

Petri Nets and Dependability

Simona Bernardi¹, Andrea Bobbio², and Susanna Donatelli¹

¹ Dipartimento di Informatica, Università di Torino, Torino, Italy
{bernardi,susi}@di.unito.it

² Dipartimento di Informatica, Università del Piemonte Orientale, Alessandria, Italy
{bobbio}@di.unipmn.it

Abstract. Dependability evaluation main objective is to assess the ability of a system to correctly function over time. There are many possible approaches to the evaluation of dependability: in these notes we are mainly concerned with dependability evaluation based on probabilistic models. Starting from simple probabilistic models with very efficient solution methods we shall then come to the main topic of the paper: how Petri nets can be used to evaluate the dependability of complex systems.

1 Introduction

The term *dependability* is normally used to refer to the ability of an element (hardware or software component, plant or whatever complex system) to correctly perform its intended function, or *mission*, over time.

In this paper we are interested in the *quantitative evaluation* of dependability, a research field that has many practical implications, as: 1) the analysis of risks and safety; 2) the specification and contract document of a system - it is usually the case that the definition of a new system also includes requirements about dependability, that is to say on how much we can rely on the system being built, whether this is a software product, an automation system, or a bridge; 3) incidence of maintenance in the life cycle of a system - being able to estimate the dependability of an object allows to predict how often it will break down, with the consequence of additional costs on maintenance, and to take decision on the balance between investing more time and money on the construction of the system and having bigger maintenance costs; 4) dimensioning of technical assistance sector: being able to predict how often a component of a car will break down allows to estimate the number of spare components needed over a certain time period, the costs of the repairs during the warranty period, and the planning of the preventive maintenance.

To study the evolution over time of the dependability of a system it is necessary to be able to foresee when and how its component will be subject to malfunctioning, and how the malfunctioning of a component may affect the system behavior. There are two major approaches: measuring of physical, existing systems, and evaluation of abstract models of the (existing or planned) systems.

For what concerns measures, we can distinguish the following classes:

◇ *Field measures.*

It assumes that the system is already operational, and that is possible to measure the quantities of interest without altering the system behavior. Field measures should be collected for a “statistically significant” period.

◇ *Single component measures.*

Only a subset of the components are placed under test to collect measures. This is usually accompanied by acceleration technique, that generate, in a short period of time, the same conditions that the component will encounter along a much longer period.

◇ *Prototype measures.*

A prototype of the system is built, and measures are taken on the prototype under normal operational condition. This technique is very expensive, and it can be used only for goods of large consume or that involve an expected high safety risk.

The approach based on models requires first the construction of abstract mathematical models that describe the behavior of the system: the quantity of interest is the computed through the analysis and solution of the model. Modeling for dependability suffers of the same problems as any other modeling approach: the choice of the right level of abstraction for the quantity/property we want to evaluate and the complexity of the model solution.

Models are usually distinguished according to the following characteristics:

◇ *Modeling language.*

A model can be described in terms of basic quantities and mathematical expressions that relate the overall behavior to the basic quantities, as well as through an high level language with a well defined semantics, as for example Petri nets or queueing networks, or application-specific languages, in which the language elements have a direct counterpart on the system basic components. In general, the higher level is the language, the easier is to define the model, and usually the harder is to solve it. By model solution we mean the evaluation of the quantity of interest, in our case it is usually a direct definition of the dependability of the system.

◇ *Solution methods.*

There are two large classes of solution methods: analytical techniques and simulation. Analytical techniques assume that it is possible to derive from the model a set of equations that describe the quantities under evaluation, and that there are mathematical techniques, exact or approximate, to solve the equations. Simulation consists instead in executing the model on a computer a number of times that is sufficient to provide a statistically acceptable estimation of the quantities of interest. There are high level modeling languages for which simulation is the only viable analysis technique, as it is usually the case for application-oriented languages, in which the semantics of the basic elements of the language is not defined in mathematical terms, but through a piece of program code.

Topic of these notes is indeed to describe various modeling approaches to the evaluation of dependability, with a large emphasis on Petri nets.

Measures and models should by no means be considered as competitors. The evaluation of dependability requires the synergic use of both: measures can indeed be used to set the model parameters, while models can be used to drive the, usually expensive, measuring activity.

With the inherent complexity of modern systems it is extremely difficult to precisely and deterministically describe the physical, technological, and environmental factors

and interactions that provoke a system malfunctioning. It is instead very much accepted that the time to failure of a system (how long does it take for the system to go into a faulty state) is not a deterministic quantity, but a random variable of a, generally unknown, distribution. A similar rationale supports also the idea that, if a system can be repaired, then the time to repair (how long it takes for the system to be repaired) is also a random variable.

The above observations led to the development of a probabilistic approach to the quantitative evaluation of system dependability, that is the main topic of this paper.

Let us now introduce the terminology used in this paper.

Faults, errors and failures. We adhere to the terminology discussed in [43], and we say that when the delivered service of a system deviates from fulfilling the system intended function, then the system has a *failure*. A failure is due to a deviation from the correct state of the system, known as *error*. Such a deviation is due to a given cause, for instance related to the physical state of the system, or to a bad system design. This cause is called a *fault*. We shall generically refer to fault, errors, and failures as the “FEF elements.”

Systems and components. We view a system as built out of elementary components. We shall first discuss dependability of a component in isolation (that can be seen as a single-component system), to then introduce the dependability of a system as a function of the dependability of its components. When dealing with a single component we do not distinguish the FEF elements, so we equivalently refer to a component as being faulty/non-faulty, failed/not-failed, not-working/working, down/up, which are current terms in the literature. For a system with a simple structure like whose presented in Sub-Section 5.1 we distinguish between fault in a component and failure of the system (provoked by one or more faulty components). The reader should nevertheless be aware that it is also common of the literature to use the generic term failure, so that the failure of one or more component provokes the failure of the system.

Net classes. Since the main topic of the paper is on quantitative analysis based on probabilistic approach, we shall consider Petri nets with timed transitions. The time associated to transition is either zero (immediate transition) or it is a delay described by an exponentially distributed random variable. We shall use two specific net classes: Generalized Stochastic Petri Nets (GSPN) [1], and their colored counterpart Stochastic Well-formed Nets (SWN) [18].

The paper is organized as follows: in Section 2 the basic concepts of dependability are introduced. Section 3 presents two combinatorial techniques for the dependability analysis of systems consisting of a number of independent components: the reliability block technique and fault tree analysis. Section 4 describes state enumeration techniques that can be applied also for the dependability analysis of systems in which the independence assumption among components does not hold. The material presented in Sections 2,3 and 4 derives from [13]. Section 5 introduces dependability modeling using Petri Nets. Two examples are presented: a simple one, representing a system with two independent components, and a more complex example with several interacting components that is a simplified version of the case study analyzed in [10]. Section 6

describes a systematic, compositional approach to the construction of Stochastic Petri Net models for dependability and Section 7 describes the application of such approach to the automation system domain. The material presented in Sections 6 and 7 has been taken from [8, 7, 4]. Finally, conclusions are written in Section 8.

2 Basic Concepts of Dependability

A first step towards quantitative evaluation of dependability is the definition of the dependability quantity. The definition of dependability takes different flavors depending on whether we consider a system that, once broken, stays broken forever, or a system that, once broken, it is repaired and goes through cycles of correct functioning and repairs. In the first case the measure to be considered is the *reliability*, in the second case is the *availability*, as we shall see in the following.

2.1 Dependability of Non-repairable Components: Reliability

The first case considered is that of non-repairable components, that is to say the system under study is seen as a monolithic component that, once it is broken/malfunctioning, it will stay in that state forever. In this case the dependability of the system is characterized by the reliability quantity. A commonly accepted definition of reliability [60] is:

The reliability of a component at time t is the probability that the component correctly fulfills the assigned mission during the interval $[0, t]$, given its environmental conditions.

Observe that the definition relates the reliability of a component to its environment: it is therefore possible that the same component will have a very different reliability depending on the environment in which it is placed.

Let τ be the random variable that represents the *time to failure* of the component under study, τ being a time quantity, it is defined only for non-negative values. The *probability distribution function* of the variable τ is:

$$F(t) = \text{Prob}\{\tau \leq t\} \quad (1)$$

that defines the probability that the system is malfunctioning at time t . The following properties hold true for $F(t)$:

$$\begin{cases} F(0) = 0 \\ \lim_{t \rightarrow \infty} F(t) = 1 \\ F(t) \text{ not decreasing in } t \end{cases} \quad (2)$$

The reliability function is defined as the complement of $F(t)$:

$$R(t) = \text{Prob}\{\tau > t\} = 1 - F(t) \quad (3)$$

that defines the probability that the system is still working properly (still up) at time t (and since the system is not repairable, being up at time t it means that no fault has taken place). The following properties hold true for $R(t)$:

$$\begin{cases} R(0) = 1 \\ \lim_{t \rightarrow \infty} R(t) = 0 \\ R(t) \text{ not increasing in } t \end{cases} \quad (4)$$

If $F(t)$ is derivable, the probability density function of the variable τ is:

$$f(t) = \frac{dF(t)}{dt} = -\frac{dR(t)}{dt} \quad (5)$$

where $f(t)dt$ is the probability that the value of τ falls in between t and $t + dt$, that is to say, the fault takes place in between t and $t + dt$. Moreover:

$$\int_a^b f(t) dt = \text{Prob}\{a < \tau \leq b\} = F(b) - F(a)$$

represents the probability that the fault takes place in the interval $[a, b]$.

The expected value of the variable τ , $E[\tau]$, is called *Mean Time To Failure*, and is indicated by the acronym *MTTF*.

The hazard rate. The hazard (or failure) rate represents the probability that a component gets faulty between t and $t + dt$, given that it was correctly functioning up to time t (that is to say, the hazard rate is equal to the probability density function of the τ variable, *conditioned* on the fact that the component was still working correctly at time t [51]).

$$h(t) = \text{Prob}\{t < \tau \leq t + dt | \tau > t\} = \frac{\text{Prob}\{t < \tau \leq t + dt, \tau > t\}}{\text{Prob}\{\tau > t\}} \quad (6)$$

From (6), using (5), we can derive:

$$h(t) = \frac{f(t)}{R(t)} = -\frac{1}{R(t)} \frac{dR(t)}{dt} \quad (7)$$

and solving for $R(t)$ we get:

$$R(t) = \exp \left[- \int_0^t h(x) dx \right] \quad (8)$$

that is the fundamental equation that relates reliability and hazard rate.

The classic shape of the failure rate when plotted over the time axis is that of a bathtub: the failure rate is high at the beginning of the life of the object, it then remains stable for a significant period, and it finally increases. In terms of behavior of a system, it means that most systems have a very high probability of breaking when they are new, this probability decreases while experiencing a correct functioning of the system, up to

a point in time in which the failure rate is constant, that is to say the failure rate does not depend on the particular time instant that we are considering, and up to a certain time barrier after which the aging of the system is predominant and the probability of breaking down increases as the time passes by. The bath-tube shape is particularly evident for manufacturing products, while for electronic components the aging effect is less evident.

In the technical literature it is often the case that the failure rate is a single constant value, which implies that we are assuming that the system is in its period of life corresponding to the bottom of the bath-tube: a constant failure rate is therefore equivalent to saying that the system has no memory of its past.

Which distribution for τ ? Given that the time of correct functioning of a system is a random variable, what is an adequate distribution for it? We shall present two candidate distributions: exponential and Weibull.

The main characteristic of the exponential distribution is that the failure rate is constant, and, vice-versa, any distribution with a constant failure rate is exponential [63]. Given a constant failure rate, $h(t) = \text{cost} = \lambda$, from (7) and (5) we can derive:

$$\begin{aligned} F(t) &= 1 - e^{-\lambda t} \\ R(t) &= e^{-\lambda t} \\ f(t) &= \lambda e^{-\lambda t} \\ h(t) &= \lambda \end{aligned} \tag{9}$$

The mean value of τ is $MTTF = 1/\lambda$, that is to say, the failure rate has a clear physical meaning: it is the inverse of the failure rate.

The exponential distribution is known as *memoryless* since the reliability conditioned on the fact that the component has been working correctly already for a duration $t = a$, is equal to the reliability at time $t = 0$.

The Weibull distribution is:

$$F(t) = 1 - \exp \left[- (t/\eta)^\beta \right]$$

where $\eta > 0$ is the scaling parameter (displacement on the x-axis), and $\beta > 0$ is the shaping parameter. Changing β we get a different characterization of the failure rate:

$$\beta < 1 \implies h(t) \text{ decreasing}$$

$$\beta = 1 \implies h(t) \text{ constant}$$

$$\beta > 1 \implies h(t) \text{ increasing}$$

Observe that the exponential distribution can be seen as a Weibull with $\beta = 1$. The ability to represent various behaviors of the failure rate is the major appeal of the Weibull distribution for dependability modeling.

2.2 Dependability of Repairable Components: Availability

We now consider the case of a component that, once broken, can be repaired. The behavior of a repairable component over time is therefore determined not only by the way in which it fails, but also by the way in which it is repaired, and we can consider the life of a system as an alternation between two states: Up (system is working) and Down (system is not working and it is under repair).

We assume that the intervals of correct functioning (time to failure), and the intervals of incorrect functioning (time to repair) are described by random variables. Let $\tau_1, \tau_2, \tau_3, \dots$ be the random variables of the successive duration of the up times, and $\theta_1, \theta_2, \theta_3, \dots$ the associated repair times, under the hypothesis that the repair is “regenerative”, that is to say that after the repair the component is “good as new”, then all the τ_i have the same distribution $F(t)$, and all the θ_i have the same distribution $G(t)$, and we can describe the behavior of the system with only two random variables τ , duration of the Up times, and θ , duration of the Down times. $G(t)$ represents the probability that the component is repaired in $[0, t]$, and it is called Maintainability. Similarly to what we have done for $F(t)$, we get for $G(t)$ the following expressions:

$$\begin{aligned} g(t) &= \frac{dG(t)}{dt} \\ h_g(t) &= \frac{g(t)}{1 - G(t)} \\ MTTR &= \int_0^\infty t g(t) dt \end{aligned}$$

where MTTR is the Mean Time To Repair, and $h_g(t)$, the repair rate, is the probability that the repair is terminated in the interval $[t, t + dt]$, given that the component was still unrepaired at time t . If we assume that the repair rate $h_g(t)$ is time-independent, $h_g(t) = \text{cost} = \mu$, then the maintainability is an exponential function:

$$G(t) = 1 - e^{-\mu t} \quad \text{and} \quad MTTR = \frac{1}{\mu} \quad (10)$$

The assumption of time-independence is not very realistic since, in general, the time it takes to finish a repair does depend on how long the repair has already taken, but this assumption is nevertheless often taken in the literature and in the practice, for the advantages that it offers in the solution process.

It is clear that if a system is subject to failures and repairs, the reliability function $R(t)$ is not particularly informative since for any t greater than the time of the first failure, the value of $R(t)$ is always going to be zero.

A new quantity is therefore defined, and it is called *availability*, indicated as $A(t)$. $A(t)$ is the probability that the system is Up at time t .

$$A(t) = \text{Prob}\{\text{at time } t, \text{state} = \mathbf{Up}\} \quad (11)$$

The *unavailability* $U(t)$ is instead the probability that the system is Down at time t .

$$U(t) = \text{Prob}\{\text{at time } t, \text{state} = \mathbf{Down}\} \quad (12)$$

and, since we assume that the system is either Up or Down, we have:

$$A(t) + U(t) = 1$$

The computation of $A(t)$ and $U(t)$ of a system starts from the observation that $A(t)$ ($U(t)$) is equivalent to the probability of being in the Up (Down) state at time t . But the probability of being in the Up state can be computed writing equilibrium equation, since the variation over time of the probability of being in state Up is equal to the difference between the probability of entering the Up state and the probability of leaving the state, that, assuming a fixed failure (repair) rate equal to λ (μ), amounts to the following equations:

$$\begin{cases} \frac{dA(t)}{dt} = -\lambda A(t) + \mu U(t) \\ \frac{dU(t)}{dt} = \lambda A(t) - \mu U(t) \end{cases} \quad (13)$$

Assuming that at time 0 the system is working properly, we can set $A(0) = 1$, we can solve the equations (13), and obtain:

$$\begin{aligned} A(t) &= \frac{\mu}{\lambda + \mu} + \frac{\lambda}{\lambda + \mu} e^{-(\lambda + \mu)t} \\ U(t) &= \frac{\lambda}{\lambda + \mu} - \frac{\lambda}{\lambda + \mu} e^{-(\lambda + \mu)t} \end{aligned} \quad (14)$$

And we get:

$$A(0) = 1 \quad ; \quad \lim_{t \rightarrow \infty} A(t) = A_{\infty} = \frac{\mu}{(\lambda + \mu)} \quad (15)$$

The typical shape of the availability function is made up of a transient term that exhibits an exponential decay, and a time independent term that constitute the horizontal asymptote.

Since in a repairable system $MTTF \gg MTTR$, and hence $\lambda \ll \mu$, the contribution of the transient term decays very quickly, and therefore the availability is often identified by its asymptotic behavior in (15).

If A_{∞} is the asymptotic availability, we can write

$$A_{\infty} = \frac{\mu}{\lambda + \mu} = \frac{1/\lambda}{1/\lambda + 1/\mu} = \frac{MTTF}{MTTF + MTTR} \quad (16)$$

Although the above expression has been derived under the hypothesis of constant failure and repair rate, it has been proven [24] that it holds for any distribution $F(t)$ and $G(t)$, given that $MTTF$ is the mean value of $F(t)$ and $MTTR$ is the mean value of $G(t)$.

3 Combinatorial Methods for System Dependability

Assuming that we are able to characterize the failure and repair distribution of a component, we have seen how to predict its availability. But if a system is a complex aggregate

of components, it may be difficult to associate directly to the system a failure and repair distribution. As usual in computer science, when a problem is too complex, a divide and conquer technique may lead to viable solution, and this is indeed the approach that we shall discuss next: given a number of independent components, and a well defined way of combining them into a *system configuration*, we shall see how to compute the reliability (availability) of a (repairable) system. We present two techniques: *reliability blocks* [60] and *fault trees* [59].

3.1 Reliability of Non-repairable Systems

In reliability blocks we assume that the system under study is built out of two basic configuration schemes: series configuration and parallel configuration.

