

A System Dependability Modeling Framework Using AADL and GSPNs

Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaâniche

LAAS-CNRS, University of Toulouse
7 avenue Colonel Roche
31077 Toulouse Cedex 4, France
Phone: +33(0)5 61 33 62 00, Fax: +33(0)5 61 33 64 11
{rugina, kanoun, kaaniche}@laas.fr

Abstract. For efficiency and cost control reasons, system designers' will is to use an integrated set of methods and tools to describe specifications and design, and also to perform dependability analyses. The SAE (Society of Automotive Engineers) AADL (Architecture Analysis and Design Language) has proved to be efficient for architectural modeling. We present a modeling framework allowing the generation of dependability-oriented analytical models from AADL models, to facilitate the evaluation of dependability measures, such as reliability or availability. We propose a stepwise approach for system dependability modeling using AADL. The AADL dependability model is transformed into a GSPN (Generalized Stochastic Petri Net) by applying model transformation rules that can be automated. The resulting GSPN can be processed by existing tools. The modeling approach is illustrated on a subsystem of the French Air Traffic Control System.

Keywords: dependability modeling, evaluation, AADL, GSPN, model transformation.

1 Introduction

The increasing complexity of new-generation systems raises major concerns in various critical application domains, in particular with respect to the validation and analysis of performance, timing and dependability-related requirements. Model-driven engineering approaches based on architecture description languages aimed at mastering this complexity at the design level have emerged and are being increasingly used in industry. In particular, AADL (Architecture Analysis and Design Language) [1] has received a growing interest during the last years. It has been recently developed and standardized under the auspices of the International Society of Automotive Engineers (SAE), to support the design and analysis of complex real-time safety-critical systems in avionics, automotive, space and other application domains. AADL provides a standardized textual and graphical notation for describing software and hardware system architectures and their functional interfaces. AADL may be used to perform various types of analysis to determine the behavior and the performance of

the system being modeled. The language has been designed to be extensible to accommodate analyses that the core language does not support.

Besides describing the systems' behavior in the presence of faults, the developers are interested in obtaining quantitative measures of relevant dependability properties such as reliability, availability and safety. For pragmatic reasons, the system designers using an AADL-based engineering approach are interested in having an integrated set of methods and tools to describe specifications and design, and to perform dependability evaluations. The *AADL Error Model Annex* [2] has been recently standardized to complement the description capabilities of the core language by providing features with precise semantics to be used for describing dependability-related characteristics in AADL models (faults, failure modes, repair policies, error propagations, etc.). However, at the current stage, no methodology and guidelines are available to help the developers in the use of the proposed notations to describe complex dependability models reflecting real-life systems with multiple interactions and dependencies between components. One of our objectives is to propose a structured method for AADL dependability model construction.

The AADL Error Model Annex mentions that stochastic automata such as fault trees and Markov chains can be generated from AADL specifications enriched with dependability-related information. Indeed, Markov chains are recognized to be powerful means for modeling system dependability taking into account dependencies between system components. Usually, they are automatically generated from higher level formalisms such as Generalized Stochastic Petri Nets (GSPNs). The latter allow structural model verification, before the Markov chain generation. Such verification support facilities are very useful when dealing with large models.

During the last decade, various approaches have been defined to support the systematic construction and validation of dependability models based on GSPNs and their extensions (see e.g. [3-5]). We propose to take advantage of such approaches in the context of an AADL-based engineering process, to i) build the dependability-oriented AADL model and to ii) generate dependability-oriented GSPN models from AADL models by model transformation. In this way, the complexity of GSPN model generation is hidden to users familiar with AADL but who have a limited knowledge of GSPNs. The AADL and GSPN models are built iteratively, taking into account progressively the dependencies between the components, and validated at each iteration. The dependability-related information is not embedded in the AADL architectural model. Instead, it is described separately and then plugged in the system's components. The user can easily unplug or replace the dependability-related information. This feature enhances the reusability and the readability of the AADL architectural model that can be used as is for other analyses (e.g., formal verification [6], scheduling and memory requirements [7], resource allocation with the Open Source AADL Tool Environment (OSATE)¹, research of deadlocks and un-initialized variables with the Ocarina toolset²).

To summarize, our objectives are threefold: i) present a structured and stepwise approach for building AADL dependability model, ii) show examples of model transformation rules to generate GSPNs from AADL dependability models and iii)

¹ <http://www.aadl.info/OpenSourceAADLToolEnvironment.html>

² <http://ocarina.enst.fr>

exemplify the proposed approach on a subsystem of the French Air Traffic Control System. The set of model transformation rules is meant to be the basis for the implementation of a model transformation tool completely transparent to the user. Such a tool can be interfaced with one of the existing GSPN processing tools (e.g., Surf-2 [8], Möbius [9], Sharpe [10], GreatSPN [11], SPNP [12]) to evaluate dependability/performance measures.

Compared to our work presented published in [13] and [14], we offer here a global view of our method's steps, by presenting a case study reflecting a real system. In particular, the AADL to GSPN transformation rules are developed and illustrated.

The remainder of the paper is organized as follows. Section 2 discusses related work. Section 3 presents the AADL concepts that are necessary for understanding our modeling approach. Section 4 gives an overview of our framework for system dependability modeling and evaluation using AADL and GSPNs. Section 5 presents examples of rules for transforming AADL into GSPN models. Section 6 applies our approach to a subsystem of the French Air Traffic Control System and Section 7 concludes the paper.

