1, 2, \Delta, s_1^0 \times s_2^0 \times 0, S_1 \times S_2 \times 2$. We have $(\langle r_i, q_j, x\rangle, a, \langle r_m, q_n, y\rangle) \in \Delta$ if and only if the following conditions hold:*

1. *$(r_i, a, r_m) \in \Delta_1$ and $(q_j, a, q_n) \in \Delta_2$, that is, the local components agree with the transitions of B_1 and B_2.*
2. *The third component is affected by the accepting conditions of B_1 and B_2*
   - *if $x = 0$ and $r_m \in F_1$, then $y = 1$.*
   - *if $x = 1$ and $q_n \in F_2$, then $y = 2$.*
   - *if $x = 2$ then $y = 0$.*
   - *otherwise, $y = x$.*

The third component guarantees that acceptable states from both B_1 and B_2 appear infinitely often in the product result. An example of the product of two LBA is given in figure 3.9 and 3.10.

A simpler intersection is obtained when all of the states of one of the automata are accepting. Such an intersection is used, for instance, $L(M) \subseteq L(S)$ (where M is the system model and S the specification model), because all the states of the automaton for the modeled system are accepting. Let's assume that all the states of B_1 are accepting and that the acceptance set of B_2 is F_2. Their intersection will be defined as it follows:
$B_1 \cap B_2 = \langle A, S_1 \times S_2, \Delta', s_1^0 \times s_2^0, S_1 \times F_2\rangle$
The accepting states are pairs from $S_1 \times F_2$ in which the second component is an accepting state. Moreover, $(\langle r_i, q_j\rangle, a, \langle r_m, q_n\rangle) \in \Delta'$ if and only if $(r_i, a, r_m) \in \Delta_1$ and $(q_j, a, q_n) \in \Delta_2$. Once the intersection automaton is build, it can be checked to emptiness.

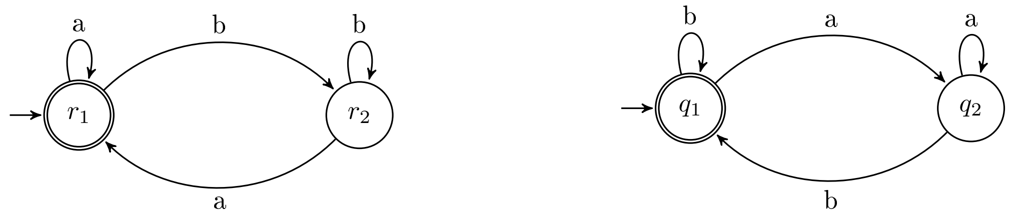


Figure 3.9: An automaton for infinite numer of a's (left) and an automaton for an infinite numer of b's (right).

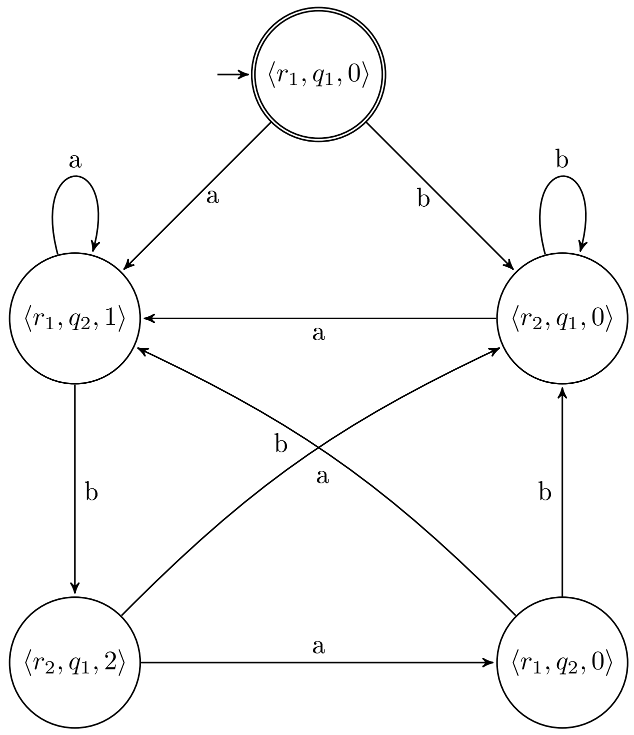

$\langle r_1, q_1, 0\rangle$
a
a
b
b
$\langle r_1, q_2, 1\rangle$
a
$\langle r_2, q_1, 0\rangle$
b
a
b
b
$\langle r_2, q_1, 2\rangle$
a
$\langle r_1, q_2, 0\rangle$

Figure 3.10: Result of product of the two LBA of figure 3.9.

Checking emptiness

The existence of the intersection is not sufficient to prove the emptiness, indeed it's necessary to check if there exists a strongly connected component that is reachable from an initial state and contains an accepting state of intersection automaton. A run that contains a strongly connected component is the counterexample of the checked property. The Depth fist search algorithm (DFS) developed by Tarjan [21] for finding strongly connected components can be used but another technique is implemented in CPN Tools a double DFS explained in [2].

3.5 On-the-Fly Model Checking

In this little section we explain the other main algorithm to check if the system satisfies the property φ. In contrast to the other algorithms in this case only the LTL formula is translated to LBA. The LBA states of the model are generated only when needed, while checking the emptiness of its intersection with the LBA of the LTL formula. This tactic is explained in [22]. There are a series of advantages in On-the-Fly procedure, such as:

1. The translation procedure of the system model into a Labeled Büchi Automaton can meet a state explosion problem, which is not possible with On-the-fly procedure. Indeed, the number of states in the LBA resulting from translation of a system's model to Büchi Automata is exponential with respect to the initial number of system's states.

2. If a counterexample is found before completing the construction of the intersection the algorithm ends.

A disadvantage can be that when a counterexample is found it is not possible to find the other error traces because the algorithm ends.

Part II

Deterministic and probabilistic dependability assessments

Employ every economy consistent with thoroughness, accuracy and reliability.

Arthur C. Nielsen - Market analyst

Chapter 4

Case study : nuclear power plant sub-system

4.1 Introduction

This work is based on a case study developed by Électricité De France (EDF) for the project "APProches de la fiabilité DYNamique pour modéliser des systèmes critiques (APPRODYN)" (described in [23] "The APPRODYN Project: Dynamic Reliability Approaches to Modeling Critical Systems"). In particular the case study touches upon a secondary circuit's sub-system of a nuclear power plant with a pressurized water reactor (PWR). The scheme representing a nuclear power plant is given in Figure 4.1.

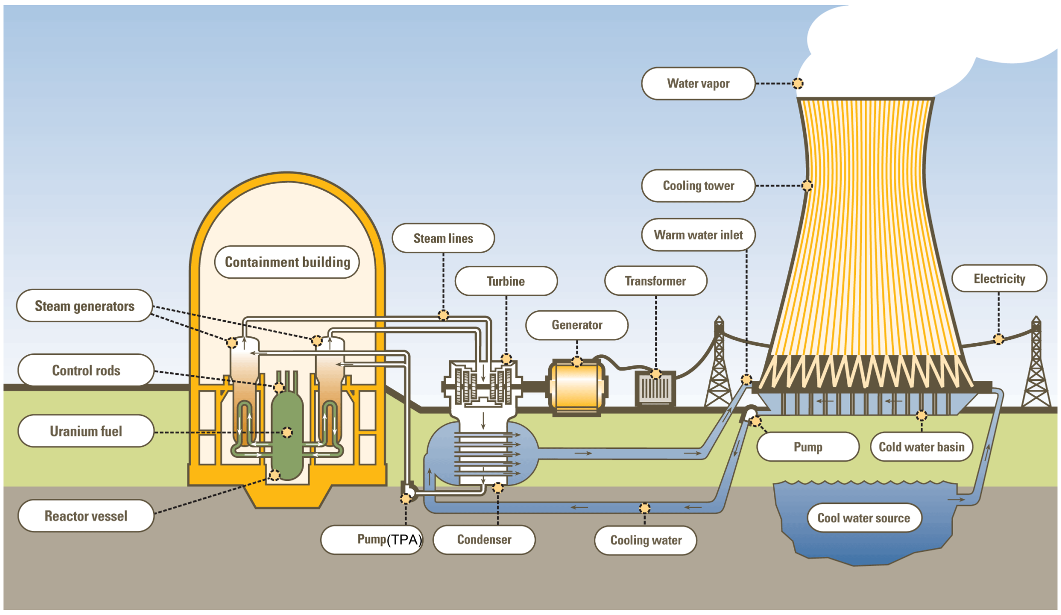


Figure 4.1: Scheme of a nuclear power plant

We can divide the system illustrated in Figure 4.1 in two different parts.

1. The primary water circuit is involved in the nuclear reaction and has irradiated components. It is composed of the reactor vessel, control rods and uranium fuel.
2. The secondary circuit is the circuit not involved in the nuclear reaction, it is the circuit which can communicate with outside environment by means of a cooling tower. It is important that the components of the secondary circuit are not be irradiated because they can bring irradiated vapor or water into the environment. The rest of the elements from the scheme (besides reactor vessel, control rods, etc) can be considered the secondary circuit.

In the case study we focus on the feedwater pump (TPA, Turbopompe alimentaire in French), illustrated in figure 4.2 giving the global installation scheme and in particular in figure 4.3. The purpose of a TPA is to inject the water from the condensator to the steam generators. Other components of the EDF system are illustrated in figure 4.2.

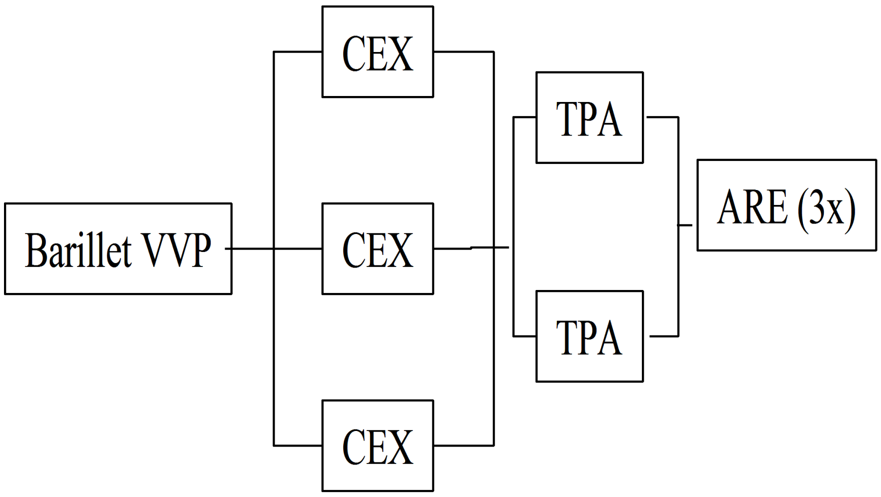


Figure 4.2: Reliability block diagram of EDF case study

For completeness an explanation of the different block are given below:

- *Barillet VVP* is the cylinder containing water for safety purposes,
- *CEX* are the extraction pumps, extracting water from condenser in Figure 4.1 are illustrated as "Pump".
- *TPA* are two feedwater pumps functioning in redundance.
- *ARE* are three valves used to set the input water flow rate to the steam generators.

Precisely two TPAs are used in parallel (see Figure 4.3). The TPA is controlled by a logical model that has as input the events from TPA and for output the orders given to TPA, this is illustrated in figure 4.4.

Detailed explanations of TPA and the logic model are given in next sections.

The TPA physical model for the case study

The composition of the TPAs system is illustrated in Figure 4.5. In this work the two TPA are equivalent from a point of view of physical and logical model. Therefore is it possible to give

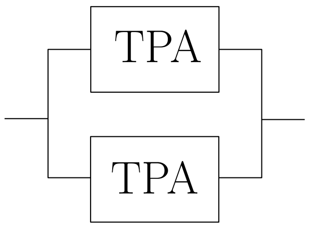


Figure 4.3: Reliability diagram block for the TPA

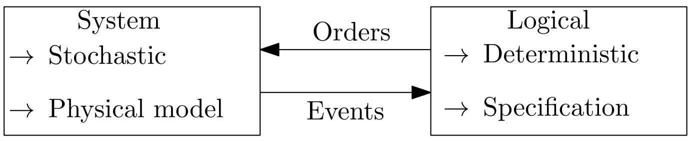


Figure 4.4: Implemented system block diagram

an explanation for one TPA. The parameters used to simulate the system's behaviour can vary, but this aspect will be addressed later.
A TPA is composed of two sub-systems, in-turbine and out-of-turbine part (in Figure 4.5 respectively "Turbine" and "Hors-Turbine"). These components are in a series configuration.

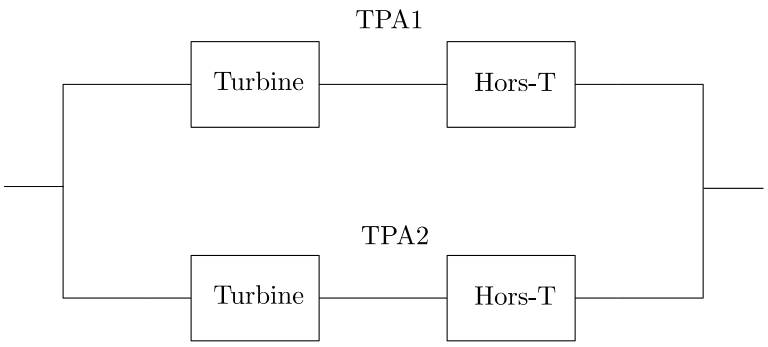


Figure 4.5: Reliability block diagram of TPAs system

The failure rate and the reparation rate of these pumps are shown in table 4.1.

TPA	$\lambda_T[h]$	$\lambda_{HT}[h]$	$\mu_T[h]$	$\mu_{HT}[h]$
1	1.475×10^{-4}	1.459×10^{-4}	1/0.5	1/24
2	4.425×10^{-4}	1.46×10^{-7}	1/24	1/144

Table 4.1: TPAs parameters, λ_x are the failure rate and μ_x the reparation rate for component x.

The considered problem the exponential law is used as the probability distribution for the failure times and Erlang for the reparation times. The scale parameter to the exponential law are $\lambda = frac1\lambda_{failure}$. Instead for the Erlang distribution, the scale parameter are $\lambda = frac1\mu_{reparation}$

and the shape parameter $k = 2$. The mean times to failure and to repair for this system are illustrated in table 4.2.

TPA	$MTTF_T[h]$	$MTTF_{HT}[h]$	$MTTR_T[h]$	$MTTR_{HT}[h]$
1	≈ 6780	≈ 6854	4	≈ 48
2	≈ 2260	≈ 6849315	≈ 48	≈ 288

Table 4.2: MTTF and MTTR of the TPAs expressed in hours, calculated from the used Exponential and Erlang laws.

An automaton for the physical model of TPA that includes the two different failure modes (in-turbine and out-of-turbine) and global parameters (the global parameters r_x, f_x are required for communication with the logical model) are given in figure 4.6.

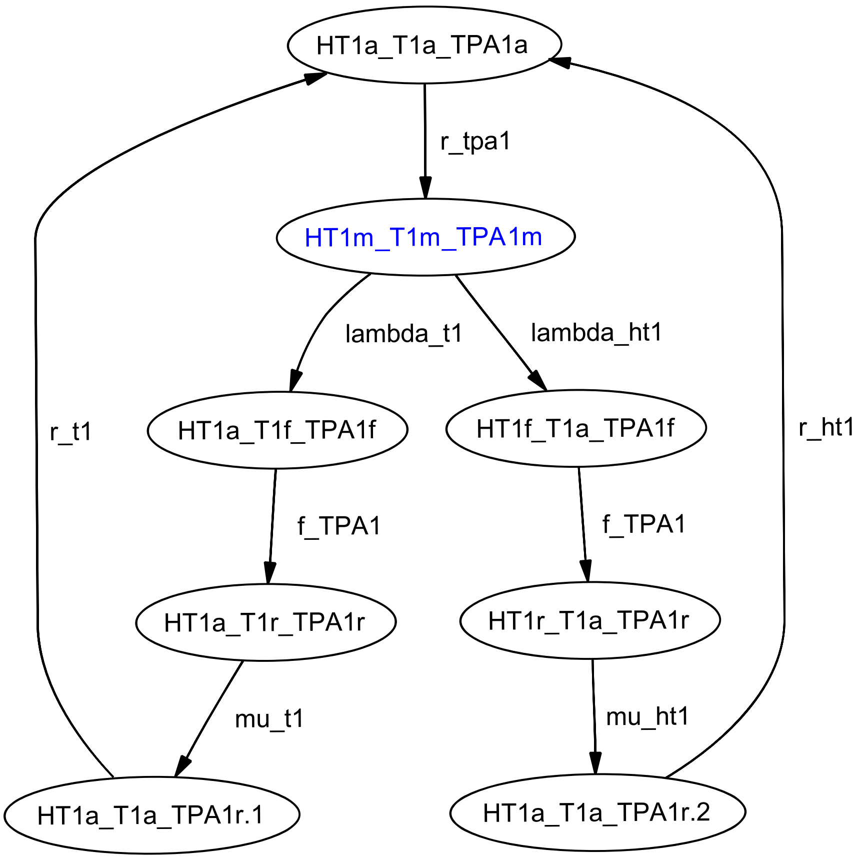
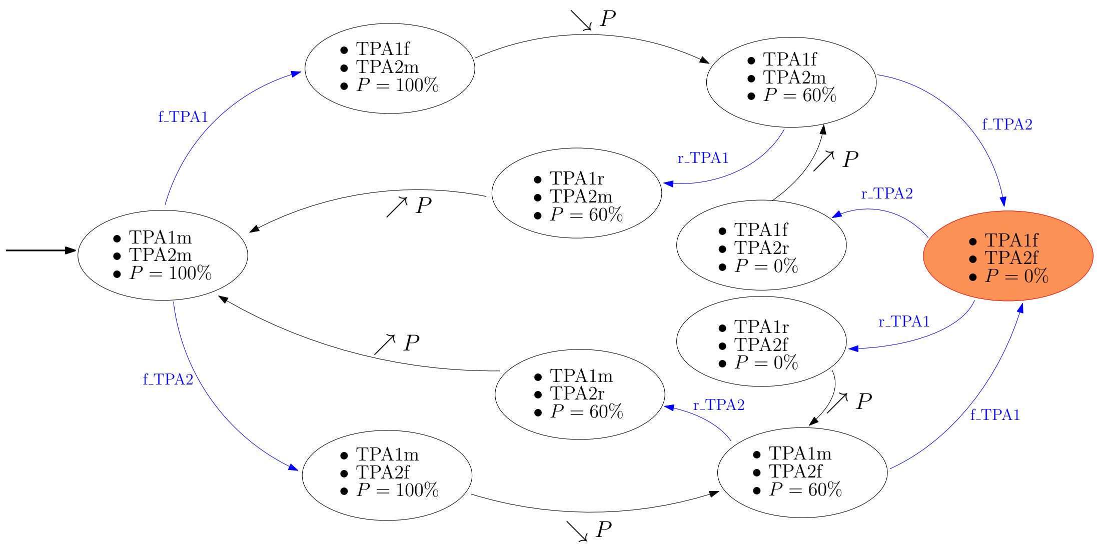


Figure 4.6: TPA's automata

An more detailed explanation of figure 4.6 is given below. The *lambda* and *mu* are respectively the failure rate λ and the reparation rate μ. In the small letters instead "a" means stop, "m" means working, "f" means failure and "r" means reparation. The model is illustrated only for one TPA, but it is identical for the other TPA.

Logical model

The logical model of the system is given in figure 4.7.