A system is the series configuration of two components if the failure of one of them provokes the failure of the whole system. If we assume that the components are statistically independent, and letting $R_1(t), R_2(t), \dots, R_n(t)$ be the reliability of the n components at time t , then the reliability of the system at time t , $R_s(t)$ is [60]:

$$R_s(t) = R_1(t) \cdot R_2(t) \cdots R_n(t) \quad (17)$$

The reliability of the system is the product of the reliability of the components, and since the $R_i(t)$ are positive values less than 1, it implies that more components we have in a series the less reliability we have.

If the failure rate of the components is constant (and therefore the $R_i(t)$ are exponential distributions of parameter λ_i), we derive from (17):

$$R_s(t) = e^{-\lambda_s t} \quad \text{with :} \quad \lambda_s = \sum_{i=1}^n \lambda_i \quad \text{and} \quad MTTF = \frac{1}{\lambda_s} \quad (18)$$

meaning that the whole system fails according to an exponential distribution.

If we want to build a more robust system, we can use the concept of parallel redundancy: the system is built out of n components, and the whole system fails only if all the components fail. Again, assuming that the components are independent, and for the simple case of $n = 2$, we can state that the unreliability $F_s(t)$ can be computed as:

$$F_s(t) = F_1(t) \cdot F_2(t) \quad (19)$$

from which we derive:

$$R_s(t) = R_1(t) + R_2(t) - R_1(t) \cdot R_2(t) \quad (20)$$

Equation (20) implies that the reliability of the system is greater than the maximum reliability of its components. In the case of constant failure rate we get:

$$R_s(t) = e^{-\lambda_1 t} + e^{-\lambda_2 t} - e^{-(\lambda_1 + \lambda_2)t} \quad \text{e} \quad MTTF = \frac{1}{\lambda_1} + \frac{1}{\lambda_2} - \frac{1}{\lambda_1 + \lambda_2} \quad (21)$$

The expression for $F_s(t)$ of (19) can be generalized to the n case, leading to:

$$[1 - R_s(t)] = \prod_{i=1}^n [1 - R_i(t)] \quad (22)$$

from which $R_s(t)$ can be computed.

The equations above allow the computation of the reliability function of systems that have a complex parallel-series composition scheme, indeed each $R_i(t)$ could be the reliability of a complex subsystem, that can be expressed with a formula that depends on whether the subsystem is the serial or parallel composition of sub-subsystems, and so on.

A special type of parallel redundancy is the so-called k out of n (indicated as $k:n$): the system has n redundant components placed in parallel, and the system works fine when at least k out of the n components works properly. An example of such a system can be a redundant water pipeline in which the required water flow is ensured as far as k pipelines are correctly transporting their flows.

The probability that exactly i components out of n work fine, in the hypothesis that all components have the same failure rate R , is given by the binomial distribution:

$$Pr\{i : n\} = \binom{n}{i} R^i (1 - R)^{(n-i)} \quad (23)$$

since the system is working in all cases in which i components, with $i > k$, are correctly working, we get:

$$R_{k:n} = \sum_{i=k}^n \binom{n}{i} R^i (1 - R)^{(n-i)} \quad (24)$$

3.2 Availability of Repairable Systems

Let us now consider the case of a system built out of *repairable* components. Again we shall consider the failure process and the repair process of the components totally independent: this hypothesis is less realistic than in the case of a single component, since it is often the case that there is a limited number of resources allocated for repairs, shared among all components, so that the computed availability may be an upper bond of the real value.

If there are n components configured in series, then the availability of the system A_s is given by:

$$A_s = A_1 \cdot A_2 \cdots A_n$$

where the A_1, A_2, \dots, A_n and A_s are evaluated for the same time instant t , or at infinity. If the constant rate hypothesis applies, then the first equation in (14) can be used.

Again, the availability of the system is smaller than the availability of the worst component, and therefore to increase the availability of a system it is advisable to act on the worst component.

For what concerns the redundant parallel systems, considering the simplifying hypothesis of two components, we have that the unavailability U_s is given by:

$$U_s = U_1 \cdot U_2 \quad (25)$$

from which we can derive

$$A_s = (1 - U_s) = A_1 + A_2 - A_1 \cdot A_2 \quad (26)$$

showing that the availability of the system is greater than the availability of the most available component.

3.3 Fault Trees

Another approach to the combinatorial study of systems built out of independent components is that of *fault trees* [59]: logical trees for the representation and analysis of the critical conditions whose combined occurrence causes a specific event, called the *Top Event (TE)*, to occur.

When the TE is one particular undesired event, then the analysis of the combination of elementary events that lead to the occurrence of the TE assumes the name of *Fault-tree analysis (FTA)*. Elementary events are of the type: component X is working properly, or component X is not working properly.

The TE is the root of the tree and the construction of the tree is usually “*top / down*” from general to specific. The FTA is particularly suited to the analysis of complex systems comprising several subsystems or components which are connected in various configurations, with a high level of redundancy. FTA is commonly used by reliability engineers dealing with aircraft, space, chemical and nuclear systems, and it is also considered in the IEC-1025 standard [36]. The interested reader can find a full treatment of the topic in [3, 31, 32, 22, 59].

The methodological approach to dependability based on FTA consists of the following steps: definition of the Top Event, construction of the Fault Tree, qualitative analysis, and qualitative analysis.

Definition of the Top Event. TE candidates are events whose occurrence may lead to unsafe operating conditions, catastrophic failure or malfunction, unaccomplishment of the assigned mission, and so on.

If more TE's need to be investigated a different tree for each one the TE's must be generated and analyzed.

Construction of the fault-tree. Once the TE has been defined, the construction of the FT proceeds by identifying the *immediate causes* for the occurrence of the TE, and their logical relationship (for example, whether the immediate causes must occur separately or simultaneously for the TE to occur).

The immediate, necessary and sufficient causes for the TE constitute the first level of the tree. Each immediate cause is now treated as a sub-top event, and the analysis proceeds to determine their immediate causes. In this way, the construction evolves iteratively from events to their causes, continuously approaching finer resolution, until a desired level of detail is reached.

Interactions between causes at each level of the iterative construction are represented by means of logic gates (usually OR and AND gates, but more complex gates can be defined, as, for example, k out of n gates), while the output of the logical gates represents the occurrence of the higher level of the tree. The events at which the construction of the tree is ended are called *terminal events*.

Qualitative Analysis of a FT. The qualitative analysis is aimed at identifying all the combinations of events that cause the top event to occur, as a function of the terminal events. Combinations are ranked according to the number of events, since the smaller the number of events that cause the TE the less resilient to failure it is likely to be our

system. The qualitative analysis of an FT consists in deriving a logical expression of the TE, in such a way that all the combinations of events whose simultaneous occurrence provokes the TE are evidenced. A combination of events whose simultaneous occurrence provokes the TE is called a *cut set* (CS) for the system. A CS that does not contain any subset which is again a CS, is minimal and is called a *minimal cut set* (MCS) or *mincut*.

Definition. A CS is a set of terminal events whose simultaneous occurrence forces the occurrence of the TE. A CS is an MCS if it does not contain any subset of terminal events that is still a CS.

Suppose the FT has m MCS denoted by K_1, K_2, \dots, K_m . According to the above definition, the occurrence of any K_i ($i = 1, 2, \dots, m$) implies the occurrence of the TE, hence:

$$TE = K_1 \text{ or } K_2 \text{ or } \dots \text{ or } K_m = OR_{i=1}^m K_i \quad (27)$$

The list of all the MCS provides a very valuable information to the analyst since it provides all the minimal sets of failure events that can provoke the TE to occur, that is the system failure event, and allows the analysts to identify the potential weak points of the system and to initiate corrective actions.

The determination of the CS proceeds iteratively in a top down fashion, starting from the TE and applying the rules of the logic algebra, guided by the gate typology, until all the terminal nodes are reached. If the FT does not contain repeated events, the above search directly provides the MCS, otherwise if the FT contains repeated events the list of the MCS must be further extracted from the obtained CS.

Quantitative Analysis. The quantitative analysis has the objective of evaluating the probability of occurrence of the TE, of the MCS and of any other intermediate event of the FT in terms of the probability of occurrence of the basic events. FTA assumes that the failures of the basic components are statistically independent. According to this assumption, the properties of the FT are completely specified if an individual probability is assigned to each single basic event.

Let A_i be a terminal event and denote $Q_i = P(\bar{A}_i)$, where \bar{A}_i stays for “component A_i not working”. To compute the probability values in either cases a mission time must be fixed. Denote the mission time T_M , hence

$$Q_i = P(\bar{A}_i, T_M) \quad (28)$$

If a component is non-repairable, then Q_i is the component unreliability computed at time T_M , if it is repairable Q_i is the component unavailability computed at time T_M .

Many FT tools accept as input parameter for each basic even only a constant failure and a constant repair rate, thus implicitly assuming that the failure and repair times of each component are exponentially distributed and restricting the analysis to this case, only. By denoting with λ_i and μ_i , respectively, the constant failure and a repair rate of component A_i , formula (28) becomes [3, 64]:

$$Q_i = \frac{\lambda_i}{\lambda_i + \mu_i} (1 - e^{-(\lambda_i + \mu_i) T_M}) \quad (29)$$

from which the value for a non-repairable component (the usual unreliability expression for a component with constant failure rate) is obtained by setting $\mu_i = 0$.

Probability of the TE. Since the TE can always be expressed in disjunctive normal form in terms of the MCS, then:

$$P(TE) = P(K_1 + K_2 + \dots, K_m)$$

where each CS K is the AND of a number (called the CSorder) of terminal events, then

$$K = \mathcal{A}_1 \mathcal{A}_2 \dots \mathcal{A}_\ell$$

and

$$Pr(K) = Pr(\mathcal{A}_1) Pr(\mathcal{A}_2) \dots Pr(\mathcal{A}_\ell) \quad (30)$$

If there are m MCS's (K_1, K_2, \dots, K_m), since the occurrence of a single MCS implies the TE, then the probability of TE is given by

$$Pr(TE) = Pr(K_1 + K_2 + \dots + K_m) \quad (31)$$

Recall that, if A and B are two events, then

$$Pr(\mathcal{A} \text{ or } \mathcal{B}) = Pr(\mathcal{A}) + Pr(\mathcal{B}) - Pr(\mathcal{A} \mathcal{B}) \quad (32)$$

and that the OR of m events requires an expansion into $(2^m - 1)$ terms involving the computation of the probability of the AND of groups of j events, ($j = 1, 2, \dots, m$).

The computation of the probability of the TE can therefore be quite complex, but we should consider that the probability of the single events are in general quite low (they are failure probabilities), and that all terms that compute the product of a significant number of basic events can be quite low. It is therefore a widely accepted practice to truncate the computation, especially considering that upper and lower bounds on $Pr(TE)$ can be computed using the probability of the single CS's and the probability of the AND of pair of CS's, as follows.

$$Pr(TE) \leq \sum_i Pr(K_i) \quad (33)$$

$$Pr(TE) \geq \sum_i Pr(K_i) - \sum_{i>j} Pr(K_i K_j) \quad (34)$$

The computation of the probability of occurrence of the TE and of the MCS is, usually, the main concern of a FTA. However, several other useful measures can be defined and evaluated, like the expected number of failed components, the main failure equivalence, and the Mean Time To Failure.

FTA is a widespread practice for the availability analysis of systems, and there are indeed a number of tools that support it, among them we cite [55, 62].

4 State Enumeration Techniques

The work of the previous section is here extended to consider systems of independent components but with arbitrary configurations, and systems in which the independence assumption does not hold. Both techniques are based on the idea of enumerating all the

possible states of the system, and to classify them as “good” or “bad”. The probability of being in a state (at time t or in steady-state) is then computed, and the reliability of the system is then obtained summing up the probability of all the good states.

If the components are independent we can provide a closed form expression for the probability of each state, thus adopting a combinatorial approach, as for reliability blocks, while if dependencies among components are to be taken into account, more complex quantitative analysis techniques need to be considered, based on the solution of the associated stochastic process.

In this section we shall first consider the problem of state enumeration, to then separately discuss the quantitative analysis for the independent case and for the dependent one, limited to the simpler case in which the associated stochastic process is a Markov chain.

Consider a system with n components and any configuration among them. The usual hypothesis is to assume that each single component can be represented by two mutually exclusive conditions or states referred to as *working* (or *Up*) and *failed* (or *Down*), identified by the state indicator variable x_i associated to the i -th component, with the following encoding:

$$x_i = \begin{cases} 1 & \text{component } i \text{ Up} \\ 0 & \text{component } i \text{ Down} \end{cases}$$

The state of the system is identified by the vector $\mathbf{x} = (x_1, x_2, \dots, x_n)$ [3]. The state space of the system Ω is the set of all the possible values of \mathbf{x} , i.e. the set of all the possible combinations of the n components being working or failed, leading to $N = |\Omega| = 2^n$.

The state space, that we shall call RG for similarity with Petri nets terminology, is a labeled directed graph whose nodes are the states of the system and each edge represents the transition between states due to failure or repair. If we assume that no multiple failures or repairs can take place at the same time, which is indeed the case for independent components working in continuous time, then there is a direct arc labelled i between state \mathbf{x} and \mathbf{x}' only if the two states differ only in the value of variable x_i .

We assume that the system as a whole can be classified according to a binary behavior: *working* or *failed*. Hence, we introduce a binary indicator variable y for the system [3, 41]:

$$y = \begin{cases} 1 & \text{system is in a working state} \\ 0 & \text{system is in a failed state} \end{cases} \quad (35)$$

The value of y is a function of the state, and we can define $y = \varphi(\mathbf{x})$, or $y = \varphi(x_1, x_2, \dots, x_n)$

The state space Ω can be partitioned in two exhaustive and mutually exclusive subsets Ω_u and Ω_d .

$$\Omega_u = \{\Omega : \varphi(\mathbf{x}) = 1\} \quad ; \quad \Omega_d = \{\Omega : \varphi(\mathbf{x}) = 0\}$$

Let $N_u = |\Omega_u|$ be the cardinality of Ω_u , and $N_d = |\Omega_d|$ the cardinality of Ω_d , then

$$\Omega = \Omega_u \cup \Omega_d \quad ; \quad \Omega_u \cap \Omega_d = \emptyset \quad ; \quad N = N_u + N_d \quad (36)$$

Observe that the system configuration is totally identified by the function $y = \varphi(\mathbf{x})$. For example, in the 2-parallel connection configuration the only system failed state is $\mathbf{x} = (0, 0)$, that is to say, both components have to be down for the whole system to fail.

The failure process defined on the state space. The evolution of the system in time can be represented by means of the succession of states passed through by the system due to failure or repair events of its components.

Denoting by $Z(t)$ the function of the time that represents the state occupied by the system at time t . For any value of t , $Z(t)$ is a random variable that assumes non negative values in the states of Ω . The probability that the system is in state \mathbf{x} at time t is denoted by $p_{\mathbf{x}}(t)$ and is defined as:

$$p_{\mathbf{x}}(t) = Pr\{Z(t) = \mathbf{x}\} \quad (37)$$

under the normalization condition:

$$\sum_{\mathbf{x} \in \Omega} p_{\mathbf{x}}(t) = 1 \quad \text{for any } t \geq 0.$$

On the failure process the following measures can be defined:

Reliability: since there are many states (all whose in Ω_u) in which the system is considered as working properly, then the Reliability of the system is obtained summing up the probability of each of these states, leading to:

$$R_S(t) = \sum_{\mathbf{x} \in \Omega_u} p_{\mathbf{x}}(t) \quad (38)$$

Availability:

$$A(t) = \sum_{\mathbf{x} \in \Omega_u} p_{\mathbf{x}}(t) \quad (39)$$

Mean sojourn time spent in a state up to time t :

$$\ell_{\mathbf{x}}(t) = \int_0^t p_{\mathbf{x}}(z) dz \quad (40)$$

System MTTF

$$MTTF = \int_0^\infty R_S(z) dz = \sum_{\mathbf{x} \in \Omega_u} \int_0^\infty p_{\mathbf{x}}(z) dz = \lim_{t \rightarrow \infty} \sum_{\mathbf{x} \in \Omega_u} \ell_i(t)$$

Average interval availability:

$$A_I(t) = \frac{1}{t} \sum_{\mathbf{x} \in \Omega_b} \ell_i(t)$$

$Z(t)$ is a stochastic process defined over the discrete state space Ω and with continuous time parameter t . The quantitative evaluation of the state probabilities expressed by equation (37), completely determines the stochastic process $Z(t)$ and, hence, the behavior of the system. If the components are statistically independent, evaluation of expression (37) can be performed by resorting to combinatorial formulas, presented in the following subsection, while the case of statistically dependent components will be treated next.

4.1 Independent Components

If the components of a system are statistically independent, the probability $p_{\mathbf{x}}(t)$ of being in a generic state \mathbf{x} with characteristic vector $\mathbf{x} = (x_1, x_2, \dots, x_n)$ at time t can be expressed as the product of the probabilities of the individual variables:

$$p_{\mathbf{x}}(t) = Pr\{(x_1(t))\} \cdot Pr\{(x_2(t))\} \cdots Pr\{(x_n(t))\} \quad (41)$$

where, thanks to the independent component assumption, each term $Pr\{(x_i(t))\}$ is given by:

$$\begin{cases} Pr\{x_i(t) = 1\} = R_i(t) \\ Pr\{x_i(t) = 0\} = 1 - R_i(t) \end{cases} \quad (42)$$

where $R_i(t)$ is the probability that component i is in working condition at time t , and coincides with the reliability of component i in case of non-repairable components or with the availability of component i in case of repairable components.

In the usual case in which the time to failure distribution of each individual component is considered exponentially distributed with failure rate λ_i , equation (42) takes the form:

$$Pr\{x_i(t)\} = \begin{cases} R_i(t) = e^{-\lambda_i t} & \text{if } x_i(t) = 1 \\ 1 - R_i(t) = 1 - e^{-\lambda_i t} & \text{if } x_i(t) = 0 \end{cases} \quad (43)$$

4.2 Markovian Methods for Dependent Components

The analysis above relies on the hypothesis that the components are independent, but this is not always the case. In the previous section we have shown how state enumeration techniques can be used to cheaply compute the reliability of a system built out of independent components. The hypothesis of independence is not always reasonable, for example the failure of a component may induce a larger load on the remaining components, thus increasing their failure rate, or two or more components have a common cause of failure (for example a computer and a video can be seen as independent, but if they use the same source of power their failure are not independent). If the components are statistically dependent, i.e. the failure or repair process of any one of them is dependent on the state of the other(s), more sophisticated techniques are necessary, able to incorporate the conditional dependencies of each component with respect to the state of the other ones.

