# A System Performance in Presence of Faults Modeling Framework Using AADL and GSPNs

Belhassen MAZIGH[1] and Kais BEN FADHEL[1]

Department of Computer Science, Faculty of Science of Monastir,
Avenue of the environment, 5019, Monastir - Tunisia
belhassen.mazigh@gmail.com

**Abstract.** The increasing complexity of new-generation systems which take into account interactions between hardware and software components, particularly the fault-tolerant systems, raises major preoccupations in various critical application domains.These preoccupations concern principally the modeling and analysis requirements of these systems.Thus, designers are interested in the verification of critical proprieties and particularly the Performance and Dependability analysis.
In this paper, we present an approach for modeling and analyzing systems with hardware and software components in the presence of faults: an approach based on Architecture Analysis and Design Language (AADL) and Generalized Stochastic Petri Nets (GSPN). This approach starts with the description of the system architecture in AADL. This description is enhanced by the use of two annexes, the existing Error Model Annex and the Activity Model Annex (proposed Annex). By applying existing transformation rules of GSPN models, we build two models: GSPNs Dependability and Performance models. Finally, we apply our composition algorithm, to obtain a global GSPN model allowing to analyze Dependability and Performance measurements.

## 1 Introduction

The quantity and complexity of systems continues to rise and generally the cost of manufacturing these systems is very high. To this end, many modeling and analysis approaches are more and more used in industry with the aim to control this complexity since the design phase. These approaches must be accompanied by languages and tools. An explicit approach presents the building of a GSPN of a complex system from the GSPNs of its components, taking into account the interactions between the components, is presented in [1]. These approaches are referred to as block modeling approach and incremental approach respectively.
AADL is among the languages having a growing interest in industry-critical systems. This language has been standardized by the "International Society of Automotive Engineers" (SAE) in 2004 [2, 3] , to facilitate the design and analysis of complex systems, critical, real-time in areas such as avionics, automotive and aerospace [4]. It provides a standardized graphical and textual notation to describe the hardware and software architectures. It is designed to be extensible

in order to adapt and analyze architectures execution that the core language does not fully support. The extensions may take the form of new properties and notations specific to analysis that may be associated with the architectural description in the form of annexes. Among these approaches, the one proposed in [5] allows specialists in AADL to obtain Dependability measures, based on a formal approach. This approach aims to control the construction and validation of Dependability models in the form of GSPN. But in reality the designers want to have a final model of the system that allows them to analyze Dependability and Performance which take into account functional and dysfunctional aspects of the system. In this paper we propose an extension to this approach so that the final model of the system allows us to analyze the attributes of Dependability and Performance measures. The outline of the paper is as follows. In Section 2, we define the AADL concepts. Then we present our approach in Section 3 and its application on a simple example in Section 4. We conclude in section 5.

## 2 AADL concepts

The AADL core language is used to analyze the impact of different architectural choices on the properties of the system [6] and [7]. An architecture specification in AADL describes how components are combined into subsystems and how they interact. Architectures are described hierarchically. Components are the basic bricks of AADL architectures. They are grouped into three categories: 1) software (process, subprogram, data, and thread), 2) hardware (processor, memory, device, bus) and 3) composite (system). AADL components may be composed of sub-components and interconnected through features such as ports. These features specify how the components are interfaced with each other. Each AADL system component has two levels of description: the component type and the component implementation. The type describes how the environment sees that component, i.e., its properties and features. Examples of features are **in** and **out** port that represent access points to the component. One or more component implementations may be associated with the same component type.
As mentioned in the introduction, AADL is designed to be extensible in order to adapt and analyze architectures execution the core language that does not fully support. The Error Model Annex is a standardized annex [3] that completes description of the capabilities of the core language AADL, providing a textual syntax with a precise semantics to be used to describe the characteristics of Dependability related to AADL architectural models. AADL error models are defined in libraries and can be associated with software and hardware components as well the connection between them. When an error model is associated with a component, it is possible to customize it by setting component-specific values for the arrival rate or the probability of occurrence of error events and error propagation declared in the error model.
In the same way as for AADL components, the error model has two levels of description: the error model type and the error model implementation. The error model type declares a set of error states, error events and error propagation

circulating through the connections and links between architecture model. In addition, the user can declare properties of type **Guard** to control the propagation. The error model implementation declares states transitions between states, triggered by events and propagation declared in the error model type [8]. Note that **in** and **out** features identify respectively incoming propagation and outgoing propagation. An **out** propagation occurs in an error model source with property of occurrence specified by the user. The error model source sends the propagation through all ports and connections of the component to which error model is associated. As a result, **out** propagation arrives at one or more error models associated with receptor components. If an error model receiver, declares **in** propagation with the same name as the **out** propagation received, the **in** propagation can influence its behavior.