• TPA1m • TPA2m • $P = 100\%$
• TPA1f • TPA2m • $P = 100\%$
• TPA1f • TPA2m • $P = 60\%$
• TPA1r • TPA2m • $P = 60\%$
• TPA1f • TPA2r • $P = 0\%$
• TPA1f • TPA2f • $P = 0\%$
• TPA1r • TPA2f • $P = 0\%$
• TPA1m • TPA2r • $P = 60\%$
• TPA1m • TPA2f • $P = 100\%$
• TPA1m • TPA2f • $P = 60\%$
f_TPA1
f_TPA2
r_TPA1
r_TPA2
$\searrow P$
$\nearrow P$

Figure 4.7: Specification automata

The small letters have the same meanings as in figure 4.6. The black arc in figure 4.7 are timed arcs associated to failure and repair times. The blue arcs are instantaneous arcs, they are necessary to model the communication between the control-machine automaton and the TPA's physical model. In the given example the model accounts for such phenomena as power (P) change. Indeed when one TPA instead of two is operating, only a part of power can be delivered by the system (60%). The system's powers should thus be adjusted to the state of the global system. We can suppose that the sensor of a broken TPA is instantaneous, so when a TPA is broken, the control-machine automaton are immediately informed about it (blue arcs). In this work implementation system's power is not modeled. The power arcs (the black arcs) are drawn in figure only for a better comprension of power changing and for a future development.

Simplicity is prerequisite for reliability.

Edsger W. Dijkstra - Computer scientist

Chapter 5

CPN models for the case study

Introduction

In this chapter we describe two different models for our analysis. A stochastic model, used for a system's performance analysis, and a deterministic model used for the model verification process. Two models are required because as we will see later the tokens used to model the TPA failures do not have a finite set of values. This problem leads to the *state explosion*, with an infinite state graph, making the verification impossible. To assure the formal definition of our CPN Tools the automata illustrated in figures 4.7 and 4.7 are used to built two CPN Tools models. In the first section we talk about the stochastic version of our system, and its functioning failures and reparations of TPA. In the second section we talk about the deterministic version, which has the same behaviour as the first one but using a "referee".

CPN models: common structures

In this section we define the data and variables in common with the two models. Three used colors are illustrated in table 5.1 (to remind, a color is a type of token/place. Generally the color sets are infinite but are restricted by "*with* ... *and*" clause. For example a color INT includes all the integer numbers but with the *with* ... *and* we can limit it to the chosen values). Sixteen different variables were defined for the system. Some of these variables are used for the monitor and performance analysis and some are used for the model behaviour verification. In particular the *local* variables are used in a transition (exactly in the starting transition for the failure time, and in the failure transition for the reparation time), their values are calculated in different ways and afterward are assigned to the output token. Figure 5.1 illustrates the resulting CPN Tools declaration spot.

5.1 Stochastic approach

This section is composed of three parts, each describing a part of CPN Tools system's model. In the first section the physical model of a generic TPA is defined. In the second section the logical model of the behaviour of the system is defined, in particular this model doesn't include timed transitions. In the last, the third section the linker between the two modules is presented: 2 modules and a top module giving the initial conditions for the system's functioning. The used colours are given in Table 5.1.

Name	Type of data associated	Explanation	Timed
UNIT	()	Default CPN Tools color set	
INT	$0, 1, 2, \dots n$	Default CPN Tools color set, integer value	
BOOL	true, false	Default CPN Tools color set	
STRING	abcsd...	Default CPN Tools color set	
TPA	pump1, pump2	The two different TPA are modelize as token	●
TPAfault	TPA, INT, INT	This token carry on the value of TPA (pump1 or pump2) and the different time of failure in-turbine and out-of-turbine.	●
StatusTPAs	TPA, BOOL	This token is used to notify about the status of TPA the control machine	●
TPA_tPanne	TPA, INT	This token is used in to control-machine model to record what TPA is broken (or repaired) and at what time (INT value).	

Table 5.1: Color set used in the CPN Tools model

TPA model

We start to implement our system taking in consideration figure 4.4 where the System or TPA model is stochastic and represents the physical behaviour of our TPAs system. To build a CPN model of the physical behaviour of a TPA we based our studies on automaton given in figure 4.6. This automaton should be translated in a CP-net. The corresponding CP-net is given in figure 5.2. As we see figure 5.2 and figure 4.6 are more or less the same, but the analogies are only graphical. Indeed in the transition of the CPN model different operation are executed for a logical consistency.

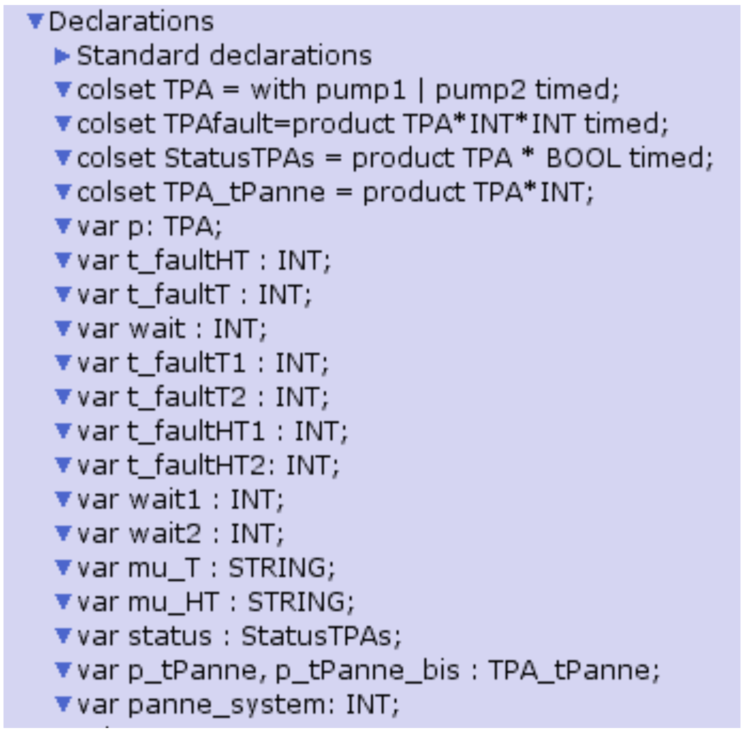
Let us describe the starting process.

Start up of a TPA

The process of starting a TPA is illustrated n figure 5.3. We can see that in the place *TPAs waiting* two token are presented, *pump1* and *pump2* at initial time (to remind the definition of marking in CPN Tools, definition 2.16). The variable *p* is involved to take on the random token (pump1 or pump2), from the place *TPAs waiting* to the transition *Starting TPAs*. The output of the transition is a complex timed token, with three different values. The actions performed in the transition *Starting TPAs* are randomly. These failure times are taken randomly from the exponential distribution (a correction of 0.5 is necessary due to the floor function that maps a real number in the smallest following integer). A series of *if* functions indicate which pumps we have and what is the nearest time to failure between in-turbine(T) and out-of-turbine(HT). In the end of this *if* a value of time @wait is associated to the token. The logic behind is the following : the pump works and after the time @wait a *failure* transition is enabled. The output token corresponds to the number of pump, the in-turbine failure time and the out-of-turbine failure time. These times are used to enable next transitions.

Name	Type of variables	Explanation	Local variables
p	TPA	Used to refer to token TPA	
t_faultHT	INT	Time of out-of-turbine failure	
t_faultT	INT	Time of in-turbine failure	
wait	INT	The minimal time between the two time defined above	
t_faultT1	INT	Time of in-turbine failure, TPA1	●
t_faultT2	INT	Time of in-turbine failure, TPA2	●
t_faultHT1	INT	Time of out-of-turbine failure, TPA1	●
t_faultHT2	INT	Time of out-of-turbine failure, TPA2	●
wait1	INT	The minimal time between the t_faultT1 and t_faultHT1	●
wait2	INT	The minimal time between the t_faultT2 and t_faultHT2	●
mu_T	STRING	In-turbine reparation rate, is a STRING because CPN Tools doesn't support the REAL value, but with the function *Real.fromString* is possible to use it	
mu_HT	STRING	Out-of-turbine reparation rate	
t_faultT	INT	Time of in-turbine failure	
status	StatusTPAs	Used for communication between the TPA model and the control-machine (specification) model	
p_tPanne	TPA_tPanne	Used in the control-machine model to know which pump is broken and at what time.	
p_tPanne_bis	TPA_tPanne	Like the variable p_tPanne	
panne_system	INT	Used to know at what time the system fails	

Table 5.2: Set of variables used in the CPN Tools model



```
▼Declarations
  ▶Standard declarations
  ▼colset TPA = with pump1 | pump2 timed;
  ▼colset TPAfault=product TPA*INT*INT timed;
  ▼colset StatusTPAs = product TPA * BOOL timed;
  ▼colset TPA_tPanne = product TPA*INT;
  ▼var p: TPA;
  ▼var t_faultHT : INT;
  ▼var t_faultT : INT;
  ▼var wait : INT;
  ▼var t_faultT1 : INT;
  ▼var t_faultT2 : INT;
  ▼var t_faultHT1 : INT;
  ▼var t_faultHT2: INT;
  ▼var wait1 : INT;
  ▼var wait2 : INT;
  ▼var mu_T : STRING;
  ▼var mu_HT : STRING;
  ▼var status : StatusTPAs;
  ▼var p_tPanne, p_tPanne_bis : TPA_tPanne;
  ▼var panne_system: INT;
```

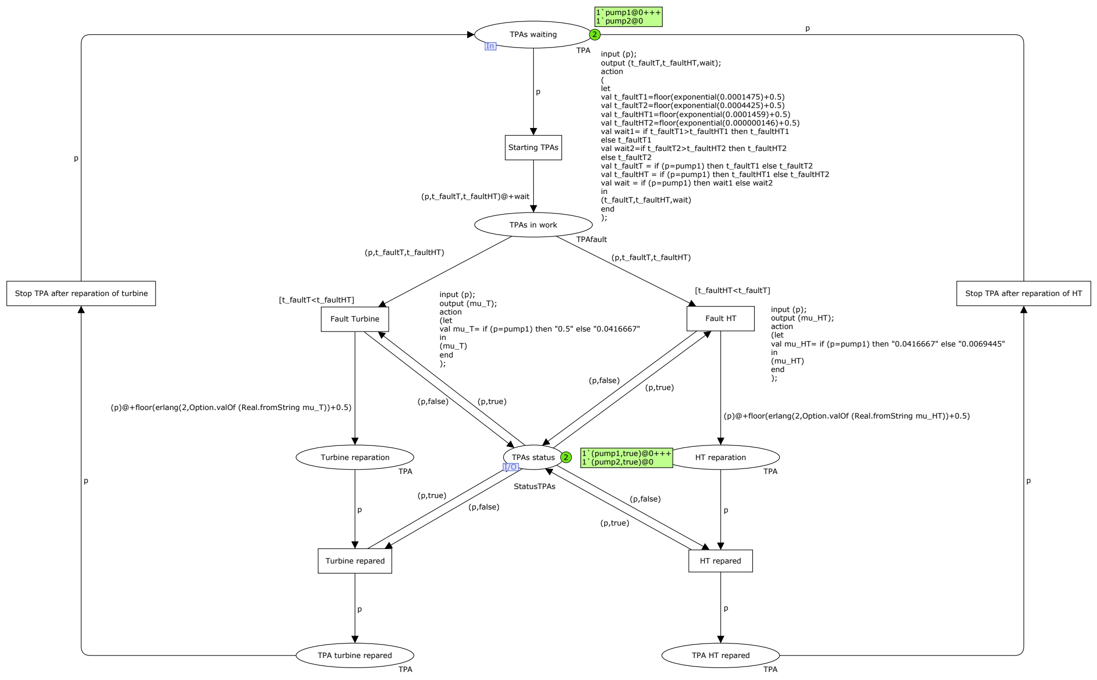
Figure 5.1: Declarations in CPN Tools for the case study

This procedure is the same for the first TPA and for the second TPA. The system can continue to function only after both TPA have started.

Failure of a TPA

Figure 5.5 illustrates the part that models the failure of a TPA. The two arcs from the place *TPAs in work* are linked to the two different failure modes. The simulation clock is at 171 (the time of the token pump2). It means that the token pump2 can be used to enable a transition. The choice of which transition is enabled is defined by the two guard conditions functions. The transition *Fault Turbine* is enabled if the random time of *in-turbine failure* time is smaller than the *out-of-turbine failure* time. Otherwise the transition *Fault HT* will be enabled.
The operations executed in the transitions have the same meaning as in the transition *Starting TPAs* (it means a token is assigned a "hold" value in the next place) but in this case only the reparation rate μ is assigned. An important place is the place *TPAs status*. This place is the link between the TPA model and the control-machine (specification) model. Its work is very simple, it has 2 tokens of type $StatusTPAs =$ TPA, BOOL. When the pump is ok the value of the token is true, when a pump is broken the boolean value is false. The fault transitions can be enabled only if all the incoming arcs are linked to places that contain a token which can satisfy the arc inscription. For example, the transition *Fault Turbine* will be enabled only if:

- The place *TPAs in work* has a token (p, t_faultT, t_faultHT),
- The place *TPAs status* has a token (p, true).

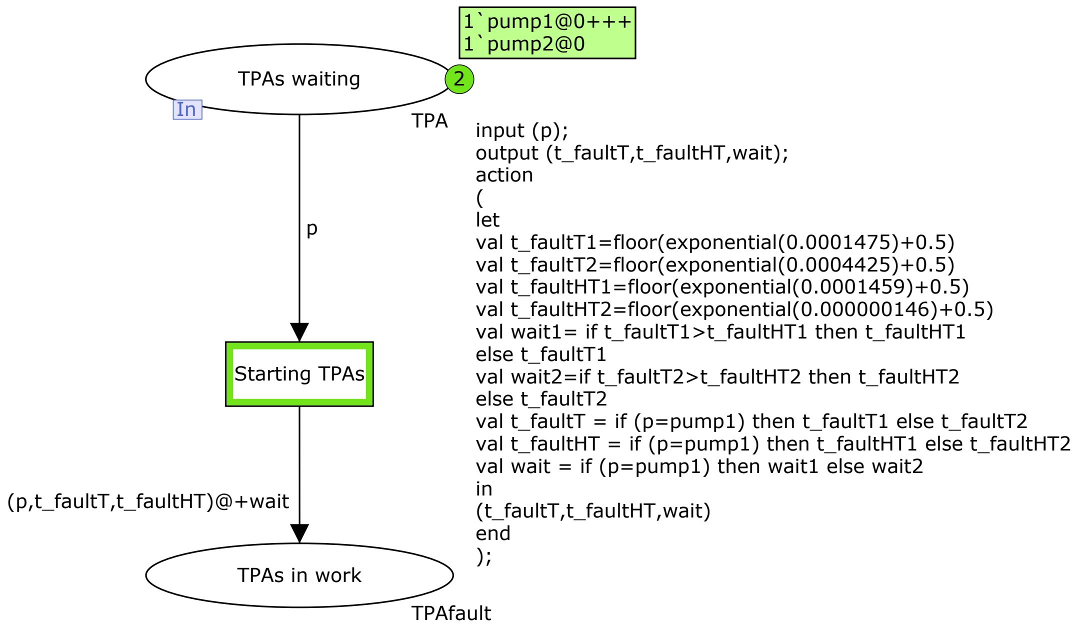


```
input (p);
output (t_faultT,t_faultHT,wait);
action
(
let
val t_faultT1=floor(exponential(0.0001475)+0.5)
val t_faultT2=floor(exponential(0.0004425)+0.5)
val t_faultHT1=floor(exponential(0.0001459)+0.5)
val t_faultHT2=floor(exponential(0.000000146)+0.5)
val wait1= if t_faultT1>t_faultHT1 then t_faultHT1
else t_faultT1
val wait2=if t_faultT2>t_faultHT2 then t_faultHT2
else t_faultT2
val t_faultT = if (p=pump1) then t_faultT1 else t_faultT2
val t_faultHT = if (p=pump1) then t_faultHT1 else t_faultHT2
val wait = if (p=pump1) then wait1 else wait2
in
(t_faultT,t_faultHT,wait)
end
);
```

```
input (p);
output (mu_T);
action
(let
val mu_T= if (p=pump1) then "0.5" else "0.0416667"
in
(mu_T)
end
);
```

```
input (p);
output (mu_HT);
action
(let
val mu_HT= if (p=pump1) then "0.0416667" else "0.0069445"
in
(mu_HT)
end
);
```

Figure 5.2: TPA, CPN model associated to the automaton from figure 4.6



```
input (p);
output (t_faultT,t_faultHT,wait);
action
(
let
val t_faultT1=floor(exponential(0.0001475)+0.5)
val t_faultT2=floor(exponential(0.0004425)+0.5)
val t_faultHT1=floor(exponential(0.0001459)+0.5)
val t_faultHT2=floor(exponential(0.000000146)+0.5)
val wait1= if t_faultT1>t_faultHT1 then t_faultHT1
else t_faultT1
val wait2=if t_faultT2>t_faultHT2 then t_faultHT2
else t_faultT2
val t_faultT = if (p=pump1) then t_faultT1 else t_faultT2
val t_faultHT = if (p=pump1) then t_faultHT1 else t_faultHT2
val wait = if (p=pump1) then wait1 else wait2
in
(t_faultT,t_faultHT,wait)
end
);
```

Figure 5.3: Starting of TPA: CPN model

where p has same value (in our system, pump1 or pump2). The logic behind is that a transition *Fault Turbine* will be enabled only if a pump is working. When a fault transition is fired a

```
input (p);
output (t_faultT,t_faultHT,wait);
action
        (
        let
        val t_faultT1=floor(exponential(0.0001475)+0.5)
        val t_faultT2=floor(exponential(0.0004425)+0.5)
        val t_faultHT1=floor(exponential(0.0001459)+0.5)
        val t_faultHT2=floor(exponential(0.000000146)+0.5)
                val wait1=
                        if t_faultT1>t_faultHT1 then t_faultHT1
                        else t_faultT1
                val wait2=
                        if t_faultT2>t_faultHT2 then t_faultHT2
                        else t_faultT2
                val t_faultT =
                        if (p=pump1) then t_faultT1
                        else t_faultT2
                val t_faultHT =
                        if (p=pump1) then t_faultHT1
                        else t_faultHT2
                val wait =
                        if (p=pump1) then wait1
                        else wait2
        in
                (t_faultT,t_faultHT,wait)
        end
        );
```

Figure 5.4: SML code of *Starting TPAs* transition

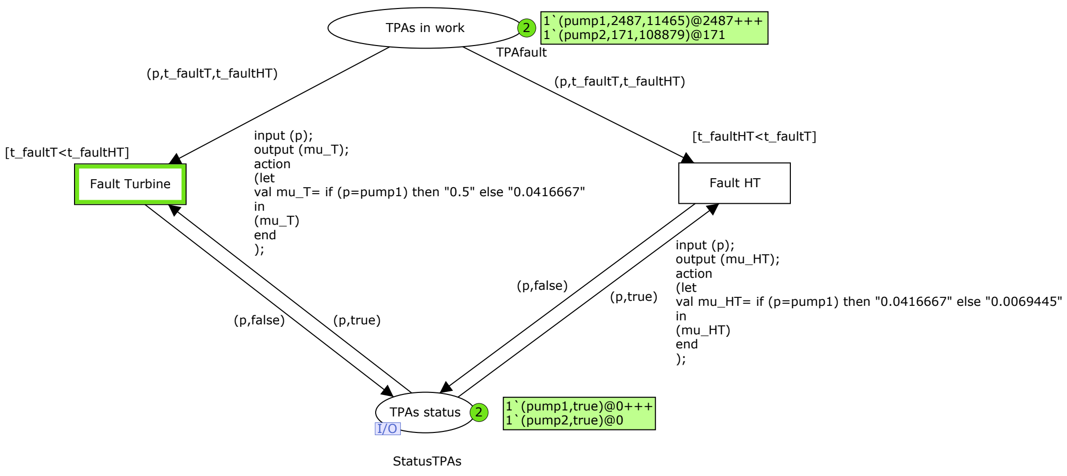
correspondent token (p, false) is placed in place of (p, true) token in *TPAs status*. This is used to notify the control-machine automaton that a pump is broken.

Reparation of a TPA

The model of the TPAs reparations is illustrated in figure 5.7.