Consider a system with two components whose state space is the Cartesian product of the state spaces of the components and is depicted in Figure 1; if $\lambda_1 \neq \lambda'_1$ then the failure rate of the first component depends on the state of the second component (for example the failure rate of the first component increases if the second component is not working properly). This dependency does not allow to compute the reliability of the system in the simple product form of Equation (41), and we need to solve the associated stochastic process. Continuous Time Markov Chains (CTMC) are stochastic process with a good tradeoff between expressiveness and solution cost.

A very large literature exists on the topic, the interested reader may refer, for example, to [51, 25, 42, 64]. Reliability analysis through CTMC is also dealt with in the international standard IEC1165 [37].

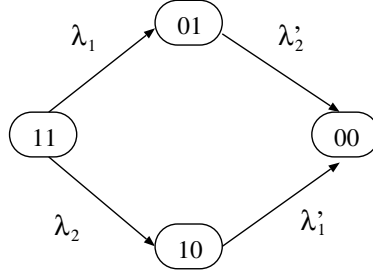


Fig. 1. The state space of a two components system

Let $Z(t)$ be a stochastic process defined over the discrete state space Ω . $Z(t)$ is a Continuous Time Markov Chain (CTMC) [51, 64] if, given any ordered sequence of time instants $(0 < t_1 < t_2 < \dots < t_m)$, the probability of being in state $\mathbf{x}^{(m)}$ at time t_m depends only on the state occupied by the system at the previous instant of time t_{m-1} and not on the complete sequence of state occupancies. This property, that is usually referred to as the *Markov property*, can be rephrased by saying that the future evolution of the process only depends on the present state and not on the past. More formally the Markov property may be written as:

$$\begin{aligned}
 &P\{Z(t_m) = \mathbf{x}^{(m)} | Z(t_{m-1}) = \mathbf{x}^{(m-1)}, \dots, Z(t_1) = \mathbf{x}^{(1)}\} \\
 &= P\{Z(t_m) = \mathbf{x}^{(m)} | Z(t_{m-1}) = \mathbf{x}^{(m-1)}\}
 \end{aligned} \tag{44}$$

If we number the states from 1 to N , for example taking the lessicographical order, then we can define the *transition probability matrix* of the process:

$$\mathbf{P}(u, v) = [p_{ij}(u, v)]$$

of dimension $(N \times N)$ whose entries are the transition probabilities $p_{ij}(u, v)$ defined as:

$$p_{ij}(u, v) = P\{Z(v) = j | Z(u) = i\} \quad (u \leq v) \tag{45}$$

$p_{ij}(u, v)$ represents the probability that the Markov chain $Z(t)$ is in state j at time v , given it was in state i at time u , and it is called the *transition probability* between state i and j . For the transition probabilities $p_{ij}(u, v)$, the following initial conditions hold:

$$p_{ii}(v, v) = 1 \quad ; \quad p_{ij}(v, v) = 0 \tag{46}$$

Further, let $p_i(t)$ be the (unconditional) probability that the system is in state i at time t . $p_i(t)$ is the state occupancy probability, or simply the *state probability*, and is defined as:

$$p_i(t) = Pr\{Z(t) = i\} \tag{47}$$

and $\mathbf{p}(t) = [p_i(t)]$ denotes the row vector of dimension $(1 \times N)$ whose entries are the state probabilities $p_i(t)$ defined in (47). $\mathbf{p}(t)$ is called the *state probability vector* of the process.

If the Markov process is homogeneous¹ we can derive the following equation:

$$\frac{d\mathbf{p}(t)}{dt} = \mathbf{p}(t) \cdot \mathbf{Q} \quad \text{with initial condition} \quad \mathbf{p}(0) = \mathbf{p}_0 \quad (48)$$

where \mathbf{p}_0 is a probability vector describing the initial conditions, and \mathbf{Q} is a matrix, called infinitesimal generator, whose elements q_{ij} are the conditional probabilities of jumping in state j in a small interval Δt , given that the CTMC was in state i at the beginning of the interval. The quantities q_{ij} are called the transition rates of the process, and they are a simple function of the p_{ij} values.

Equation (48) is the fundamental equation for CTMC: it consists of a set of N first order differential equations with constant coefficients, that provide the state occupancy probabilities at time t , from which the required performance models can be computed. Various analytical and numerical solution techniques are available for the fundamental CTMC Equation ([61]).

If the Markov chain is irreducible, that is to say each state is reachable from any other state, then the limit

$$\lim_{t \rightarrow \infty} p_i(t) = \pi_i$$

always exists and it is independent of the initial state; equation 48, when t goes to infinity, simplifies to:

$$\pi \cdot \mathbf{Q} = \mathbf{0} \quad \text{with} \quad \sum_{i=1}^N \pi_i = 1 \quad (49)$$

where the normalizing condition on the right is necessary to impose that the solution is a probability vector. Equation (49) is a linear set of homogeneous equations, and can be solved with numerical solution techniques [61].

Observe that, if the components are non-repairable, then the associated MC will never be irreducible (from a state in which there is a failed component is never possible to come back to the initial state in which all components are working properly), and therefore the only reasonable measures to be computed are the probabilities at time t , and derived quantities.

From what concerns complexity, we can observe that most techniques used for the solution “at time t ” or in steady state are based on iterative methods: at each iteration the most expensive operation is a vector-matrix multiplication (vector of size equal to the number of states and matrix with a number of non-null elements equal to the number of arcs in the RG). The iteration procedure is stopped when a certain (estimated) convergence towards the actual solution is reached.

The number of iteration is usually the major factor affecting the solution cost.

5 Dependability Modeling Using Petri Nets

Specifying systems at the state space level can be an error-prone, low level activity. Petri nets have been widely recognized in the literature as an effective way to specify systems

¹ A Markov process is said to be homogeneous when the transition probabilities in matrix $\mathbf{P}(u, v)$ depend only on the length time interval $(v - u)$ and not on the values of the time instants v and u .

using a reasonably high-level formalism, while at the same time having a precise operational semantics that allows the derivation of the associated state-space. In particular the class of Stochastic Petri Nets [47, 2] (SPN) has a semantics defined through Markov chains, so they are considered a natural language to use when the stochastic process underlying the system is a Markov chain. With the term SPN we actually indicate the more general class of Petri nets with stochastic delays associated to transitions, that include various extensions like ESPN [28], GSPN [1], Stochastic Reward nets [48], and colored/parametric extensions like Stochastic Activity Networks [57], and Stochastic Well-formed Nets [18].

Indeed SPN have been widely used not only for the study of generic performance indices, but also specifically for dependability studies, as testified by the available SPN tools that allow the computation of some pre-defined dependability quantities, like SURF-2 [30], UltraSAN [23], and SPNP [20]. Among the initial works on the use of SPNs for dependability modeling and analysis we can mention [33, 35, 48, 58, 39, 46, 14]. In [29] Extended SPNs are used for carrying out sensitivity analysis of the system reliability and availability when the error coverage probability varies. The work [49] presents an overview of different classes of non Markovian Petri Nets used for dependability analysis. Net-compositionality has been adopted in [40, 56, 16, 11] to cope with the complexity of dependability modeling. Concerning works on the translation of Fault Trees into SPNs we mention the works [34, 45, 15]. In [34] and in [45] the translation into Petri Nets allows to model the dependences among system components, while in [15] SWNs are used as a target formalism of the translation to exploit the symmetries of the parametric fault trees.

But what do we gain from the use of Petri nets?

- An high level language to describe the system.
- The possibility of reusing the tools and the solution methods available for the SPN tools, including the possibility of validating qualitative properties (like liveness or deadlock freeness) that can be an important issue when the system being modeled as a complex behavior.

What do we loose by using SPN? The solution associated to an SPN is usually produced solving the associated CTMC, but the CTMC is, in principle, a “flat” structure, in which all the information on independency between components is basically lost, thus forcing the solution of the whole Markov chain with numerical methods, even in whose cases, like that of independent components, in which the solution is just the product of the solution of the components.

The section is organized as follows: we shall first show a very simple model, equivalent to a 2-parallel configuration, and then a slightly more complex case, in which a model of a system is modified to include the presence of faults. We conclude the section with the presentation of a (simplified version) of a case study of a “dependability mechanisms”.

5.1 Simple PN Models of Dependability

The very simple SPN model of Figure 2 represents a system with two independent components. Each SPN is a simple sequence place - transition - place, where the tran-

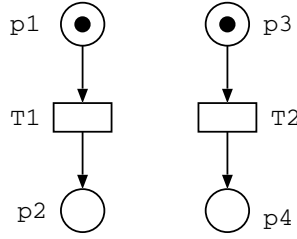


Fig. 2. A simple SPN model of a two component system

sition represents the failure. The Markov chain of this SPN is exactly that of Figure 1 if $\lambda_i = \lambda'_i$ and it is equal to the rate of transition T_i . If $\lambda_i \neq \lambda'_i$ then it is necessary to resort to marking dependent rates.

How do we compute the reliability/unreliability of the system? Again, as for CTMC based approaches, we can sum up the probability of the Up and Down states, where the Up and Down states depend on the system configuration. But how can we specify a system configuration? There are two possible approaches: an implicit one, in which it is the definition of the measure that encodes the configuration, and an explicit one, in which the configuration is reported in the net.

If we assume the implicit approach, and we want to express a series configuration, then to compute the unreliability of the system we have to sum up the probability of all states in which either place p_2 is marked, or place p_4 is marked, or both. For the parallel configuration we sum over a single state: that with a token each in places p_2 and p_4 .

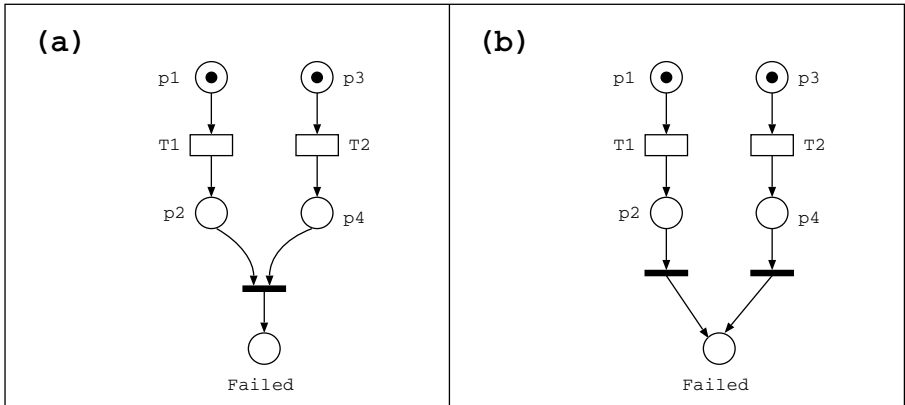


Fig. 3. Explicit modeling of the failure state

If we want to make the configuration explicit we can add a place *Failed*, as it is done in Figure 3. The place is connected to places p_2 and p_4 in different manners, to reflect the parallel configuration in Figure 3(a), and the series one in Figure 3(b).

The simple SPN model of Figure 2 can be modified so as to take into account repairs, leading to the SPN model of Figure 4 where one repair transition per component (named T_3 and T_4) has been added. Again, we can define the configuration in an implicit manner through the definition of the availability/unavailability metrics, or we can explicitly define the configuration in the model. Indeed the modification of the net is not as simple as in the unrepairable components case, since we need to put a token in place *Failed* without removing the tokens from the places P_2 and P_4 , moreover we need to model also the fact that, in consequence of a component repair, the whole system can be working again, and this may not be trivial.

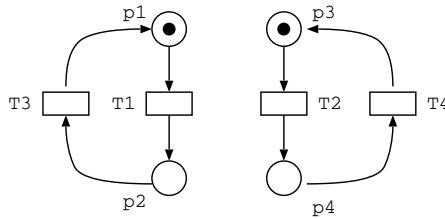


Fig. 4. Modeling of repair actions

The approach based on reliability blocks can be translated into SPN with a limited effort, nevertheless since the SPN solution requires the solution of the Markov chain, then this translation makes sense only if we need to insert dependencies between the components (for example a simple failure rate dependency).

A similar argument holds true also for fault trees. There has been a number of translation defined from fault trees of various flavors to various flavors of SPN, always with the objective of being able to include dependencies [33, 45], or to exploit symmetries of the fault tree also in the solution process [15].

In the previous SPN models we have assumed that a component has only two states, Up and Down, represented as distinct places, but a model can be a much more complicated net. In the next section we shall introduce a simplified version of the model of a piece of software that provides a sort of parallel redundancy. Although being a simplified version of the real code, it has indeed a more articulated structure than the two-states approach discussed in these SPN introductory examples, moreover it is a good example of the problem related to the modeling of the restart of normal operation after a failure.

5.2 A More Complex PN Model of Dependability: The Local Voter

The Local Voter mechanism (LV) presented in this sub-section is a simplified version of a fault-tolerance mechanism designed and implemented within the TIRAN EEC project [17] and studied in [10]. A fault-tolerance mechanism is basically a piece of code aimed at improving the reliability of a complex software system. LV aims at masking occurrences of faults during the execution of the code of an application process.

Fault masking is achieved by the adoption of a spatial redundancy (as in the $k : n$ case that we have seen for reliability blocks) of the execution of the piece of code and by the voting on the results coming from the replicas. Depending on the voting technique adopted in the LV and on the spatial redundancy, a limited number of faults may be masked; for instance, by using a majority voting algorithm and by running concurrently K copies, up to $\lfloor \frac{K-1}{2} \rfloor$ faults can be made transparent for an application process.

In [10] the purpose of the modeling activities was to evaluate the “goodness” of the mechanism both from a qualitative (i.e., correctness with respect to the design specification) and from a quantitative point of view (i.e., performance and dependability). In particular, concerning the quantitative analysis of the LV two important issues were addressed: first, the amount of overhead induced by the use of the mechanism with respect to not to use it in the application execution; second, the probability of voting success.

Observe that the first measure is not a dependability measure in a strict sense, but it is aimed at evaluating the cost of “dealing with faults”, in particular the cost of masking them whenever possible. The second measure can be considered instead a reliability measure, since we can consider as the “assigned mission” of our system the ability to vote on an agreed value.

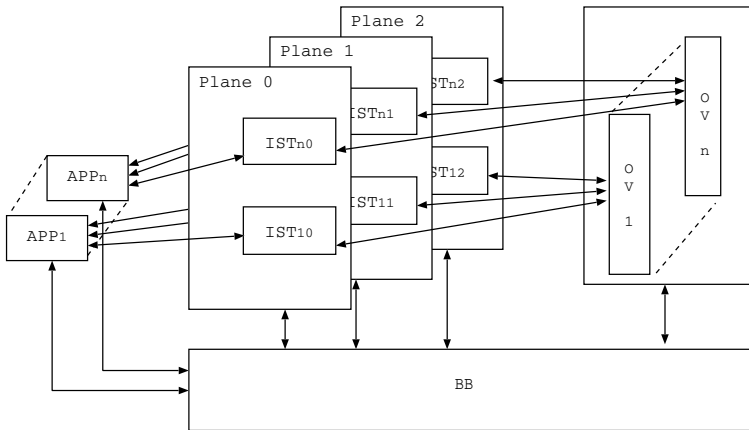


Fig. 5. Representation of the local voter mechanism

Figure 5 shows a graphical representation of the simplified LV; the LV can be used concurrently by several application processes and three replicas are considered per application.

The replicas are executed on separate “planes”, that naturally correspond to separate processing nodes. The application process APP_i that uses the LV mechanism is split in two parts, a part that does not require a replicated execution, and a part that requires it. If there are n applications that use LV, then each application has its distinct piece of code to be executed.

The three replicas of an application i , called IST_{i0} , IST_{i1} and IST_{i2} in the figure, receive the same input data from the application.

When a replica IST_{ij} ends its computation, it sends its output data to the appropriate voting OV_i ; there is one voting task per application.

The components of the local voter interact with the backbone BB, which is a sort of run-time support for the TIRAN library of mechanisms which handles all exceptions as well as the recovery actions. All interactions among tasks are based on communication through mailbox.

Table 1 lists the acronyms used for the different tasks, and for each task lists how many copies of that task there are in a LV that serves n applications.

Table 1. Acronyms

Acr. description	no. of copies
APP application	n
IST replicated software to vote upon	$3 * n$
OV output voter	n
BB backbone	1

Each OV_i is characterized by a timer which is set and starts to count-down for expiration as soon as OV_i receives the first output from one of the replicas of the corresponding application APP_i . If either all the three replicas of APP_i or two of them are received before the time-out expiration then the timer is disabled and a voting on the available replicas is carried out. In any case, OV_i will send a message to the BB to notify the voting outcome on the available replicas and, if it is the case, the time-out expiration. The BB is in charge of notifying the termination of the elaboration to the application and of restarting the system in case of a time-out occurrence.

The SWN Model of the Local Voter. The following assumptions were made to model LV: tasks communicate in an asynchronous manner via mailboxes, and there is one mailbox for each ordered pair of tasks; time required to prepare a message is in general negligible, while the time to actually transmit it from the task output buffer to the recipient mailbox is not. With respect to the graphical representation, we have used cross-lined places to emphasize mailboxes and shadowed boxes to delimit portions of the nets that correspond to “recovery actions”, and that will be explained in a second step. Moreover, we have adopted the SWN syntax of the GreatSPN [5] tool: net objects, i.e., places and transitions, are denoted as *name*|*label* where *name* is the name of the object and *label* is the label, τ labels are omitted. Labels are used for net composition.

Colors have been used to identify applications, planes and to distinguish the output value as “termination with normal operation” or “termination under abnormal conditions”. Three color classes have been defined:

AP is the color class of applications that can request a replicated execution of their code, and it is defined as $AP = \{ap1, \dots, apn\}$;

P is the color of the planes, and there are always three planes, therefore $P = \{pl1, pl2, pl3\}$;

Exc is the color used to distinguish the positive or negative outcome of a LV activity, and it is built out of two static subclasses $Exc = Exc1 \cup Exc2$, where $Exc1 = \{e1\}$ means that there has been a time-out expiration, while $Exc2 = \{e2\}$ means that there was no time-out expiration.