## 3 Modeling approach

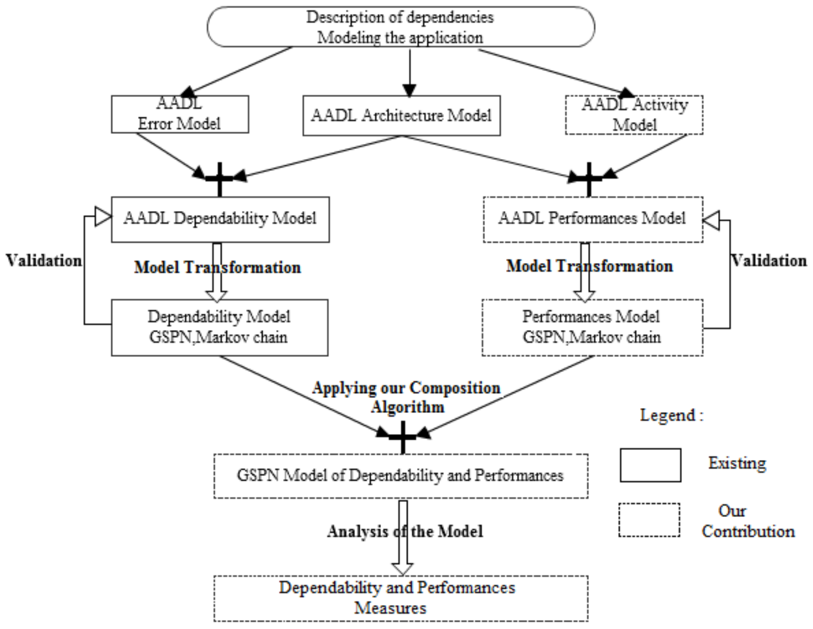


**Fig. 1.** Proposed Approach

We can describe our method, presented in Figure 1, into five main steps:

1. The first step focuses on the modeling of the system architecture in AADL.

2. The second step concentrates on building an AADL Dependability Model and a AADL Performance Model:
   – Building Dependability model is done by the association of AADL Error models to the components of AADL Architectural Model, see [5] for more details.
   – Building performance model is done by the association of our AADL Activity models to the components of AADL Architectural Model. An Activity Model is similar to the Error Model, this model works as an Error Model, but the state change is performed by passing from a reliable state to another with integration of performance metrics associated to the properties that must be defined in AADL language. Syntactically it is inspired by the standard Error Model. Activity models are devoted to describe the components activities.
3. The third step is to build two GSPN models, of Dependability and Performance, from two AADL models using the transformation rules presented in [5].
4. The fourth step is dedicated to the application of our composition algorithm. This algorithm receives as an input two GSPN models, a GSPN model of Dependability and a GSPN model of Performance, each model is composed of sub-networks of components and sub-networks of dependencies. We obtain a global GSPN model which allows to analyze Dependability and Performance measurements for hardware and software systems in the presence of faults.
5. The fifth step is dedicated to analyzing the global GSPN model to obtain measures of Dependability and Performance. This last step is entirely based on classical algorithms for processing GSPN models and it is not the subject of this work, and therefore will not be detailed here.

The next section presents the application of our approach to a simple example.

## 4 Application of our approach

To illustrate our approach, we use a simple system constituted by a processor PR which executes a user process P. The processor allocation is made according to the following policy: we define a quantum of time (e.g. q) for the processor PR. A process P sends an allocation request for the processor PR. If the processor is free then it accepts the request, the process pass to the execution state. The process can pass into a blocking state (blocked) if it expects other resource (e.g. end of an input output). If the execution time is smaller than the quantum then the process completes its task and passes to termination state otherwise the processor interrupt the execution of process, so that another process could be executed (the processor is retained for the current process until the end of quantum). When the process passes into the blocking state the processor can be allocated to another process. The processor is not necessarily free. The process then moves to the ready state. The ready state is the standby state of the processor. It is clear

that there is a structural dependence between the processor PR and process P. Defects in materials could be propagated and influence behavior of software associated with it.
We will first establish an AADL model of Dependability, with the corresponding transformation steps into GSPN and we do the same thing to develop an AADL Performance model. Finally, we apply our composition algorithm to obtain a global Performance model in presence of software and hardware faults.

### 4.1 Construction of Dependability model

```
Error Model comp_hard
    features
    Error_Free : initial error state;
    Activation : error state;
    Erroneous : error state;
    Failed : error state;
    Fault : error event {Occurrence => poisson $\lambda_{1h}$};
    Temp_Fault : error event {Occurrence => poisson $\lambda_{2h}$};
    Perm_Fault : error event {Occurrence => poisson $\lambda_{3h}$};
    Restart : error event {Occurrence => poisson $\lambda_{4h}$};
    Recover : error event {Occurrence => poisson $\lambda_{5h}$};
End comp_hard;
Error Model implementation comp_hard.general
    transitions
    Error_Free - [Fault]- > Activation;
    Activation - [Perm_Fault]- > Failed;
    Activation - [Temp_Fault]- > Erroneous;
    Failed - [Restart]- > Error_Free;
    Erroneous - [Recover]- > Error_Free;
End comp_hard.general;
```

**Fig. 2.** Error Model of hardware component

We propose generic error models (without propagation) for the hardware and software components (Figure 2 and 3) inspired by the works [8], [9], [10] and [11]. These two models are respectively associated with the implementation of the processor PR and the process P components. Because the fact that the process P is running on the processor PR, the errors in the processor (hardware faults) can affect the process executed as follows:

