Stepwise Construction and Refinement of Dependability Models

Cláudia Betous-Almeida and Karama Kanoun
LAAS-CNRS
7, Avenue du Colonel Roche
31077 Toulouse Cedex 4 - France
E-mail: {almeida,kanoun}@laas.fr

Abstract

This paper presents a stepwise approach for dependability modeling, based on Generalized Stochastic Petri Nets (GSPNs). The first-step model called functional-level model, can be built as early as system functional specifications and then completed by the structural model as soon as the system architecture is known, even at a very high level. The latter can be refined according to three different aspects: Component decomposition, state and event fine-tuning and distribution adjustment to take into account increasing event rates. We define specific rules to make the successive transformations as easy and systematic as possible. This approach allows the various dependencies to be taken into account at the right level of abstraction: Functional dependency, structural dependency and those induced by non-exponential distributions. A part of the approach is applied to an instrumentation and control system (I&C) in power plants.

1. Introduction

Dependability evaluation plays an important role in critical systems’ definition, design and development. Modeling can start as early as system functional specifications, from which a high-level model can be derived to help in analyzing dependencies between the various functions. However the information that can be obtained from dependability modeling and evaluation becomes more accurate as more knowledge about the system’s implementation is incorporated into the models.

The starting point of our work was to help (based on dependability evaluation) a stakeholder of an I&C system in selecting and refining systems proposed by various contractors in response to a Call for Tenders. To this end, we have defined a stepwise modeling approach that can be easily used to select an appropriate system and to model it thoroughly. This modeling approach is general and can be applied to any system, to model its dependability in a progressive way. Thus, it can be used by any system’s developer.

The process of defining and implementing an I&C system can be viewed as a multi-phase process starting from the issue of a call for tenders by the stakeholder. The call for tenders gives the functional and non-functional (e.g., dependability) requirements of the system and asks candidate contractors to make offers for possible systems/architectures satisfying the specified requirements. A preliminary analysis of the numerous responses by the stakeholder, according to specific criteria, allows the pre-selection of two or three candidate systems. At this stage, the candidate systems are defined at a high level and the application software is not entirely written. The comparative analysis of the pre-selected candidate systems, in a second step, allows the selection of the most appropriate one. Finally, the retained system is refined and thoroughly analyzed to go through the qualification process. This process is illustrated in Figure 1. Even though this process is specific to a given company, the various phases are similar to those of a large category of critical systems.

Dependability modeling and evaluation constitute an efficient support for the selection and refinement processes, thorough analysis and preparation for the system’s qualification. Our modeling approach follows the same steps as the development process. It is performed in three steps as described in Figures 1 and 2:

Step 1. Construction of a functional-level model based on the system’s specifications;

Step 2. Transformation of the functional-level model into a high-level dependability model, based on the knowledge of the system’s structure. A model is generated for each pre-selected candidate system;

Step 3. For the retained system, refinement of the high-
level model into a detailed dependability model.

The remainder of the paper is organized as follows. Section 2 describes the functional-level model. The high-level dependability model is presented in Section 3. Section 4 deals with the structural model’s refinement and Section 5 presents a small example of application of the proposed approach to an I&C system. Finally, Section 6 concludes the paper.

2. Functional-level model

The derivation of the system’s functional-level model is the first step of our method. This model is independent of the underlying system’s structure. Hence, it can be built even before the call for tenders, by the stakeholder. It is formed by places representing possible states of functions. For each function, the minimal number of places is two (Fig. 3): One represents the function’s nominal state (F) and the other its failure state (F̅).

In the following, we assume only one failure mode, but it is applicable in the same manner when there are several failure modes per function. Between states F and F̅, there are events that manage changes from F to F̅ and vice-versa. These events are inherent to the system’s structure that is not specified at this step, as it is not known yet. The model containing these events and the corresponding places, is called the link model (M_L). Note that the set \{F, M_L, F̅\}, that constitutes the system’s GSPN model, will be completed once the system’s structure is known.