Since the system is specified compositionally, it is a very natural choice to model each component of Figure 5 as an isolated SWN, to then compose them. This approach simplifies the model construction and allows model reuse, but it might make more complex the modeling of whose activities that require the knowledge of the global state, as for the restart activity after a failure.

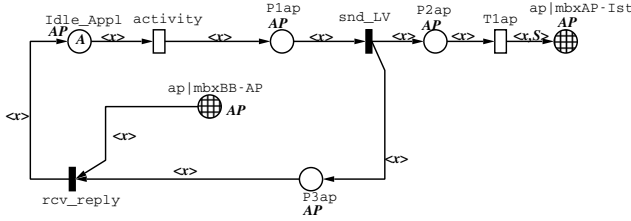


Fig. 6. The application model

Figure 6 shows the SWN model of the APP_i , that cyclically execute their own activity, broadcast the input to their replicas (tasks IST_{ij}), and wait for a message of termination of elaboration coming from the backbone BB.

Figure 7 shows the SWN model of a copy of the code to be executed on the different planes: it is assumed that all replicas are activated at the beginning and then suspend themselves waiting for a message from the APP tasks. There are $|AP| \times |P|$ replicas, i.e., one for each application and for each plane. Each replica (x, y) waits for the input message (x, y) from the application x . When such message is received the replica of application x on plane y starts its activity, modeled by timed transition *comp*, and then sends the result of the computation to OV.

Figure 8 shows the SWN model of the output voter OV: there is an OV for each application that can use LV.

Each OV executes the voting algorithm (majority voting 2 out of 3) on replicas of the same application, independently from the others. OV waits for the replicas outcome from the three different planes. As soon as the first outcome is received, a timeout for reception of the other two replicas outcome is set (transition *setTOforx*). Then three situations may occur:

- C1** all the three outcomes are received before the time-out expiration (transition *recv3noTO* fires) and voting on the three outcomes takes place;
- C2** the time-out has expired and two of the three outcomes have been received (transition *recv2&TO* fires), and a vote on the two replicas takes place;
- C3** the time-out has expired and only one of the three outcomes has been received by OV (transition *recv1&TO* fires).

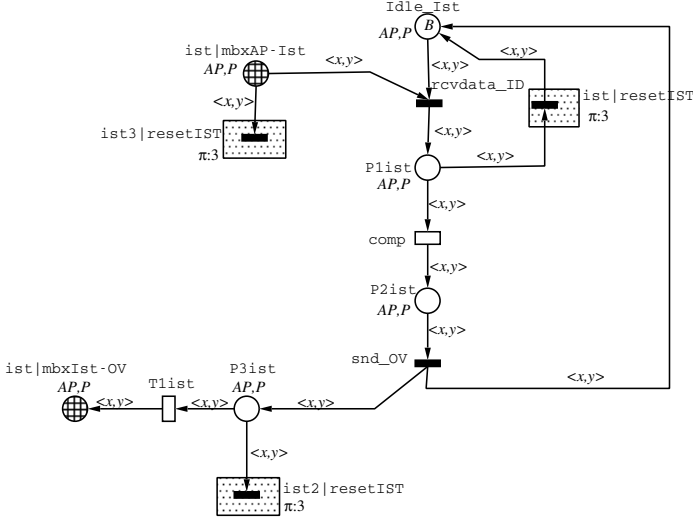


Fig. 7. The model of the replicated code

Under condition C1 an exception message of type $e2$ (no time-out has occurred) is sent to the backbone BB; in cases C2 and C3 a message of exception of type $e1$ (a time-out has occurred) is sent to BB. Observe that we are not passing on to the backbone the information on whether the vote was successful or not, although this will be a trivial extension, since the success or failure of the 2-out-of-3 algorithm is modeled in detail in the SWN of Figure 8.

When the message is sent to BB, OV waits for an acknowledge from BB to return back into its idle state. Observe that we are assuming that no direct answer goes back directly from OV to APP, not even in the case of a “normal” 3-out-of-3 voting, since we impose that all restarts are caused by BB.

Figure 9 shows the SWN model of the BB task, or, more precisely, of that part of BB devoted to interactions with LV. BB is in an idle state until it receives an exception message coming from OV. If the exception is of type $e2$, i.e., no time-out has occurred, then BB sends an acknowledge to OV and to the application. If instead the exception is of type $e1$, then a time-out has occurred, and therefore a reset operation is needed, before sending back the messages to OV and to APP.

Local Voter without Recovery Actions: An Open Model. A first analysis was performed for the case of a “single run” for each application. In order to obtain the complete model the single nets have to be composed using the program *algebra* [10], a program associated to the GreatSPN tool that allows superposition of nets over places and transitions. The nets used are the one without shadowed portion and, since in the non-shadowed portion no message is passed from OV to BB, each application is executed only once. The resulting SWN net has been solved, for the single application case, using the reachability graph construction of GreatSPN, that produces 68 tangible states.

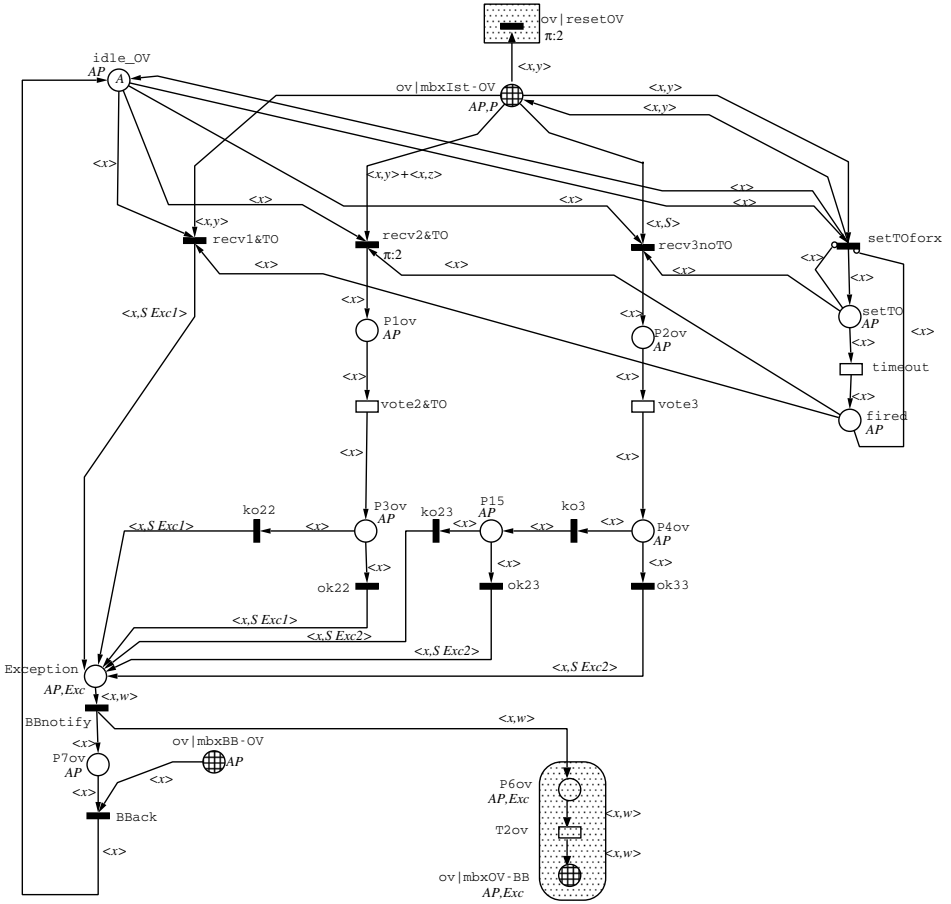


Fig. 8. The model of the output voter

There are 7 dead markings. Three of them correspond to the case of time-out expiration after OV receives the results coming from one replica and is waiting for the results to be sent by the other two replicas. The three markings differ from the identity of the replica that has sent the results to the OV before the time-out expiration. All components, except OV and APP, are in their initial states (idle state), APP and OV are both waiting for a message from BB, that will, of course, never arrives. Three deadlocks correspond to the case of time-out expiration after the results coming from any two replicas have been received. The last deadlock represents the case of reception of all the three replicas before the time-out occurrence. The qualitative behavior was judged correct by the system designer, and the modeling activity could move on to the subsequent step.

Local Voter and Recovery Actions: An Ergodic Model In a second step the recovery actions due to the time-out expiration have been added to the model. The recovery action taken by BB is:

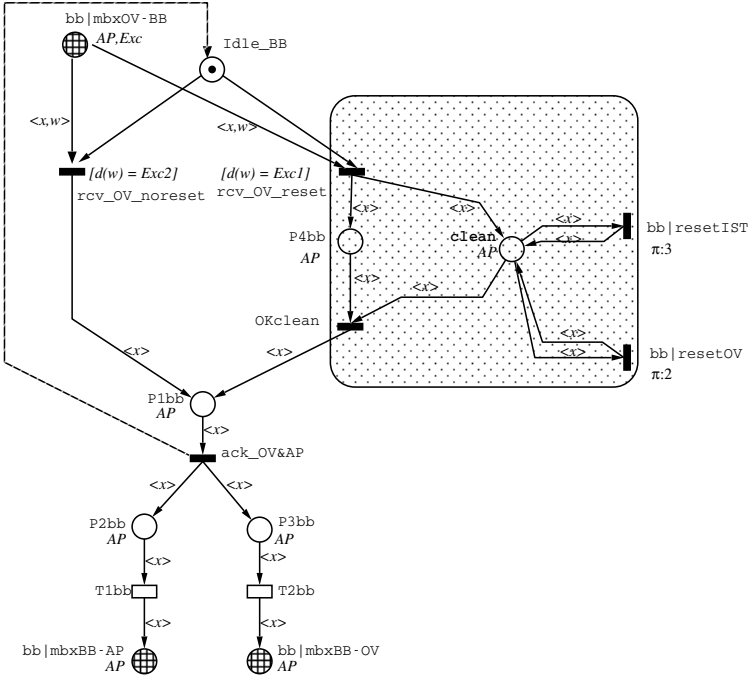


Fig. 9. The model of the backbone

- to remove messages from mailboxes that refer to the application that has caused the exception;
- to take the corresponding tasks back to their initial states.

To accomplish this BB enables a number of immediate transitions, one per model component, and they are labelled in such a way as to superpose with the resetting transitions in the model components. Observe that these transitions are assigned a different priority, mainly to avoid the generation of useless inter leavings of immediate transitions, that could significantly slow down the state space generation.

The model is obtained by composing all nets, including also the shadowed portions. The resulting SWN is ergodic (since there is a single strongly connected component in the reachability graph and only exponential and immediate transitions are present).

The reachability graph for the single application case has 106 tangible states and the initial marking is a home state. The generation takes a few seconds on a 256Mbyte Pentium 4 machine.

Local Voter without Recovery Actions and Explicit Faults In the models considered up to now no fault is explicitly included in the model, so that a time-out can expire only due to a delay in the completion of one of the replicas. In order to consider the effect of explicit faults the model of IST has been modified to include a timing transition that models the fault and that takes IST into an error state place: the modified model is depicted in Figure 10.

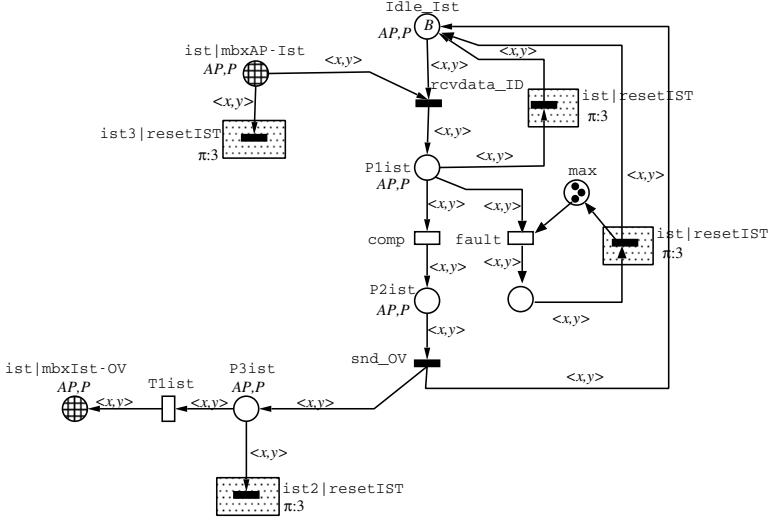


Fig. 10. The modified model of the replicated code

The model assumes that:

- only the replicas IST_{ij} can be affected by a fault and only during their computation phase;
- faults are independent.

The resulting model, for the single application case, has 119 tangible markings and there are 20 dead markings. Among them a very interesting one is the marking that represents the state of the model where all the replicas are in an error state, and this corresponds to a case in which no replica will ever reach OV, so no time-out will be set. This case was, up to the modeling phase, overlooked by the specification document and it is an example of use of Petri Nets for the correctness analysis of the mechanism.

Quantitative Results. The mechanism overhead may be analyzed using the ergodic model with a single application and the time-out deactivated, and computing the mean time to execute a computation through the local voter (inverse of the throughput of transition activity of Figure 6) divided by the mean time spent by a single replica to perform the operation (inverse of the throughput of transition comp of Figure 7).

The probability of different voting outcomes for one application cycle is given by the relative throughput of transitions $recv1\&TO$, $recv2\&TO$, $recv3\&noTO$ of Figure 8. The value of the metric depends on the length of the time-out as well as on the probability of matching of the results produced by the replicas. Observe that, not having explicit faults in the ergodic model, the time-out can expire only due to excessive delays of the computations on the planes and/or of the communications between the model components. This implies that, assuming that the results produced by the replicas always match, the probability of 3-out-of-3 voting converges to one as the length of the

time-out goes to infinity. This behavior cannot be observed, instead, if we consider the effect of explicit faults and we compute the same metric on the third model. A detailed description of the quantitative analysis of the LV can be found in [10].

6 Dependability of Complex Systems Using Petri Nets

A system can be a complex aggregation of components, and the ways in which the up and down states of a component influence the up and down states of the whole system may not be so straightforward as what we have seen in the previous sections. As we have seen in the introduction (Section 1), when the delivered service of a system deviates from fulfilling the system intended function, we say that the system has a *failure*. A failure is due to a deviation from the correct state of the system, known as *error*. Such a deviation is due to a given cause, for instance related to the physical state of the system, or to a bad system design. This cause is called a *fault*.

But if we consider a system as a set of interacting components, which is pretty much in the line with the way systems are designed nowadays, then we should consider, as pointed out in [44], that the failure of a sub-component (deviation from its intended functionality) may be perceived by the other sub-components as an external fault, thus giving rise to the so-called Fault-Error-Failure (FEF) chain.

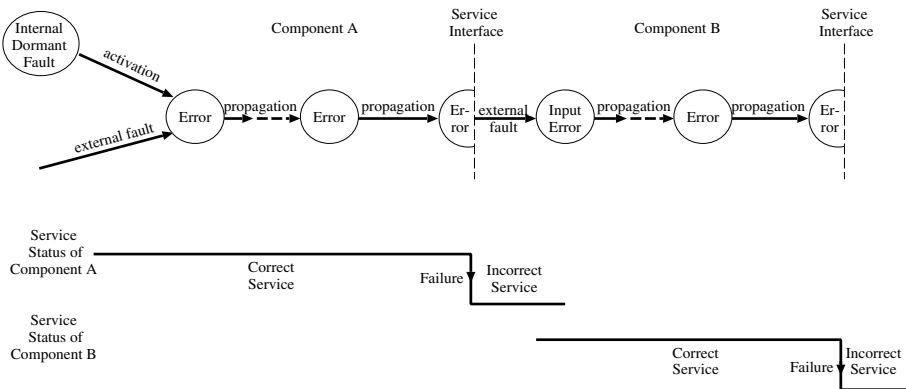


Fig. 11. Relationships between faults, errors and failures

Error propagation within and among system components is explicitly shown in Figure 11 extracted from [44]: internal propagation is caused by the computation process of the faulty component A while the external propagation from component A to component B, that receives service from A, occurs when, through internal error propagation, an error reaches the service interface of the faulty component A. Then, the service delivered by component B to component A becomes incorrect provoking the failure of component A that appears as an external fault to component B and propagates the error into B.

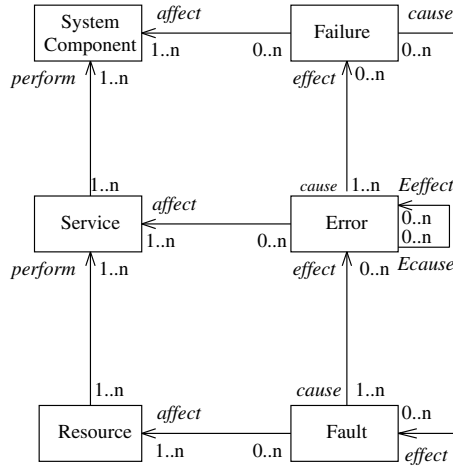


Fig. 12. The CD system description

To be able to model the FEF chain we need a more articulated vision of the system, that allows to clearly identify the components that can be affected by a fault, the system elements on which faults can induce erroneous behaviors, and how these behaviors may lead to a (sub)system failure. Although with a different aim in mind, the work in [27] introduces a view of the system as a layered structure in which a system component realizes a certain function by using a set of resources. Since the pattern of usage of resources can be quite complicated, an intermediate level is added, called services. This point of view on system behavior leads to a three layer structure, represented as an UML Class Diagram (CD) in the left portion of Figure 12.

The right portion of the CD shows instead the FEF chain: relationships between faults, errors and failures as well as their propagation among the system entities is captured by the *cause-effect* associations. Once customized on a specific application the CD shows which faults provoke which errors and which (set of) errors provoke a failure, that is to say a deviation from the function delivered by the system. The diagram also connects each type of fault, error and failure with the corresponding system entity affected by it, so a fault may affect a resource, an error may affect a service performed on one or more faulty resources, and a failure may affect the whole system if errors are not recovered in due time.

Moreover, if a service is affected by an error, the error can be propagated to another service either performed by the same resource (i.e., internal propagation) or by another resource communicating with the former (i.e., external propagation). This error propagation is represented by the *Ecause-Effect* association.

Since a failure of a system component can be perceived by another component as an external fault (as described in [44]), an association exists between the failure and the fault classes. This is an aspect that adds additional complexity to the modeling and is out of the scope of these notes.

We would like to use the high-level information provided by the CD of Figure 12 to drive the construction of PN models. To do this we need first to introduce the PSR

modeling approach that has been presented in [27], and then to modify it to allow for the treatment of the FEF aspects.