- If the fault is temporary, it can transmit errors in the process. The error sent by the processor leads the process in state relating to the activation of fault (state '*Detect_ERR*').
- If the fault is permanent, this failure has two consequences: stopping the software components and synchronizing the restoration actions since the

relaunch of software components is conditioned by the end of the repair of hardware component on which they were executed.

```
Error Model comp_soft
   Features
   Error_Free : initial error state;
   Detect_ERR : error state;
   ERR_ND : error state;
   ERR_D : error state;
   Erroneous : error state;
   Failed : error state;
   Fault : error event {Occurrence => poisson $\lambda_{1s}$};
   NonDetect : error event {Occurrence => poisson $\lambda_{2s}$};
   Eliminate : error event {Occurrence => poisson $\lambda_{3s}$};
   PerceiveFail : error event {Occurrence => poisson $\lambda_{4s}$};
   Detect : error event {Occurrence => poisson $\lambda_{5s}$};
   Temp_Fault : error event {Occurrence => poisson $\lambda_{6s}$};
   Perm_Fault : error event {Occurrence => poisson $\lambda_{7s}$};
   Restart : error event {Occurrence => poisson $\lambda_{8s}$};
   Recover : error event {Occurrence => poisson $\lambda_{9s}$};
End comp_soft;
Error Model implementation comp_soft.general
   transitions
   Error_Free − [Fault]− > Detect_ERR;
   Detect_ERR − [NonDetect]− > ERR_ND;
   Detect_ERR − [Detect]− > ERR_D;
   ERR_ND − [Eliminate]− > Error_Free;
   ERR_ND − [PerceiveFail]− > Failed;
   ERR_D − [Temp_Fault]− > Erroneous;
   ERR_D − [Perm_Fault]− > Failed;
   Failed − [Restart]− > Error_Free;
   Erroneous − [Recover]− > Error_Free;
End comp_soft.general;
```

**Fig. 3.** Error Model of software component

Figure 4 shows only what is added to the error model associated with the processor in order to describe the structural dependency after a recovery failure. The error model type for processors, *comp_hard*, is completed with lines $R_O^1$ and $R_O^2$ in order to include 'out' error propagation declarations ('*PR_Failed*','*PR_Ok*'), '*PR_Failed*' causes the processes failures while '*PR_Ok*' is used to synchronize the repair of the processor with the restart of the process. The error model implementation, *comp_hard.general*, takes into account the sender side of the recovery dependency, it declares one transition triggered by each of the two newly introduced 'out' propagation (see lines $R_O^{11}$ and $R_O^{22}$ of Figure 4). When one of the 'out' propagation occurs, the processor remains in the same state and the propagation remains visible until the processor leaves this state. Figure 5 shows what is added to the error model associated with a process in order to describe the structural dependency. The error model type, *comp_soft*, is com-

```
      Error Model comp_hard
         features
         [...]
R_O^1    PR_Failed : out error propagation {Occurrence => fixed q1};
R_O^2    PR_ok : out error propagation {Occurrence => fixed q2};
      End comp_hard;
      Error Model implementation comp_hard.general
         transitions
         [...]
R_O^11   Failed − [out PR_Failed]− > Failed;
R_O^22   Error_Free − [out PR_ok]− > Error_Free;
      End comp_hard.general;
```

**Fig. 4.** Error Model for processor component

pleted with lines $L_0, L_1 and\ L_2$. Line $L_0$ declares an additional state in which the process is allowed to restart and Lines $L_1$ and $L_2$ declares 'in' propagation. The error model implementation, *comp_soft.generale*, takes into account the recipient side of the structural dependency by declaring five transitions triggered by the 'in' propagation '*PR_Failed*' (see lines $L_1^1, L_1^2, L_1^3, L_1^4$ and $L_1^5$ of Figure 5) and leading the process from each state (other than '*Failed*') to the '*Failed*' state. The process is authorized to move from the '*Failed*' state to '*InRestart*' state only when it receives the '*PR_Ok*' propagation (see line $L_2^1$ of Figure 5). Since the recovery procedure is now engaged by the '*InRestart*' state , the AADL transition ($Failed - [Restart]- > Error\_Free$) will be replaced by the transition ($Failed - [PR\_Ok]- > InRestart$, see line R).

```
      Error Model comp_soft
         features
         [...]
L_0     InRestart : error state;
L_1     PR_Failed : in error propagation;
L_2     PR_ok : in error propagation;
      End comp_soft
      Error Model implementation comp_soft.general
         transitions
         [...]
L_1^1   Error_Free − [inPR_Failed]− > Failed;
L_1^2   Detect_ERR − [inPR_Failed]− > Failed;
L_1^3   ERR_ND − [inPR_Failed]− > Failed;
L_1^4   ERR_D[inPR_Failed]− > Failed;
L_1^5   Erroneous − [inPR_Failed]− > Failed;
L_2^1   Failed − [inPR_OK]− > InRestart;
R       InRestart − [Restart]− > Error_Free;
      End comp_soft.general
```