Unlike in the *Starting TPAs* transition, in the reparation section we have defined the @wait time in the arc inscription. In this case the Erlang distribution function defined in CPN Tools requires two parameters: an integer (the shape parameter) and a real (the scale parameter). CPN Tools doesn't support real numbers, to overcome this problem we assign the different μ by the p value and the resulting *mu_ T* is defined as a string. The @wait time is assigned to the arc inscription using the function *Real.fromString*.

```
(p)@+floor(erlang(2,Option.valOf (Real.fromString mu_T))+0.5)
```



```
input (p);
output (mu_T);
action
(let
val mu_T= if (p=pump1) then "0.5" else "0.0416667"
in
(mu_T)
end
);
```

```
input (p);
output (mu_HT);
action
(let
val mu_HT= if (p=pump1) then "0.0416667" else "0.0069445"
in
(mu_HT)
end
);
```

Figure 5.5: Failure of TPA: CPN model

```
input (p);
output (mu_T);
action
          (
          let
                    val mu_T =
                    if (p=pump1) then "0.5"
                    else "0.0416667"
          in
          (mu_T)
          end
          );
```

Figure 5.6: SML code of *Fault Turbine* transition

The transitions *Turbine/HT repared* are enabled only if the selected pump is broken (p, false) and after the reparation time, calculated in the arc inscription. Afterwards the token (p, false) *TPAs status* is removed and a new token (p, true) is placed in the place.

Restart of a TPA

The restart of a TPA is illustrated in figure 5.8. In this case all the transitions are instantaneous. We suppose that after the reparation a TPA is hold and restarted immediately. The presence of the transitions *Stop TPA after reparation of turbine/HT* and the places *TPA turbine/HT repared* are not essential in a timed model but we will see in the deterministic version of this model the importance of these transitions/places (in section 5.2).

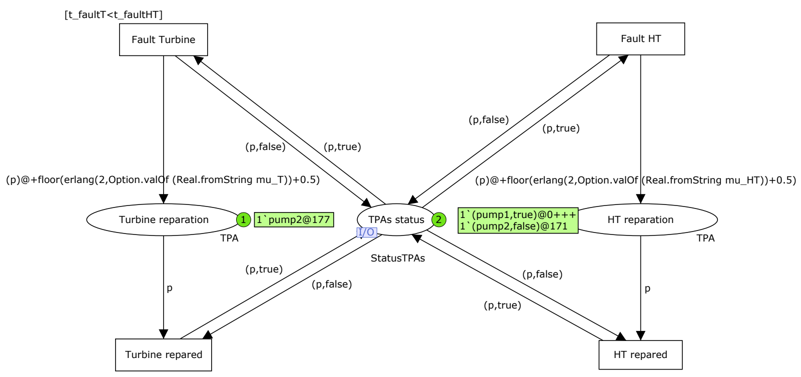


Figure 5.7: Reparation of TPA: CPN model

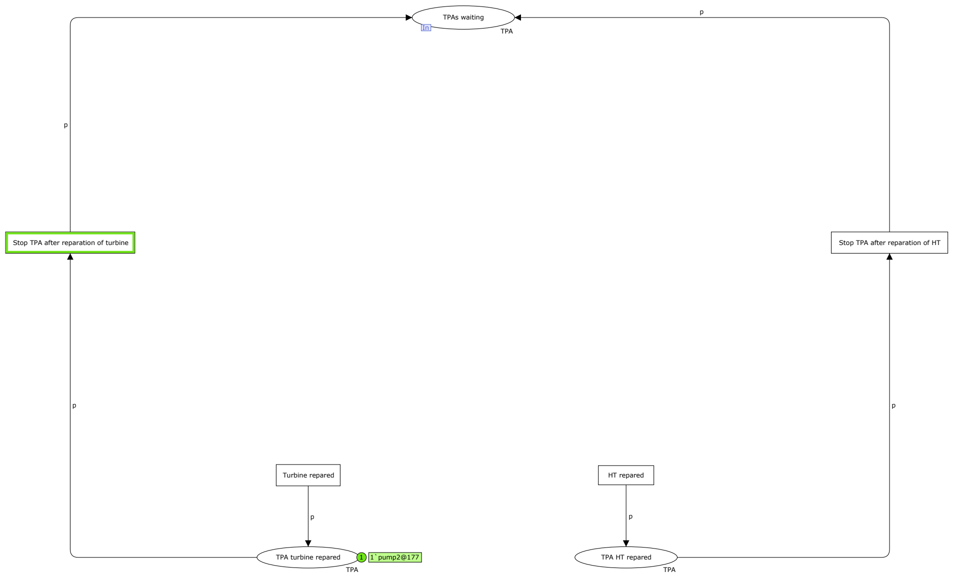


Figure 5.8: Restart of a TPA: CPN model

Control-machine model

The control-machine (logical) model is illustrated in figure 5.9. As illustrated in figure 4.4, the logical model should respect the following :

- The logical model is deterministic.
- The logical model should follow the provided specification.
- The logical model should allow to store all the information in order to carry out the performance analysis.

Figure 5.9 gives a CPN model interpretation of an the automaton in illustrated in figure 4.7. In

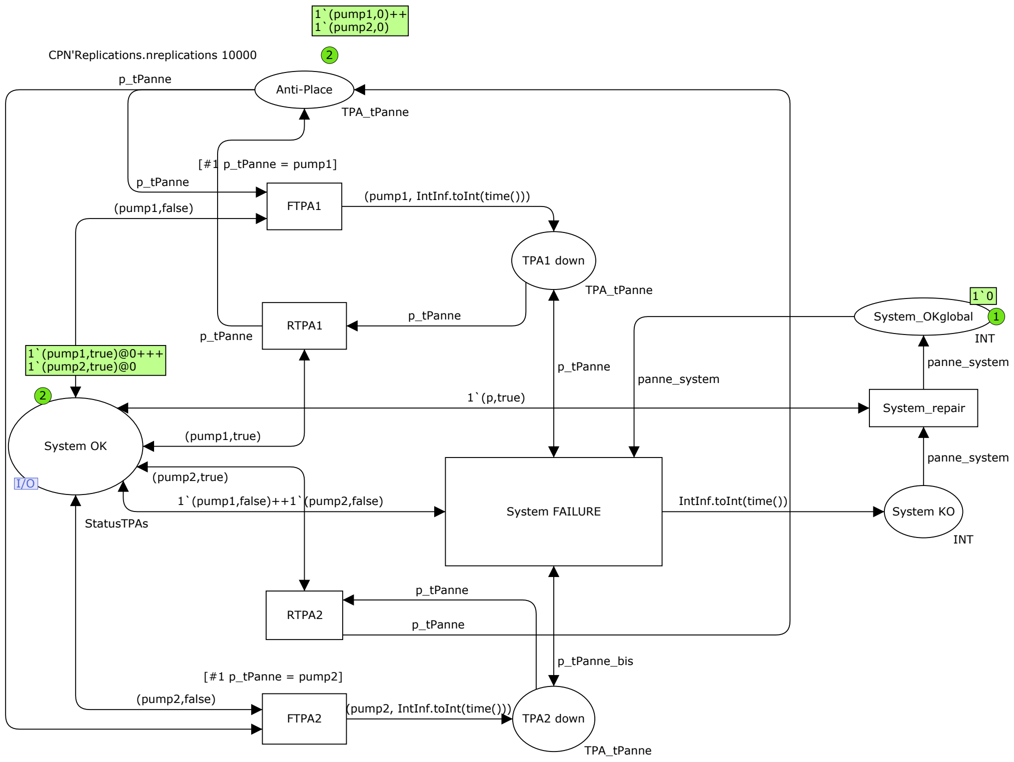


Figure 5.9: Control-machine CPN model

this logical model the place *System OK* is in "read only" mode, only the TPA model can change its contents. Let us describe different parts of this logical model.

Failure and reparation of a TPA - Logical model

The logical part involved in the control of the behaviour of the TPA system, in particular for the TPA1 is illustrated in figure 5.10, it is the same for the TPA2. It allows recording of all the parameters required to carry out the performance analysis. When the TPA1 (pump1) is broken the place *System OK* has the token (pump1, false) this allows the transition *FTPA1* that doesn't change the token in place *System OK*. An initial problem presented in this case is that when the instantaneous transition *FTPA1* is enabled, the marking of size *TPA1 down* is infinitely augmented. To resolve this an Anti-Place is implemented. CPN Tools doesn't support the inhibitor arcs, in the [12] online documentation of CPN Tools an easy trick is presented for the Anti-Place system.
The idea of the Anti-Place system is that the place looped in a two different transitions (in our case the *FTPA1* and *RTPA1*) cannot have more token that the token in the Anti-Place, because the Anti-Place enables or disables the input and the output transitions of the involved place. In this case thanks to the Anti-Place, the *FTPA1* is allowed only once, because the token *p_ tPanne* (where the first argument should be *pump1*, the second is the failure time of pump1) is removed from the Anti-Place. A next execution of the *FTPA1* transitions is allowed only if the token with pump1 is placed by the reparation transition *RTPA1*. Thus the place *TPA1 down* has at maximum one token.

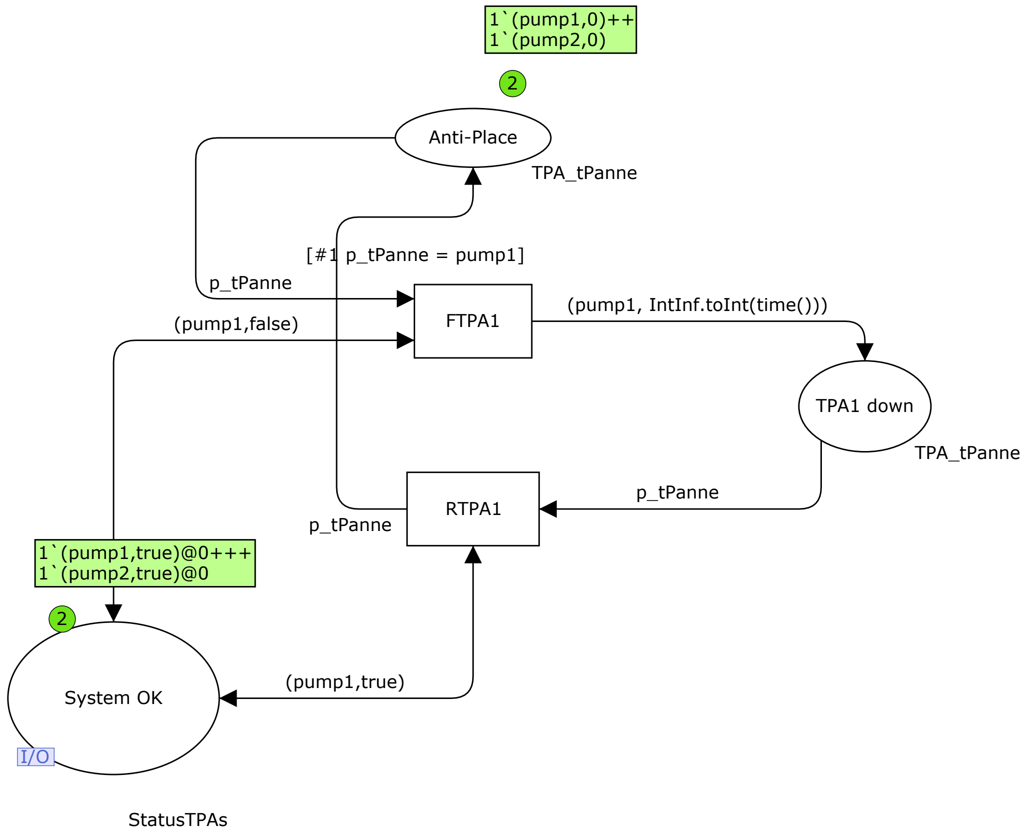


Figure 5.10: Failure and reparation of the TPA1: CPN model

The arcs inscriptions are used for the monitoring system in the performance analysis.

System failure - Logical model

From the automaton illustrated in figure 4.7 and the reliability block from diagram of figure 4.5 we know that a system is broken if the two TPA are broken. This means that the transition *System FAILURE* is enabled when the places *TPA1 down* and *TPA2 down* have the tokens that indicate the failure of the two TPA and the place *System OK* has the tokens (pump1, false) and (pump2, false). Afterward a token of type INT with the time of system's failure is placed in the place *System KO*.

Restart from a system failure - Logical model

The system is down when 2 TPA are down and restarts when one of the TPA is repared. In this case the transition *System_repair* is enabled only if there is a token in the place *System KO* and a pump is repaired. It is an instantaneous transition. In this case we can have a concurrency between the transitions *System_repair* and *RTPA1/2* but it does not entail implementation problem behaviour of the system. Indeed CPN Tools executes the instantaneous transitions before the timed transitions and the instantaneous transitions in the logic model cannot change the place *System OK*. Thanks to the arc with inscription (see figure 5.11) :

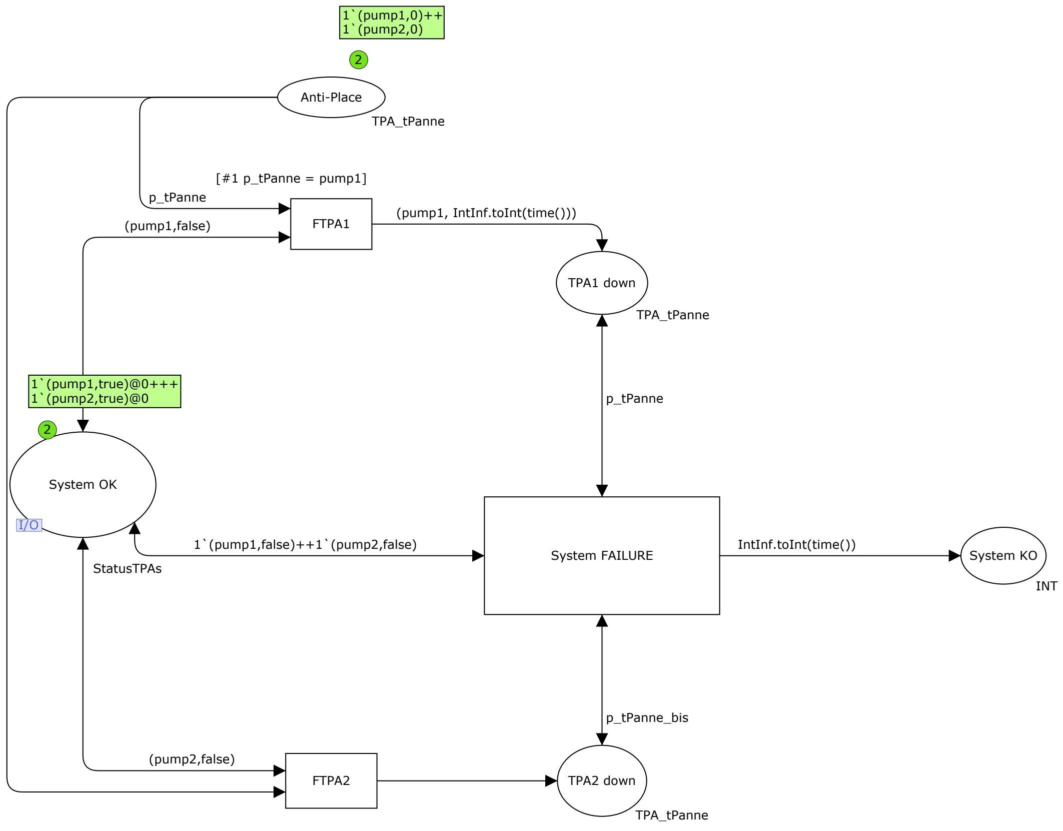


Figure 5.11: System failure in the logical model

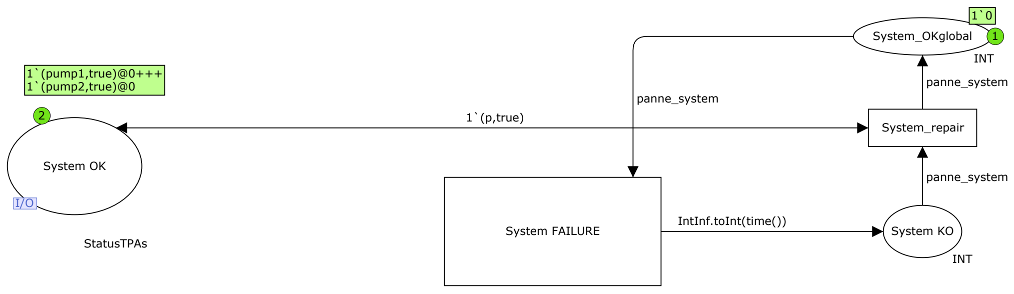


Figure 5.12: Restarting from a system failure state

```
1‘(pump1, false)++1‘(pump2, false)
```

the transition *System FAILURE* cannot be enabled if a pump is repared, neither if we have some token in the places *TPA(1,2) down*. This is another certainty of the correct behaviour of the system. The involved part of the reparation after a system failure is illustrated in figure 5.12. Without the places and the transitions of the right side of figure the system cannot be repaired after a system failure.

System's hierarchisation - Top module

After building the two modules of our system it is necessary to develop a Top module with the initial conditions and the shared place (a shared place by the 2 modules. It is named *Fusion places* in CPN Tools).

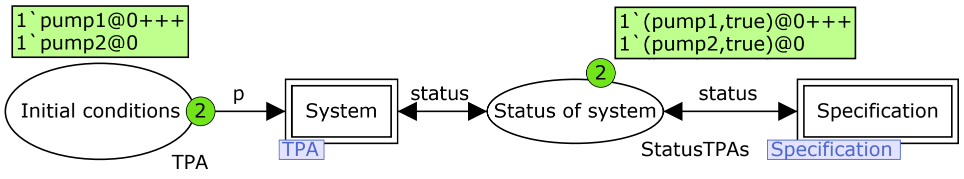


Figure 5.13: Top module of the case study system

The place *Initial conditions* has the initial marking of the TPA module, in this place we can set how TPA are functioning in our system. For example we can easily make simulation with two TPAs of type *pump1* and four TPAs of type *pump2*. The place *Status of system* has an initial marking of pump1 and pump2 in a true value, similarly if we increase the number of pumps in the place *Initial conditions* we should augment also the tokens in this place. The transitions showen in the figure 5.13 are simply the *TPA model* for *System* and *Logical model* for *Specification*. In this case the transitions are actually the *substitution transitions*.

5.2 Deterministic models

A determinization of a timed model is simple. Substitution all the timed transitions with instantaneous transitions is the first step (see figure 5.14 for the deterministic vrsion of the TPA CPN model).

However time in our case is not simply a variable. Time gives an order to a series of events. Without time, all the two module, *TPA* and *Specification* are deterministic and in this case some transition are not forbidden by the time. Thus making if possible that the system reaches irrelevant states (see for example figure 5.15).

In Figure 5.15 we can see that pump2 can be broken and repaired while the pump1 is still in waiting state. This is not possible in our stochastic system. In theory, with this problem the pump2 can be started and broken infinitely and the pump1 can be always in a waiting status. The solution is in limiting the state space, but how? CPN Tools supports the option for priority the transitions. However for a analysis tool which we use for perform the verification analysis (its name is ASAP [24]) the priority transitions are not supported. Thus, our idea is to use a place which enables a module or the other with a special token. Like a classical mutual exclusion problem, it is like two trains in a bridge with only one railway ; somehow it should be allowed that one train at time passes the bridge, in the meanwhile the other waits in a station near the bridge. We have applied the same method to our deterministic model, but in our case the order is not simply "go on the bridge" or not. An additional complexity is present in ordering the 2 TPAs. This complexity is approached using the "referee" place.