6.1 A Layered Approach to Modeling: The PSR

The PSR is a model construction approach in which the PN model is organized into three levels: resources, services and processes. Resources are at the bottom level, and they provide operations for the services, where a service is basically a complex pattern of use of the resources. Services are then requested by the application model placed at the highest level, called process level (and that we have called “System Component” in the CD, since it is more intuitive, although it is not the original term used in [27]).

PSR provides a schema of how the resource, service, and process levels nets should look like, and a compositional operator to compose them.

Figure 13, bottom part, depicts a model of a resource: a resource can be idle, and it can offer one or more operations op_i through the sequence of actions start operation, operation, end operation. Each transition of the sequence has an associated label (shown in *italics* in the figure), that is used for composition with the service level, and that is derived from the association $perform(Resource, Service)$ of the CD diagram of Figure 12, prefixing it with an $S_$ or with an $E_$ to indicate Start and End of operation, and postfixed with the operation index. Depending on the type of operation it may be necessary to acquire a lock (transition $lockRes$ and $unlockRes$). Figure 13, in the middle, models a service. A service can be requested by a process through the pair of labels of start and end service ($S_perform(Service, System\ Component)$, $E_perform(Service, System\ Component)$). Once activated the service can request resource operation via the $S_perform(Resource, Service)_i$, $E_perform(Resource, Service)_i$ labels.

The upper part of Figure 13 depicts a skeleton of the process model that uses services: the request of a service is performed through the label $perform(Service, System\ Component)$ and through a matching function that maps the label into the pair of labels $S_perform(Service, System\ Component)$, $E_perform(Service, System\ Component)$.

Each level is defined through net composition operators based on transition superposition (“horizontal composition”), then resource level is composed with the service level, and the resulting net is composed with the process level also through transition superposition (“vertical composition”). Transition superposition of nets is based on transition labels: two transitions of equal label in two separate nets are fused into a single one: the formal definition of the superposition operator for GSPN can be found in [27], and the SWN extension is instead presented in [10]. Therefore if $\{R_i\}$, $\{S_k\}$, $\{P_m\}$, are the sets of GSPN (or SWN) models representing resources, services, and processes, respectively, and if \parallel is the transition superposition operator, then the full model of the system is given by:

$$\begin{aligned} R &= R_1 \parallel R_2 \parallel \dots \parallel R_{n_r} \\ S &= S_1 \parallel S_2 \parallel \dots \parallel S_{n_s} \\ P &= P_1 \parallel P_2 \parallel \dots \parallel P_{n_p} \\ PSR &= R \parallel S \parallel P \end{aligned}$$

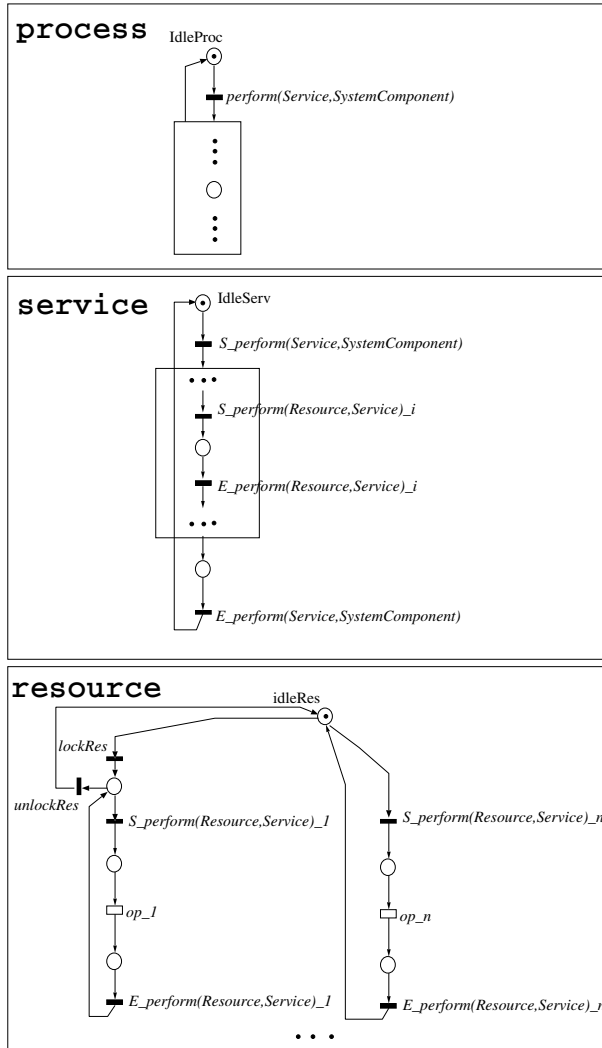


Fig. 13. Resource, service and process models

From now on the term PSR will refer to a structure of the models according to Figure 13.

6.2 PSR and FEF Elements

The PSR originally defined in [27] is not adequate for dependability modeling, since it does not take into consideration the interactions with the FEF elements. In [7, 4] the PSR has been modified by changing the basic models of the resources, services, and process so as to allow interactions with the FEF elements, and by extending the formula to compose the $\{R_i\}$, $\{S_k\}$, and $\{P_m\}$ with models of the FEF elements.

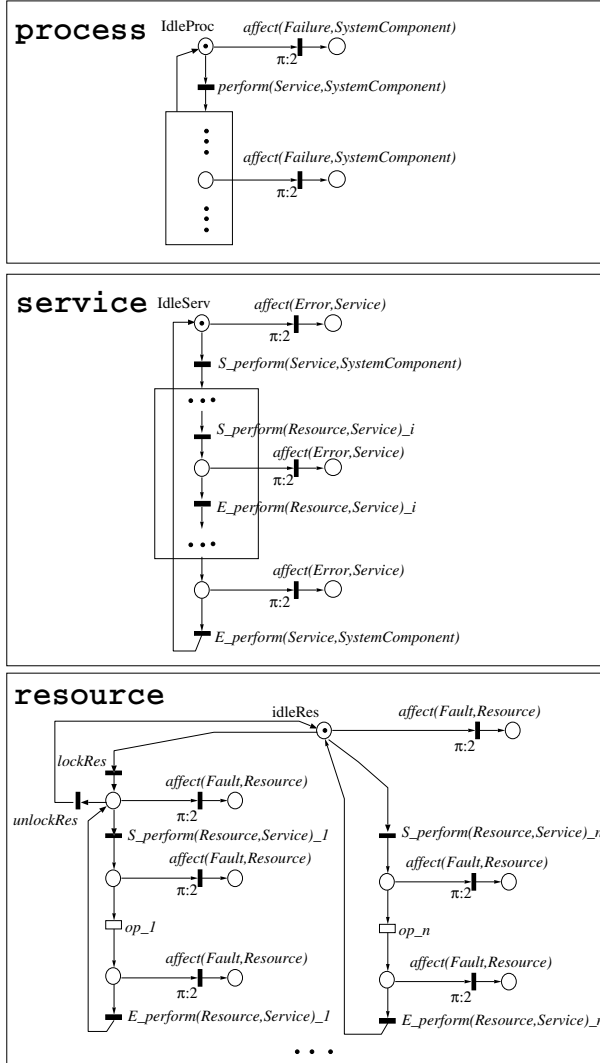


Fig. 14. The modified models of resources, services, and processes

Figure 14 presents the modification to the basic PSR models. Following the CD scheme of Figure 12, we assume that faults affect only the behavior of the resources, errors are perceived at the service level, while failure are a concern of System Components, and therefore of the process level.

Figure 14, bottom part, models a resource. From each state in which a fault can be perceived by the resource a transition labeled $\text{affect(Fault, Resource)}$ has been added (to be used for synchronization with the fault model) which takes the resource into a faulty state. Again, $\text{affect(Fault, Resource)}$ is the association that, in the CD diagram of a

specific application, relates a specific type of fault to a specific type of resource affected by that fault.

Also for service and process models (Figure 14 in the middle part and at the bottom part, respectively), transitions to be used for synchronization with an error and a failure model, respectively, have been added which take the services/processes into an anomalous state.

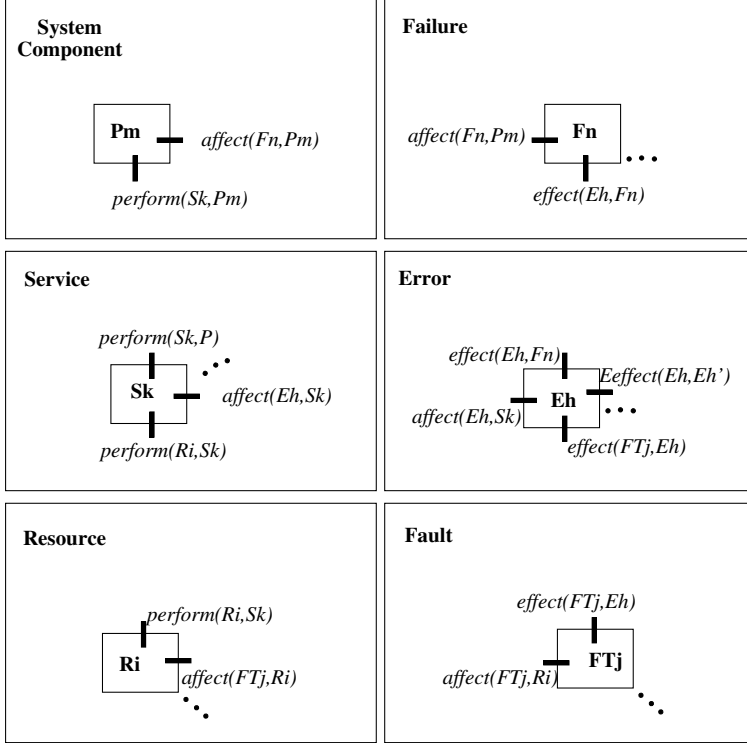


Fig. 15. Organization of the Petri net models in the layered approach

Now that we have the modified resource, service, and process models we need to modify the PSR construction: indeed the $\{R_i\}$, $\{S_k\}$, and $\{P_m\}$ models have to be integrated with the models of the FEF elements. Figure 15 assumes that there are a number of models for faults, errors, and failures, called $\{FT_j\}$, $\{E_h\}$, and $\{F_n\}$, respectively, and it provides a schematic view of how they are organized in three levels: fault models are placed at the resource level, error models at the service level and failure models at process level. Each model is depicted as a box with explicit interface transitions that are labelled according to the associations defined in the CD of Figure 12.

Each level is obtained through horizontal composition, and the global model is obtained through vertical composition of the levels, as for the original PSR. Therefore if $\{R_i\}$, $\{FT_j\}$, $\{S_k\}$, $\{E_h\}$, $\{P_m\}$, $\{F_n\}$ are the sets of GSPN (or SWN) models represent-

ing resources, faults, services, errors, processes, and failure, and if \parallel is the transition superposition operator, then the full model of the system is given by

$$\begin{aligned} R &= R_1 \parallel R_2 \parallel \dots \parallel R_{n_r} \parallel FT_1 \parallel FT_2 \parallel \dots \parallel FT_{n_r} \\ S &= S_1 \parallel S_2 \parallel \dots \parallel S_{n_s} \parallel E_1 \parallel E_2 \parallel \dots \parallel E_{n_e} \\ P &= P_1 \parallel P_2 \parallel \dots \parallel P_{n_p} \parallel F_1 \parallel F_2 \parallel \dots \parallel F_{n_f} \\ PSR &= R \parallel S \parallel P \end{aligned}$$

The proposed approach to the construction of Petri net models for dependability shares similarities with other approaches. In [54] there is an example of organization of dependability GSPN model into layers to separate the architecture model, the service model and the failure modes model, although without explicitly modeling the FEF chain. The modeling of the FEF is made more explicit instead in [12], although the approach taken is that of a top down hierarchical approach more than flat compositional as in the PSR.

6.3 PN Models of the FEF Elements

The compositional approach depicted in Figure 15 requires the definition of models for the FEF elements. In the following we present a library of FEF element models that respect the transition interface of the boxes of Figure 15. The original definition of the library can be found in [8]. The library is built starting from a classification of faults, errors, and failures into a hierarchy of classes that has been devised in the DepAuDE [26] EEC project, and whose complete description can be found in [9]. The classification of the FEF elements in DepAuDE was heavily inspired by the work in [43], customized on the automation system field that was the application target of DepAuDE.

The hierarchy of the classes has a counterpart in a hierarchy of PN components. To set the field to the description of the hierarchy of the FEF models we need first to define the notion of PN component and that of hierarchy for PN components.

PN component is a GSPN system [1] (or an SWN one [18]) labeled over transitions, parametric with respect to transition rates (weights) and/or initial marking and with an associated list *RESULTS* of performance results to be computed and/or verified and a list *CONSTR* of constraints to be verified.

Hierarchy of PN models. Hierarchy in a Class Diagram involves a notion of *inheritance*, that involves the structure of a class (attributes, operation names, and associations) as well as behavior (operations). In GSPN the inheritance of the structure of a class is reflected into 1) inheriting parameters (rate/weight, initial marking), results to be computed, constraints to be verified, and possibly adding new ones; 2) inheriting, and in case modifying, the labels associated to either places or transitions. Inheritance of the dynamic behavior of the super-class is reflected in either maintaining the same net structure in the sub-class or modifying it by applying transformation rules that preserve the *behavioral inheritance* [66]. Two main notions of behavioral inheritance are introduced in [66]: *protocol inheritance* and *projection inheritance*. Although they have

been defined for labeled transition systems, it is rather straightforward to use them for the reachability graphs (RGs) of SPN models. Intuitively, let p and q be two SPN models representing the behavior of a class P and of its super-class Q , respectively; protocol inheritance can be verified by not allowing to fire transitions that are present in p and not in q (i.e., *blocking new actions*) and by checking whether the RGs of p and q are equivalent. Projection inheritance can be verified, instead, by considering not observable the transitions that are present in p and not in q (i.e., *hiding the effect of new actions*) and by checking whether the RGs of p and q are equivalent. In both cases, branching bisimulation [53] is used as equivalence relation. Branching bisimulation belongs to the class of observational equivalence, in which two systems are equivalent if an external observer cannot discriminate between them. Of course two systems may or may not be equivalent depending on what an observer is allowed to see. In our context an observer is allowed to see all transition labels, unless otherwise stated, that is to say the labels that are used to compose models.

The two basic notions of inheritance are combined [66] in order to obtain a stronger and a weaker notion. *Stronger* inheritance is preserved if both protocol and projection inheritance are satisfied. *Life cycle* inheritance is the weaker notion: the set of transitions present in p and not in q is partitioned into “not-observable” and “not-allowed to fire” such that the observable behavior of P equals the behavior of Q .

Observe that the proposed rules for inheritance only consider the net functional behavior, the stochastic behavior may not be preserved, and usually it is not.

Fault Models. Figure 16(A) is the classification of faults: the root class of the inheritance tree describes a generic fault; the first level of the inheritance tree distinguishes *Physical Fault*, *Design Fault*, *Interaction Fault* and *Malicious Logic*. Physical faults are characterized by two input attributes that allow to specify the maximum time during which the fault is active and can be perceived by the system (*duration*) and the frequency of its occurrence (*fault_rate*). Attribute *fault_dormancy*, the length of time between the occurrence of a fault and the appearance of the corresponding error, is considered instead as a metric to be evaluated. The different type of usage of class attributes is denoted by prefixing the attribute name with a specific symbol (i.e., “\$” = input attribute, “/” = metric to be evaluated, “/\$” = metric to be evaluated and validated).

Physical Faults may be either considered permanent or temporary: their discrimination depends on the values assigned to the input attributes *min-duration* and *max-duration* as emphasized by the constraint written in the note symbol. Permanent physical faults and temporary physical faults are further specialized by several sub-classes. For example, temporary physical faults are discriminated in *DevTemp Physical Faults*, that is internal faults due to the development phase; *Transient Physical Faults*, that is faults induced by environmental phenomena; *Intermittent Physical Faults*, i.e., internal physical defects that become active depending on a particular point-wise condition.

Transient and intermittent physical faults classes are enriched with some input parameters, such as: *latency_rateN* and *latency_rateB*, representing the rate of transient fault activation in case of normal conditions and burst conditions, respectively, *persistence_rate*, representing the rate of fault deactivation and *latency_rate*, representing the rate of intermittent fault activation.

Dotted boxes in Figure 16(A) represent classes that are not described here: the interested reader can find a complete description in [9].

GSPN component models for faults have been built according to the hierarchy view of Figure 16(A): each GSPN model is an elaboration of previous generic Petri net models of fault generator proposed in [50] where only physical faults are considered and they are classified with respect to their persistence in permanent and temporary, the latter being further specialized in transient and intermittent.

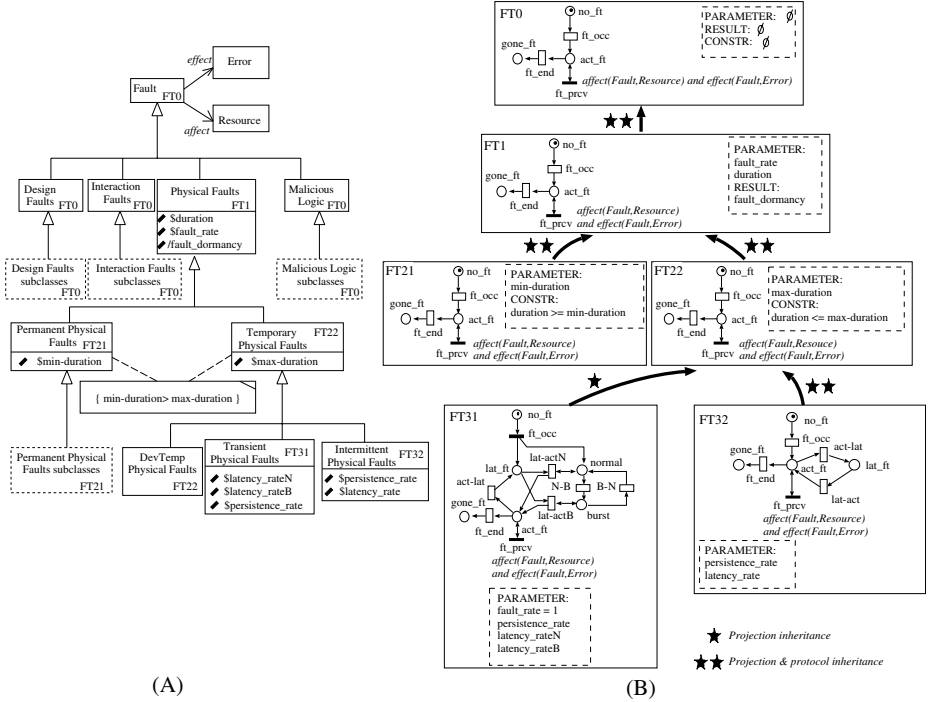


Fig. 16. GSPN component models of fault classes

In Figure 16(B) each box is a labelled GSPN component: a GSPN net with a set of parameters, results to be computed, and constraints to be verified. For sake of graphical clarity the rates of transitions are not shown in the figure, they are listed in the text when needed. Labels are associated to transitions, they determine the observable part of the net behavior and they identify the transitions that can be used for the composition with another model.