**Fig. 5.** Error Model for process component

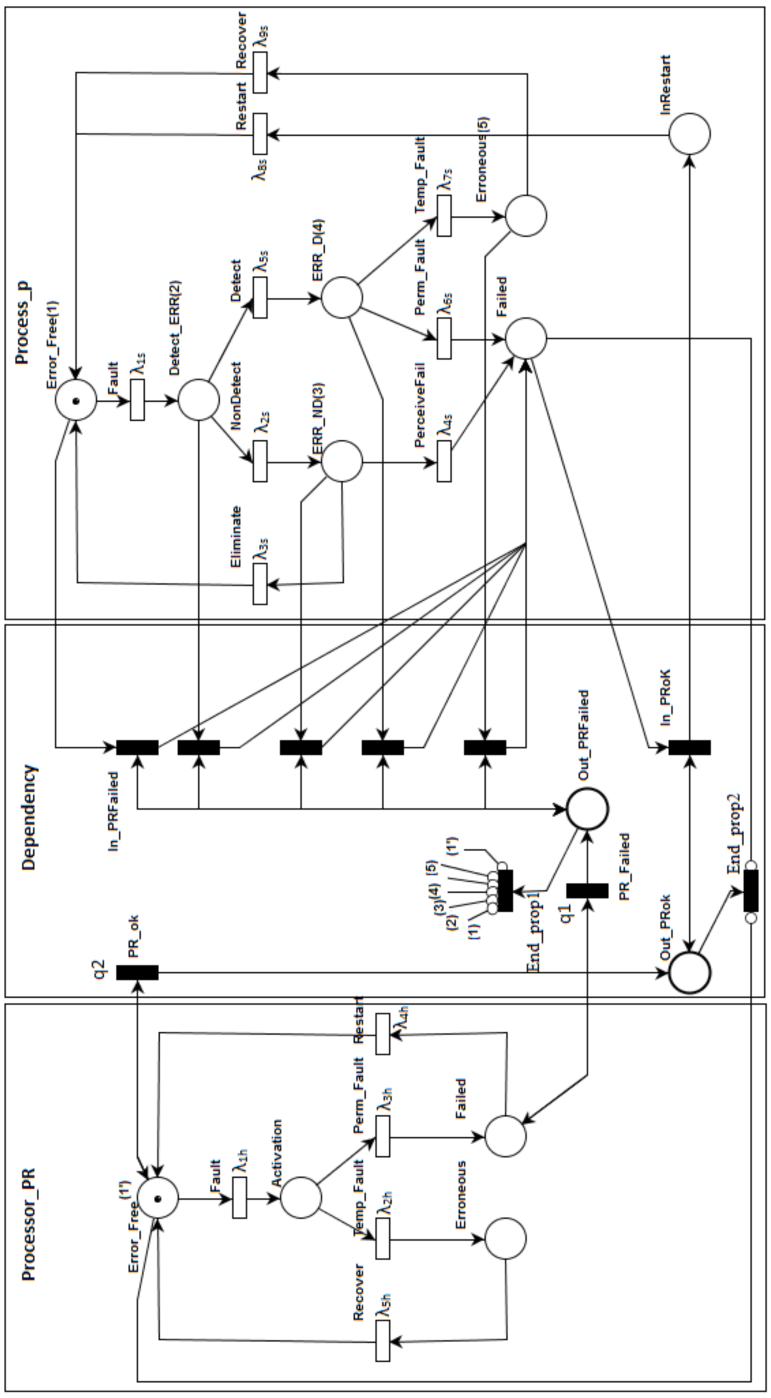


**Fig. 6.** Dependability GSPN Model

By applying transformation rules presented in [5], Figure 6 shows the GSPN obtained when transforming the AADL model corresponding to the process P linked to the processor PR. We note that places with dark circles are places with capacity of one token.

## 4.2 Construction of Performances model

The following section presents the steps for constructing the AADL Performance model with corresponding transformation steps. For more clarity, we model each component (process P or processor PR) in the presence of internal events and 'in' propagation and then we integrate the 'out' propagation following the description of the system. Figure 7 shows the activity model associated with the processor. As for the error model, we associate the activity model, *processor_PR.imp*, to the implementation of the component processor. The processor is initially free. It will be occupied if it receives an allocation request, '*request*'. It can go from '*Busy*' state to the '*Exp_termination*' state if the '*End_quantum*' event is activated with a rate $\lambda_{6h}$. Or it can pass to the '*Free*' state if it receives an 'in' propagation '*FreePR*'. From the '*Exp_termination*' state it can return to its initial state with a rate $\lambda_{7h}$.

```
Activity Model processor_PR
    Features
    Free : initial state;
    Busy : state;
    Exp_termination : state;
    End_quantum : event {occurrence => poissonsλ6h};
    Initialization : event {occurrence => poissonsλ7h};
    request : in propagation;
    FreePR : in propagation;
End processor_PR;
    Activity Model implementation processor_PR.imp
    transitions
    Free − [in request]− > Busy;
    Busy − [End_quantum]− > Exp_termination;
    Exp_termination − [Initialization]− > Free;
    Busy − [in FreePR]− > Free;
End processor_PR.imp;
```

**Fig. 7.** Activity Model for the processor component

The GSPN model of the processor PR (Figure 8) is obtained by applying the transformation rules (incomplete model).