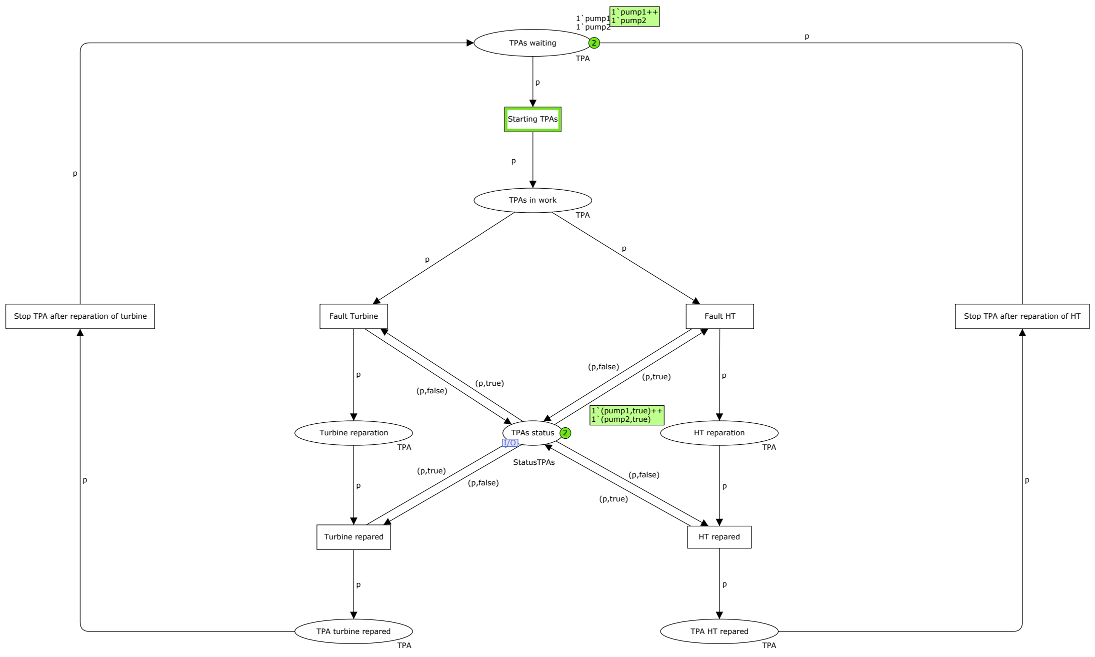


Figure 5.14: The TPA model without timed transitions

The "referee" place

We can divide the principal problem of a "referee" in four little problems. Firstly we can consider the TPA *failure* problem, secondly the TPA *reparation*, thirdly the TPA *restart* problem, at last the *system restart* problem. To resolve these problems a series of *flags* is defined, different flag for each problem. These flags are tokens with a STRING color set, which allows a compact representation of those. The initial value of the place *TOKEN* is as follows (see figure 5.16 for its representation) :

```
2 ‘ "OKTPA"++
1 ‘ "SYSOK"++
1 ‘ "TPA"
```

The hearts of the "referee" place is a special token we named as *golden token*. This token can have three values: *TPA*, *SPEC* or *REP_ TPA*. The particularity of the golden token is that is used to switch between the TPA model or the Specification model. The place *TOKEN* can have only once at time.

Failure problem

In case of failure of a TPA this failure can be allowed only when the 2 TPA are in a working status. For that a token *1‘"TPAS"* is placed in *TOKEN* when the transition *Starting TPAs* occurs. To enable the failure transitions it is required that two tokens *TPAS* are placed in the place *TOKEN* (see figure 5.17). If a failure transition is fired the golden token of type *TPA* is substituted with the value *SPEC* and the TPA model remains in this status. Only when the Specification model gives the golden token to the TPA, the TPA can perform others transitions.

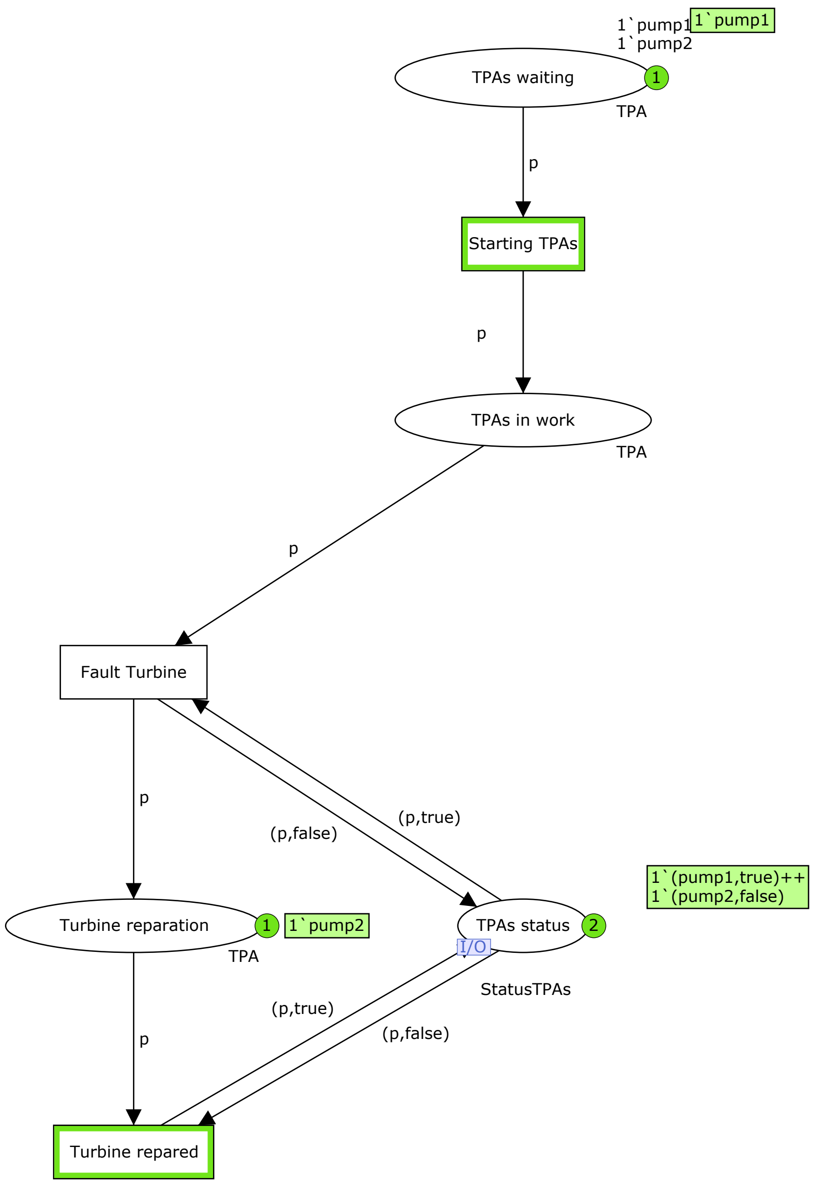


Figure 5.15: An error in the first version of the deterministic system

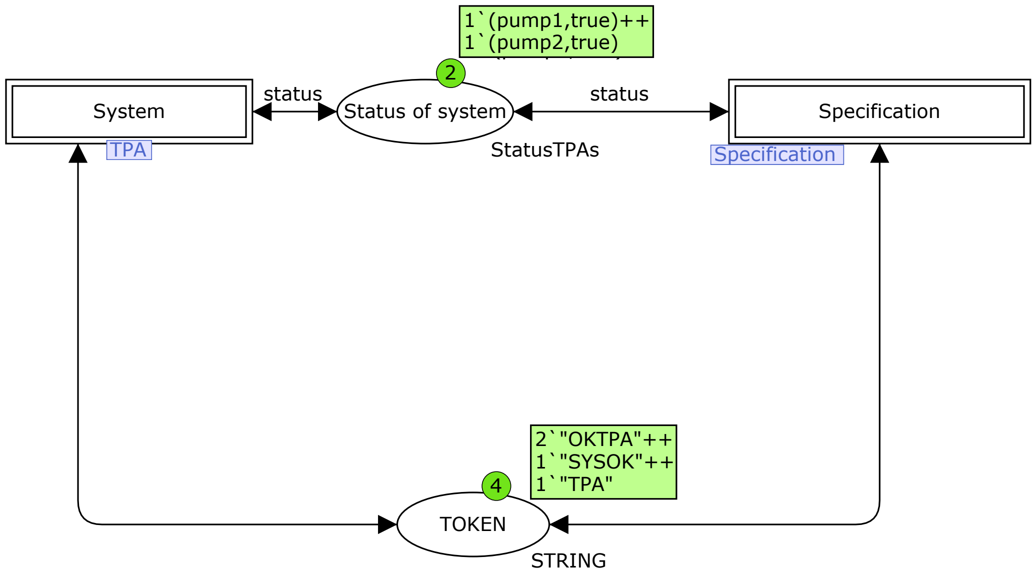


Figure 5.16: The "referee" place named *TOKEN* in CPN Tools

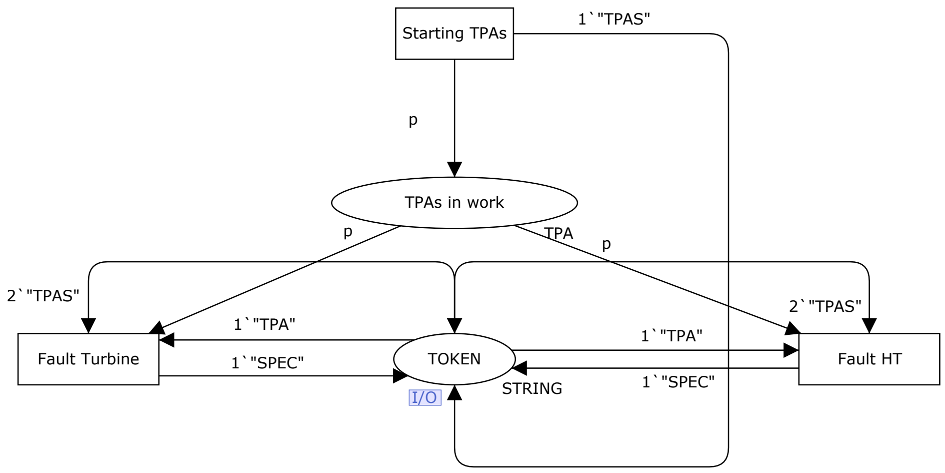


Figure 5.17: The *failure* problem, TPA model

The specification model executes the only transition that can be fired, for example if the pump1 is going to be broken, the specification fires the transition *FTPA1*, gives back the golden token to the TPA model and takes the token *OKTPA* (see figure 5.18). This indicates a broken TPA. After that another failure or reparation transition can be fired.

Reparation problem

Suppose that after a first failure we have a reparation, in this case only the reparation transition should be enabled. Thus a combination of tokens is removed.

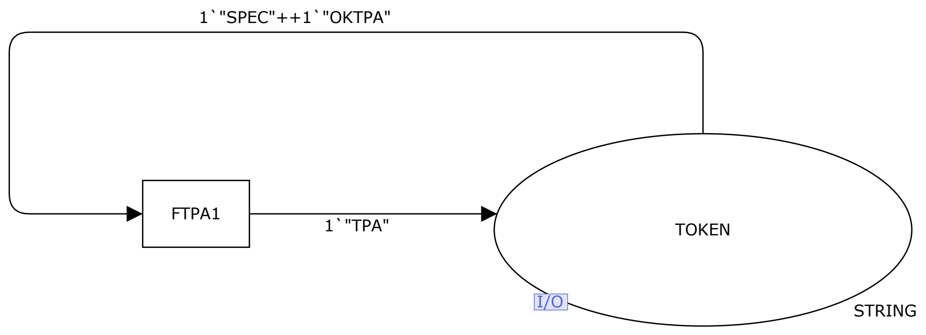


Figure 5.18: The *failure* problem, specification model

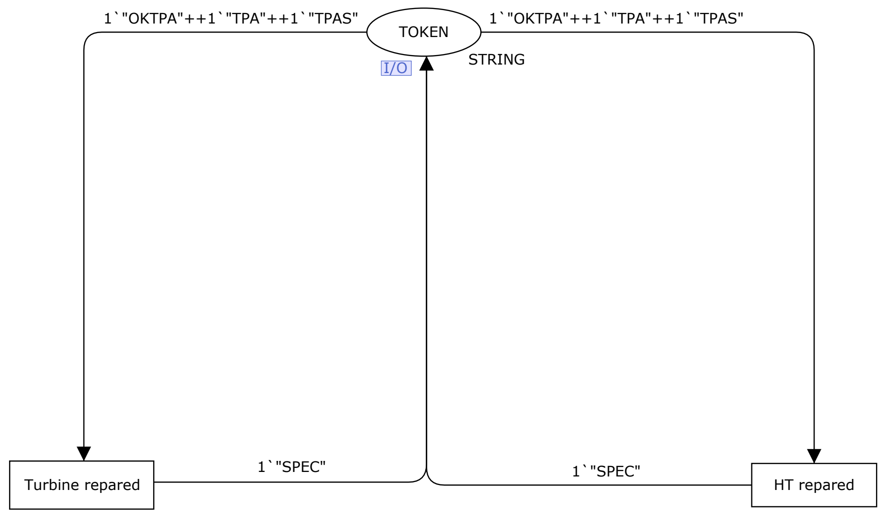


Figure 5.19: The *reparation* problem, TPA model

In figure 5.19 we can see that the tokens removed when a *reparation transition* is fired are the following :

```
1‘"OKTPA"++1‘"TPA"++1‘"TPAS"
```

These token are removed for the following reasons :

- *TPA* token stops the TPA model
- *TPAS* token interrupts the failure transitions (the failure transitions requires 2 tokens *TPAS*)
- *OKTPA* token is removed because this makes it possible that only a reparation transitions in the specification model can be enabled.

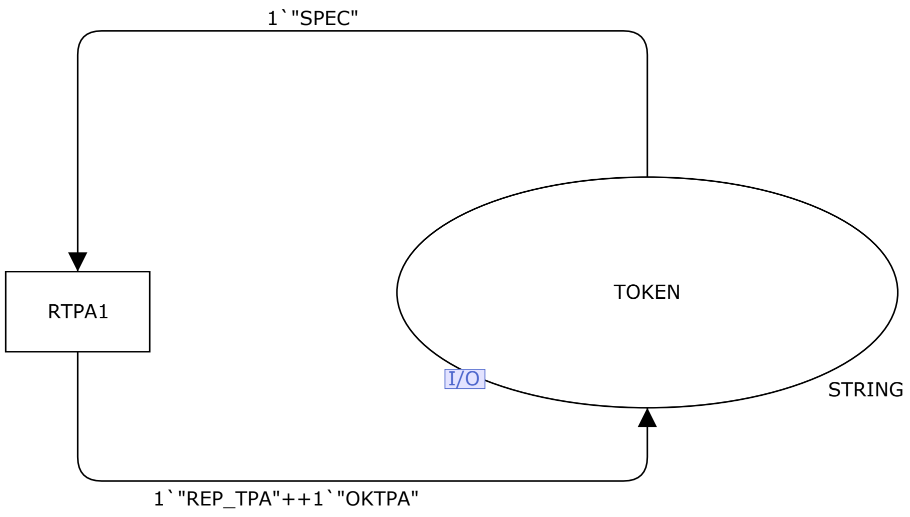


Figure 5.20: The *reparation* problem, specification model

In the specification model given in figure 5.20 we can see the token *REP_ TPA*, which forces the restart of the pump, and the token *OKTPA*, which indicates the reparation of the pump.

Restart problem

For the restart problem (see figure 5.21) it is checked if the system is not in a failure state using the *SYSOK* token. The token *REP_ TPA* is removed and tokens *OKTPA* and *TPA* are placed in the *TOKEN* place. Thanks to this the next transitions is *Starting TPAs*, which brings back the system at the time when both pumps are in working state.

System failure problem

In this case the two pumps are broken. When firing the transition *System FAILURE* the specification model should remove the token *SYSOK* from the "referee" and to remove a token *TPAS* to forbid the execution of all failure transitions, because at this moment only a reparation should be enabled thanks to the token *OKTPA* (see figure 5.22 for the CPN implementation). If a pump is repaired the transition *System_ repair* in the specification model is fired. As a result the pump restarted.

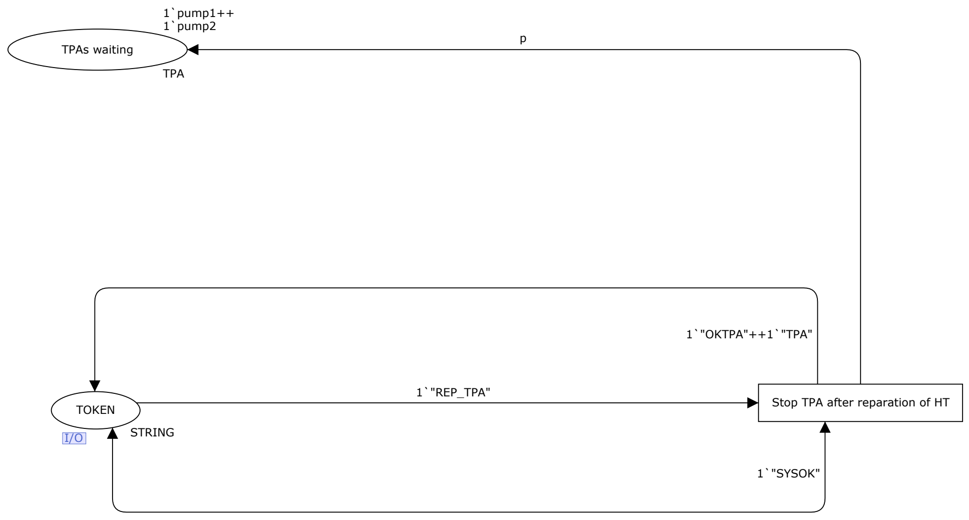


Figure 5.21: The *restart* problem, TPA model

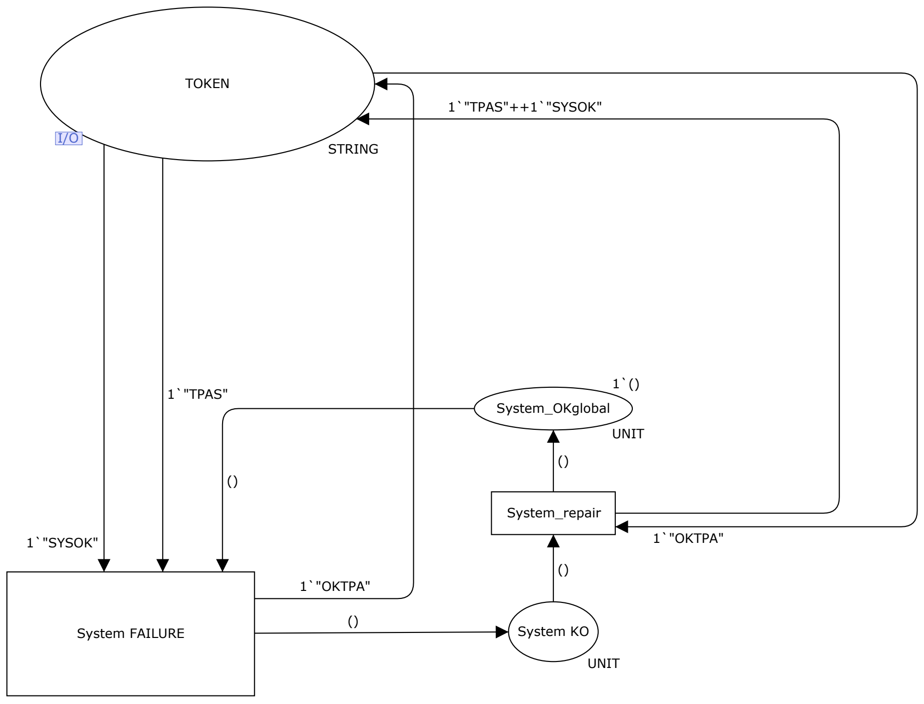


Figure 5.22: The *system failure* problem, specification model

Thanks to this we can have a working pump going to a failure status while the other pump is in reparation. This is also possible for our stochastic system, although the probability of such an event is not very high because in average the repair durations are significantly smaller than the failure durations. The complete deterministic versions of the TPA system and the specification system are given in figures 5.23 and 5.24 respectively.