2 Background and Related Work

To the best of our knowledge there are no contributions similar to our work in the current state of the art. Most of the published work on analyses using AADL has focused on the extension of the language capabilities to support formal verifications. For example, the COTRE project [6] provides a design approach bridging the gap between formal verification techniques and requirements expressed in Architecture Description Languages. AADL system specifications can be imported in the newly defined COTRE language. A system specification in COTRE language can be transformed into timed automata, Time Petri nets or other analytical models. Also, a transformation from AADL models to Colored Petri Nets, aiming at formally verifying certain properties through model checking, is presented in [15]. However, as far as we are aware of, published work does not address generation of dependability-oriented quantitative evaluation models from AADL specifications.

Considering the problem of generating dependability evaluation models from model-driven engineering approaches in a more general context, a significant amount of research has been carried out based on UML (Unified Modeling Language) [16]. For example, the European project HIDE ([17], [18]) proposed a method to automatically analyze and evaluate dependability attributes from UML models. It defined several model transformations from subsets of UML diagrams to i) GSPNs, Deterministic and Stochastic Petri Nets and Stochastic Reward Nets to evaluate dependability measures, ii) Kripke structures for formal verification and iii) to Stochastic Reward Nets for performance analysis. Also, [19] proposes an algorithm to synthesize dynamic fault trees (DFT) from UML system models. Other interesting approaches have been developed, aiming at obtaining performance measures by transforming UML diagrams (activity diagrams in [20], sequence and statechart diagrams in [21]) into GSPNs.

Similarly to UML users, the AADL users are interested in using modeling approaches allowing them to derive dependability evaluation models from AADL specifications. The approach proposed here aims at fulfilling this objective.

3 AADL Concepts

In the AADL, systems are particular composite components modeled as hierarchical collections of interacting application components (processes, threads, subprograms, data) and a set of execution platform components (processors, memory, buses, devices). The application components are bound to the execution platform. The AADL allows analyzing the impact of different architecture choices (such as scheduling policy or redundancy scheme) on a system's properties [22].

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` ports that represent access points to the component. One or more component implementations may be associated with the same component type, corresponding to different implementation structures of the component in terms of subcomponents, connections (between subcomponents' ports) and operational modes.

Dynamic aspects of system architectures are captured with the AADL operational mode concept. Different operational modes of a system or a system component represent different system configurations and connection topologies, as well as different sets of property values to represent changes in non-functional characteristics such as performance and reliability. Mode transitions model dynamic operational behavior and are triggered by events arriving through ports. Operational modes may represent fault-tolerance modes or different phases in a phased-mission system. This dynamics may influence dependability measures (i.e., availability), thus operational modes are taken into account in the dependability model.

An *AADL architectural model* can be annotated with dependability-related information (such as faults, failure modes, repair policies, error propagation, etc.) through the standardized Error Model Annex. *AADL error models* are defined in libraries and can be associated with application components, execution platform components, and device components, as well as the connections 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 for error events and error propagations declared in the error model.

In the same way as for AADL components, error models have 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 (internal to the component) and error propagations³. Occurrence properties specify the arrival rate or the occurrence probability of events and propagations. The error model

³ We will refer to error states, error events, error propagations and error transitions without the qualifying term `error` in contexts where the meaning is unambiguous (note that `error states` can model error-free states, `error events` can model repair events and `error propagations` can model all kinds of notifications).

implementation declares error transitions between states, triggered by events and propagations declared in the error model type.

Figure 1 shows a simple error model, without propagations, considering two types of faults: temporary and permanent. A temporary fault leads the component in an erroneous state while a permanent fault leads it in a failed state. A temporary fault can be processed and the component recovers regaining its error free state. A permanent fault requires restarting the component.

Error Model Type [independent]
<pre> error model independent features Error_Free: initial error state; Erroneous: error state; Failed: error state; Temp_Fault: error event {Occurrence => poisson λ_1}; Perm_Fault: error event {Occurrence => poisson λ_2}; Restart: error event {Occurrence => poisson μ_1}; Recover: error event {Occurrence => poisson μ_2}; end independent; </pre>
Error Model Implementation [independent.general]
<pre> error model implementation independent.general transitions Error_Free-[Perm_Fault]->Failed; Error_Free-[Temp_Fault]->Erroneous; Failed-[Restart]->Error_Free; Erroneous-[Recover]->Error_Free; end independent.general; </pre>

Fig. 1. Error model example without propagations

Interactions between the error models of different components are determined by interactions between components of the architectural model through connections and bindings. Out propagations are sent out of a component through all features connecting it to other components. Thus, out propagations have an impact on any receiving component that declares an in propagation with the same name. In some cases, it is desirable to model how error propagations from multiple sources are handled. This is modeled by further customizing an error model to a system component by specifying filters and masking conditions for propagations by using Guard properties associated with its features.

AADL allows modeling logical error states independently from the operational modes of a component. It also allows establishing a connection between the logical error states and the operational modes. For example, operational mode transitions may be constrained, through the use of Guard_Transition properties applied to ports, to occur depending on the error state configuration of several components.

Several examples are given throughout the paper to illustrate how propagations and Guard_Transition properties are handled in AADL.

4 The Modeling Framework

For complex systems, the main difficulty for dependability model construction arises from dependencies between the system components. Dependencies are of several types, identified in [4]: structural, functional, those related to the fault-tolerance and those associated with recovery and maintenance policies. Exchange of data or transfer of intermediate results from one component to another is an example of functional dependency. The fact that a thread runs on a processor induces a structural dependency between them. Changing the operational mode of a component according to a fault tolerance policy (e.g., leader/follower) represents a fault tolerance dependency. Sharing a maintenance facility between several execution platform components leads to a maintenance dependency. Having to follow a strict recovery order for application components is an example of recovery dependency. Functional, structural and fault tolerance dependencies are grouped into an architecture-based dependency class, as they are triggered by physical or logical connections between the dependent components at architectural level. On the other hand, recovery and maintenance dependencies are not always visible at architectural level.