The behavior of *Fault* super-class of Figure 16(A) corresponds to the GSPN component model FT_0 of Figure 16(B) characterized by three states: the fault is not present (place no_ft), the fault is active and it may be perceived by a system entity (place act_ft) and the fault is terminated (place $gone_ft$). The fault occurrence is represented by the firing of transition ft_occ , and when the fault is active it can be perceived (transi-

tion ft_prcv) by a system entity, causing an error situation. Transition ft_prcv is labeled so as to allow synchronization with the affected system entity (i.e., the resource model of Figure 14) and with an error model. The fault termination is represented by the firing of transition ft_end . Since neither attributes nor constraints are specified for the *Fault* super-class, the lists of parameters, results and constraints of the corresponding GSPN model FT_0 are empty.

Classes *Design Faults*, *Interaction Faults*, *Malicious Logic*, and their corresponding sub-classes, present the same behavior of the more general class *Fault*, so that the GSPN model FT_0 is reused to represent these classes also.

The class *Physical Faults* is associated with the GSPN model FT_1 that inherits from FT_0 and adds to the parameter list $fault_rate$ and $duration$ and to the result list $fault_dormancy$. The parameters and the result correspond to the homonyms attributes defined in the *Physical Fault* class.

The net structure of FT_0 has been maintained, but rates of transitions ft_occ and ft_end have been defined as functions of the added parameters, i.e., $w(ft_occ) = fault_rate$ and $w(ft_end) = 1/duration$.

The behavior of *Permanent Physical Faults* and of *Temporary Physical Faults* classes is represented by the GSPN models FT_{21} and FT_{22} , respectively. Both the models inherit from model FT_1 , add a parameter (the parameter *min-duration* for model FT_{21} and the parameter *max-duration* for model FT_{22}) and maintain the same net as FT_1 . A fault is classified permanent if it lasts more than min-duration, and it is classified temporary if it lasts less than max-duration, with the constraints, derived from the note symbol of the CD of Figure 16(A), that min-duration is greater than max-duration. The interaction of the models with the corresponding resource and error models (that amounts to the labels associated to transitions) is also inherited from FT_1 .

With respect to the fault models proposed in [50], where permanent faults remain always active while temporary faults once occurred after a certain amount of time eventually disappear, both the fault models FT_{21} and FT_{22} are characterized by a termination state (i.e., place *gone_ft*) and the represented fault classes are discriminated by the fault duration.

Temporary faults can still be distinguished into intermittent and transient faults. Intermittent faults, once occurred, are characterized by alternating periods in which they are active, and they can be perceived by the system entity, and periods in which they are latent and hence they do not cause any error. Transient faults, instead, disappear a certain amount of time after their activation; however, unlike generic temporary faults, they are characterized by a complex mechanism of activation that depends on the condition of the external environment.

The behavior of *Transient Physical Faults* class is represented by the GSPN model FT_{31} in which a fault moves from the latent state to the active state with a different rate depending on the environment conditions. Under normal condition, represented by the place *normal* marked, transition *lat-actN* with rate parameter equal to *latency_rateN* will fire, while under “burst” condition, represented by the place *burst* marked, transition *lat-actB* with rate parameter equal to *latency_rateB* will fire.

The behavior of *Intermittent Physical Faults* class is represented by the GSPN model FT_{32} , in which firing of transition *act-lat* (with rate parameter equal to *per-*

sistence_rate) brings the state of the fault from active to latent and, vice-versa, firing of transition *lat-act* (with rate parameters equal to *latency_rate*) changes the fault state from latent to active.

GSPN models FT_{31} and FT_{32} inherit from FT_{22} : for the structure, new parameters have been added with respect to the parameter list of FT_{22} and for FT_{31} the parameter *fault_rate* is now not relevant (and it has been set to the default value of 1), since the fault activation depends upon the two transitions *lat-actN* and *lat-actB*. From the behavioral point of view the GSPN model FT_{32} strongly inherits from FT_{22} , i.e., it preserves both the projection and the protocol inheritance, while the GSPN model FT_{31} preserves only the projection inheritance, that is to say, if any of the transitions *act-lat*, *lat-actN*, and *lat-actB* is used in a synchronization with another model, then it may be the case that FT_{31} is not able to act as FT_{22} .

Finally, the sub-classes *DevTemp* of temporary physical faults and sub-classes of permanent physical faults inherit the behavior of their super-classes and they have been represented by the same GSPN models associated to the latter.

All the fault GSPN models described above can have more than one label for each transition; in particular, transition *ft-prcv* is characterized by two labels: one is used to interact with the resource model affected by the fault and the other is used to interact with the corresponding error model.

Error Models. Errors are deviations from the correct state of the system that may cause a subsequent failure [44]; they are caused by faults affecting the resources of the system and they are related to the services performed by the faulty resources. A classification of errors is given by the CD of Figure 17(A), taken from [9], that considers only errors caused by physical faults, and discriminate them depending on which type of resource has been affected. The type of resources considered in DepAuDE are processing, memory, and communication.

The super-class *Error* of the hierarchy/logical view is modeled by the GSPN ER_0 - shown in Figure 17(B). The class is characterized by two attributes that are mapped in two results to be computed on ER_0 : *error_latency*, the length of time between the occurrence of an error and the appearance of the corresponding failure, and *PE*, the probability of error. Note that for some results, as *PE*, it is already possible to give their definitions, since their computation is based only on local information; the definition of other results, as *error_latency*, requires instead information on the whole system.

The GSPN model ER_0 is characterized by four states: the error is not present (place *no_err*), the error is generated (place *pot_err*), the error is occurred (place *error*) and the error has been detected (*detected*). Places *error* and *detected* are used to define the result *PE* as the probability that one of the places is marked. The error can be caused by either a fault occurred in a resource or by the error propagation effect: the error generation is represented by the firing of transition *cause* that is labeled so as to ensure synchronization with caused fault or error model. The labels are derived from the associations *effect(Fault, Error)* and association *Eeffect(Error, Error)* of the CD of Figure 17(A). In general, ER_0 contain as many transition “cause” (i.e., with input place *no_err* and with output place *pot_err*) as the number of GSPN models representing potential causes of the error. The occurrence of the error in the corresponding service is represented by the firing of transition *err_occ*, properly labeled to ensure synchronization with the service

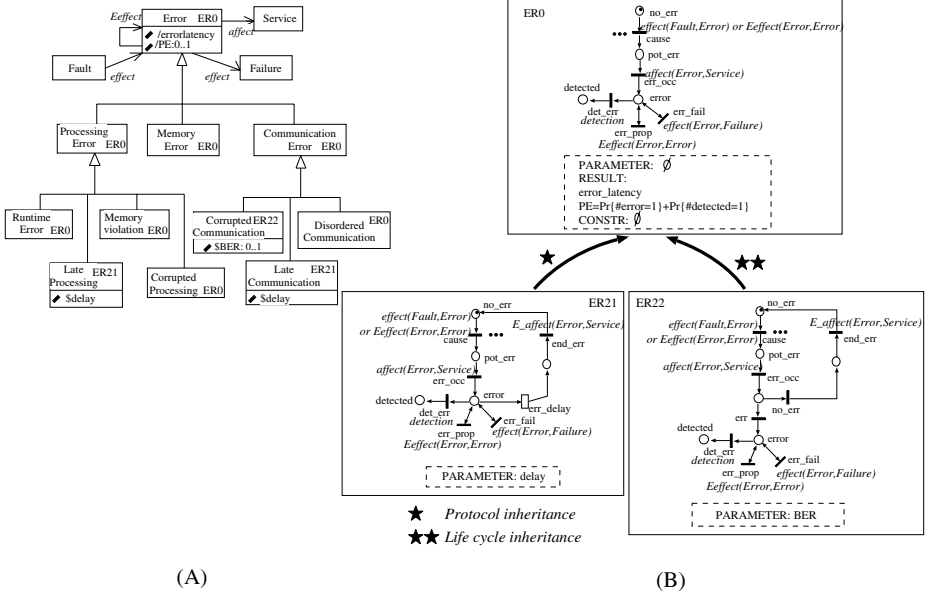


Fig. 17. GSPN component models of error classes

model. Transition det_err represents error detection carried out by some other model, synchronized through the label $detection$.

Test transitions err_prop and err_fail are instead interface transitions for an error model and for a failure mode model, respectively.

Classes *Processing Error*, *Memory Error*, *Communication Error*, *Runtime Errors*, *Memory Violation*, *Corrupted Processing* and *Disordered Communication* present the same behavior of the super-class *Error*, so that GSPN model ER_0 is reused to represent the behavior of these classes also.

Late Processing and *Late Communication* classes are characterized by an input attribute, $delay$, whose values indicate the delay caused in the execution of the corresponding erroneous function. Their behavior is modeled by the GSPN ER_{21} where $delay$ has been added to their parameter list. Moreover, the model contains a pair of causal connected transitions: err_delay , that represents the delay caused by the error, and end_err that brings the error model to its initial state (no_err). Timed transition err_delay is characterized by a rate equal to $w(err_delay) = 1/delay$, immediate transition end_err is, instead, an interface transition and has to be synchronized with the service model in order to bring it from an erroneous state to a normal state. Protocol inheritance is preserved for model ER_{21} ; indeed if transition err_delay is not allowed to fire the RG of ER_{21} is equal to the RG of ER_0 .

Finally, GSPN model ER_{22} has been associated to the *Corrupted Communication* class characterized by the input attribute BER (Bit Error Rate). BER has been added to the parameter list of ER_{22} and it has been assigned to the weight of the immediate transition err . For model ER_{22} life cycle inheritance is preserved, when considering transition err non observable and transitions no_err and end_err not allowed to fire.

Failure Models. Failures are deviations of the service delivered by the system with respect to the system intended function. The CD shown in Figure 18(A) associates to a generic failure mode two metrics to be computed and verified: *PF*, i.e., the probability of failure, and *RF*, rate of failure. The CD represents a classification of failures with respect to their impact on the system, that is whether their occurrences are considered acceptable or not depending on the criticality level associated to the system process they affect. The different failure mode assumptions are represented by the sub-classes: *Halting Failure*, *Degrading Failure* and *Repairing Failure*. Halting failures cause the system activity not to be any longer perceptible by the user. Depending whether the absence of system activity takes form of a frozen output or of silence, they are further classified in passive failures and in silent failures, respectively. Degrading failures still allow the system to provide a subset of its specified behavior. Repairing Failure requires instead that faulty resources originating the failure be replaced or repaired before the system activity continues. Repairing actions are undertaken during the failure treatment phase and are performed by proper mechanisms (association *address*).

The failure hierarchy of Figure 18(A), defined in [9] can be exploited to construct GSPN model components representing different failure modes. The main purpose of GSPN models representing failure modes is to synthesize in a unique place the set of (erroneous) states that have equivalent consequences on the system. These models correspond to the failure mode layer described in [54] that allows to arrange an SPN model in a manner suitable for the analysis of different levels of service degradation. In Figure 18(B) two skeletons of GSPN models representing a generic failure mode and a repairing failure mode, respectively, are depicted. The model F_0 is characterized by three main states: *no_fail*, *pot_fail* and *fail*, respectively meaning the absence of failure, the occurrence of the error conditions causing it and the failure occurrence. Several error conditions may cause the occurrence of a failure: the firing of transition *cond_i* represents the occurrence of one of such conditions; since, in general, the failure occurrence is caused by a combination of errors, *cond_i* is a multi-labeled transition with labels derived from association *effect(Error, Failure)* for synchronization with the error models. Transition *fail_occ* has to be synchronized with a System Component model, so that its label is derived from association *affect(Failure, SystemComponent)*.

Concerning the result list, the GSPN model is characterized by two metrics derived from the homonyms attributes of *Failure* class: *PF*, defined as the probability the place *fail* is marked, and *RF* defined as the throughput of transition *fail_occ*.

Model F_1 , representing a repairing failure mode, contains one transition more with respect to F_0 : *fail_repair* that is an interface transition to be synchronized with reconfiguration mechanism models. Model F_1 respects protocol inheritance.

7 A Methodological Approach to the Construction of Petri Nets Models for Dependability in the Automation System Domain

The Petri Net component models and their organization into a three-layered structure described in Section 5 constitute a first support in the construction of a Petri Net model suitable to be analyzed through either numerical or simulation techniques. But a number of points are still open: how does the modeler identifies, in the system being modeled,

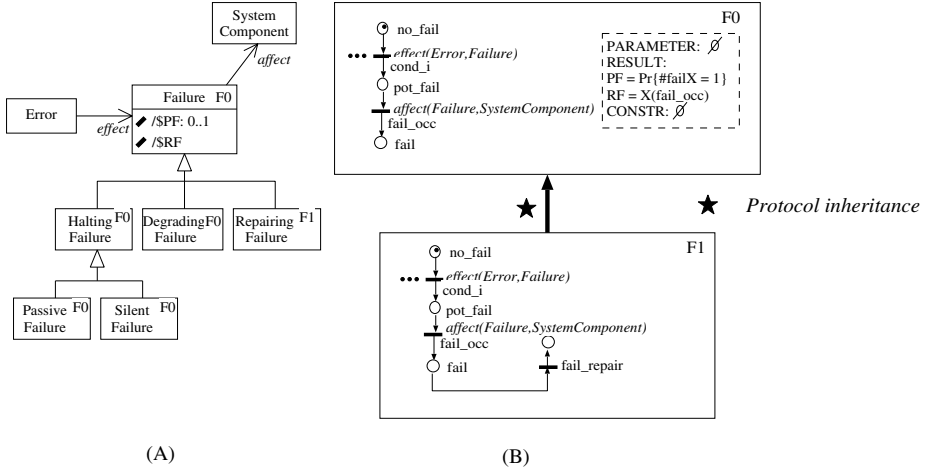


Fig. 18. GSPN component models of failure mode classes

the resources operations, the services and the system components required by the PSR approach? And while modeling a system that includes some FT mechanisms, where should the mechanisms models be placed in the context of Figure 15? In this section we try to give an answer to these questions by restricting the scope to a particular domain, that of automation systems. A larger treatment of the topic can be found in [7, 4].

This work was developed in the EEC-IST project DepAuDE [26] as part of a methodological effort to support the analyst from the early phases of the project (collection of dependability requirements) down to the definition, validation and dependability evaluation of fault tolerance strategies adopted for automation systems [9].

The DepAuDE methodology follows the approach of integrating different notations during the dependability process as suggested by emerging standard like IEC 60300 [21]. In particular, UML Class Diagrams (CDs) and SPN are used in the methodology with different roles, but the information contained in the CDs is exploited to drive the SPN modeling process.

CDs are meant as a support for the requirements collection and/or for structuring and/or reviewing for completeness already available requirements. A set of predefined CDs for the automation system domain (called *generic* CD scheme) and guidelines on how to produce from it the *customized* one that refers to the target application are provided.

Stochastic Petri nets - in particular GSPN and SWN – are used to support dependability design validation and evaluation through modeling. The methodology supports the construction of *PN evaluation scenarios*. A PN scenario consists of a set of PN model components and of their interactions, plus the set of model parameters and the set of performance and validation properties of interest.

The methodology helps the analyst in the construction of a PN scenario by providing a set of predefined reusable PN models for some of the UML classes, a suggested structure of interaction of the model components, guidelines on how to extract infor-

mation from Class Diagrams, and in particular on the Class Diagram instantiated on the specific application, automatic translation from UML State-Charts and Sequence Diagrams into PN, and a suggested approach to the dependability analysis.

The issue of re-use of high level information available from the UML design has been a major concern also for the European Esprit project HIDE [16], that has devised an integrated environment supporting dependability analysis of UML-based system design from the early stages, based on the automatic generation of PN from a number of UML diagrams that encode specific dependability aspects in a rather abstract form. The goal being the evaluation from the early stages of the design the resulting model is obviously rather abstract (since a limited amount of information is available), which is an advantage from a computational point of view, but may be not sufficiently detailed to allow also qualitative properties to be checked.

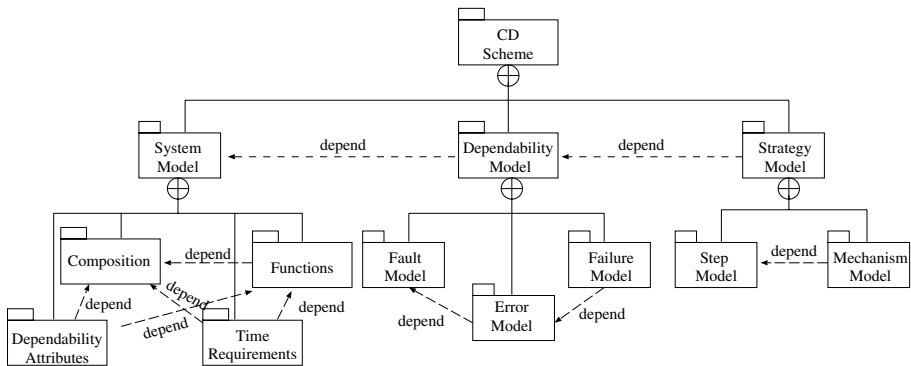


Fig. 19. Scheme hierarchy

UML Class Diagrams. The CDs of the generic CD scheme of the DepAuDE methodology are grouped into the hierarchical structure of *packages* represented in Figure 19, where each non-leaf package of the structure encapsulates a set of inner packages together with their definition dependency relationships. The scheme is therefore constituted by a set of CDs that describe the system in terms of automation components, automation functions, dependability attributes, and timing requirements (left branch in Figure 19), a set of CDs that describe the dependability model in terms of the FEF chain (central branch), and a set of CDs devoted to the strategy model (right branch) that is seen as a set of dependability actions/steps, that can be achieved through a number of software “mechanisms”. The *Fault Model*, *Error Model* and *Failure Model* packages contain, respectively, the hierarchy views of fault, errors and failures described in sub-Section 6.3.