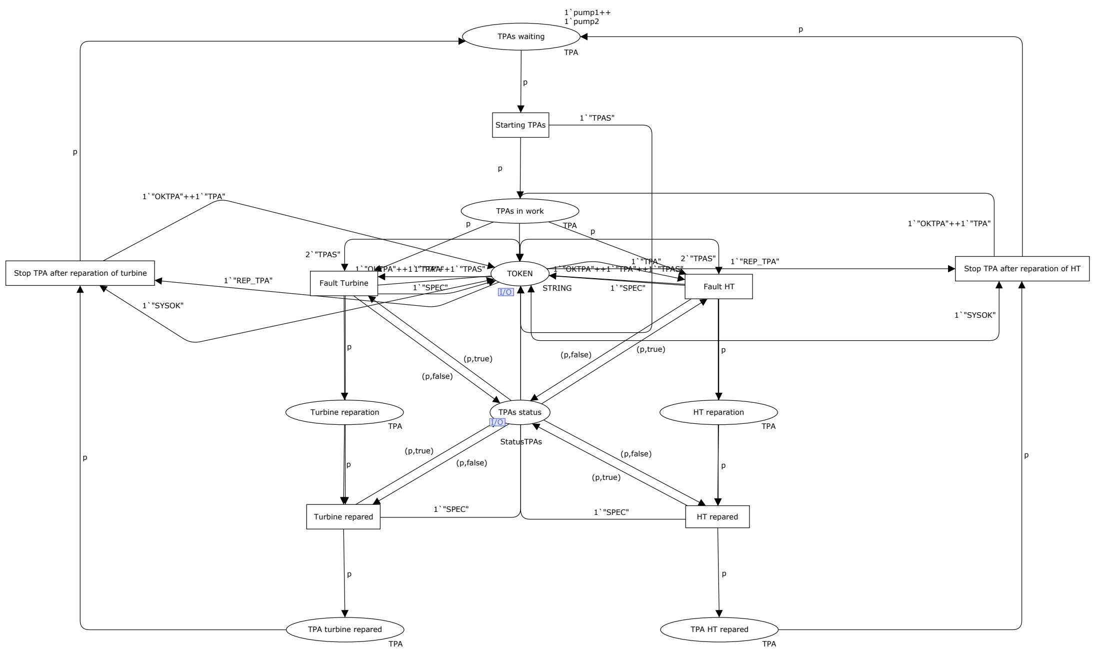


Figure 5.23: The deterministic version of the TPA model

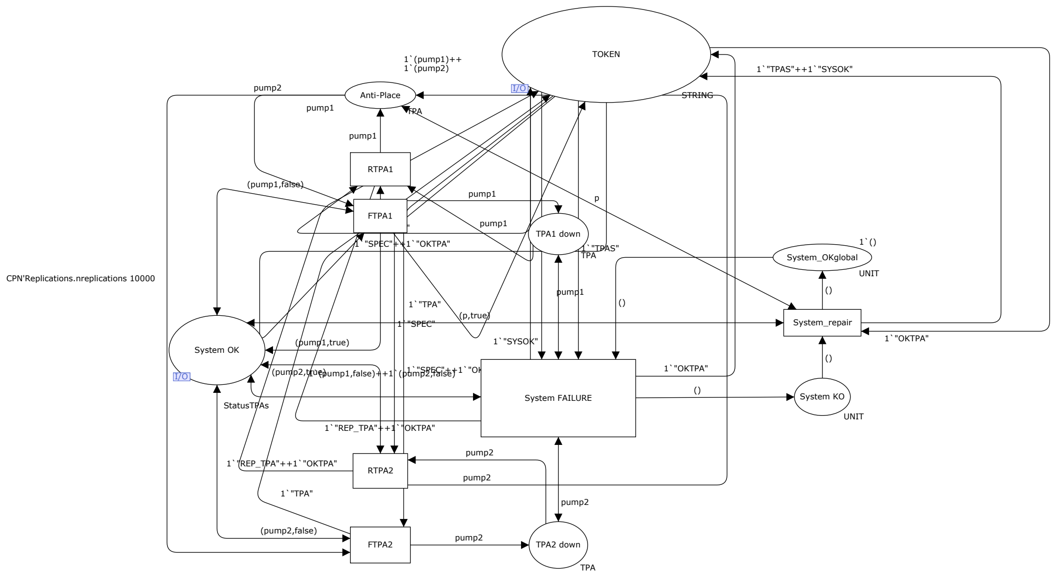


Figure 5.24: The deterministic version of the specification model

A careful analysis of the process of observation in atomic physics has shown that the subatomic particles have no meaning as isolated entities, but can only be understood as interconnections between the preparation of an experiment and the subsequent measurement.

Erwin Schrödinger - Physicist

Chapter 6

Model usage

Introduction

In this chapter the two types of model usage and their implementation will be discussed. We will start with the system performance analysis and present the the verification analysis afterwards. The outline of this chapter is illustrated schematically in figure 6.1.

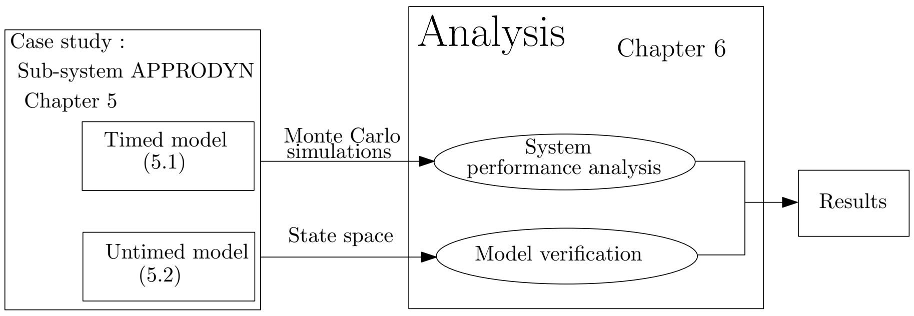


Figure 6.1: Chapter outline

6.1 System performance analysis

CPN Tools is adapted for a performance analysis. In the online documentation [12] there is an explanation of how to carry out a performance analysis using the monitors. For the performance analysis a Monte Carlo simulations have been done. The monitors record the data for one simulation, after that the CPN Tool engine makes a series of basic evaluations of these data (means, confidence intervals, etc). We should record the time when a particular transition occurs, afterwards CPN Tools gives us the mean over all the simulations. A synthesis of parameters provided by the performance analysis is given in figure 6.2.

Monitors in performance analysis

An explanation of CP-nets monitor is given in definition 2.17. In our case four monitors are applied to record three different mean times: the *MTTFF*, the *MTTR*, the *MTBF* and

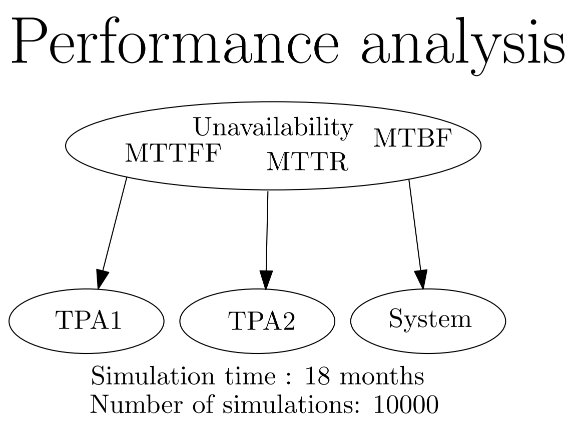


Figure 6.2: A synthesis of the performance analysis parameters.

the last to record the *Unavailability* (the unavailability is the complementary probability of the availability). These monitors are replicated for the two TPAs and for the system. We will explain these monitors for the TPA1, same code with only slightly different variables is used for TPA2 and the system.

Composition of a monitor in CPN Tools

According to the online documentation [12], *Each monitor has a number of different functions that are used for different purposes. Some examples of functions for monitors are:*

- *predicate function* for checking if the relevant condition is fulfilled,
- *observation function* for extracting information from the net,
- *action function* for doing something relevant with the extracted data,
- *initialization function* for initializing a monitor before a simulation starts,
- *stop function* for concluding a monitor when a simulation ends.

Some monitors will have one of each of these functions, and some monitors will only have a subset of these functions. For example for our aim we use a *Generic data collection monitor* which allow to execute a particular observation function written by us if the predicate function is verified. With the *Generic data collection monitor* we cannot modified the *action function* which is hidden. These functions are explained with the explanation of the different monitors. A detailed explanation of the monitors is given below.

MTTFF monitor

The MTTFF monitors should record only the time of the first entity failure. They are applied to the *failure transitions* corresponding to the entity (for the TPAx it is *FTPAx*, for the system it is *System FAILURE*). It is composed of the following functions:

- Predicate function:

```
fun pred (bindelem) =
let
  fun predBindElem
  (Specification 'FTPA1 (1,{p_tPanne})) = if #2 p_tPanne = 0
                                           then true else false
    | predBindElem _ = false
in
  predBindElem bindelem
end
```

In this case the second element of *p_tPanne* is the time to failure, this time is recorded only after the *failure* transition. For the initial failure it is always 0.

- Observer function: if the *predicate* functions is true then the observer function is executed.

```
fun obs (bindelem) =
let
  fun obsBindElem
  (Specification 'FTPA1 (1, {p_tPanne})) =
                            IntInf.toInt(time())
    | obsBindElem _ = ~1
in
  obsBindElem bindelem
end
```

In this case we extract the value of the global clock using the function *IntInf.toInt(time())*.

- Initialization function: we should not initialize the monitors by construction.

```
fun init () = NONE
```

- Stop function:

```
fun stop () = NONE
```

In this case the monitor stops automatically when the simulations ends, so no stop function is defined.

MTTR monitor

The MTTR monitors should record every reparation time. When a *reparation* transition is enabled a difference between the failure time and the transition time should be calculated. We use the failure time to calculate the difference because we suppose that the reparation is immediately performed after failure. It is applied at *RTPAx*, for the *TPAx* at *System_repair* for the system. It is composed of the following functions:

- Predicate function :

```
fun pred (bindelem) =
let
  fun predBindElem (Specification 'RTPA1 (1, {p_tPanne}))
                 = true
      | predBindElem _ = false
in
  predBindElem bindelem
end
```

When the transition *RTPA1* is enabled the monitor is activated.

- Observer function : if the *predicate* functions is true then the observer function is executed.

```
fun obs (bindelem) =
let
  fun obsBindElem
  (Specification 'RTPA1 (1, {p_tPanne})) =
        IntInf.toInt(time()) - #2 p_tPanne
      | obsBindElem _ = ~1
in
  obsBindElem bindelem
end
```

In this case we extract the value of the global clock using the function *IntInf.toInt(time())*.

- Initialization : like for MTTFF monitors, we should not initialize the monitors.
- Stop function : like for the MTTFF monitors, no stop function is defined.

MTBF monitor

The MTBF monitors should record every failure time. They are similar to MTTR monitors with the only difference in the *observer* function. When a *failure* transition is enabled the difference between the time of the last failure and the new failure should be calculated. It is applied at *FTPAx* for the TPAx, at *System FAILURE* for the system. It is composed of the following functions:

- Predicate function: the same as for the MTTR, modified to work with the FTPA transitions.
- Observer function: if the *predicate* functions is true then the observer function is executed.

```
fun obs (bindelem) =
let
  fun obsBindElem
  (Specification 'FTPA1 (1, {p_tPanne})) =
         IntInf.toInt(time()) - #2 p_tPanne
      | obsBindElem _ = ~1
in
  obsBindElem bindelem
end
```

It is the same as MTTR monitors and is adapted to the *failure* transition, FTPA1.

- Initialization : equal to MTTR monitors.
- Stop function : equal to MTTR monitors.

Unavailability monitor

The *Unavailability* monitors should record the duration of state when the entity is broken. In our case we can count how much time the token spends in the *broken* place (for the TPAx, it is the place *TPAx down*). In our system we can have at maximum one token at time in the *broken* places, then the probability is very simple to extract. The monitor is a *Marking size* type monitor which has all the functions hidden, we can only count the number of tokens for simulation and of course perform a Monte Carlo simulations.

The stop simulation monitor

The system is built to perform simulations infinitely. There are no dead markings which stop the simulations. A *breakpoint* monitor is defined to halt simulations. In *breakpoint* monitor we can only define the *predicate* function. In our case we stop the simulations at 18 months, the system time is expressed in hours thus 18 months correspond more or less to 13152 hours. The simulation clock is advanced only when a transition is enabled thus it is not possible to stop the simulation at an exact time. The stop error is in general only in a few days after 18 months which is an acceptable error.

- Predicate function :

```
fun pred () =
  IntInf.toInt(time()) > 13152
```

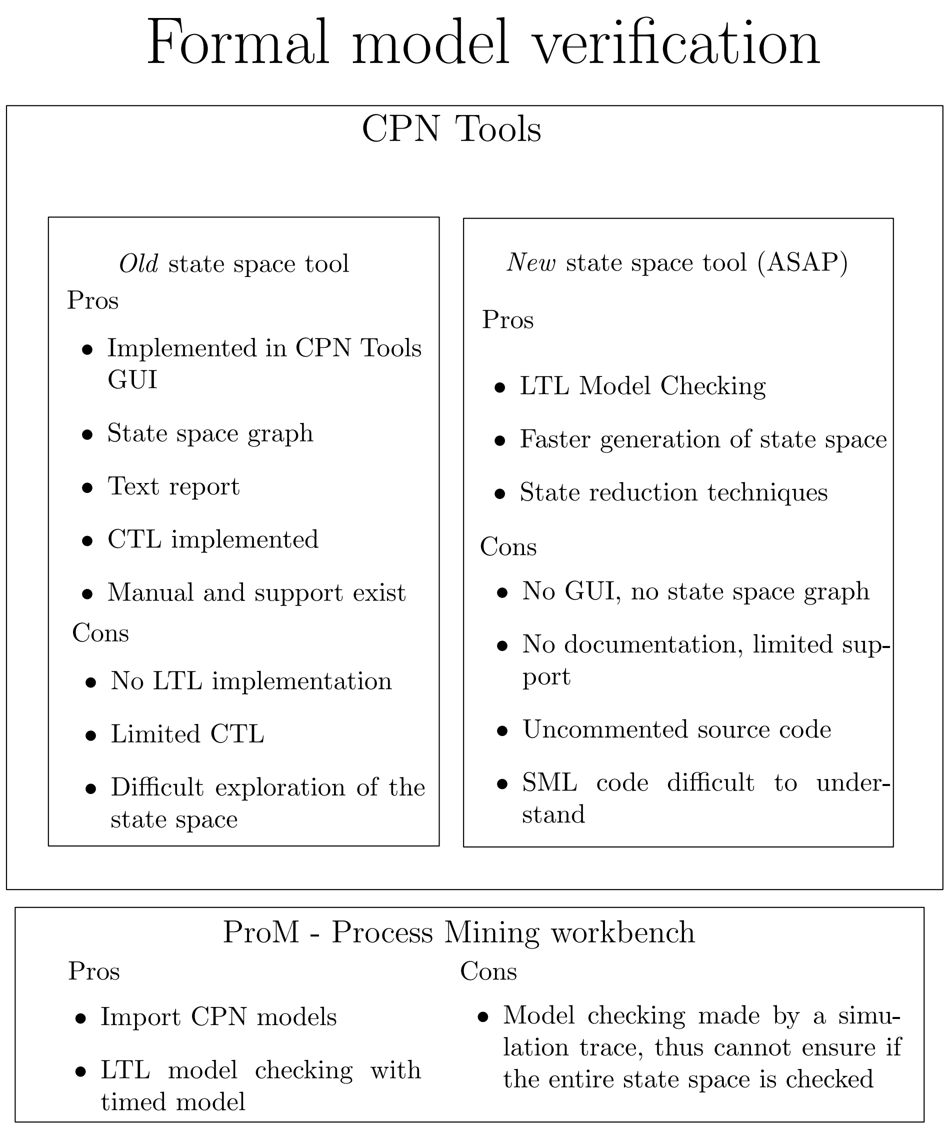
Note that it is easy to make our system not repairable using the monitors. For that if we place a monitor of type *Place content breakpoint* in to the place *System KO* and the simulations stop when a token is in the *System KO* place.

6.2 Formal model verification

CPN Tools possesses two tools to perform a state space analysis, the first is the classical *state space generator* included in CPN Tools, with a manual and explained in the online documentation. The other is named ASAP ASCoVeCo State Space Analysis Platform) presented in [24]. A version of ASAP with a GUI (graphical user interface) was developed but is no longer supported. Since version 3.0 of CPN Tools, the state space tool of ASAP has implicitly been part of CPN Tools. ASAP is not exposed in the GUI, however, and lacks support for some features of CPN Tools. Some advantages and disadvantages of the two state space tools are explained in figure 6.3. Let us start by detailing the first state space tool.

Old state space tool

In fact the prefix *old* is improper, since this tool is not old. It is the state space tool officially implemented in CPN Tools. It is stable, and it allows to verify some basic proprieties like the *home/dead makings*. In this case a text report with different informations is saved. These informations are:



Formal model verification

CPN Tools

Old state space tool

Pros

- Implemented in CPN Tools GUI
- State space graph
- Text report
- CTL implemented
- Manual and support exist

Cons

- No LTL implementation
- Limited CTL
- Difficult exploration of the state space

New state space tool (ASAP)

Pros

- LTL Model Checking
- Faster generation of state space
- State reduction techniques

Cons

- No GUI, no state space graph
- No documentation, limited support
- Uncommented source code
- SML code difficult to understand

ProM - Process Mining workbench

Pros

- Import CPN models
- LTL model checking with timed model

Cons

- Model checking made by a simulation trace, thus cannot ensure if the entire state space is checked

Figure 6.3: Summary of the two state space generators

- *Statistics* of the state space and the strongly connected components graph (number of arcs, nodes and if the state space is fully calculated or not).
- *Boundedness Properties* with various bound, like the maximum or the minimum marking of a place.
- *Home Properties* if there are some *home marking*.
- *Liveness Properties* including *Dead Markings*, *Dead Transition Instances*, *Live Transition Instances*.

- *Fairness Properties* to check if all the paths are "fair" in the sense that if the automaton enters in a state infinitely often, it must take every (or not, depends of the properties) possible transition from that state.

The state space is easy to draw with the CPN tools GUI or with another software Graphviz as is illustrated in figures 2.10 and 2.9. Our problem being large and complex, we cannot draw the entire state space here. Indeed our state space contains 100 arcs and 72 nodes, and thus is too big for an A4 layout, but is calculated and drawn with Graphviz.

State space exploration and CTL model checking

The state space exploration can be performed following the online documentation of CPN Tools. In this case however we should specify the state of the state space we are interested. To perform this analysis we respond to four question:

1. How many states and what are the state where the TPA_1 is broken?
2. What is the minimum path from a broken TPA_1 to a state where the entire system is in failure?
3. Is it always true that the TPA_1 is restarted immediately after its reparation ?
4. For all the existing paths, is it possible to arrive to a system failure state?

The first two question are answered with the informations contained in the state space tools manual, presented in the online documentation of CPN Tools. The last two are briefly presented with a version of CTL implemented but no longer supported in CPN Tools. The next section explain the new state space tool, ASAP.

ASAP state space tool

The largest part of this work was to understand and use ASAP to perform an LTL model checking analysis in our CPN model. Unfortunately ASAP has no documentation and an only limited support obtained from exchanges with a developer of this tool, watching his Youtube videos or trying to understand the SML code. A version of ASAP with a GUI was developed but the developer suggests us to ignore the GUI version and use the version implemented in CPN Tools. With this version we have tried to use a toy model to understand how ASAP works and afterwards we have used the same methodology for our case study system. The schema representing the steps of model verification using CPN Tools is given in figure 6.4.

Toy model, an LTL analysis

This part is divided into different paragraphs, every paragraphs starts with a letter representing the various steps from figure 6.4. The toy model used for this test is illustrated in figure 6.5.

As we have defined in chapter 2, the *coverability* graph of the toy model is given in figure 6.6.

a. We have three different markings, these markings have been considered as states of the Kripke structure (figure 6.7).

The toy model in CPN Tools is presented in Figure 6.8. We have used a UNIT color set for this model.