A structured approach is necessary to model dependencies in a systematic way, to avoid errors in the resulting model of the system and to facilitate its validation. In our approach, the AADL dependability-oriented model is built in an iterative way. More concretely, in the first iteration, we build the model of the system's components, representing their behavior in the presence of their own faults and repair events only. They are thus modeled as if they were isolated from their environment. In the following iterations, we introduce dependencies in an incremental manner.

The rest of this section is structured as follows. A general overview of our modeling framework is presented in subsection 4.1. In subsection 4.2, we illustrate how dependencies are modeled in AADL in the context of our approach. Subsection 4.3 presents briefly how a GSPN model is generated from the AADL model.

4.1 Overview

An overview of our iterative modeling framework, which is decomposed in four main steps, is presented in Figure 2.

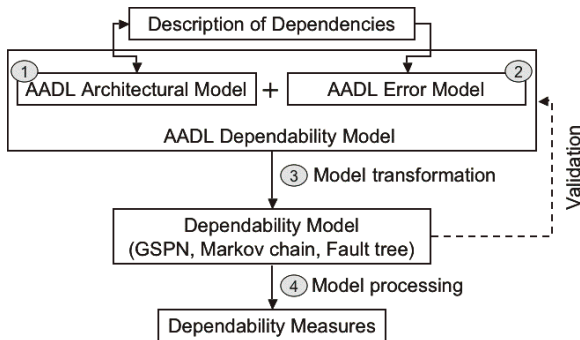


Fig. 2. Modeling framework

The *first step* is devoted to the modeling of the system architecture in AADL (in terms of components and operational modes of these components). This AADL architectural model may be available if it has been already built for other purposes.

The *second step* concerns the building of the AADL error models associated with components of the architectural model. The error model of the system is a composition of the set of components' error models, taking into account the dependencies between these components.

The description of architecture-based dependencies between components of the system is based on the analysis of the connections and bindings present in the architectural model. The corresponding error model is built based on the description of dependencies. Making maintenance and recovery assumptions may lead to the addition of components in the architectural model.

The architectural model and the error model of the system form a dependability-oriented AADL model, referred to as the *AADL dependability model* further on.

The *third step* aims at building a dependability evaluation model, from the AADL dependability model, based on model transformation rules. Here, we focus on generating a GSPN from the AADL model.

The *fourth step* is devoted to the processing of the dependability evaluation model (in our case under the form of a GSPN) to evaluate quantitative measures characterizing dependability attributes. This step is entirely based on existing GSPN processing algorithms and tools. Therefore, it is not considered here.

To obtain the AADL dependability model, the user must perform the first and second steps described above. The third step is intended to be automatic in order to hide the complexity of the GSPN to the user.

The iterative approach can be applied to the first two steps only or to the first three steps together. In both cases, the AADL dependability model is updated at each iteration. Modeling a dependency may either require to only add information in the model or to modify the existing model and to add new information (i.e., states and propagations). In the latter case, the AADL dependability model can be validated against its specification, based on the analysis and validation of the GSPN model, after each iteration.

To evaluate dependability measures, the user must specify state classes for the overall system. For example, if the user wishes to evaluate reliability or availability, it is necessary to specify the system states that are to be considered as failed states. If in addition, the user wishes to evaluate safety, it is necessary to specify the failed system states that are considered as catastrophic. In AADL, state classes are declared by means of a *derived error model* for the overall system describing the states of a system as Boolean expressions referring to its subcomponents' states.

4.2 Modeling with Dependencies in AADL

Architecture-based dependencies can be derived from the AADL architectural model. To these dependencies one has to add recovery and maintenance dependencies. The full set of dependencies can be summarized in a *dependency block diagram* to provide a global view of the system components and interactions. In the dependency block diagram, each component and each dependency are represented as distinct blocks. Blocks are connected through arcs. Their directions identify the directions of

dependencies. This diagram and the AADL architectural model are used to build the AADL error model progressively. Once the AADL error models of the components are built, the dependencies are added gradually. The order for introducing dependencies does not impact the final AADL dependability model. However, it may impact the reusability of parts of the model. Thus, the order may be chosen according to the context of the targeted analysis. Generally, fault tolerance and maintenance dependencies are modeled at the end, as their description strongly depends on the architecture.

It is noteworthy that not all the details of the architectural model are necessary for the AADL dependability model. Only components that have associated error models and all connections and bindings between them are necessary.

The rest of this subsection presents guidelines for modeling i) an architecture-based dependency and ii) a maintenance or recovery dependency.

4.2.1 Architecture-Based Dependency Modeling

The architecture-based dependency is supported by the architectural model and must be modeled in the error models associated with dependent components, by specifying respectively outgoing and incoming propagations and their impact on the corresponding error model. An example is shown in Figure 3. Figure 3-a presents the AADL architectural model (*Component 1* sends data to *Component 2*). Figure 3-b shows the corresponding dependency block diagram (the behavior of *Component 2* depends on that of *Component 1*). Figure 3-c presents the AADL dependability model where an error model is associated with each component to describe the dependency.