However, systems generally perform more than one function. In this case we have to look for dependencies between these functions due to the communication between them. We distinguish two degrees of dependency: Total dependency and partial dependency. Figure 4 illustrates examples of the two degrees of functional dependency between two functions F_1 and F_2. F_3 is independent from both F_1 and F_2.

Case (a) Total dependency – F_2 depends totally on F_1 (noted F_2 → F_1): If F_1 fails, F_2 also fails;
Case (b) Partial dependency – F_2 depends partially on F_1 (noted F_2 ↔ F_1): F_1’s failure does not induce
F₂’s failure. In fact, F₁’s failure puts F₂ in a degraded state that is represented by place F₂d. This state is marked whenever F₁ is in its failure state and F₂ in its nominal one. In Figure 4(b), the token is removed from F₂d as soon as F₁ returns to its nominal state, however other scenarios might be considered.

3. High level dependability model

The high level dependability model is formed by the function’s states and the link model that gathers the set of states and events related to the system’s structural behavior. This behavior is modeled by the so-called structural model and then it is connected to F and F places through an interface model. The link model is thus made up of the structural model and of the interface model.

The structural model represents the behavior of the hardware and software components taking into account fault-tolerance mechanisms, maintenance policies as well as dependencies due to the interactions between different components.

The interface model connects the structural model with its functional state places by a set of immediate transitions.

In this section we concentrate mainly on the interface model. In particular, we assume that the structural model can be built by applying one of the many existing modular modeling approaches (see e.g., [5, 9, 10, 11]), and we focus on its refinement in section 4. Note that the structural models presented in this section are not complete. We present simple examples to help understand the notion of interface model before presenting the general interfacing rules.

3.1 Examples of interface models

For sake of simplicity, we first consider the case of a single function then the case of multiple functions.

Single Function: Several situations may be taken into account. Since the two most important cases are the series and the combination series-parallel components, we limit the illustrations to these two basic cases which allow modeling of any system. More details are given in [3].

Series case: Suppose function F carried out by a software component S and a hardware component H. Then, F and F places’ markings depend upon the markings of the hardware and software components models (Fig. 5).

Multiple Functions: Consider two functions (the generalization is straightforward) and let \{C₁\} (resp. \{C₂\}) be the set of components associated to F₁ (resp. F₂). We distinguish the case where functions do not share resources (such as components or repairmen), from the case where they share some. Examples of these two cases are presented hereafter.
3.2. Interfacing rules

The interface model \( M_I \) connects the system’s components with their functions by a set of transitions. This model is a key element in our approach. Particular examples of interface models have been given in Figures 5 to 7. In this section the general organization of the interface model is presented. Interfacing rules have been defined in formal terms. However, the main rules are stated here in an informal manner.

Upstream and downstream \( M_I \) have the same number of immediate transitions and the arcs that are connected to these transitions are built in a systematic way:

- **Upstream \( M_I \):** It contains one function transition \( t_F \) for each series (set of) component(s), to mark the function’s up state place, and one component transition \( t_CX \) for each series, distinct component that has a direct impact on the functional model, to unmark the function’s up state place.
  - Each \( t_F \) is linked by an inhibitor arc to the function’s up state place, by an arc to the function’s up state place and by one bidirectional arc to each initial (ok) component’s place;
  - Each \( t_CX \) is linked by an arc to the function’s up state place and by one bidirectional arc to each failure component’s place.

- **Downstream \( M_I \):** It contains one function transition \( t'_F \) for each series (set of) component(s), to unmark the function’s failure state place, and one component transition \( t'_CX \) for each series, distinct component that has a direct impact on the functional model, to mark the function’s failure state place.
  - Each \( t'_F \) is linked by an arc to the function’s failure state place and by one bidirectional arc to each initial (ok) component’s place;
  - Each \( t'_CX \) is linked by an inhibitor arc to the function’s failure state place, by an arc from the function’s failure state place and by one bidirectional arc to each component’s failure place.
4. Refinement of the structural model

We assume that the structural model is organized in a modular manner, i.e., it is composed of sub-models representing the behavior of the system’s components and their interactions. For several reasons, the first model that is built, starting from the functional-level model, may be not very detailed. One of these reasons could be the lack of information in the early system’s selection and development phases. Another reason could be the complexity of the system to be modeled. To master this complexity a high level model is built and then refined progressively.