Class attributes are used to represent either parameters, whose values have to be provided as input to the specifications, or measures to be computed or upper/lower bounds whose values have to be provided as input to the specification and to be validated

at later stages of the development. We have chosen to discriminate these different types of usage of class attributes by prefixing the name of the attributes with a specific symbol (“\$”, “/” or “/\$”, respectively).

Elements of a generic CD can be customized on a specific application with the help of a set of guidelines [9]. In the customized CD the value of the class attributes and of the association multiplicities have been set and new classes and associations are added.

The customized CD still refers to classes, and not to objects, but certain classes and associations have been made more specific using information from the application. We now illustrate a few Class Diagrams of the generic CD scheme, trimmed so as to simplify explanation, while still, we hope, containing enough information for what will be discussed later.

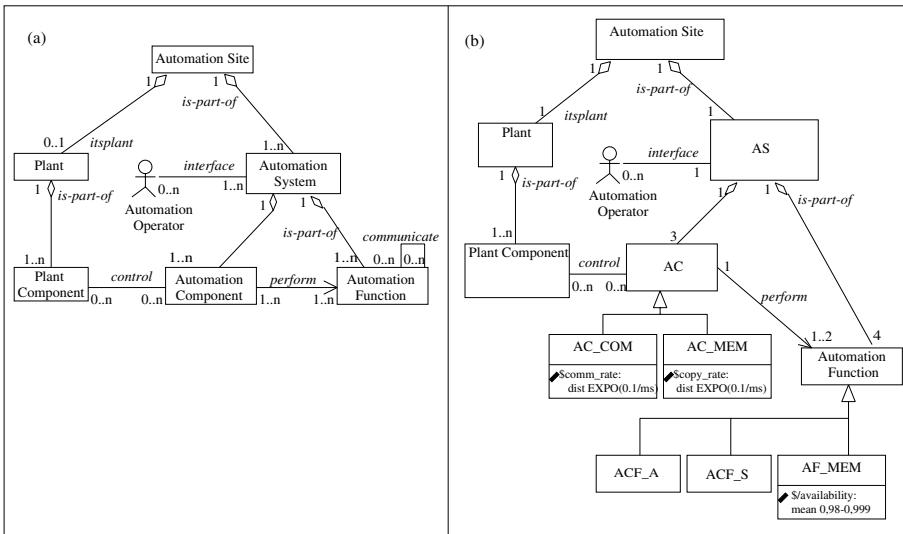


Fig. 20. The CD Structure of Composition (a) and its instantiation to the running example (b)

Figure 20(a) describes a portion of the system: an automation site is a plant (optional) together with one or more automation systems. An automation system is composed by a set of automation functions, that can *communicate* among them, and by a set of automation components that can be used to *perform* one of more automation functions. An automation component *controls* zero or more plant components. An *instantiated* version of the CD of Figure 20(a) will contain application specific information.

Let us consider, for example, a cyclic application that activates two concurrent processes: each process reads a sample input from a plant, elaborates the future state, saves the new state in memory and produces the new output for the plant. The memory units can be affected by physical faults that may cause errors in the automation functions. Communication units are instead assumed not affected by faults. To increase the dependability of the automation system a fault-tolerance strategy has been devised con-

sisting of error detection, error diagnosis and error recovery. The error detection step uses a standard watchdog mechanism while error diagnosis and recovery steps are implemented by a recovery mechanism. If the watchdog expires, it sends a notification message to a software recovery mechanism, that provides to terminate the watchdog and check the status of the automation system: if no error is present then it is a false alarm, and the watchdog is simply reinitialized. If instead an error is present then a recovery action is carried out. Measures to be computed are the availability of the automation functions and the probability of failure of the automation system.

Figure 20(b) shows an instantiated version of the CD depicted in Figure 20(a) to the example: there are two types of automation components dealing with communication (AC_{COM}) and memory (AC_{MEM}), three types of automation functions dealing with asynchronous communication (ACF_A), synchronous communication (ACF_S), and memory (AF_{MEM}), and a single type of automation system AS .

Classes AC_{COM} and AC_{MEM} are characterized, respectively, by the input attributes *comm_rate*, representing the communication rate, and *copy_rate*, representing the rate of the copy operation. The values assigned to these attributes are exponential probability distribution functions with parameter $\lambda = 0.1$. Class AF_{MEM} is characterized, instead, by a metric to be evaluated and verified against the requirements that is the mean availability of the automation functions performed by the memory components whose value has to be included in the interval $[0.98, 0.999]$.

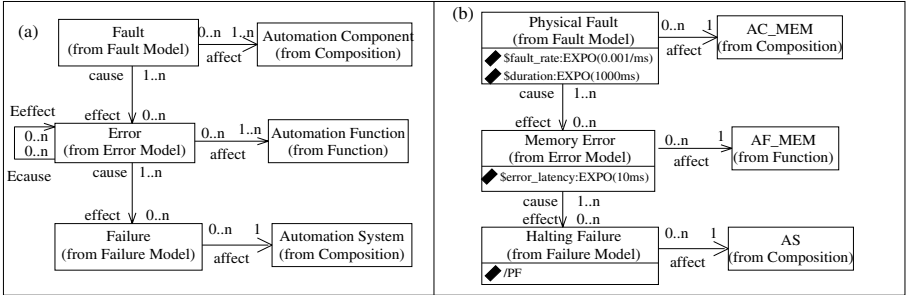


Fig. 21. The generic *FEF* chain (a) and its instantiation to the running example (b)

Figure 21(a) is a trimming of the CD of the FEF chain. Once customized on a specific application it shows which faults cause which errors, how errors propagates, and which (set of) errors cause a failure, that is to say a deviation from the service delivered by the system. The diagram also connects each type of fault, error and failure with the corresponding system components affected by it. The *instantiated* version is shown in Figure 21(b): it contains only one type of faults (*Physical Faults*), a single type of error (*Memory Error*) and a failure (*Halting Failure*). The instantiation of the *affect* relationship relates the fault to the AC_{MEM} component only, the error to function AF_{MEM} , and failure to AS . Values are set to the input attributes of *Physical Fault* and *Memory Error* classes (i.e., the attributes prefixed with the “\$” symbol), while the attribute *PF* (pre-

fixed with the “/” symbol) emphasizes that the probability of failure of the automation system is a measure of interest to be computed.

The CD description of a system contains a lot of useful information for the construction of PN evaluation scenarios, in particular we have observed the following relationships: (1) the *package structure* provides indication on the organization of PN component models; (2) the *aggregations* provides information that allows to identify PN components and the composition formulae; (3) *binary general associations* (associations from now on) among classes indicate interactions and can therefore be used to identify labels for PN model composition; (4) *classes* are rich of *attributes* that are useful to set rate parameters (input and/or upper-lower bound attributes) and to define the performance/dependability indices (output measures and upper-lower bound attribute to be checked); (5) information on the *FEF chain* is fundamental to set the relationship among the PN models of faults, errors, failures, and system components; (6) *hierarchies* can indicate reuse of PN components through inheritance [65]; (7) the *Strategy Model* package allows to identify which mechanisms are used for which fault, error, or failure (dependability strategy).

Composition Scheme of PN Models. To use the PSR in the automation system domain we need to identify the main PN models involved and their interactions. This identification is again driven by the CDs. The organization of PN models into layers is described by Figure 22. The package structure with the three branches of Figure 19 is reflected in the organization into three columns of Figure 22, while to decide in which level to place the various PN models we have considered the FEF chain first (Figure 21). Faults are at the lowest level of the chain, and they have therefore been placed at the resource level. Consequently also automation components (AC), that are affected by faults, have been placed at the same level. With a similar reasoning errors and automation functions (AF) have been placed at the service level, and failure models and automation system (AS) have been placed at the process level.

This is depicted in Figure 22 by the set of boxes AC_i for automation components, FT_j for the fault models, AF_k for the automation functions, ER_h for the error models, AS for the automation system, and $FAIL_n$ for the failure models.

The composition of automation components with faults requires a proper assignment of labels to interface transitions for horizontal composition, that can be derived from the associations *affect(Fault, Automation Component)* of the CD scheme in Figure 21.

The labels for the composition of automation functions with errors and for error propagation are derived from the association *affect(Error, Automation Function)* and *Effect(Error, Error)*, from the CD of Figure 21.

The inter-level interaction between service and resource is given by the association *perform(Automation Component, Automation Function)* of the CD of Figure 20 and by the cause-effect association between faults and errors of Figure 21.

The labels for the composition of automation system with failure are derived from the association *affect(Failure, Automation System)*, while the propagation from error to failure is based on the association *effect(Error, Failure)*.

The software mechanisms are instead placed either at the service or at the process level, depending on whether they address errors or failures.

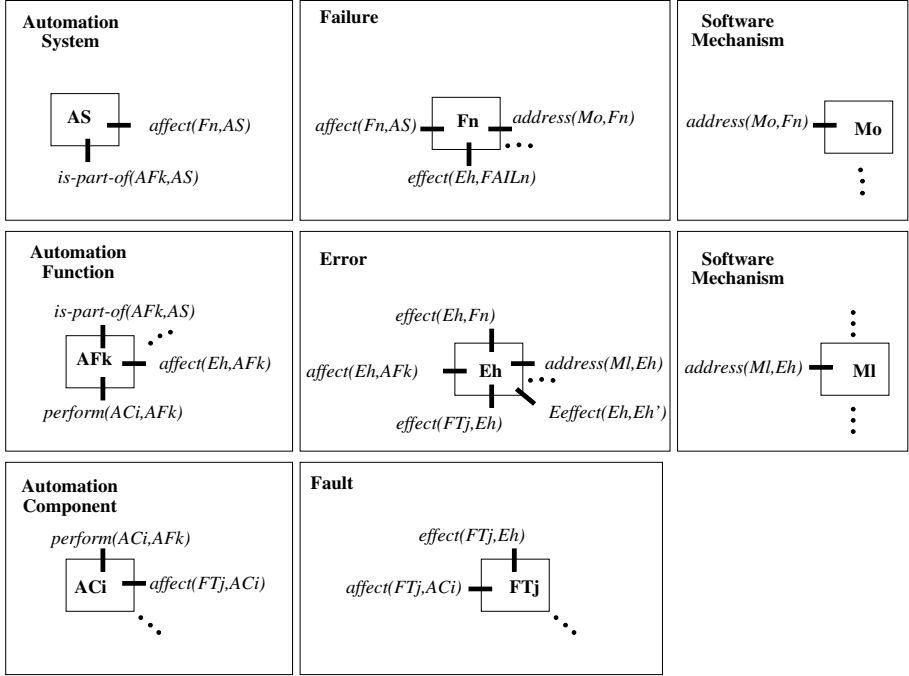


Fig. 22. Organization of the PN models in the DepAuDE methodology

Observe that the relationship between software mechanisms and automation functions is through the error models, although there are cases, like the Local Voter presented in Sub-Section 5.2, in which it seems more natural to have a direct relationship between the mechanisms and the functions, but unfortunately, no information of the subject is contained in the CDs, so that no general guidelines can be derived.

Getting an Executable PN Model. Once the basic structure of the PN models has been identified it is necessary to complete them with a number of information related to the specific application. To reach this goal we have identified a number of steps that will be illustrated through an example.

Step 1. Select the concrete classes of the customized CD scheme amenable to a PN model

The set of PN component models can be identified by examining the customized CD scheme to select the classes that are relevant from a quantitative point of view. Good candidates are classes of the customized CD scheme that contain attributes specifying input parameters, metrics to be computed and/or to be verified.

If we assume a class level specification, then all GSPNs are initialized with a single token, while in case of object level specification the identities of the objects come into play. In the context of modeling of distributed object software, an in-depth treatment of the specification level is reported in [67], where a formalism derived from Colored Petri Nets (CPN) [38] has been defined. Having used GSPN for the class models, SWN [18] is a natural choice at the object level to keep track of the object identities.

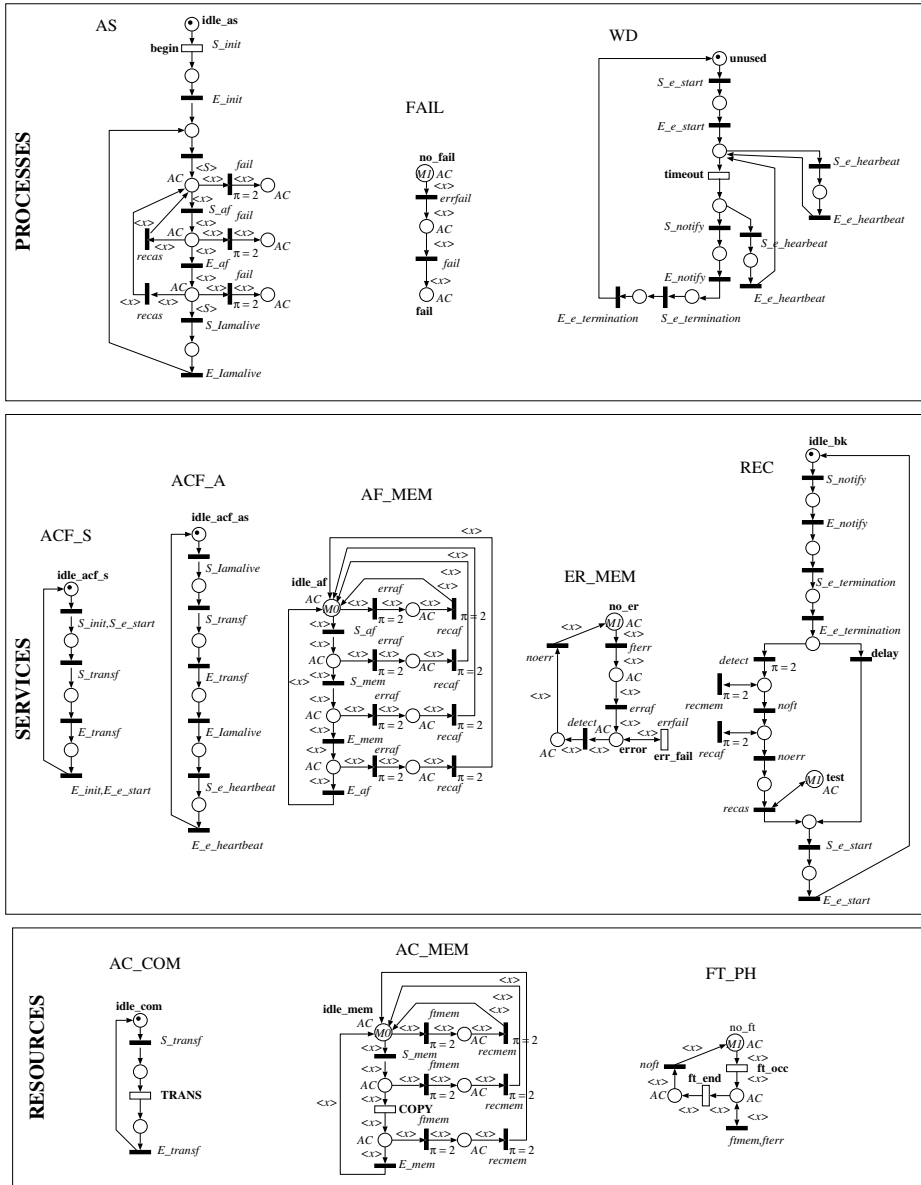


Fig. 23. Example of analyzable PN model: the set of PN components

Example. The customized CD scheme allows the identification of the PN components shown in Figure 23.

At the resource level, left to right, we have: the communication unit, the memory units, and the physical fault models. For AC_{COM} and AC_{MEM} we have reused the resource models of Figure 13 and of Figure 14, respectively, with transmission and copy

as basic operations, while adding reset transitions in AC_{MEM} . For FT_{PH} the predefined physical fault model FT_0 of Figure 16 has been used (with reset).

At service level, left to right, there are: the synchronous communication function used by the automation system to initialize the watchdog, the asynchronous communication function used by the automation system to send signals to the watchdog, the memory automation functions, the errors affecting the memory automation functions, and the recovery mechanism. Models ACF_A and ACF_S are simplified models of communication presented in [6]. AF_{MEM} follows the basic skeleton of service model of Figure 14 with “reset” transitions. ER_{MEM} is an error memory model obtained by refining the PN model component ER_0 of Figure 17. Model REC has been produced from the high level design specification of the recovery mechanism.

At process level there are: the automation system, the halting failure mode of the automation system, and the watchdog mechanism. Model AS has been produced from the high level design specification of the automation system. Model $FAIL$ is a customization of the predefined failure mode model F_0 of Figure 18. WD is a simplification of the watchdog model resulting from the automatic translation of the State-Chart specification of the watchdog [11].

Colors have been used to keep track of the multiple copies of AC_{MEM} , FT_{PH} , and ER_{MEM} , as well as to model two parallel subprocesses of AS .

Step 2. Customize the composition rules using the associations of the customized CDs. The names/rolenames of the binary general associations defined in the CDs have been used to characterize the labels of the PN components in Figure 23.

If object level specification is assumed, and therefore SWN models are used, further “control” SWN component models may be necessary, to do the right association between colors, as, for example, in the case of AC and AF model, to associate to each *Automation Component* the corresponding *Automation Function*.

Example. The associations named *affect* allow to define three synchronization labels: *ftmem*, to synchronize the memory model and the faults, *erraf*, to synchronize the automation functions and the memory, and finally, *fail*, to synchronize the automation system and the halting failure model. New labels are introduced for the interaction between the recovery mechanism and the memory error model (*detect*, *noerr*) and between the recovery mechanism and the automation functions model (*recaf*). We can then define the sets of labels for the horizontal compositions of the resource level ($L_{res} = \{ftmem\}$), of the service level ($L_{srv} = \{erraf\}$, and $L'_{srv} = \{detect, noerr, recaf\}$), and the process level ($L_{pr} = \{fail\}$).