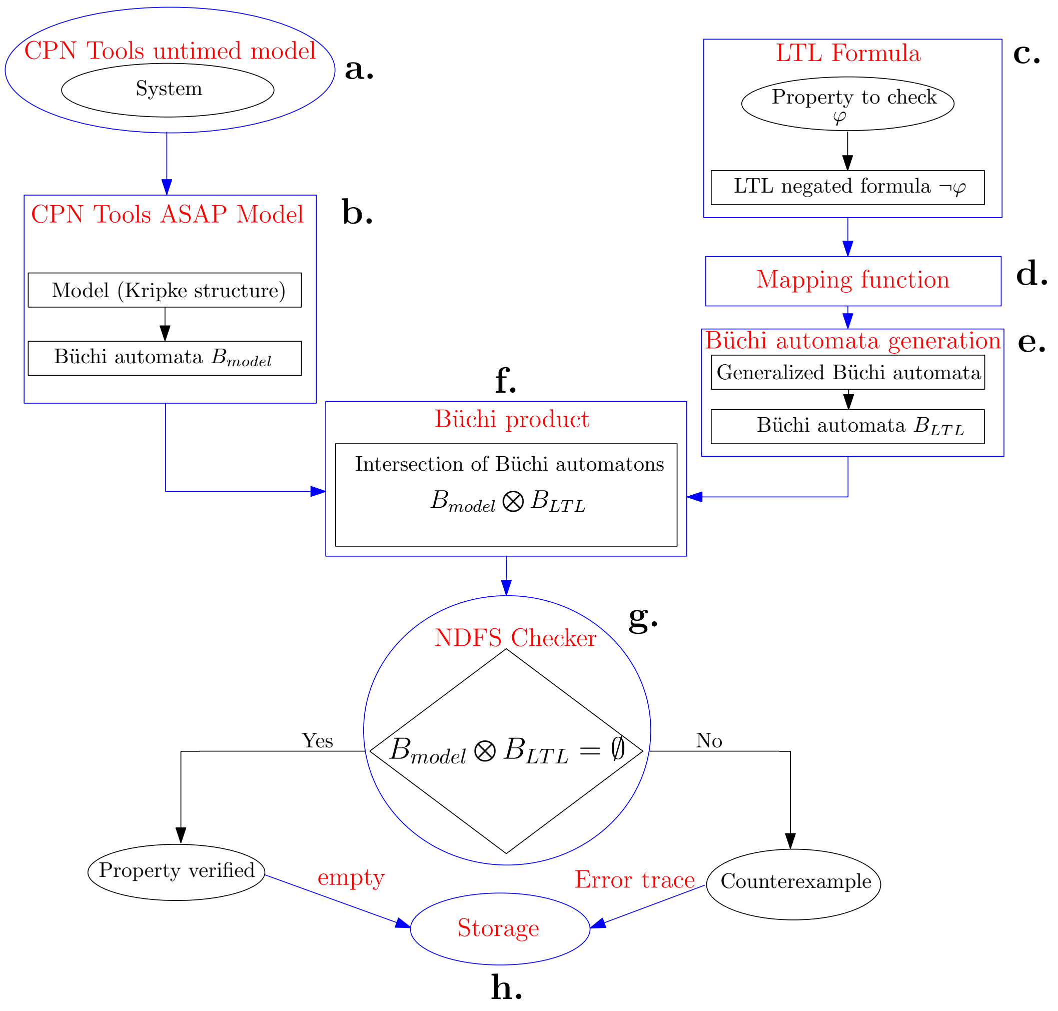

CPN Tools untimed model
System
a.
CPN Tools ASAP Model
Model (Kripke structure)
Büchi automata B_{model}
b.
LTL Formula
Property to check φ
LTL negated formula $\neg\varphi$
c.
Mapping function
d.
Büchi automata generation
Generalized Büchi automata
Büchi automata B_{LTL}
e.
f.
Büchi product
Intersection of Büchi automatons
$B_{model} \otimes B_{LTL}$
g.
NDFS Checker
$B_{model} \otimes B_{LTL} = \emptyset$
Yes
No
Property verified
Counterexample
empty
Error trace
Storage
h.

Figure 6.4: Model checking algorithm: schema

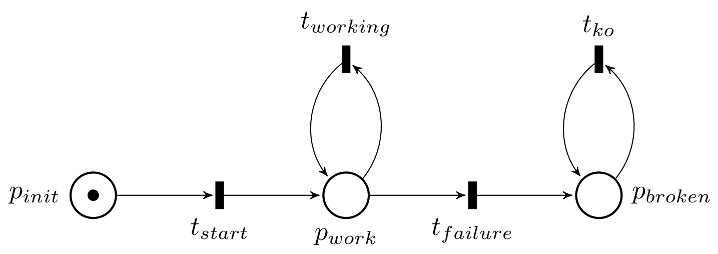


Figure 6.5: Petri net of our toy model

b. The CPNToolsModel element used by ASAP is not explicitely given, but is explored using the DFS (deep first search) method. This model is however similar to that given in Figure 6.9, with only a different way to declare the states and events. The Büchi Automata of this model

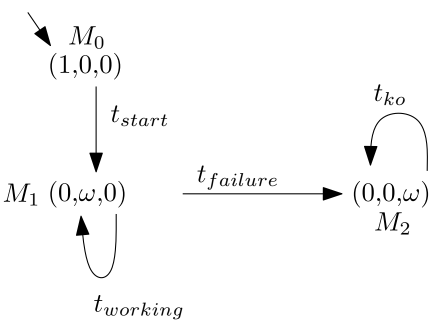


Figure 6.6: The coverability graph of the toy model, we can see the ω value defined in chapter 2.

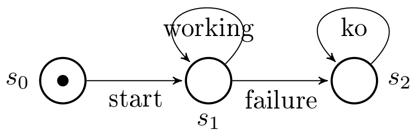


Figure 6.7: Kripke structure of the toy model

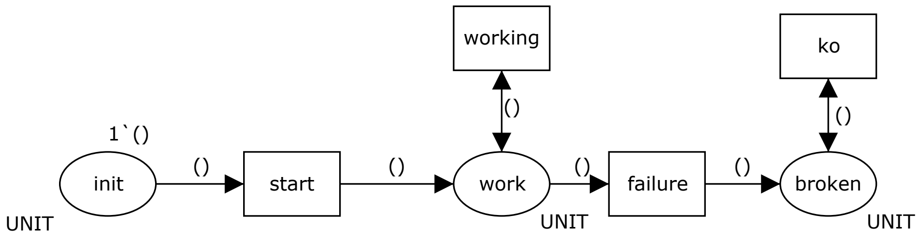


Figure 6.8: The toy model in CPN Tools

is also hidden, we can suppose that it is similar to a general definition of Büchi Automata, illustrated in figure 6.10.

c. We want check if it is possible that the toy system cannot be broken. For doing this we use an following LTL formulae like: *FUTURE(GLOBALLY work-state)*. It means that from an existing path we can remain in a selected state infinitely. This formula can be strange, because we cannot select the starting state, thus we should use the *FUTURE* to avoid this problem. The definition of AP is given by the mapping function, in our case a state is a CPNToolsModel state. Unfortunately this method is very complicated. We should explore the CPNToolsModel sates with a series of commands like:

```
val cpnstateinit =#1(hd(CPNToolsModel.getInitialStates()))
val cpneventinit= hd(#2(hd(CPNToolsModel.getInitialStates())))
val nextstatoCPN = CPNToolsModel.nextStates(cpnstateinit ,cpneventinit)
```

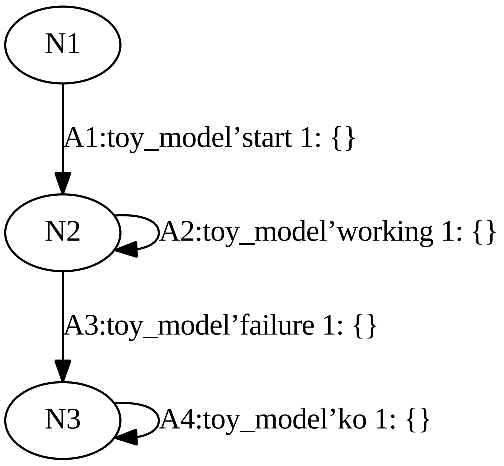


Figure 6.9: The coverability graph drawn by Graphviz

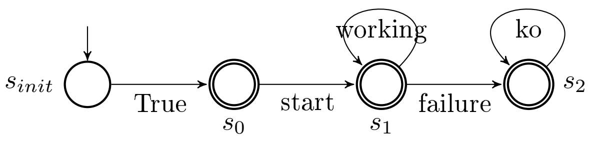


Figure 6.10: Büchi Automata of the toy model

For a complex system this is not practicable. This is the result of the LTL formula in CPN Tools:

```
open ASAP
open PLTLSyntax

val selfloop = ATOMIC statoCPN
val ok = ATOMIC statook
val formula = FUTURE(GLOBALLY ok)
```

d. The mapping function is the heart of our procedure, it is also one of the only things defined by us and without examples or manuals is really complicated to manage. We have sent different emails to the support list of CPN Tools to resolve this problem and we have made a mapping function but it has some limitations.

```
structure MyFormula = struct
type state =
CPNToolsModel.state * CPNToolsModel.event list

val (transitionTable', initials', accepting') =
   PLTLSyntax.translate formula

fun map' (ATOMIC ap) =
(if (ap=statook) then "ap" else "notap")
 | map' (NOT(NOT f)) = String.concat [map' f,"\n␣"]
 | map' (NOT f) = String.concat ["NOT␣", map' f]
 | map' TRUE = "TRUE"
 | map' FALSE = "FALSE"
 | map' _ = raise Error "Invalid␣automaton"


val labels = Vector.map
(fn (_,l) => List.map map' l) transitionTable'


fun map ap = (fn s:state => (if ((#1 s) = statook)
        then true else false))

val (transitionTable, initials, accepting) =
PLTLSyntax.map (transitionTable', initials', accepting') map
end
```

One limitations for example is thath we cannot check a formula with two inputs as states.

e. The generation of the Büchi Automata is automatic, it need only to put the correct input at the functor *BuchiSimulator* (functor definition in 2.20). We can semplify the Büchi Automata for obtain a small version.

```
(*Optimization of the BuchiExpression*)

structure help = LTLCompressor(
structure Automaton = MyFormula)
structure help2 = RemoveNonAcceptingSCC(
structure Automaton = help)
structure help3 = FinalSCCOptimizer(
structure Automaton = help2)
structure smallmodel = RemoveRedundantStates(
structure Automaton = help3)

(*BuchiModel*)
structure Buchimodello= BuchiSimulator(
structure Expression = smallmodel
)
```

The resulting Büchi Automata obtained by a manual exploration of the states is illustrated in figure 6.11.

f. The product between the two automata is illustrated in figure 6.12. As the other figure, this graph is extracted with a manual exploration of the net. We can see that the product is non-deterministic and exists two principal path. The first path with the states with the label s_2 (label of the state s_2 of the LTL Büchi Automata) is our error trace. It is explain that we can have a path that cannot remain at infinitive in the *work* status. The other with the state s_3 show

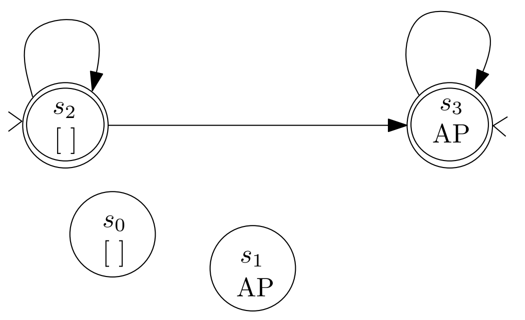


Figure 6.11: The LTL formula translated in a simplified Büchi Automata by CPN Tools

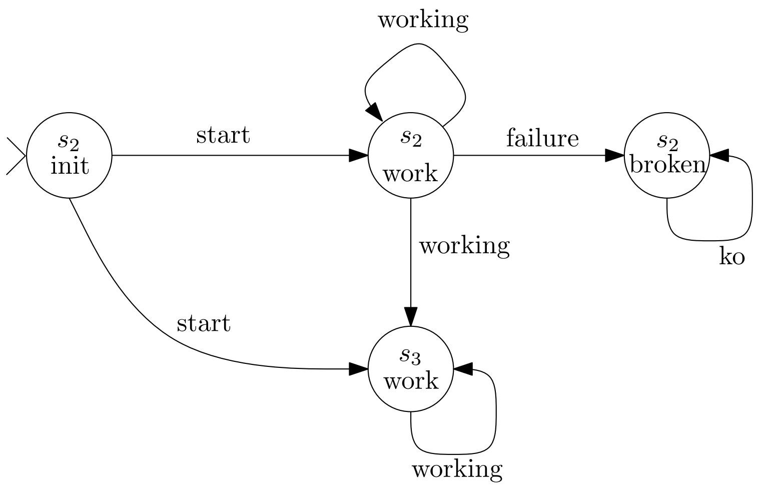


Figure 6.12: The product of the two Büchi Automata

that is it possible to have one path wich the toy model is always in a working status. This is our interpretation. The result should be checked with another functor that make a Nested Depth First Search (NDFS) (two depth first search nestled) for find a strongly connected components which is our counter example explained in the next paragraph.

g. The NDFS algorithm is performed by the functor named *NDFSLTLChecker*. Unfortunately this functor return every time an error, maybe this error is given by the definition of the Storage but it is difficult to say it. The storage is defined in the next paragraph.

```
structure verifica =NDFSLTLChecker(
structure Model = modello
structure Storage = RStorage)
```

h. The storage have a signature defined as *REMOVE_STORAGE*:

```
structure BuchiHash = ProductHashFunction (
structure Hash = CPNToolsHashFunction2)
structure RStorage = HashCompactionStorage(
structure HashState = BuchiHash
structure HashWord  = Hash31ToHash31)
```

We can see the *CPNToolsHashFunction2* which is an hash function used to map the data of the different states in our storage. In this case the functor *ProductHashFunction* is used to hashing the intersection of the Büchi Automata. The others functors are used to compress the data in the storage. Unfortunately as said before, this is not much clear, the NDFS functor works but when we try to use the function *check* (defined in NDFS functor) with the SML code illustrated below we have a generic SML error.

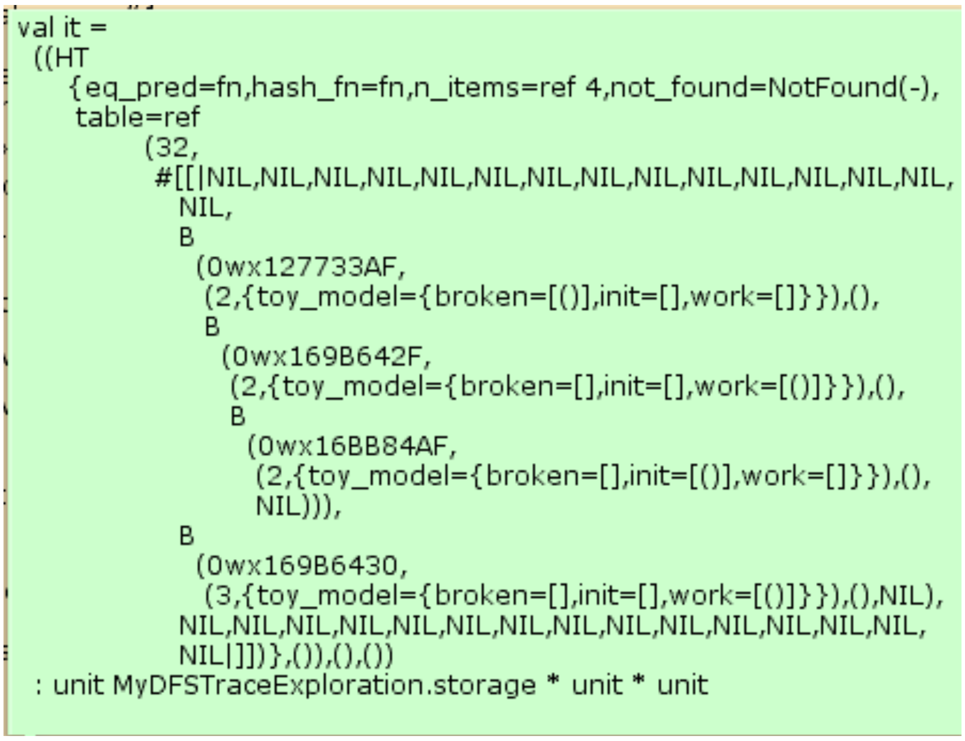
```
fun transarc (state,events) = events
fun transstate (state) = state

val counterex= verifica.check transarc
            transstate
          {
 a_initial = (),
 arc_hook = fn _ => (),
 s_initial = (),
 state_hook = fn _ => ()
}
(verifica.emptyStorage{init_size = 1})
(modello.getInitialStates ())
```

To obtain some result we have explored the product with a single DFS with another storage, without the data compression.

```
structure StorageDFS =HashStorage(
structure Model = modello
structure Hash = BuchiHash)

structure MyDFSTraceExploration=DFSExploration(
        structure Model=modello;
        structure Storage=StorageDFS);
```

The code illustrated below is used for make the exploration:

```
MyDFSTraceExploration.explore (fn (_,n) => n) (fn e=>e)
 { a_initial = (),
   arc_hook = fn _ => (),
   s_initial = (),
   state_hook = fn _ => (),
   t_initial = (),
   pre_trace_hook = fn (_, _, _, _, storage') => ((), storage'),
   post_trace_hook = fn (_, _, _, _, storage') => ((), storage')
 }
(StorageDFS.emptyStorage { init_size = 0 } ())
(modello.getInitialStates())
```



```
val it =
  ((HT
     {eq_pred=fn,hash_fn=fn,n_items=ref 4,not_found=NotFound(-),
      table=ref
            (32,
             #[[|NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,
               NIL,
               B
                 (0wx127733AF,
                  (2,{toy_model={broken=[()],init=[],work=[]}}),(),
                  B
                    (0wx169B642F,
                     (2,{toy_model={broken=[],init=[],work=[()]}}),(),
                     B
                       (0wx16BB84AF,
                        (2,{toy_model={broken=[],init=[()],work=[]}}),(),
                        NIL))),
               B
                 (0wx169B6430,
                  (3,{toy_model={broken=[],init=[],work=[()]}}),(),NIL),
               NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,NIL,
               NIL|]])},()),(),())
  : unit MyDFSTraceExploration.storage * unit * unit
```

Figure 6.13: The exploration result of our intersection automata in CPN Tools

The result of this exploration is illustrated in figure 6.13. We can see the four state present in our draw automata of figure 6.12 thus the figure drawn is correct but we can check the state space strongly connected component only manually.

ProM - Process Mining workbench

Another software which takes a CPN model and perform an LTL analysis is ProM. ProM is capable also to do a performance analysis but this is not the focus of our work. With ProM is easy to make an LTL analysis but is not formal, is done with a simulation log of our system. Of course if we have a simulation log we can use a timed model and if we perform an high number of simulations we can say with a high trustworthiness level that a property is verified or not but we cannot say it in formal terms.

Success represents the 1% of your work which results from the 99% that is called failure.

Soichiro Honda - Engineer

Chapter 7

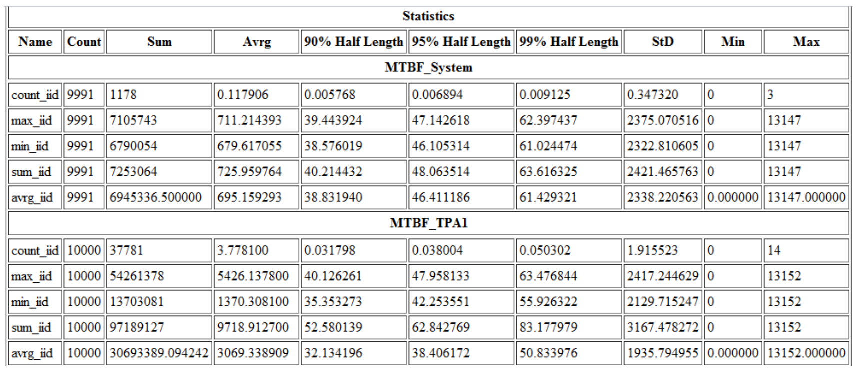
Results

Introduction

This chapter is divided in two sections with the results of the *system performance analysis* and of *formal verification analysis.* We start to describe the results of the *performance analysis* based on 10000 simulations calculated in a simulation period of 18 months.