As soon as more detailed information is available concerning the system’s composition and events, governing component evolution, the structural model can be refined.

Another refinement may be done regarding event distributions. Indeed, an assumption is made that all events governing the system’s behavior are exponentially distributed, which, in some cases, is not a good assumption. In particular, failure rates of some components may increase over time.

Model refinement allows detailed behavior to be taken into account and leads to more detailed results compared to those obtained from a high level model. In turn, these detailed results may help in selecting alternative solutions for a given structure. For our purpose, we consider three types of refinement: Component, state/event and distribution. Given the fact that the system’s model is modular, refinement of a component’s behavior is undertaken within the component sub-model and special attention should be paid to its interactions with the other sub-models. However, in this paper due to the lack of space, we will mainly address the new dependencies created by the refinement, without discussing those already existing.

Component refinement consists in replacing a component by two or more components. From a modeling point of view, such a refinement leads to the transformation of the component’s sub-model into another sub-model. Our approach is to use the same transformation rules as those used for the interface model presented in section 3.

State/event fine-tuning consists in replacing, by a subnet, the place/transition corresponding to this state/event. We define basic refinement cases, whose combination covers most usual possibilities of state/event refinement.

For distribution adjustment, we use the method of stages. Consider an event whose distribution is to be transformed into a non-exponential one. This method consists in replacing the transition associated with this event, by a subnet. We have adapted already published work to take into account dependencies between the component under consideration and components with which it interacts. This is done without changing the sub-models of the latters.

A section is devoted to each refinement type.

4.1. Component decomposition

Consider a single function achieved by a single software component on a single hardware component. Suppose that the software is itself composed of N components. Three basic possibilities are taken into account (combinations of these three cases model any kind of system):

- The N components are redundant, which means that they are structurally in parallel;
- The N components are in series;
- There are Q components in parallel and R+1 components in series (with Q+R=N).

Our goal is to use refinement rules identical, as far as possible, to the ones used in Section 3.

In the following we explain how a single component is replaced by its N components. These decompositions are respectively called parallel, series and mixed.

4.1.1. Parallel decomposition. Consider software S’s decomposition into two redundant components S1 and S2. Thus, S’s up state is the result of S1 or S2’s up states, and S’s failure state is the combined result of S1 and S2’s failure states.

4.1.2. Series decomposition. Consider the decomposition of software S into two series components S1 and S2. Hence, this case is identical to the one presented in Fig. 5 when replacing F by S, H by S1 and S by S2.

Figure 8 gives a GSPN model of this case. The generalization to N components is straightforward. It is worth mentioning that the interface model between the system and its components is built exactly in the same manner as the interface model between a function and its associated components.

4.1.2. Series decomposition. Consider the decomposition of software S into two series components S1 and S2. Hence, this case is identical to the one presented in Fig. 5 when replacing F by S, H by S1 and S by S2.
4.1.3. Mixed decomposition. Suppose S is composed of three components: S1, S2 and S3, where S3 is in series with S1 and S2, that are redundant. This case is identical to the example presented in Fig. 6 when replacing F by S and H by S3.

4.1.4. Conclusion. In all the cases illustrated above, we have considered only one token in each initial place. Identical components can be modeled by a simple model with K tokens in each initial place. When refining the behavior of such components, a dissymmetry in their behavior may appear. Indeed, this is due to the fact that some components that have the same behavior at a given abstraction level, may exhibit a slightly different behavior when more details are taken into account. If this is the case, one has to modify the model of the current abstraction level before refinement. This may lead to changing the interface model either between the functional-level and the structural model, or between two successive structural models. This is the only case where refinement leads to changing the model at the higher level.