The resource layer model R , the service layer model S and process layer model P are then obtained by applying the composition operator $||$ over transition labels:

$$\begin{aligned}
 R &= \left(AC_{COM} \underset{0}{|} AC_{MEM} \right) \underset{L_{res}}{||} FT_{PH}, \\
 S &= \left\{ \left[\left(ACF_A \underset{0}{|} ACF_S \right) \underset{0}{|} AF_{MEM} \right] \underset{L_{srv}}{||} ER_{MEM} \right\} \underset{L'_{srv}}{||} REC, \\
 P &= \left(AS \underset{L_{pr}}{||} FAIL \right) \underset{0}{||} WD
 \end{aligned}$$

A similar procedure allows to identify the labels for the vertical composition of layers based on the associations *perform* between automation components and automation functions, and on the associations *effect* between physical faults and memory errors, and between memory errors and halting failure. New labels are also added to represent: the interactions among the recovery mechanism model and the component models laying at resource level and at process level, the interactions among the automation system model and the automation (communication) functions models and, finally, the interactions among the watchdog model and the automation communication functions models.

The final PN model *PSR* is then obtained using the parallel composition of the various levels upon the identified labels, according to the PSR methodology.

Step 3. Define the initial marking of the composed PN.

A complete definition of the initial marking is possible only when system design specification is available, by composing the initial marking of the components, that may require information on the object identities.

Example. The initial marking is based on the assumptions that there is one communication unit, and that physical faults affects only one the first memory. The marking parameter M_0 has one token per color of the class $C = C_1 \cup C_2 = \{c_1\} \cup \{c_2\}$ and it is used for AC_{MEM} and AF_{MEM} . The marking parameter M_1 , defined as the single color c_1 , is used for FT_{PH} , ER_{MEM} and $FAIL$.

Step 4. Initialize the rate/weight parameters of the PN composed model and define the results.

The rate/ weight parameters and performance/ dependability indices should be defined according to the values set to the input and output attributes of the customized classes. The remaining ones are added and initialized by the modeler.

Example. From the customized CDs of Figure 20(b) and of Figure 21(b) we can identify the following input parameters for the PN model: *comm_rate*, *copy_rate*, *fault_rate* and *duration* representing the communication rate, the rate of copy operation, the rate of the fault occurrence and its duration, respectively.

The metrics to be evaluated and/or validated are specified by three attributes: *availability*, defined in class AF_{MEM} of Figure 20(b), and *PF*, defined in the *Halting Failure* class of Figure 21(b), specifying the probability of failure. These information extracted from the CDs are only indications, no formal definition is associated to them, and they have to be defined by the modeler.

Step 5. Perform the analysis.

To perform the analysis it may not be a straightforward task, since it may require a modification of the PN model: as exemplified by the Local Voter mechanism in Sub-Section 5.2 we point out that for certain types of indices it may be necessary to perform a transient analysis in which the states representing a failure are made absorbing, while if instead a recovery strategy is being evaluated it is likely that the model should be made ergodic.

Example. We have used GreatSPN tool [52] to construct the PN component models depicted in Figure 23 and the program algebra [10] to carry out their composition. The

reachability graph of the final SWN model contains 115 tangible markings, 778 vanishing markings and 4 dead markings (failure of the automation system). The model can be used for the computation of the probability of failure defined as the probability that place *fail* becomes marked within time t , i.e., $Pr\{M[fail](x) = 1, x \leq t\}$. The modified ergodic model (in which a restart from failure has been modeled) is characterized by 103 tangible markings and 1051 vanishing markings. The ergodic model can be used for the computation of the mean availability of the memory automation function that can be affected by error that is defined as the probability that place *error* is empty.

8 Conclusions

In this paper we have introduced the quantitative evaluation of dependability based on a probabilistic approach, following an order of presentation that somehow reflects also be the historical development of the dependability field.

Starting from the dependability of simple systems, that can be expressed as relatively simple formulas of the dependability of the elementary components of the system, following a divide and conquer approach, we have then discussed the role of state enumeration techniques for dependability, and in particular of state enumeration techniques based on Continuous Time Markov Chain.

Since state enumeration is a low-level, error-prone activity, the researchers have looked with interest into higher level formalisms as Petri nets, and in particular to whose classes of Petri nets that have an underlying stochastic process semantics as either the simple Markov Chain, like SPN, or as the more complex form of Markov Regenerative Process, like Markov Regenerative SPN [19].

When building a model of complex systems for dependability, the interplay between the system components and the FEF elements plays a central role, we have therefore also presented a systematic, compositional approach to the construction of SPN models for dependability. The approach has been exemplified with an example taken from the automation system domain.

Acknowledgements

We would like to thank Andras Horváth that has been the co-author of the work [10], part of which has been reused in this paper, as well as the person who implemented the program *algebra* for net-composition in the *GreatSPN* tool.

References

1. M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with Generalized Stochastic Petri Nets*. J. Wiley, 1995.
2. M. Ajmone Marsan, A. Bobbio, and S. Donatelli. Petri Nets in performance analysis, an introduction. In W. Reisig and G. Rozenberg, editors, *Lectures in Petri Nets: basic models*, pages 211–256, Berlin, Germany, 1998. Springer Verlag. LNCS, Vol 1491.
3. R.E. Barlow and F. Proschan. *Statistical Theory of Reliability and Life Testing*. Holt, Rinehart and Winston, New York, 1975.

4. S. Bernardi. *Building Stochastic Petri Net models for the verification of complex software systems*. PhD thesis, Dipartimento di Informatica, Università di Torino, April 2003.
5. S. Bernardi, C. Bertinello, S. Donatelli, G. Franceschinis, G. Gaeta, M. Gribaudo, and A. Horváth. GreatSPN in the new millenium. Technical report, In Tools of Aachen 2001, International MultiConference on Measurement, Modelling and Evaluation of Computer-Communication System, 2001.
6. S. Bernardi and S. Donatelli. Performance Validation of Fault-Tolerance Software: A Compositional Approach. In *Proc. of the International Conference on Dependable Systems and Networks, DSN'01*, pages 379–388, Göteborg, Sweden, July 2001. IEEE Computer Society ed.
7. S. Bernardi and S. Donatelli. Building Petri net scenarios for dependable automation systems. In *Proc. of the 10th International Workshop on Petri Nets and Performance Models (PNPM2003)*, pages 72–81, Urbana-Champaign, Illinois (USA), September 2003. IEEE Computer Society ed.
8. S. Bernardi and S. Donatelli. Stochastic Petri nets and inheritance for dependability modelling. In *Proc. of the 10th Pacific Rim International Symposium on Dependable Computing (PRDC04)*, Papeete, tahiti (French Polynesia), March 2004. IEEE C.S. to be published.
9. S. Bernardi, S. Donatelli, and G. Dondossola. Methodology for the generation of the modeling scenarios starting from the requisite specifications and its application to the collected requirements. Deliverable D1.3b - DepAuDE IST Project 25434, June 2002.
10. S. Bernardi, S. Donatelli, and A. Horváth. Special section on the practical use of high-level Petri Nets: Implementing Compositionality for Stochastic Petri Nets. *Journal of Software Tools for Technology Transfer (STTT)*, 3(4):417–430, August 2001.
11. S. Bernardi, S. Donatelli, and J. Merseguer. From UML Sequence Diagrams and Statecharts to analysable Petri Net models. In *Proceedings of the 3rd International Workshop on Software and Performance*, pages 35–45, Rome (Italy), July 2002.
12. C. Betous-Almeida and K. Kanoun. Stepwise Construction and Refinement of Dependability Models. In *Proc. of the International Conference on Dependable Systems and Networks, DSN'02*, pages 515–524, Washington, D.C., USA, June 2002. IEEE Computer Society ed.
13. A. Bobbio. Teoria e Metodi di affidabilità. Dispense COREP - Dipartimento di Informatica, Università del Piemonte Orientale, Alessandria, Italia (in italian).
14. A. Bobbio. Petri Nets Generating Markov Reward Models for Performance/Reliability Analysis of Degradable Systems. In Puigjaner, R. et al., editors, *Modeling Techniques and Tools for Computer Performance Evaluation. Proceedings of the Fourth International Conference, 1988, Palma, Spain*, pages 353–365, New York, NY, USA, 1989. Plenum.
15. A. Bobbio, G. Franceschinis, R. Gaeta, and L. Portinale. Parametric Fault Tree for the Dependability Analysis of Redundant Systems and Its High-Level Petri Net Semantics. *IEEE Trans. Software Eng.*, 29(3):270–287, 2003.
16. A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. Dependability analysis in the early phases of UML-based system design. *International Journal of Computer Systems Science & Engineering*, 16(5):265–275, September 2001.
17. O. Botti, V. De Florio, G. Deconinck, F. Cassinari, S. Donatelli, A. Bobbio, A. Klein, H. Kufner, R. Lauwereins, E. Thurner, and E. Verhulst. TIRAN: Flexible and Portable Fault Tolerance Solutions for Cost Effective Dependable Applications. In P. Amestoy, P. Berger, M. Daydé, I. Duff, V. Frayssé, L. Giraud, and D. Ruiz, editors, *Proceedings 5th Int. Euro-Par Conference on Parallel Processing (Europar'99) - Lecture Notes in Computer Science Vol. 1685*, pages 1166–1170. Springer Verlag, Berlin, Germany, Toulouse, France, Aug. 31-Sep. 3 1999.
18. G. Chiola, C. Dutheillet, G. Franceschinis, and S. Haddad. Stochastic Well-Formed coloured nets for symmetric modelling applications. *IEEE Transaction on Computers*, 42(11):1343–1360, November 1993.

19. H. Choi, V.G. Kulkarni, and K. Trivedi. Markov Regenerative Stochastic Petri Nets. *Performance Evaluation*, 20:337–357, 1994.
20. Gianfranco Ciardo and Kishor S. Trivedi. SPNP: The Stochastic Petri Net Package (Version 3.1). In *Proc. 1st Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'93)*, pages 390–391. IEEE Comp. Soc. Press, 1993.
21. International Electrotechnical Commission. IEC-60300-3-1: Dependability Management. IEC, 3 rue de Varembe CH 1211 Geneva, Switzerland, 2001.
22. S. Contini and A. Poucet. Advances on fault tree and event tree techniques. In In A.G. Colombo and A. Saiz de Bustamante, editor, *System Reliability Assessment*, pages 77–102. Kluwer Academic P.G., 1990.
23. J.A. Couvillion, R. Freire, R. Johnson, W. Douglas Obal II, M.A. Qureshi, M. Rai, W.H. Sanders, and J.E. Tvedt. Performability Modeling with UltraSAN. *IEEE Software*, 8(5):69–80, 1991.
24. D.R. Cox. *Renewal theory*. Chapman & Hall, London, 1962.
25. D.R. Cox and H.D. Miller. *The theory of stochastic processes*. Chapman and Hall, London, 1965.
26. DepAuDE EEC-IST project 2000-25434. <http://www.depaude.org>.
27. S. Donatelli and G. Franceschinis. The PSR methodology: integrating hardware and software models. In *Proc. of the 17th International Conference in Application and Theory of Petri Nets, ICATPN '96*, Osaka, Japan, June 1996. Springer Verlag. LNCS, Vol 1091.
28. J. B. Dugan, K. S. Trivedi, R. M. Geist, and V. F. Nicola. Extended Stochastic Petri Nets: Applications and Analysis. In Gelenbe, E, editor, *PERFORMANCE'84: Models of Comput. System Performance, Proc. of the 10th Int. Symp., Paris*, pages 507–519, Amsterdam, 1984. Elsevier.
29. J.B. Dugan and K.S. Trivedi. Coverage modelling for dependability analysis of fault tolerant systems. *IEEE Transaction on Computers*, 38(6):775–787, 1989.
30. C. Béounes et al. SURF-2: A Program for Dependability Evaluation of Complex Hardware and Software Systems. In *23rd Int. Symp. on Fault-Tolerant Computing*, pages 668–673, Toulouse (France), 1993.
31. N.B. Fuqua. *Reliability Engineering for Electronic Design*. Marcel Dekker Inc., New York, 1987.
32. E.J. Henley and H Kumamoto. *Reliability Engineering and Risk Assessment*. Prentice Hall, Englewood Cliffs, 1981.
33. G. S. Hura. A Petri Net Approach to Enumerate all System Success Paths for Reliability Evaluation of a Complex System. *Microelectron. Reliab. (GB)*, 22(3):427–428, 1982.
34. G. S. Hura and J. W. Atwood. The Use of Petri Nets to Analyze Coherent Fault Trees. *IEEE Trans. Reliab. (USA)*, 37(5):469–474, 1988.
35. O. Ibe, A. Sathaye, R. Howe, and K. S. Trivedi. Stochastic Petri Net Modeling of VAXcluster Availability. In *Proc. Third Int. Workshop on Petri Nets and Performance Models (PNPM89)*, pages 112–121, Kyoto (Japan), 1989.
36. IEC-10125. *Fault Tree Analysis*. IEC-Standard-No. 10125, 1990.
37. IEC-61165. *Application of Markov techniques*. IEC-Standard-No. 61165, 1995.
38. Jensen K. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*, volume 1. Springer-Verlag, 1997. Monographs in Theoretical Computer Science, ISBN:3-540-60943-1.
39. K. Kanoun, M. Borrel, T. Moreteveille, and A. Peytavin. Modeling the Dependability of CAUTRA, a Subset of the French Air Traffic Control System. In *Proceedings of the 26th Int. Symp. Fault-Tolerant Computing (FTCS-26)*, pages 95–515, Sendai (Japan). LAAS-REPORT.

40. K. Kanoun and Borrel M. Dependability of fault-tolerant systems. Explicit modeling of the interactions between hardware and software. In *Proc. of the 2nd Annual IEEE International Computer Performance and Dependability Symposium (IPDS'96)*, pages 252–261, Urbana Champaign, USA, Sept 1996. IEEE-CS Press.
41. A. Kaufmann, D. Grouchko, and R. Cruon. *Mathematical Models for the Study of the Reliability of Systems*. Academic Press, 1977.
42. V. G. Kulkarni. *Modeling and Analysis of Stochastic Systems*. Chapman & Hall, 1995.
43. J. C. Laprie. Dependability – Its attributes, impairments and means. In B. Randell, J.C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, pages 3–24. Springer Verlag, 1995.
44. Randell B. Laprie J.C. and Avizienis A. Fundamental Concepts of Dependability. Technical report, LAAS - NewCastle University - UCLA, 2001. LAAS Report no. 01-145, NewCastle University Report no. CS-TR-739, UCLA CSD Report no. 010028.
45. M. Malhotra and K.S. Trivedi. Dependability Modelling using Petri net based models. *IEEE Transactions on Reliability*, 44(3):428–440, 1995.
46. M. Ajmone Marsan, A. Bobbio, G. Conte, and A. Cumani. Performance analysis of degradable multiprocessor systems using Generalized Stochastic Petri Nets. *Distributed Processing Technical Committee Newsletter*, 6(SI-1):47–54, 1984.
47. M.K. Molloy. Performance analysis using Stochastic Petri Nets. *IEEE Transaction on Computers*, 31(9):913–917, September 1982.
48. J. Muppala, G. Ciardo, and K. Trivedi. Stochastic reward nets for reliability prediction. *Communications in Reliability, Maintainability and Serviceability*, 1(2):9–20, July 1994.
49. J. Muppala, R. Fricks, and K. S. Trivedi. Techniques for System Dependability Evaluation. In W. Grassman, editor, *Computational Probability*, pages 445–480, The Netherlands, 2000. Kluwer Academic.
50. I. Mura, S. Chiaradonna, and A. Bondavalli. Modelli teorici e pratici per la rappresentazione del processo di guasto. Progetto di ricerca PDCC-ENEA: Aspetti specifici e tecniche di tolleranza ai guasti. (in italian).
51. A. Papoulis. *Probability, Random Variables and Stochastic Processes*. Mc Graw Hill, New York, 1965.
52. Performance Evaluation group of Torino. The GreatSPN tool. <http://www.di.unito.it/~great-spn>.
53. L. Pomello, G. Rozenberg, and C. Simone. A Survey of Equivalence Notions for Net Based Systems. *Lecture Notes in Computer Science; Advances in Petri Nets 1992*, 609:410–472, 1992.
54. M Rabah and K. Kanoun. Performability evaluation of multipurpose multiprocessor systems: the “separation of concerns” approach. *IEEE Transactions on Computers. Special Issue on Reliable Distributed Systems*, 52(2):223–236, February 2003.
55. R. A. Sahner and K.S. Trivedi. Reliability Modeling using SHARPE. *IEEE Transactions on Reliability*, R-36(2):186–193, June 1987.
56. W.H. Sanders and L.M. Malhis. Dependability Evaluation Using Composed SAN-Based Reward Models. *Journal of Parallel and Distributed Computing*, 15(3):238–254, 1992.
57. W.H. Sanders and J.F. Meyer. Stochastic Activity Networks: Formal Definitions and Concepts. *Lecture Notes in Computer Science*, 2090:315–??, 2001.
58. W. Schneeweiss. *Petri Nets for Reliability Modelling*. LiLoLe-Verlag GmbH, Hagen (Germany), 1999.
59. W.G. Schneeweiss. *The Fault Tree Method*. LiLoLe Verlag, 1999.
60. M.L. Shooman. *Probabilistic reliability: an engineering approach*. Mc Graw Hill, 1968.
61. W.J. Stewart. *Introduction to the Numerical Solution of Markov Chains*. Princeton University Press, 1994.

62. K.J. Sullivan, J.B. Dugan, and D. Coppit. The Galileo Fault Tree Analysis Tool. In *Proceedings of the 29th Annual International Symposium on Fault-Tolerant Computing*, Madison, Wisconsin, 1999. IEEE.
63. K. Trivedi. *Probability & Statictics with Reliability, Queueing & Computer Science applications*. Prentice Hall, 1982.
64. K. Trivedi. *Probability & Statistics with Reliability, Queueing & Computer Science applications*. Wiley, II Edition, 2001.
65. W.M.P. Van der Aalst. Inheritance of Dynamic Behavior in UML. In Daniel Moldt ed., editor, *Proc. of the 2th Workshop on Modelling of Objects, Components and Agents, MOCA'02*, Aarhus, Denmark, August 2002. Technical Report ISSN 0105-8517, Dept. of Computer Science, University of Aarhus.
66. W.M.P. Van der Aalst and T. Basten. Life-cycle inheritance: A Petri-net based approach. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248, pages 62–81. Springer Verlag, Berlin, 1997.
67. X. Xie and S.M. Shatz. Development of Class-level and Instance-level Design Model for Distributed Systems. *International Journal of Informatica, special issue on Component Based Software Development*, 25:465–474, 2001.