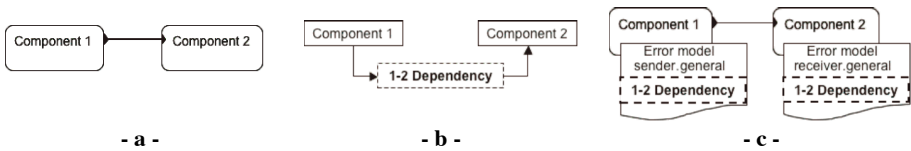


Fig. 3. Architecture-based dependency

The error model of Figure 4 is associated with *Component 1*. It takes into account the sender-side dependency from *Component 1* to *Component 2*. This error model is an extension of the one of Figure 1 that represents the behavior of a component as if it were isolated. The error model of Figure 4 declares an *out* propagation *Error* (see line d1) in the type and an AADL transition triggered by the *out* propagation in the implementation (see line d2).

The error model *receiver.general* associated with *Component 2* is not shown here but is similar. The only difference is the direction of the propagation *Error*. This *in* propagation triggers a state transition from *Error_Free* to *Failed*.

When *Component 1* is in the erroneous state, it sends a propagation through the unidirectional connection. As a consequence, the incoming propagation *Error* causes the failure of the receiving component *Component 2*. The *in* - *out* propagations *Error* defined respectively in the error model instance associated with *Component 2* and with *Component 1* have identical names. In the rest of the paper, such propagations are referred to as *name matching propagations*.

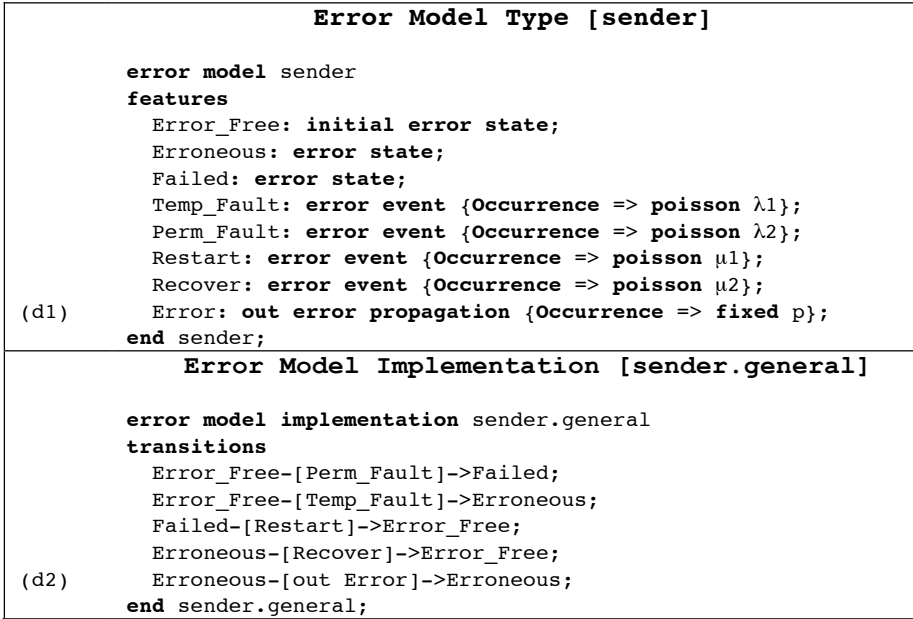


Fig. 4. Error model example with dependency

In real applications, architecture-based dependencies usually require describing how error propagations from multiple sources are handled by the receiver component. This is achieved by using `Guard` properties in which Boolean expressions are used to specify the consequences of a set of propagations occurring in a set of sender components on a receiver component.

4.2.2 Maintenance and Recovery Dependency Modeling

Maintenance dependencies need to be described when repair facilities are shared between components or when the maintenance or repair activity of some components has to be carried out according to a given order or a specified strategy.

Components that are not dependent at architectural level may become dependent due to the fact that they share maintenance facilities or to the synchronization of the maintenance activities. Thus, the architectural model might need some adjustments to support the description of dependencies related to the maintenance policy. As error models interact only via propagations through architectural features (i.e., connections, bindings), the maintenance dependency between components' error models must also be supported by the architectural model. This means that besides the system architecture components, we may need to add a component representing the shared repair facilities to model the maintenance dependencies. Figure 5-a shows an architectural model example where *Component 3* and *Component 4* do not interact (there is no architecture-based dependency between them). However, if we assume that they share one repairman, it is necessary to represent the repairman at the level of the architectural model, as shown in Figure 5-b in order to model the maintenance dependency between these components.

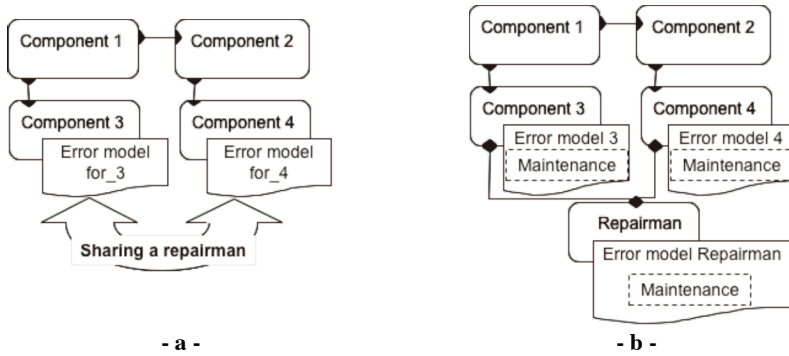


Fig. 5. Maintenance dependency

Also, the error models of dependent components with regards to their recovery might need some adjustments. For example, to represent the fact that *Component 3* can only restart if *Component 4* is running, one needs to distinguish between a failed state of *Component 3* and a failed state where *Component 3* is allowed to restart.