4.2. State/Event fine-tuning

In GSPNs, places correspond to system’s states and timed transitions to events that guide state changes. The fine-tuning of places/transitions allows more detailed behavior to be modeled. Refinement has been studied in Petri nets ([13, 12]) and more recently in Time Petri Nets [8].

Our goal is to detail the system’s behavior by refining the underlying GSPN. Our sole constraint is to ensure that the net’s dynamic properties (aliveness, boundness and safeness), at each refinement step, are preserved. The main motivation for model refinement is to have more detailed results about system behavior, that better reflect reality.

We define three basic refinement cases. Combinations of these three cases cover most usual situations for dependability models’ refinement. They are given in Table 1.

<table>
<thead>
<tr>
<th>TR1: Separation into two events</th>
<th>Two competing events</th>
</tr>
</thead>
<tbody>
<tr>
<td>TR2: Sequence of events</td>
<td>Refinement of the action represented by transition T</td>
</tr>
<tr>
<td>TR3: State refinement</td>
<td>t1 = p1 \equiv \text{prob. of firing} t_1, \ t2 = p2 \equiv \text{prob. of firing} t_2 and \ p_1 + p_2 = 1</td>
</tr>
</tbody>
</table>

Table 1. State/Event refinement

Fig. 9(b). The resulting model is presented in Fig. 9(c).

Finally, we model the error detection efficiency by applying TR3. Detected errors allow immediate system’s repair. We then add a perception latency (transition T_{1/2}), Fig. 9(d). This latency is important to be modeled because, as long as the non-detected error is not perceived, the system is in a non-safe state. Repair can be performed only after perception of the effects of such errors.

This is a small example of a state/event refinement application. Other details can be added to the model using the cases presented in this section.

4.3. Distribution adjustment

It is well known that the exponential distribution assumption is not appropriate for all event rates. For example, due to error conditions accumulating with time and use, the failure rate of a software component might increase.

The possibility of including timed transitions with non-exponential firing time is provided by the method of stages [7]. This method transforms a non Markovian process into a Markovian one, by decomposing a state (with a non exponential firing time distribution) into a series of k successive states. Each of these k states will then have a negative exponential firing time distribution, to simulate an increasing rate. In GSPNs, a transition, referred to as extended transition, is replaced by a subnet to model the k stages.

The transformation of an exponential distribution into a non-exponential one might create new timing dependencies. Indeed, the occurrence of some events in other components might affect the extended transition. For example, the restart of a software component might lead to the restart of
the component under consideration (that has an increasing failure rate) and thus stop the accumulation of error conditions, bringing back the software under consideration to its initial state.

In previously published work [1, 2], the dependency between events is modeled only by concurrent transitions enabled by the same place. This is not very convenient when several components interact with the component under consideration, as it could lead to changing their models. We have adapted this extension method to allow more flexibility and take into account this type of dependency.

The salient idea behind our approach is to refine the event’s distribution without changing the sub-models of the components, whose behavior may affect the component under consideration (when assuming a non-exponential distribution).

In the rest of this section, we first present the extension method presented in [2] and then present our adapted extension method.

4.3.1. Previous work. Concerning the transitions’ timers, three memory policies have been identified and studied in the literature, namely, resampling, age memory and enabling memory. The latter is well adapted to model the kind of dependency that is created when modeling system’s dependability as mentioned above. It is defined as follows: At each transition firing, the timers of all the timed transitions that are disabled by this transition are restarted, whereas the timers of all the timed transitions that are not disabled hold their present values.

In [1] and [2] an application of the enabling memory policy in structural conflict situations has been given. It concerns the initial model of Fig. 10, in which transition $T_1$ to be extended is in structural conflict with transition $T_{res}$.

When applying the enabling memory policy as given in [2] to transition $T_1$ of Fig. 10, the resulting model is presented in Fig. 11. In this figure, the $k$ series stages are modeled by transitions $t_c^i$, $i=1,2,3$ and $T_1$, and places $P_1, P_2$ and $P_3$. Token moving in these places is controlled by the control places $P_{c1}, P_{c2}$ and $P_{c3}$.