Figure 9 shows the Activity Model associated with the process P. Initially, the process is in '*Ready*' state. It passes to '*Execution*' state when it receives an 'in' propagation '*Grant*' (it means that the processor starts its execution). It can go from '*Execution*' state to a *Ready* state if it receives an 'in' propagation

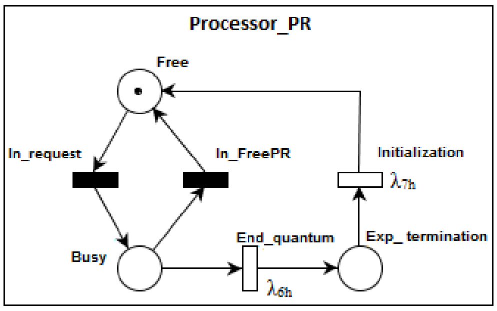
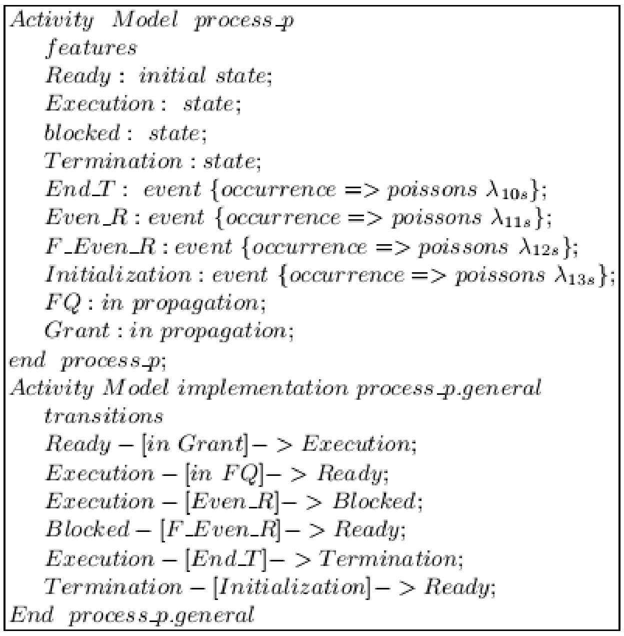


**Fig. 8.** GSPN model for the processor component

'*FQ*' (it is temporarily suspended to allow the execution of another process). It passes from the '*Execution*' state to the '*blocked*' state if the '*Even_R*' event is activated with a rate $\lambda_{11s}$. When the '*F_Even_R*' event occurs it passes to '*Ready*' state. It can go from the '*Execution*' state to a '*Ready*' state, if the '*End_T*' event is activated with a rate $\lambda_{10s}$.



```
Activity Model process_p
   features
   Ready : initial state;
   Execution : state;
   blocked : state;
   Termination : state;
   End_T : event {occurrence => poissons λ10s};
   Even_R : event {occurrence => poissons λ11s};
   F_Even_R : event {occurrence => poissons λ12s};
   Initialization : event {occurrence => poissons λ13s};
   FQ : in propagation;
   Grant : in propagation;
end process_p;
Activity Model implementation process_p.general
   transitions
   Ready − [in Grant]− > Execution;
   Execution − [in FQ]− > Ready;
   Execution − [Even_R]− > Blocked;
   Blocked − [F_Even_R]− > Ready;
   Execution − [End_T]− > Termination;
   Termination − [Initialization]− > Ready;
End process_p.general
```

**Fig. 9.** Activity Model for the process component

By applying the transformation rules we obtain the GSPN model of the process P (figure 10, incomplete model).

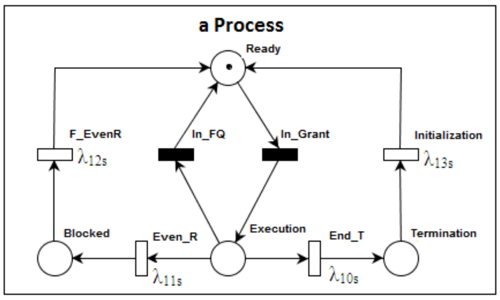


**Fig. 10.** GSPN model for the process component

Figure 11 shows just what is added to the activity model associated with the processor in order to describe the interaction between process P and the processor PR. The activity model type for processor, *processor_PR*, is completed with lines $L_O$ and $L_1$ (see figure 11) in order to include 'out' error propagation declarations such as:

- Line $L_O$ declares an 'out' propagation '*FQ*', which indicate the end of quantum. Its name matches the name of the 'in' propagation declared in the activity model type, *process_p* (see figure 9).
- Line $L_1$ declares an 'out' propagation '*Grant*', which indicates that the processor has given permission to move the process in '*Execution*' state. Its name matches the name of the 'in' propagation declared in the activity model type, *process_p* (see figure 9).

The activity model implementation, *processor_PR.imp*, declares two transitions (lines $L'_O$ and $L'_1$) triggered by the newly introduced 'out' propagation '*FQ*' and '*Grant*'. When one of these two 'out' propagation occurs, the processor remains in the same state and the propagation remains visible until the processor leaves this state.