4.3 AADL to GSPN Model Transformation

The GSPN model of the system is built from the transformation of the AADL dependability model following a modular approach and taking into account the dependency block diagram.

The GSPN of the global system is structured as a set of interacting subnets, where a subnet is associated with a component or a dependency block identified in the dependency block diagram. Two types of GSPN subnets are distinguished: 1) a *component subnet* is associated with each component and describes the component's behavior in the presence of its own faults and repair events; and 2) a *dependency subnet* models the behavior associated with the corresponding dependency. In the AADL dependability model, each dependency is modeled as part of each of the error models involved in the dependency. GSPN dependency subnets are obtained from information concerning a particular dependency existing in (at least) two dependent error models. The global GSPN contains one subnet for the behavior of each component in the presence of its own faults and repair events, and one subnet for each dependency between components. It has the same structure as the dependency block diagram. The modular structure of the GSPN allows the user to validate the model progressively; as the GSPN is enriched with a subnet each time a new dependency is added in the error model of the system. So, if validation problems arise at GSPN level during iteration i , only the part of the current error model corresponding to iteration i is questioned.

5 Transformation Rules

In the next three subsections we present successively AADL to GSPN transformation rules for 1) isolated components, 2) name matching in – out propagations in







dependent components and 3) systems with operational modes necessary to describe fault tolerance dependencies. All transformation rules are defined to ensure that the obtained GSPN is correct by construction: bounded, live and reversible. They are aimed to be systematic in order to prepare the transformation automation. Also, the resulting GSPN is tool-independent, i.e., we do not use tool-specific features or predicates. It is worth noting that this section only presents a small set of transformation rules. A more complete set is presented in [23].

5.1 Isolated Components

In the case of an isolated component or in the case of a set of independent components, the AADL to GSPN transformation is rather straightforward, as an error model represents a stochastic automaton. The number of tokens in a component subnet is always one, as a component can only be in one state.

Table 1 shows the basic transformation rules.

Table 1. Basic AADL error model to GSPN transformation rules

AADL error model element	GSPN element	
State	Place	
Initial state	Token in the corresponding place	
Event	GSPN transition (timed or immediate)	
Occurrence property of an event	Distribution or probability characterizing the occurrence of associated GSPN transition	 Timed
		 Immediate
AADL transition (Source_State-[Event] -> Destination_State)	Arcs connecting places (corresponding to AADL Source_State and Destination_State) via GSPN transition (corresponding to AADL Event)	

By applying the transformation rules presented in Table 1 to the error model shown in Figure 1, we obtain the GSPN of Figure 6.

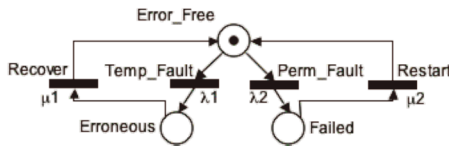


Fig. 6. GSPN corresponding to the error model of Figure 1

5.2 Transforming in - out Name Matching Propagations

In the most general case, an out propagation declared in a propagation sender error model could trigger n AADL transitions in this same error model (e.g., a *Failed* propagation could be propagated out both from a *FailStopped* and a *FailRandom* states). Name matching in propagations could be declared in $r \geq 2$ propagation receiver error models and trigger m_j AADL transitions in each j ($j = 1 \dots r$) receiver error model. We identified and analyzed several transformation rules for the same AADL specification of in - out name matching propagations. Some of the rules are convenient when an out propagation has only one receiver. On the other hand, these rules are hard to automate in case there are several receivers (i.e., the in propagation is declared in several components' error models) for the same out propagation. Also, the choice of a transformation rule for in - out name matching propagations impacts the transformation rules for systems with operational modes. The transformation rule for in - out name matching propagations we present here is very well adapted for the case where an out propagation has several receivers. It also simplifies the definition of the transformation rule for systems with operational modes. We first present an example of a pair of in - out name matching propagations declared in two connected components in Figure 7. Then we illustrate the chosen transformation rule on this example.

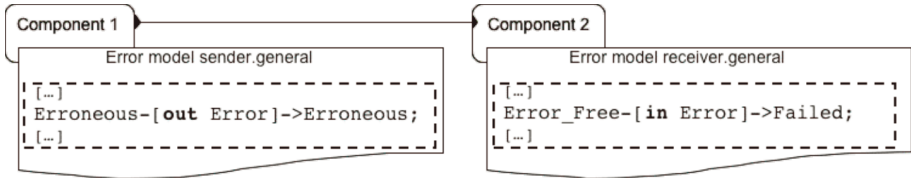


Fig. 7. Sender and Receiver – name matching propagations

In Figure 7, *Component 1* plays the role of the propagation sender and it sends propagations named *Error* through the connection that arrives at *Component 2*. *Component 2* plays the role of a receiver. If it receives a propagation named *Error*, it moves from *Error_Free* to *Failed* state.

The transformation rule consists in decoupling the in and out propagations in the GSPN through an intermediary place that represents the fact that the out propagation *Error* occurred, as shown in Figure 8 (*InOut_Error* place). A token arrives in the *InOut_Error* place when a GSPN transition (*Out_Error*) corresponding to the out propagation (and characterized by its Occurrence property) occurs. The existence of a token in the *InOut_Error* place leads to the firing of an immediate GSPN transition *In_Error* (if the place *Error_Free* in *Component 2* is marked) that corresponds to the in propagation. The intermediary place is emptied when the place corresponding to the source of the out propagation is empty and the GSPN transition corresponding to the in propagation is not enabled. We do not empty this place at the occurrence of the GSPN transition corresponding to the in propagation, as we need to memorize

the occurrence of the `out` propagation until all effects (immediate GSPN transitions) of the propagation occur. This memory is used in other transformation rules.