After removal of the token from $S$ by firing of transition $T_{res}$, the clearing of places $P_1, P_2$ and $P_3$ is accomplished in two steps. As soon as $S$ becomes empty, immediate transitions $t_1, t_2$ and $t_3$ are fired as many times as needed to
remove all tokens from these three places. At the end of this step, places $P_{c3}$ and $P_3$ are marked with one token each. The return to the initial state is then performed by immediate transition $t_4$ that puts one token in place $P_{c1}$, after places $P_1$, $P_2$ and $P_3$ are empty.

4.3.2. Enabling memory with external dependencies. Our approach replaces the transition to be extended by two subnets: One internal to the component, to model its internal evolution, and a dependency subnet, that models its interaction with other components. The initial model is given in Figure 12(a). In this model we assume that $T_1$, $T_{dis1}$ and $T_{dis2}$ are exponentially distributed. Suppose that in refining $T_1$’s distribution, its timer becomes dependent on on $T_{dis1}$ and $T_{dis2}$. The transformed model is given in Fig. 12(b). A token is put in $P_{dep}$ each time the timer of transition $T_1$ has to be restarted, due to the occurrence of an event that disables the event modeled by $T_1$ (firing of $T_{dis1}$ or $T_{dis2}$ in other components models). Like in the previous case, this is done in two steps. As soon as place $P_{dep}$ is marked, $t_1$, $t_2$ and $t_3$ are fired many times as needed to remove all tokens from these three places. The return to the initial state is performed by transition $t_4$ that removes a token from place $P_{dep}$ and puts one token in place $P_{c1}$, after places $P_1$, $P_2$ and $P_3$ are empty.

Note that transitions $t_1$, $t_2$ and $t_3$ replace respectively $t_1$, $t_2$ and $t_3$. Also, we simplified Fig. 12(b), by replacing place $P_3$ by an inhibitor arc between $t_4$ and $P_{c1}$. Thus, the two major differences between Figures 11 and 12(b) are: 1) Place $P_1$ of Fig. 12(b) is replaced by an inhibitor arc going from place $P_{c1}$ to immediate transition $t$; 2) Place $P_{dep}$, that manages dependencies between this net and the rest of the model, is added.

5. Application to I&C systems

In this section we illustrate our modeling approach. Due to space limitations we only present a small part of it.

We start by presenting the functional-level model for a general I&C system. Then we describe how the high-level dependability model is built for one of the I&C systems. Finally we show some results concerning a small part of a dependability model is built for one of the I&C systems.

An I&C system performs five main functions: Human-machine interface (HMI), processing (PR), archiving (AR), management of configuration data (MD), and interface with other parts of the I&C system (IP). The functions are linked by the partial dependencies: HMI $\leftarrow$ {AR, MD}, PR $\leftarrow$ MD, AR $\leftarrow$ MD and IP $\leftarrow$ MD. These relations are modeled by the functional-level model depicted in Fig. 13.

To illustrate the second step of our modeling approach, we consider the example of an I&C system composed of five nodes connected by a Local Area Network (LAN). The mapping between the various nodes and their functions is given in Fig. 14. Note that while HMI is executed on four nodes, Node 5 runs three functions. Nodes 1 to 4 are composed of one computer each. Node 5 is fault-tolerant: It is composed of two redundant computers. The initial structural model of this I&C is built as follows:

- Node 1 to Node 3 – in each node, a single function is achieved by one software component on a hardware component. Its model is similar to the one presented in Figures 5 and 15 (that will be explained later);

- Node 4 – has two functions that are partially dependent. Its functional-level model will be similar to $F_1$ and $F_2$’s functional-level model given in Fig. 4(b). Its structural model will be similar to the one depicted in Fig. 7, followed by a model slightly more complex than the one of Figure 15;

- Node 5 – is composed of two hardware components with three independent functions each. Its structural model is more complex than the one given in Figure 15 due to the redundancy.
Figure 13. Functional-level model for I&C systems

- LAN – the LAN is modeled at the structural level by the structural dependencies that it creates.