7.1 Performance analysis

The data from different monitors in CPN Tools are written in different text files and in a single folder for one set of simulations. CPN Tools provides a report file in HTML with different statistical analyses. A part of the report file is illustrated in figure 7.1. We note that these data

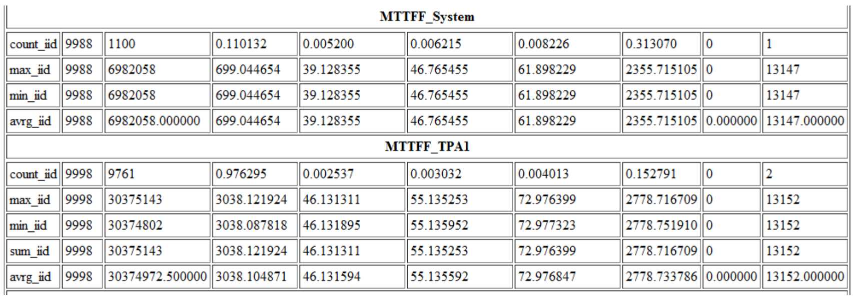


Statistics									
Name	Count	Sum	Avrg	90% Half Length	95% Half Length	99% Half Length	StD	Min	Max
MTBF_System									
count_iid	9991	1178	0.117906	0.005768	0.006894	0.009125	0.347320	0	3
max_iid	9991	7105743	711.214393	39.443924	47.142618	62.397437	2375.070516	0	13147
min_iid	9991	6790054	679.617055	38.576019	46.105314	61.024474	2322.810605	0	13147
sum_iid	9991	7253064	725.959764	40.214432	48.063514	63.616325	2421.465763	0	13147
avrg_iid	9991	6945336.500000	695.159293	38.831940	46.411186	61.429321	2338.220563	0.000000	13147.000000
MTBF_TPA1									
count_iid	10000	37781	3.778100	0.031798	0.038004	0.050302	1.915523	0	14
max_iid	10000	54261378	5426.137800	40.126261	47.958133	63.476844	2417.244629	0	13152
min_iid	10000	13703081	1370.308100	35.353273	42.253551	55.926322	2129.715247	0	13152
sum_iid	10000	97189127	9718.912700	52.580139	62.842769	83.177979	3167.478272	0	13152
avrg_iid	10000	30693389.094242	3069.338909	32.134196	38.406172	50.833976	1935.794955	0.000000	13152.000000

Figure 7.1: CPN Tools report

are not correct.

For example, in figure 7.1 we can see that the minimum value of the MTBF of the System is 0. It is not possible, because we suppose that at time 0 the system is operative. Another thing illustrated if figure 7.2 is that the third column *avrg* (meaning the arithmetical mean of the selected value, in this case the MTTFF) for the System, has a value of 699 hours. This is not possible, because theoretically the MTTFF of the TPA$_1$ is about 3000 hours and the other TPA is in a parallel configuration, the system's MTTFF must be in average higher than the MTTFF of one of the entities. This brought us to check the text file of simulation results and we



MTTFF_System									
count_iid	9988	1100	0.110132	0.005200	0.006215	0.008226	0.313070	0	1
max_iid	9988	6982058	699.044654	39.128355	46.765455	61.898229	2355.715105	0	13147
min_iid	9988	6982058	699.044654	39.128355	46.765455	61.898229	2355.715105	0	13147
avrg_iid	9988	6982058.000000	699.044654	39.128355	46.765455	61.898229	2355.715105	0.000000	13147.000000
MTTFF_TPA1									
count_iid	9998	9761	0.976295	0.002537	0.003032	0.004013	0.152791	0	2
max_iid	9998	30375143	3038.121924	46.131311	55.135253	72.976399	2778.716709	0	13152
min_iid	9998	30374802	3038.087818	46.131895	55.135952	72.977323	2778.751910	0	13152
sum_iid	9998	30375143	3038.121924	46.131311	55.135253	72.976399	2778.716709	0	13152
avrg_iid	9998	30374972.500000	3038.104871	46.131594	55.135592	72.976847	2778.733786	0.000000	13152.000000

Figure 7.2: CPN Tools report error

have encountered some little errors. For example, for the System the row *count*, second column, shows us in how many experiments a system had a failure (see figure 7.2). If we read the file, there are 1100 failures, but if we load all the text files there are 1102. It is a little mistake but it is important because it is used as basis for the arithmetical mean calculation, instead the arithmetical mean is calculated on the basis of the number given in the first column, 9988, the meaning of which is unknown.

Another error is present in a file containing the reference numbers of experiments, and if those presented a failure. We have checked some of these references and in some cases the reference has an error. Fortunately CPN Tools saves all the text files of the simulations, making it possible to perform the analysis of *Monte Carlo simulations* with MATLAB.

Monte Carlo simulations analysis in MATLAB

To perform the analysis we have imported all the text files with a MATLAB script, analyzed and plotted the results. We have made different plots of the different dependability times. For all of the different mean times we have plotted its *probability density function*. We will start by talking about the MTTFF.
The Mean Time To First Failure is an explicit term, for non reparable systems it is known as Mean Time To Failure. We can see in figure 7.3 three different histograms. The histograms of the TPAs have forms of an exponential distribution whereas the System's MTTF is uniformly distributed. This is linked to the exponential distributions of TPAs' MTTF. Intuitively, the failure rate of a TPA is constant, thus there is no particular time region where the system has the higher probability to fail.

The next mean time which we deal with is the MTTR. The Mean Time To Reparation represent the mean time between two reparations. We can see that the distribution of the MTTR of the TPA_1 (and more evidently that of the TPA_2) is close to an Erlang distribution. The empirical data confirm the theoretical distribution used in the CPN model. It is possible that the reparation time of the system is very small. For example we suppose to have the TPA_1 in reparation and 1 hour before the reparation is accomplished, the TPA_2 fails. The system comes to a failure status but after 1 hour the TPA_1 is rebooted. The reparation time of the system in this case is 1 hour.
The last graphs illustrate the different times: MTBF (Mean Time Between Failures), the Mean Down Time MDT, and the Mean Up Time MUT. As we have seen in figure 1.2 in the first chapter, the MTBF is the sum of MDT and MUT. The three MTBF with the MDT and MUT are illustrated in figures 7.5, 7.6 and 7.7.

The MTBF is the analogous of the MTTR for the failures instead of the reparations.

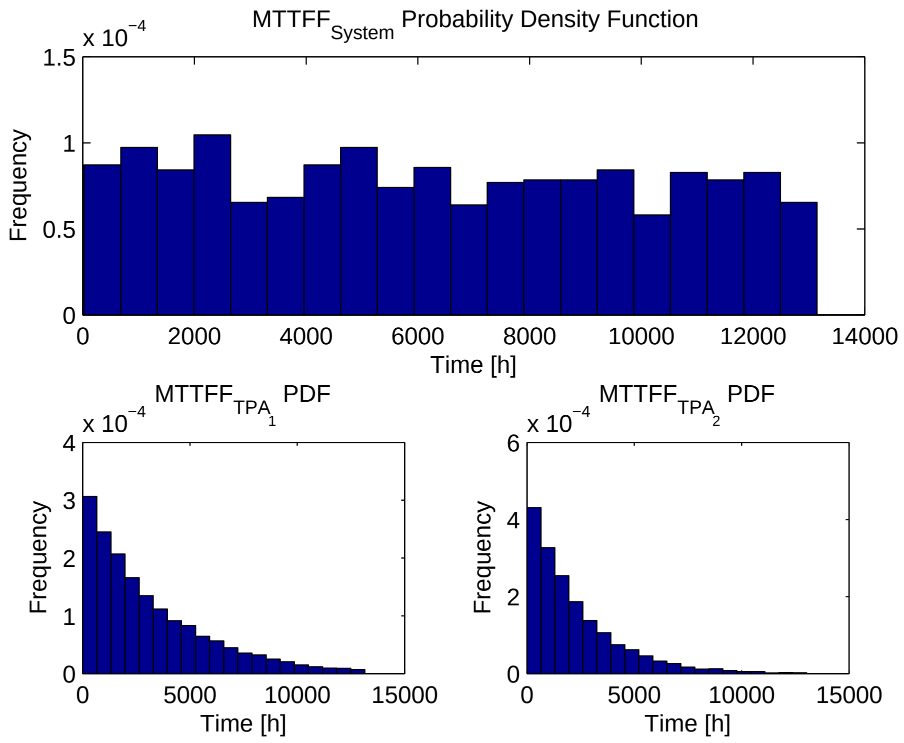


Figure 7.3: Mean Time To First Failure: results

Some relevant statistical results are illustrated in table 7.1.

Entity	MTTFF	MTBF	MTTR	MUT	MDT
TPA$_1$	3111.7	3144.2	26.4	3020.1	26.4
TPA$_2$	2235.2	2259.2	48	2135.9	48
System	6340.5	6312.8	16.4	6290.9	16.4

Table 7.1: System performance results

We can see in table 7.1 that the value of MDT and MTTR are equal. Indeed, we have supposed that the reparation starts immediately after the failure.
We can calculate other parameters like in how many experiments an entity has at least one failure. In 10000 experiments for the different entities the results are :

- TPA$_1$ = 9762,
- TPA$_2$ = 9965,
- System = 1102.

From these data we can affirm for example that empirically, for the given set of simulations, the TPA$_2$ is less reliable than the TPA$_1$.

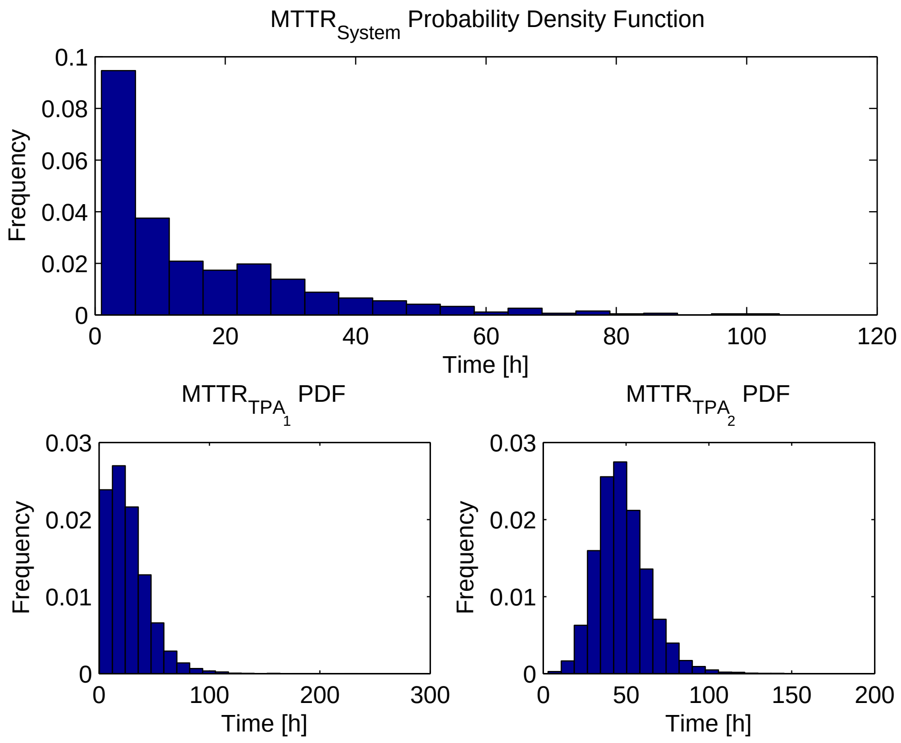


Figure 7.4: Mean Time To Reparation: results

7.2 Verification analysis

In this section the results of the formal verification analysis are given.

Old state space tool

We illustrate below a part of the data produced and reported by the state space tool of CPN Tools. We have ignored the *Boundedness Properties* and *Fairness Properties* sections for readability reasons.

```
- Statistics -
State Space, status: Full
Nodes: 72; Arcs: 100; Secs: 0
Scc Graph
Nodes: 1; Arcs, Secs: 0
- Home Properties -
Home Markings: All
- Liveness Properties -
Dead Markings, Dead Transition Instances: None
Live Transition Instances: All
```

We can see from the report that the entire state space is composed of 72 nodes (a node is a states and it is composed of the marking of all CPN model places) and 100 arcs. All the transitions

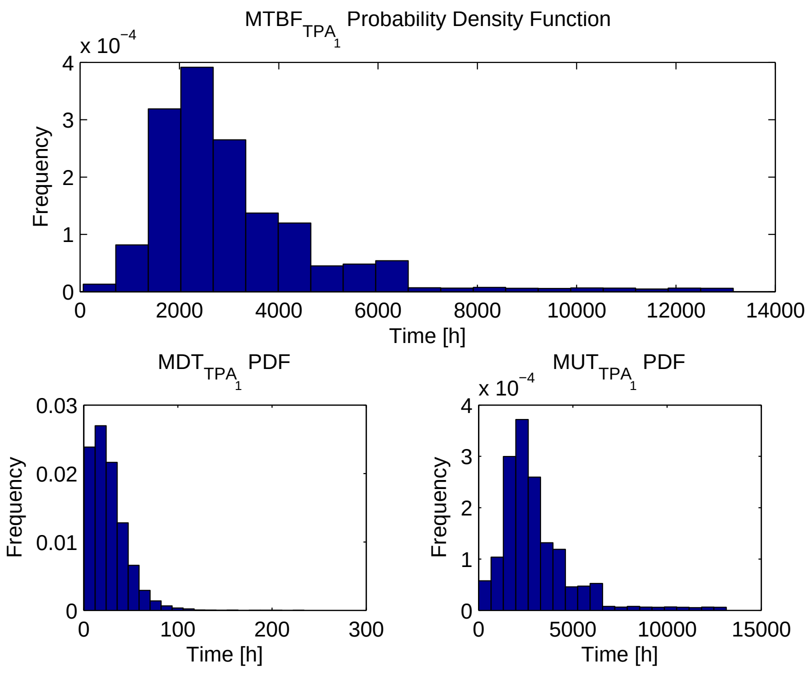


Figure 7.5: Mean Time Between Failures of TPA$_1$: results

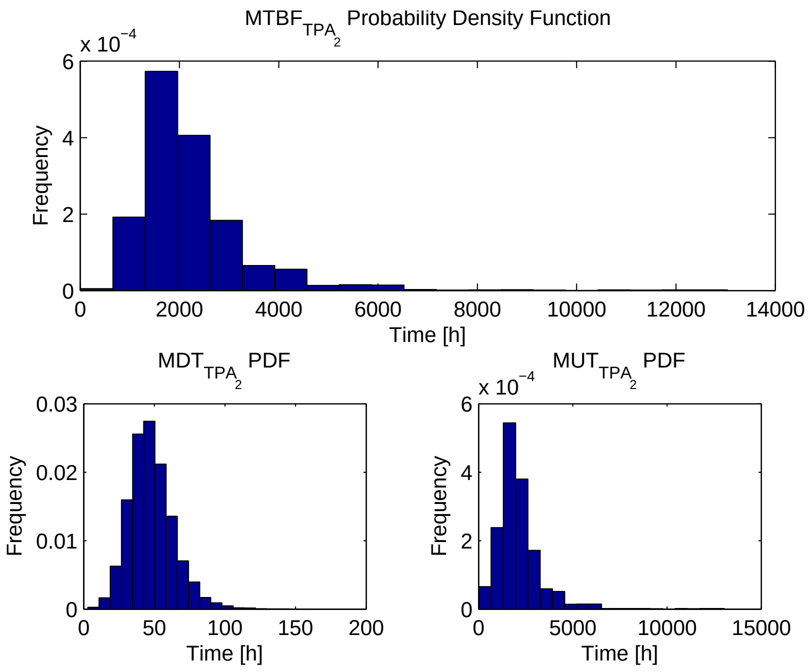


Figure 7.6: Mean Time Between Failures of TPA$_2$: results

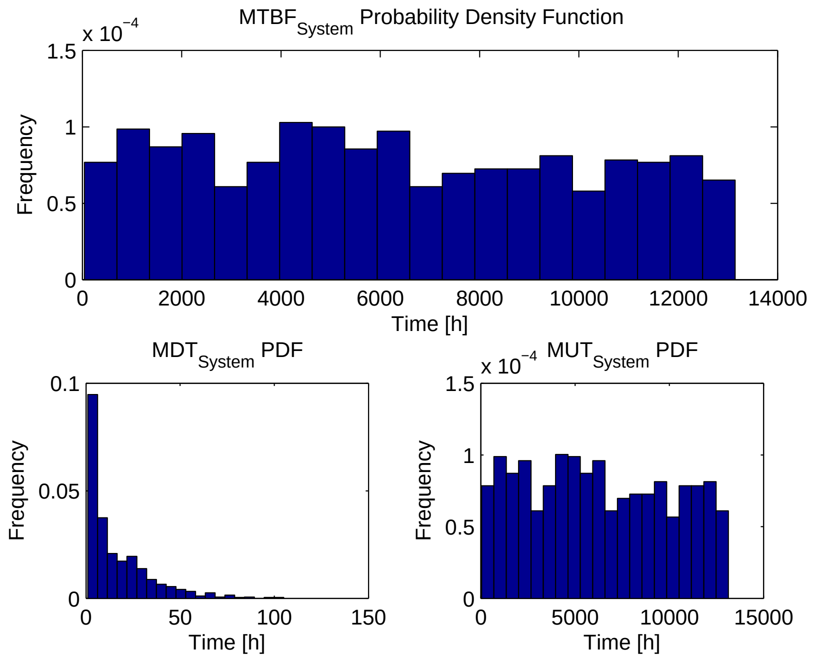


Figure 7.7: Mean Time Between Failures of the system: results

are *Live*, we have no *Dead markings* and all the nodes are *Home markings*, meaning that we can always repair our system. From every node we can return in the initial node.

State space exploration and CTL model checking

The answers to the four question illustrated in section 6.2 are illustrated below. The first two questions are answered with the information contained in the state space tools manual, available in the online documentation of CPN Tools [12]. The last two are briefly presented with a version of CTL implemented in CPN Tools but no longer supported.

Question 1. How many states and what are the states where the TPA$_1$ is broken?

Answer 1. After the computation of the state space we can use a query like this:

```
PredAllArcs(
 fn a => case ArcToBE a of
 Bind.Specification 'FTPA1 (1, {}) => true
 | _ => false)
```

This query gives us the arcs where the transition FTPA is verified, and with the function reported below we can discover the nodes.

```
st_Arc arc_number
```

Then we can answer the question, there are six different nodes: $[9, 30, 26, 28, 25, 11]$ (at every computation of the state space the identification number of the nodes can change).

Question 2. What is the minimum path from a broken TPA_1 to a state where the entire system is in failure?

Answer 2. We can use the same method as for the *Question 1* to find in which nodes the system is in failure, and afterwards use the function reported below to find the minimum path that links the two nodes.

```
Reachable'(init_node,final_node)
```

For example from the node 9 to the node 34 the minimum path is: $[9, 14, 26, 34]$

Question 3. Is it always true that the TPA_1 is restarted immediately after its reparation?

Answer 3. To answer this question we use the module ASK-CTL implemented in CPN Tools. There exists a little manual of this module and thanks to this we can make a few queries.

```
 fun Isrestarted a =
(Bind.TPA'Starting_TPAs(1, {p=pump1})=ArcToBE a);

val myASKCTLformula =
MODAL(EV(AF("Is it possible to restart the TPA1?",
            Isrestarted)))

eval_node myASKCTLformula 9
```

The manual is in the online documentation of CPN Tools [12]. In this case the command *Bind* and *ArcToBE* are taken from the state space tool manual. The aim of the first function is to verify if the arc has the transition *TPA'Starting_ TPAs(1, p=pump1)* in its bindings. This function is used in the formula declared above. The *myASKCTLformula* is our CTL formula which can be translated as *for all the path from a selected node, Isrestarted should be TRUE in a finite number of steps.* For example we start from the node number 9 and the answer is *TRUE*. This is clearly correct, we have seen in the *Question 2* that there is a path which guarantees this, in fact from the node 9 we can have a failure of the system, then is it logical that there exists a path which can restart the TPA_1.

Question 4. For all the existing paths, is it possible to arrive to a system failure state?