Similarly, Figure 12 shows what is added to the activity model associated with the process in order to describe the interaction between process P and the processor PR. The activity model type for process, *process_p*, is completed with lines $S_O$ and $S_1$ (see figure 12) in order to include 'out' error propagation declarations such as :

- Line $S_O$ declares an 'out' propagation '*request*', to indicate that the process requires the processor. Its name matches the name of the 'in' propagation declared in the activity model type, *processor_PR* (see figure 7).

```
    Activity Model processor_PR
       features
       [...]
L_O    FQ : out propagation {occurrence => Fixed λ8h};
L_1    Grant : out propagation {occurrence => poissons λ9h};
    End processor_PR;
    Activity Model implementation processor_PR.imp
       transitions
       [...]
L'_1   Busy − [out Grant]− > Busy;
L'_0   Exp_Termination − [out FQ]− > Exp_Termination;
    End processor_PR.imp
```

**Fig. 11.** Activity Model for processor component with interaction

– Line $S_1$ declares an 'out' propagation '$FreePR$'. Its name matches the name of the 'in' propagation declared in the activity model type, *processor_PR* (see figure 7).

The activity model implementation, *process_p.general*, declares three transitions (lines $S_3, S_4$ *and* $S_5$ of figure 12) triggered by the newly introduced 'out' propagation '*request*' and '$FreePR$'.

```
    Activity Model process_p
       features
       [...]
S_0    request : out propagation {occurrence => poissons λ14s};
S_1    FreePR : out propagation {occurrence => Fixed λ15s};
    End process_p;
    Activity Model implementation process_p.general
       transitions
       [...]
S_3    Ready − [out dmdp]− > Ready;
S_4    Blocked − [out FreePR]− > Blocked;
S_5    Termination − [out FreePR]− > Termination;
    End process_p.general;
```

**Fig. 12.** Activity Model for process component with interaction

Figure 13 shows the GSPN model obtained by applying the transformation rules.

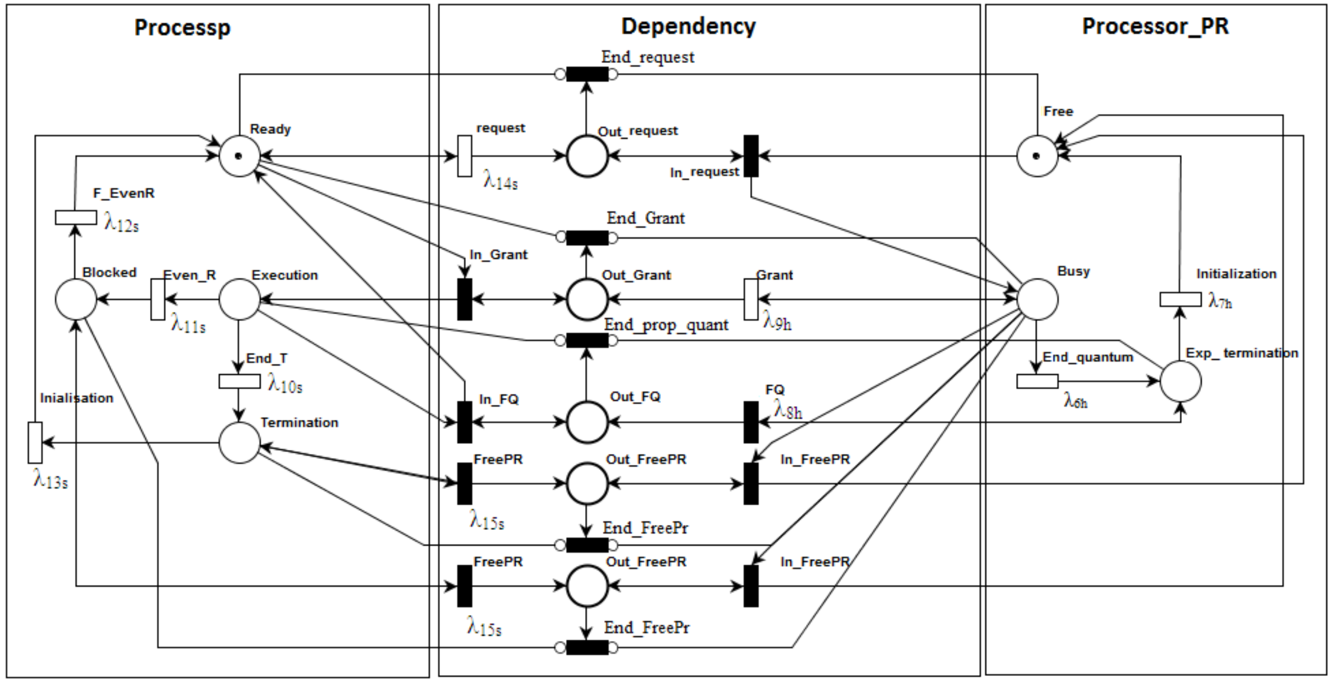


**Fig. 13.** GSPN Model of Performance

### 4.3 Application of composition algorithm

Now after the construction of the Performance and Dependability GSPN models, we apply our composition algorithm on these two models. Each model is composed of components sub-nets and dependencies subnets. Each component has a GSPN model of Dependability and a GSPN model of Performance. The basic idea of this algorithm is to connect each component sub-net of Performance with the corresponding component sub-net of Dependability. This algorithm is defined to ensure that the obtained global GSPN model is correct by construction (bounded, safe and no deadlock). The main steps of our composition algorithm are the following:

- For each sub-network component of Performance, if the component has no replicas, we add a bidirectional arc which connects the transitions of Performance model component with the place that represents the initial state of the corresponding Dependability model. This rule reflects that if the component is in a state of Performance model, it can move to another state only if there is no activation of a fault. Note that the number of tokens in a sub-net component is always 1 because at a given time a component can be only in one state. It is clear that if there is activation of a temporary fault, after adding a bidirectional arc, the component remains in waiting until the resolution of this fault since transitions in the Performance model are disabled. We note that if there is a place in a Performance model where the activation of a temporary fault does not exist, for all transitions that represent the output transitions of this place, the bidirectional arc is eliminated. In our case if the processor PR in a free state, a temporary fault will never be activated. Now if a permanent fault occurs, the component must regain its initial state. The rule of the link consists in adding timed transitions, and link with a bidirectional arc the initial place of Performance model with the transition *Restart* of Dependability model.
- if the component has replicas, each replica has the same GSPN model of Performance and Dependability. In first step, the addition of bidirectional arcs is applied to each replica. Then immediate transitions are added to represent the switching between the GSPN models.

By applying this algorithm on our models, we obtain Figure 14 which shows the GSPNs models of processor PR and Figure 15 which shows the GSPN models of the process P. The two models constitute the global model of the studied system.