The GSPN place `NoPropag` and the associated immediate transitions with probability $(1-p)$ and 1 respectively model the situation where the propagation does not occur when `Component 1` is in an `Erroneous` state. If the probability of occurrence of the `out` propagation is equal to 1, then this subnet is not necessary.

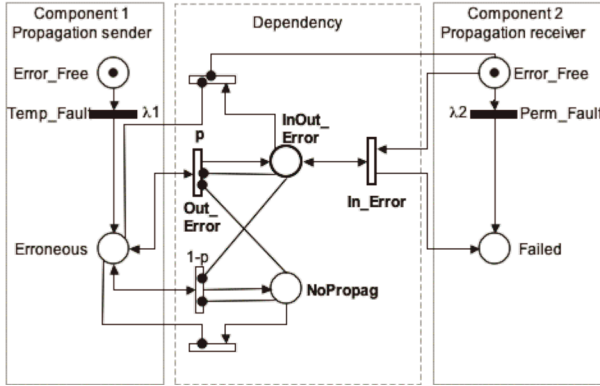


Fig. 8. Propagation from sender to receiver - transformation rule

In the general case of n AADL transitions triggered by an `out` propagation, with name matching `in` propagations in several receiver error models, one GSPN transition is created for each AADL transition triggered by the `out` propagation in the sender error model. Also, one intermediary place is created for each `out` propagation. One GSPN transition is created for each AADL transition triggered by the `in` propagation in the receiver error models. Consequently, the number of GSPN transitions (N_{tr}) describing the AADL propagation is given by:

$$N_{tr} = 4 * n + n * \sum_{j=1}^r m_j, \forall r \geq 1 \tag{1}$$

- where $n =$ the number of AADL transitions triggered by the `out` propagation in the sender error model;
- $r =$ the number of receiver error models;
- $m_j =$ the number of AADL transitions triggered by the `in` propagation in the receiver error model j .

The first term of equation (1) represents the number of GSPN transitions that model the `out` propagation, i.e., 4 GSPN transitions for each one of the n `out` propagations. The second term represents the number of GSPN transitions that model the `in` propagation, i.e., one GSPN transition for each pair of `in` - `out` propagations.

Figure 9-a shows an example of AADL dependability model with one sender and two receivers. It is transformed into the GSPN of Figure 9-b.

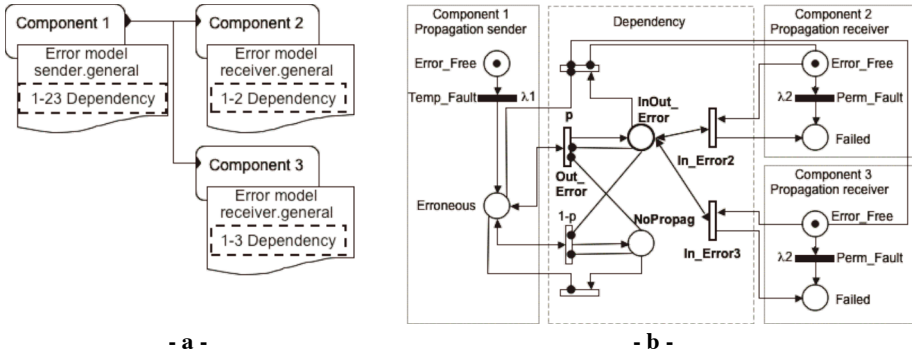


Fig. 9. Propagation from sender to two receivers

Naturally, when transforming large AADL models formed of many components, the size of the corresponding GSPN increases. The state space size depends on the number of components and on the dependencies between them. We have analyzed the state space for GSPNs obtained using our transformation rules from AADL models with several dependent components. Indeed, the more independent or loosely coupled components are, the larger the state space gets. To address this problem, GSPN reduction methods, such as those mentioned in [24], may be efficiently used before processing it to obtain the underlying Markov chain.

5.3 Systems with Operational Modes

In AADL, there are several mechanisms for connecting logical error states and operational mode transitions. For space limitation reasons, in this section, we focus on the AADL to GSPN transformation rules for *Guard_Transition* properties, which allow constraining a mode transition to occur depending on the error state configuration of several components of a system. The rest of this subsection presents successively the AADL modeling of an example of a system with operational modes, using *Guard_Transition* properties, and illustrates the proposed transformation rule on this example.

5.3.1 AADL Dependability Modeling of *Guard_Transition* Properties

We first present an example of a modal AADL system in Figure 10 and we show in Figure 11 the association of a *Guard_Transition* property with the ports involved in mode transitions. The error state configuration necessary to allow a mode transition is expressed as a Boolean expression referring to error states and propagations.