The complete high level dependability model for this system is composed of 41 places and 19 tokens. The other two I&C systems of our case study are composed of 76 places and 38 tokens and of 27 places and 13 tokens. It is worth mentioning that these model sizes correspond to the high-level models. After refinement, the models are much larger, as it is illustrated on the following example.

Let us consider the simple case of Fig. 5. The associated detailed structural model is given in Fig. 15 in which the \( S_{k.o} \) place of Fig. 5, corresponds to either place \( S_{r.d} \) or \( S_{s.r} \). The detailed GSPNs presented are obtained using the rules described in section 4.2. The following assumptions and notations are used:

- The activation rate of a hardware fault is \( \lambda_{h} \) (Tr\(_{1}\)) and of a software fault is \( \lambda_{s} \) (Tr\(_{3}\));
- A permanent hardware fault (resp. software) is detected by the fault-tolerance mechanisms with probability \( d_{h} \) (resp. \( d_{s} \) for software faults). The detection rate is \( \delta_{h} \) (Tr\(_{5}\)) for the hardware, and \( \delta_{s} \) (Tr\(_{7}\)) for the software;
- The effects of a non detected error are perceived with rate \( \pi_{h} \) (Tr\(_{4}\)) for the hardware, and rate \( \pi_{s} \) (Tr\(_{8}\)) for the software;
- Errors detected in the hardware component require its repair: repair rate is \( \mu \) (Tr\(_{5}\));
- Permanent errors in the software may necessitate only a reset. The reset rate is \( \rho \) (Tr\(_{6}\)) and the probability that an error induced by the activation of a permanent software fault disappears with a reset is \( r \) (Tr\(_{7}\));
- If the error does not disappear with the software reset, a re-installation of the software is done. The software’s re-installation rate is \( \sigma \) (Tr\(_{10}\)).

Note that a temporary fault in the hardware may propagate to the software (Tr\(_{11}\)) with probability \( p \). We stress that when the software component is in place \( S_{r.d} \) or \( S_{s.r} \), it is in fact not available, i.e., in a failure state.

Also, when the hardware is in the repair state, the software is on hold. The software will be reset or re-installed as soon as the hardware repair is finished. Due to the size of the subsequent model, this case is not represented here.

Thus, from the original 4 places model, we have a 15 places model after refinement.

6. Conclusions

Our modeling approach follows in the footsteps of most of the existing work on dependability modeling. Where this approach is unique is in the inclusion of the system’s functional specifications into the dependability model, by means of a functional-level model. Also, it allows modeling of one system from its functional specification up to its implementation. The existing refinement techniques are conceived in order to preserve the result values. On the contrary, ours provides more accurate models and associated results.

Thus, the modeling approach presented in this paper gives a generally-applicable process for system’s analysis, based on generalized stochastic Petri nets. This process involves a stepwise refinement in which dependencies are introduced at the appropriate level of refinement. A careful and precise definition of the constructs and of the refinement process is given. Indeed, we have shown how starting from functional specifications, a functional-level model can be
Figure 15. Structural model of a software and a hardware components

transformed progressively into a dependability model taking into account the system’s structure. We have also shown how the structural model can be refined to incorporate more detailed information of the system’s behavior. Refinement is a very powerful tool for mastering progressively model construction. It will allow experimented, but not necessarily specially-trained, modelers to analyze the dependability of one or several systems and compare their dependability at the same level of modeling abstraction, if required.

The approach was illustrated here on simple examples related to a specific structure of an instrumentation and control system in power plants. However, we have applied this approach to three different I&C systems to identify their strong and weak points, in order to select the most appropriate one.

Acknowledgements

The authors wish to thank Mohamed Kãâniche for his helpful comments on an earlier version of this paper. We also wish to thank the anonymous reviewers for their useful suggestions for improvement.

References