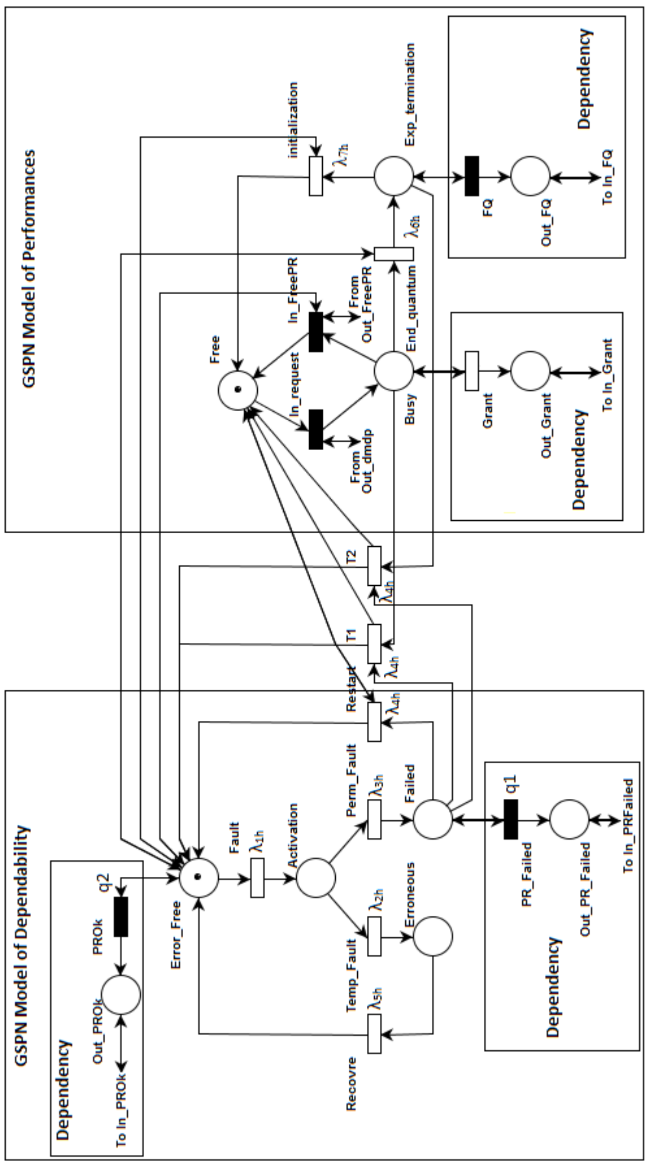


**Fig. 14.** The first part of the Global GSPN model: processor model

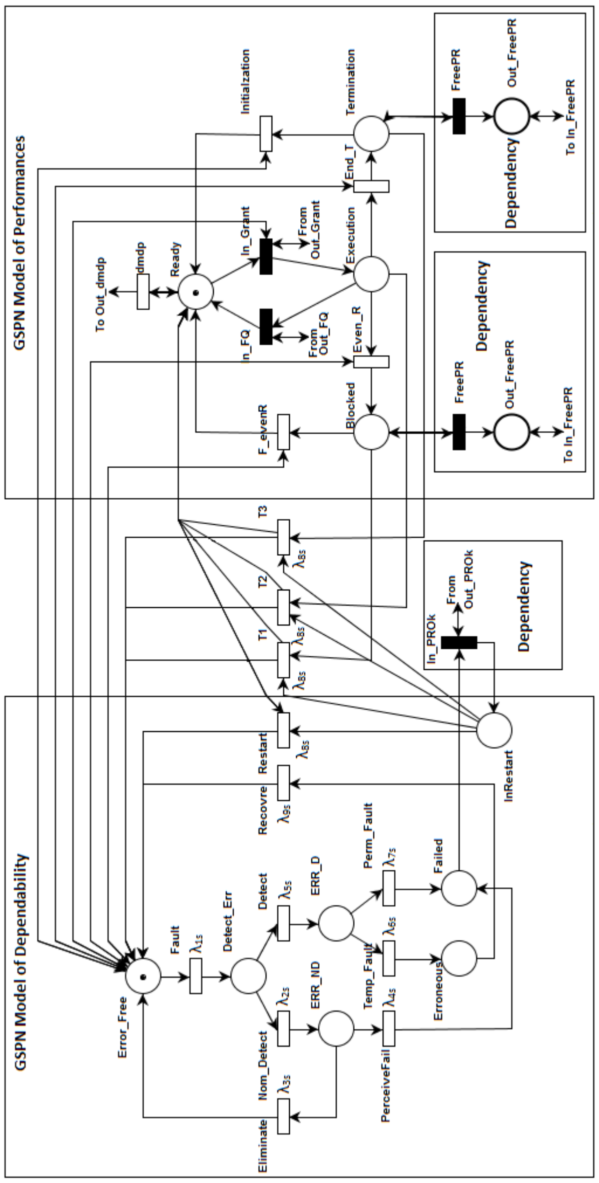


**Fig. 15.** The second part of the Global GSPN model: process model

## 5 Conclusion

In this paper, we have presented an approach based on AADL and Generalized Stochastic Petri Nets for modeling and analyzing Performance and Dependability of systems in the presence of faults. This approach consists of several steps. After modeling the system architecture in AADL, two AADL models are obtained, Dependability and Performance models. They are transformed in two GSPN models by applying transformation rules presented in [5]. Finally, by applying our algorithm we build the global model related to the studied system. Our composition algorithm was implemented in Java language. From Performance and Dependability models of hardware and software components, our algorithm builds a global GSPN model. The obtained GSPN model is a file type PNML exchange format which can be analyzed by tools that support this format such as Tina toolbox [12]. In [13] we applied this approach on the ABS anti-lock complex system.

## References

1. K. Kanoun, M. Borrel, Fault-tolerant system Dependability: Explicit modeling of hardware and software component-interactions, IEEE Transactions on Reliability, 49, (2000).
2. SAE-AS5506. Architecture Analysis and Design Language, SAE, (2004).
3. SAE-AS5506/1, Architecture Analysis and Design Language Annex Volume 1. SAE, (2006).
4. Ana Elena Rugina, Karama Kanoun, Mohamed Kaaniche, System Dependability modeling using AADL langage, 15eme Congres de Maitrise des Risques et de Surete de Fonctionnement, Lille, (2006).
5. Ana-Elena Rugina, Dependability Modeling and Evaluation: From AADL to Stochastic Petri nets, PhD.D. thesis, Institut National Polytechnique de Toulouse, France, (2007).
6. Thomas Vergnaud. Modelisation des Systemes Temps reel Repartis Embarques pour la Generation Automatique d Application Formellement Verifiees. PhD thesis, Ecole Nationale Superieure des Telecommunications, (2006).
7. P. Feiler, B. Lewis, and S. Vestal. The SAE Architecture Analysis & Design Language (AADL) A Standard for Engineering Performance Critical Systems. In Proceedings of the 2006 IEEE Conference on Computer Aided Control Systems Design, (2006).
8. Rogerio de Lemos, Cristina Gacek, Alexander B. Romanovsky, Architecting Dependable Systems IV, Lecture Notes in Computer Science 4615, Springer, (2007).
9. Jean-Claude Laprie, Dependable Computing: Concepts, Limits, Challenges, Invited paper to FTCS-25, the 25th IEEE International Symposium on Fault-Tolerant Computing,Pasadena, California, USA, June 27-30, Special Issue, (1995).
10. M. Borrel, Interactions entre composants materiel et logiciel de systemes tolerant aux fautes - Caracterisation - Formalisation - Modelisation - Application a la surete de fonctionnement du CAUTRA, LAAS-CNRS, These de doctorat, Toulouse, France, (1996).

11. A. Bondavalli, S. Chiaradonna, F. D. Giandomenico, J. Xu, Fault-tolerant system Dependability: Explicit modeling of hardware and software component-interactions, Journal of Systems Architecture, 49, (2002).
12. B. Berthomieu, P.-O. Ribet, F. Vernadat, The tool TINA – Construction of Abstract State Spaces for Petri Nets and Time Petri Nets, International Journal of Production Research, Vol. 42, No 14,(2004).
13. Kais Ben Fadhel, une approche de modelisation et d'analyse des performances et de la surete de fonctionnement : Transformation d'une specification AADL en GSPN, Master de recherche, faculte des sciences de Monastir, Tunisie, (2012).