In Figure 10, the system is represented using the AADL graphical notation. It contains two identical active components and two operational modes (*Comp1Primary* and *Comp1Backup*). The system is initially in mode *Comp1Primary*. The transition from mode *Comp1Primary* to mode *Comp1Backup* occurs when propagations arrive through the ports *Send2* of *Comp1* and *Send1* of *Comp2*. In this case, the second

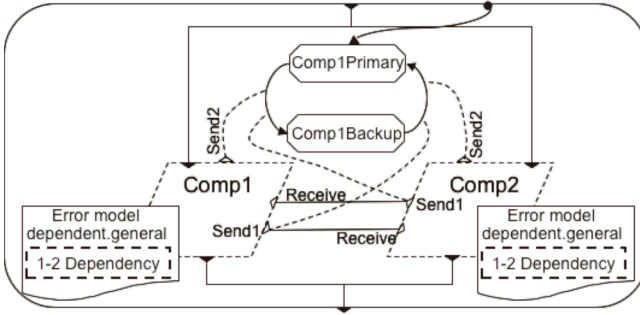


Fig. 10. Example architectural model for a system with operational modes

```

thread Comp
features
  Send1, Send2: out event port;
  Receive: in event port;
end Comp;

thread implementation Comp.generic
annex Error_Model {**
  Model => dependent.general;
**};
end Comp.generic;

system SystemLevelModes
end SystemLevelModes;

system implementation SystemLevelModes.generic
modes
  Comp1Primary: initial mode;
  Comp1Backup: mode;
  Comp1Primary-[Comp1.Send2, Comp2.Send1]->Comp1Backup;
  Comp1Backup-[Comp1.Send1, Comp2.Send2]->Comp1Primary;
subcomponents
  Comp1: system Comp.generic;
  Comp2: system Comp.generic;
connections
  event port Comp1.Send1->Comp2.Receive;
  event port Comp2.Send1->Comp1.Receive;
annex Error_Model {**
(g1)   Guard Transition =>
(g2)   (Comp1.Send2[FailedVisible] and Comp2.Send1[Error_Free])
(g3)   applies to Comp1.Send2, Comp2.Send1;
(g4)   Guard Transition =>
(g5)   (Comp2.Send2[FailedVisible] and Comp1.Send1[Error_Free])
(g6)   applies to Comp1.Send1, Comp2.Send2;
**};
end SystemLevelModes.generic;

```

Fig. 11. Guard_Transition property associations

component must take over and provide the service. The transition from mode *Comp1Backup* to mode *Comp1Primary* occurs when propagations arrive through the ports *Send2* of *Comp2* and *Send1* of *Comp1*. The same error model is associated with both *Comp1* and *Comp2*. It is based on the error model for isolated components (see Figure 1). It declares in addition an out propagation *FailedVisible*, which notifies the failure of the component and which is used in the *Guard_Transition* properties.

Guard_Transition properties are associated with the ports involved in mode transitions. A mode transition occurs only if the *Guard_Transition* property associated with the port named in it evaluates to TRUE. In our example, the mode transition from mode *Comp1Primary* to *Comp1Backup* occurs when *Comp1* sends the *FailedVisible* out propagation while *Comp2* is *Error_Free* (see lines g1-g3 of Figure 11). The complementary condition must hold for the occurrence of the transition from *Comp1Backup* to *Comp1Primary* (see lines g4-g6 of Figure 11).

5.3.2 Transforming *Guard_Transition* Properties

We illustrate the transformation rule on the example of a system with operational modes described in Figure 10 and Figure 11. Then, we discuss the use of this rule.

Figure 12 shows the GSPN corresponding to the first *Guard_Transition* property (lines g1-g3) of Figure 11. The GSPN models of *Comp1* and *Comp2* are incomplete in this figure.

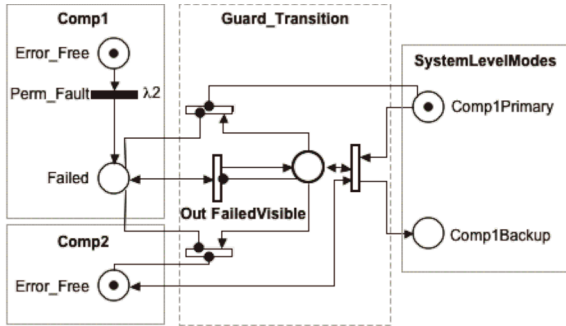


Fig. 12. GSPN modeling of the *Guard_Transition* property

Operational modes are directly mapped to Petri net places.

The transformation rule assumes that the Boolean expression of the *Guard_Transition* property is in disjunctive normal form. If it is not the case, the Boolean expression must first be transformed into disjunctive normal form. Each conjunction (referring to states and/or propagations) is transformed into an immediate GSPN transition connected with:

- places corresponding to the states and out propagations referred to in the AND expression via bi-directional arcs or inhibitor arcs (depending whether there are negations in the Boolean expression or not).
- places corresponding to operational modes referred to in the mode transition triggered by the port having the *Guard_Transition* property.

If, in our example above, the Boolean expression in disjunctive normal form were formed of several conjunctions, then several GSPN transitions would be connected to the places *Comp1Primary* and *Comp1Backup*.

An intermediary place corresponding to an out propagation is emptied when the place corresponding to the source of the out propagation is empty and the GSPN transitions corresponding to *Guard_Transition* conjunctions are not enabled.

If an out propagation name-matches an in propagation in a receiver component and is referred to in a *Guard_Transition* property declared in another receiver component, the same intermediary place is used both for the name matching GSPN subnet and for the *Guard_Transition* subnet. The intermediary place is emptied when both emptying conditions related to the name-matching propagations rule and to the *Guard_Transition* rule are true. An example is shown in Figure 13. We consider the system presented in Figure 9 contains a third component, *Comp3*, and that *Comp1* is connected to it. The error model associated with the newly introduced *Comp3* declares an in propagation *FailedVisible*. The *Guard_Transition* is transformed as above and the name matching propagation of *Comp1* and *Comp3* reuses the intermediary place representing the occurrence of the *FailedVisible* propagation.