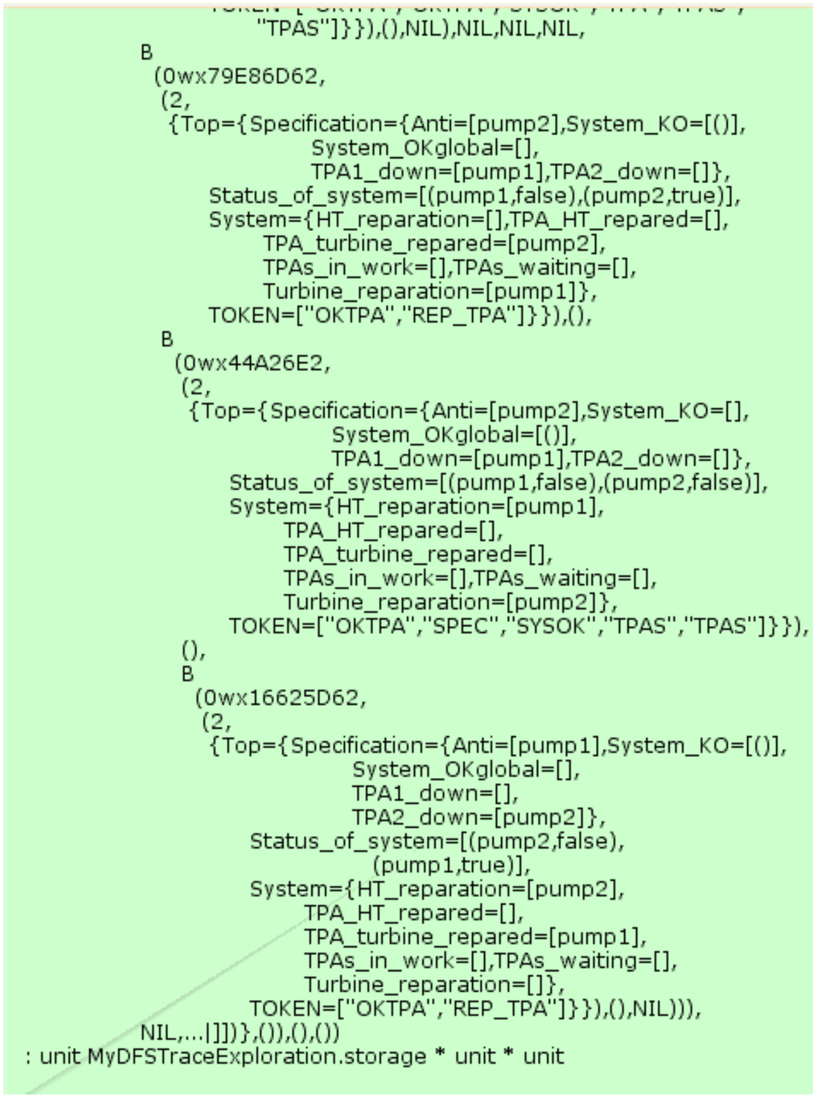
Answer 4. The answer is *TRUE*, unfortunately we should verify the formula for every node. The SML code is illustrated below.

```
fun Isgoingtofailure a =
(Bind.Specification 'System_FAILURE (1, {})= ArcToBE a) ;

val myASKCTLformula =
MODAL(POS(AF("Is it possible to breakdown the system?",
     Isgoingtofailure)))

eval_node myASKCTLformula 9
```

The formula can be translated as *Does there exist a path from the current node that can bring us to encounter a System Failure transition?*

ASAP state space tool

The part made in the section 6.2 is applied in the same way for the case study system model. In this case the only thing that changes is the state space which is more complex, because it includes a marking in all the places. We cannot draw or explore easily the state space, then it is not possible to verify the LTL properties. Perhaps is it possible to save the DFS exploration results but we don't know how. A part of the results obtained from the DFS exploration is reported in figure 7.8.

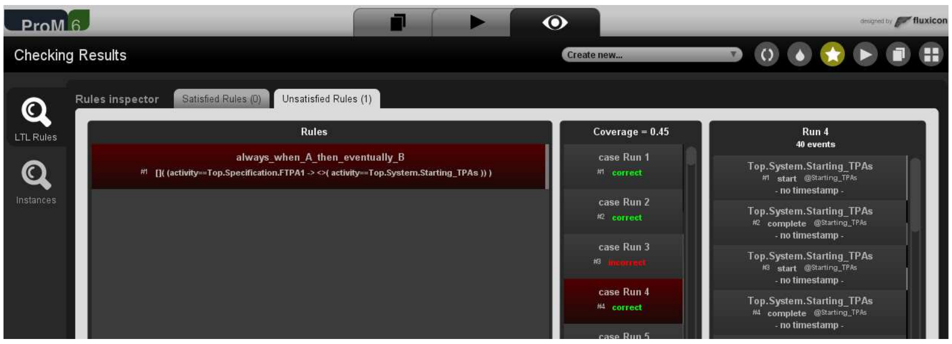


```
                          "TPAS"]}}),(),NIL),NIL,NIL,NIL,
               B
                (0wx79E86D62,
                 (2,
                  {Top={Specification={Anti=[pump2],System_KO=[()],
                                       System_OKglobal=[],
                                       TPA1_down=[pump1],TPA2_down=[]},
                       Status_of_system=[(pump1,false),(pump2,true)],
                       System={HT_reparation=[],TPA_HT_repared=[],
                               TPA_turbine_repared=[pump2],
                               TPAs_in_work=[],TPAs_waiting=[],
                               Turbine_reparation=[pump1]},
                       TOKEN=["OKTPA","REP_TPA"]}}),(),
                 B
                  (0wx44A26E2,
                   (2,
                    {Top={Specification={Anti=[pump2],System_KO=[],
                                         System_OKglobal=[()],
                                         TPA1_down=[pump1],TPA2_down=[]},
                         Status_of_system=[(pump1,false),(pump2,false)],
                         System={HT_reparation=[pump1],
                                 TPA_HT_repared=[],
                                 TPA_turbine_repared=[],
                                 TPAs_in_work=[],TPAs_waiting=[],
                                 Turbine_reparation=[pump2]},
                         TOKEN=["OKTPA","SPEC","SYSOK","TPAS","TPAS"]}}),
                   (),
                   B
                    (0wx16625D62,
                     (2,
                      {Top={Specification={Anti=[pump1],System_KO=[()],
                                           System_OKglobal=[],
                                           TPA1_down=[],
                                           TPA2_down=[pump2]},
                           Status_of_system=[(pump2,false),
                                             (pump1,true)],
                           System={HT_reparation=[pump2],
                                   TPA_HT_repared=[],
                                   TPA_turbine_repared=[pump1],
                                   TPAs_in_work=[],TPAs_waiting=[],
                                   Turbine_reparation=[]},
                           TOKEN=["OKTPA","REP_TPA"]}}),(),NIL))),
               NIL,...|]])},()),(),())
  : unit MyDFSTraceExploration.storage * unit * unit
```

Figure 7.8: Output example of the DFS exploration

ProM - Process Mining workbench

With ProM we execute 1000 simulations limited to 20 steps, there are no particular reasons for the 20 steps but we should limit the simulations. We have tried to verify an LTL formula like, *Always when the* TPA_1 *is in failure eventually we can restart it* (in LTL syntax $\Box FTPA1 \rightarrow \Diamond Starting_T PAs$, the arrow means the imply operator). The result of this query is illustrated in



Rules

always_when_A_then_eventually_B
#1 []((activity==Top.Specification.FTPA1 -> <>(activity==Top.System.Starting_TPAs)))

Coverage = 0.45

case Run 1
#1 correct

case Run 2
#2 correct

case Run 3
#3 incorrect

case Run 4
#4 correct

case Run 5

Run 4
40 events

Top.System.Starting_TPAs
#1 start @Starting_TPAs
- no timestamp -

Top.System.Starting_TPAs
#2 complete @Starting_TPAs
- no timestamp -

Top.System.Starting_TPAs
#3 start @Starting_TPAs
- no timestamp -

Top.System.Starting_TPAs
#4 complete @Starting_TPAs
- no timestamp -

Figure 7.9: The results of the LTL formula in ProM

figure 7.9 . The property is not fully satisfied, it is verified in only 45% of the cases. Actually the 45% is the limit of the ProM LTL techniques. We can say that ProM performed an LTL *statistical* model checking, with the simulation trace which cannot assure the formal verification. In our case indeed the limitation to 20 steps for simulations is the explication of the only 45% of the verified true value.

A conclusion is the place where
you get tired of thinking.

Arthur Bloch - Author of the Murphy's Law books

Chapter 8

Conclusions and future development

8.1 Conclusions

In this work we have developed a way to model stochastically arriving failures and reparations in a system with a CP-nets, afterwards we have modified the system model to be deterministic. It was not an easy task, and involved the usage of some *Golden token* to control the two different nets, like two different processes. The verification of the correct behaviour of the model has required time and patience. It is not possible to develop an algorithm to make the procedure for transforming a stochastic model into a deterministic model automatic. Not only the temporal transitions should be removed but it is necessary to check the correct logical behaviour of the model. CPN Tools as the CP-net software is better suited for a *performance analysis*. The different probability distributions allow to model various systems. The implemented monitors provide data from the simulations, thus the parameters of the performance analysis can be calculated. After a little time and with the help of the on-line documentation this part of the task is not difficult to manage. Some problems evolved when dealing with the report data automatically saved by CPN Tools, which can be not accurate. We have however succeeded to calculate all desired performance parameters correctly with MATLAB, directly using the files saved for each simulation set.
More important difficulties arose while performing the *verification analysis*. The CPN state space tool produces and saves a report, allowing to draw the state space of a model. This part implemented in *ASAP* seems however more convenient. Unfortunately, no documentation is developed for that, and the ASAP tool is not officially supported. Some aims have been nevertheless achieved thanks to the patience of a developer (M.Westergaard). *ASAP* has the ability to perform an LTL model checking analysis and to make different state explorations. This seem quite complicated, but is still possible. From M.Wesergaard's blog we have learned that the ASAP integration into CPN Tools GUI is not scheduled for the moment. The simulator written in SML wil perhaps be changed to a Java simulator next year. This has some advantages, for example a major part of users will be able to develop their own functions easily.
It is finally interesting to implement the data exchange between the ProM software and the CPN Tools. It would however be useful only for the performance analysis, and less suited for the model verification.

8.2 Future developments

An important future development of our model consists in its extension, *i.e* in considering a more complete and complex case study. In particular, we can add the extraction pumps CEX, model the power of the system considering different power profiles (indeed, the power influences the functioning of the TPAs). It would also be interesting to add a small amount of time to simulate the start of the reparation or the time required to reboot the TPA after the reparation. These times will influence the MDT and the MUT. Indeed in this case the MDT should not be equal to the MTTR, because the reparation will not start immediately after the failure (due to failure detection delay for example).
Concerning the model verification, without a full manual of *ASAP* functors it is rather difficult to perform the LTL analysis and to use all the computational power offered by ASAP. Verification that we can perform is limited. In theoretical works and practical applications of the automata verification, a software named *SPIN* is often used. Another famous software is *UPPAAL* which allows to account for the timed automata and to verify the properties using the *Timed CTL*. However our purpose was to explore the capacities of the CPN Tools in terms of model checking, we have thus not chosen other softwares to perform the analysis. In the next version of CPN Tools (the fourth version), *ASAP* integration will probably be accomplished, enabling the performance of the type of analyses that other softwares can not perform. In the meanwhile one is to study the SML code and to test different procedures.

List of acronyms

AP	Atomic Proposition
APPRODYN	APProches de la fiabilité DYNamique pour modéliser des systèmes critiques
ASAP	ASCoVeCo State space Analysis Platform
CPN Tools	CPN Tools
CTL	Computation tree logic
DES	Discrete Event System
DFS	Depth fist search algorithm
EDF	Électricité De France
GMEC	Generalized Mutual Exclusion Constraints
IDPA	Integrated Deterministic and Probabilistic Dependability Analysis
LTL	Linear temporal logic
FTA	Fault tree analysis
GUI	Graphical Interface Unit
KP	Kripke structure
LBA	Labeled Büchi Automaton
LGBA	Labeled Generalized Büchi Automaton
NDFS	Nested Depth First Search
PWR	Pressurized water reactor
RAMS	Reliability Availability Maintainability Safety
RBD	Reliability Block Diagram
TPA	Turbo Pompe Alimentaire

Bibliography

[1] "Ieee guide for information technology - system definition - concept of operations (conops) document," *IEEE Std 1362-1998*, p. i, 1998.

[2] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model checking*. Cambridge, MA, USA: MIT Press, 1999.

[3] G. ZWINGELSTEIN, "Sûreté de fonctionnement des systèmes industriels complexes," *Techniques de l'ingénieur. Informatique industrielle*, vol. 9, no. S8250, pp. 1–32, 1999.

[4] J. Ligeron, "Le cercle des fiabilistes disparus ou critique de la raison fiabiliste–," 2006.

[5] IEEE, "Ieee standard dictionary of measures of the software aspects of dependability," *IEEE Std 982.1-2005 (Revision of IEEE Std 982.1-1988)*, pp. 1–34, 2006.

[6] B. Bordbar, L. Giacomini, and D. Holding, "Uml and petri nets for design and analysis of distributed systems," in *Control Applications, 2000. Proceedings of the 2000 IEEE International Conference on*, pp. 610–615, IEEE, 2000.

[7] K. Jensen and L. Kristensen, *Coloured Petri nets: modeling and validation of concurrent systems*. Springer-Verlag New York Inc, 2009.

[8] L. Bernardinello and F. De Cindio, "A survey of basic net models and modular net classes," in *Advances in Petri Nets 1992* (G. Rozenberg, ed.), vol. 609 of *Lecture Notes in Computer Science*, pp. 304–351, Springer Berlin / Heidelberg, 1992.

[9] W. Reisig, "Place/transition systems," *Petri Nets: Central Models and Their Properties*, pp. 117–141, 1987.

[10] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541 –580, apr 1989.

[11] A. Giua, F. DiCesare, and M. Silva, "Generalized mutual exclusion contraints on nets with uncontrollable transitions," in *Systems, Man and Cybernetics, 1992., IEEE International Conference on*, pp. 974–979 vol.2, 1992.

[12] M. Westergaard, "Cpn tools online documentation." http://cpntools.org/documentation/start.

[13] R. Harper, “Programming in standard ml,” *Working draft available at: http://www. cs. cmu. edu/˜ rwh/smlbook*, 1997.

[14] D. Giannakopoulou and F. Lerda, “From states to transitions: Improving translation of ltl formulae to büchi automata,” in *Formal Techniques for Networked and Distributed Sytems - FORTE 2002* (D. Peled and M. Vardi, eds.), vol. 2529 of *Lecture Notes in Computer Science*, pp. 308–326, Springer Berlin / Heidelberg, 2002.

[15] F. Somenzi and R. Bloem, “Efficient büchi automata from ltl formulae,” in *Computer Aided Verification* (E. Emerson and A. Sistla, eds.), vol. 1855 of *Lecture Notes in Computer Science*, pp. 248–263, Springer Berlin / Heidelberg, 2000.

[16] M. Daniele, F. Giunchiglia, and M. Vardi, “Improved automata generation for linear temporal logic,” in *Computer Aided Verification* (N. Halbwachs and D. Peled, eds.), vol. 1633 of *Lecture Notes in Computer Science*, pp. 681–681, Springer Berlin / Heidelberg, 1999.

[17] R. Gerth, D. Peled, M. Vardi, and P. Wolper, “Simple on-the-fly automatic verification of linear temporal logic,” in *Protocol Specification Testing and Verification*, vol. 15, pp. 3–18, 1995.

[18] P. Gastin and D. Oddoux, “Fast LTL to Büchi automata translation,” in *Proceedings of the 13th International Conference on Computer Aided Verification (CAV’01)* (G. Berry, H. Comon, and A. Finkel, eds.), vol. 2102 of *Lecture Notes in Computer Science*, (Paris, France), pp. 53–65, Springer, July 2001.

[19] S. Demri and P. Gastin, “Specification and verification using temporal logics,” 2009.

[20] Z. Manna and A. Pnueli, *Temporal verification of reactive systems: safety*, vol. 2. Springer Verlag, 1995.

[21] R. Tarjan, “Depth-first search and linear graph algorithms,” in *Switching and Automata Theory, 1971., 12th Annual Symposium on*, pp. 114 –121, oct. 1971.

[22] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis, “Memory-efficient algorithms for the verification of temporal properties,” *Formal Methods in System Design*, vol. 1, pp. 275–288, 1992. 10.1007/BF00121128.

[23] N. Matta, *Supervision and Safety of Complex Systems.* Wiley, 2012.

[24] L. Kristensen and M. Westergaard, “The ascoveco state space analysis platform: Next generation tool support for state space analysis,” in *Eighth Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools*, p. 1, 2007.

Acknowledgements - Ringraziamenti

Ci sarebbero tante persone da ringraziare per il supporto e lo sviluppo di questa lavoro di tesi. La mia famiglia in primis (Franco, Rosanna, Pietro, Giancarlo, Martina, Renato, Lella, Gianna, Davide, Giorgia, Claudia, Matteo, Maria Pina, Maria Ausilia, Efisio e tanti altri), gli amici dell'università (Fabio, Giovanni, Tomaso, Paolo, Federica, Valeria, Giulia, Francesca, Alessandra, Sebastiano, Matteo, Simone, Mattia), i ragazzi dell'AVIS (Francesco, Massimo, Nicola, Donatello etc) e quelli dell'Italia dei Valori (Diego, Francesco, Davide, Lorenzo, Antonio) per gli aiuti economici durante un periodo buio della mia vita e per l'incoraggiamento che mi hanno permesso di intraprendere questa esperienza all'estero con il programma ERASMUS. Esperienza che non sarebbe potuta iniziare se l'ufficio relazioni internazionali dell'Università di Cagliari (ISMOKA) ed il mio relatore (Prof. A.Giua) non mi avesserero aiutato con rapidità a fine luglio 2011 quando tutto sembrava perso a causa di problemi burocratici. Pour Nancy je dois remercié l'ESN de Nancy, sans lequel je ne sais pas si j'aurais eu le mémé amour que j'ai maintenant pour la France et la Lorraine. Merci à mes amis erasmus Alice, Alina, Anna, , Silvia, Stefania, Luca, Antonio, Marco, Luigi, Agnieskza, Bibi, Bianca, Laura et autres... (j'aurais besoin de un autre thesis pour l'appeler tous). À la fin je voudrais remercie Génia pour l'immense travail qu'elle a fait pour m'aide dans la correction de ce mémoire et les professeurs Petin et Brînzei du CRAN pour l'opportunite qui m'ont donnée.

Alla fine ci siamo riusciti, finire gli esami è stato piú complicato di lavorare e fare la tesi. È giusto aggiungere una parte agli altri ringraziamenti, questi sono ringraziamenti per il fine esami. Esami che mi hanno tenuto in scacco per diverso tempo ma alla fine ho vinto io. Per questo devo rigraziare i colleghi Fabio, Giovanni e Claudio in particolare, senza i loro appunti, i loro consigli, ci avrei messo ancora di piú. Un ringraziamento speciale vá ad Alessandra, mi ha supportato, obbligato, istigato, minacciato a causa dello studio, qualche volta l'ho odiata. Non contenta dello stress che mi metteva mi ha addirittura interrogato negli ultimi esami, con sua notevole difficoltá (in medicina non si dimostra perché il cielo é azzurro ad esempio). Sono sempre particolarmente grato al mio primo finanziatore, mio fratello, che ha investito una parte del suo stipendio su di me, senza di lui, non lo só, non só come sarebbe andata a finire, ma nonostante tutto ci sono riuscito. Infine un ringraziamento é dovuto alla segreteria studenti della facoltá di ingegneria, senza il lavoro degli impiegati, il loro aiuto, la loro pazienza, non sarebbe stato possibile finalmente discutere la tesi nel mio paese, l'Italia.

This master thesis have won the *EFNMS (European Federation of National Maintenance Societies) Excellence Award 2014 for Master Thesis in Maintenance* during the opening ceremony of the Euromaintenance 2014, in Helsinki (Finland), 6 May 2014.