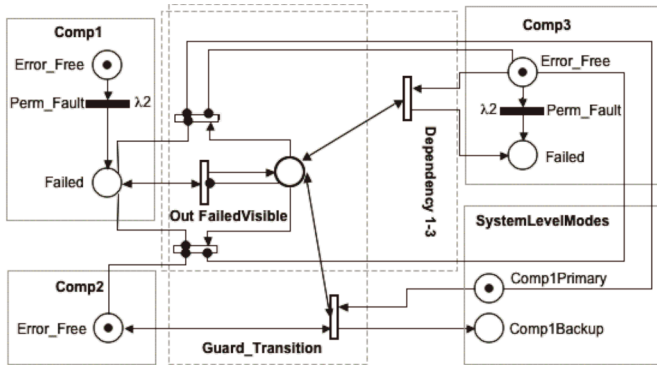


Fig. 13. GSPN modeling of the *Guard_Transition* property taking into account a name matching propagation

6 Case Study

In this section we show how our modeling framework can be used to compare the availability of two candidate architectures for a subsystem of the French Air Traffic Control System. This subsystem is designed to achieve high levels of service availability and is further detailed in [25]⁴.

In the rest of this section, we first present the AADL architectural models of the two candidate architectures in subsection 6.1. The dependency analysis based on these models is presented in subsection 6.2. Subsection 6.3 details the error models

⁴ The AADL modeling of the subsystem and the AADL to GSPN model transformation are not presented in the paper cited here.

describing some of these dependencies. Subsection 6.4 deals with the AADL to GSPN transformation while subsection 6.5 presents an example of dependability evaluation for the two candidate architectures.

6.1 AADL Architectural Models

The subsystem we consider here is formed of two fault-tolerant distributed software units that are in charge of processing flight plans (FPunit) and radar data (RDunit). Two processors can host these units. We consider two candidate architectures for this subsystem, referred to as *Configuration1* and *Configuration2*. Figure 14 presents both candidate architectures using the AADL graphical notation.

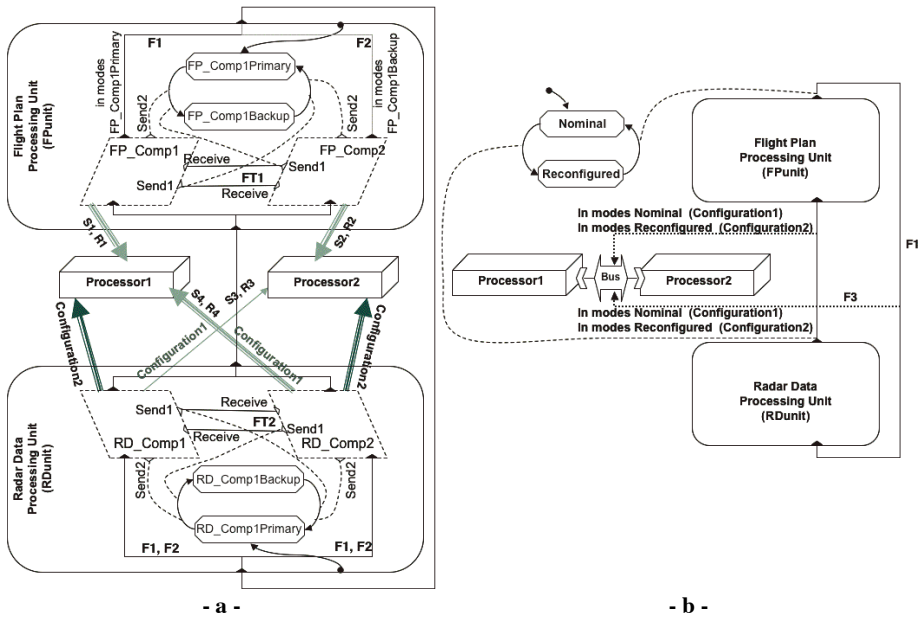


Fig. 14. AADL architectural model of Air Traffic Control System candidate architectures

The FPunit and the RDunit have the same structure (presented in Figure 10), i.e., they are formed of two replicas (threads): one having the primary role (provides the service) while the other one has a backup role (monitors the primary). Both candidate architectures use two processors. The two replicas of each software unit are bound to separate processors. In *Configuration1*, the initially primary replicas of the FPunit and RDunit (*FP_Comp1* and *RD_Comp1*) are bound to separate processors (*FP_Comp1* bound to *Processor1* and *RD_Comp1* bound to *Processor2*). In *Configuration2*, the initially primary replicas of the FPunit and RDunit are bound to the same processor, *Processor1*. The whole subsystem has two operational modes: *Nominal* and *Reconfigured*. Connections between replicas bound to separate processors are bound to a bus. Thus, the connection bindings to the bus depend on the operational mode of the subsystem. A bus failure causes the failure of the RDunit replica. The primary

replica of the FPunit exchanges data with both replicas of the RDunit. For the sake of clarity, we show the thread binding configurations in Figure 14-a and the bus and the connection bindings to the bus separately in Figure 14-b.

6.2 Dependency Analysis

The various interactions between this subsystem's components induce dependencies between them. Most of them are architecture-based, thus they are visible on the architectural model. We took into account the following dependencies:

- structural dependency between each processor and the threads that run on top of it. We assume that hardware faults can propagate and influence the software running on top of it. These dependencies (S1, S2, S3 and S4 in Figure 14) are supported by the architectural bindings of threads to processors.
- recovery dependency between each processor and the threads that run on top of it. If a thread fails, it cannot be restarted if the processor on top of which it runs is in a failed state. These dependencies (R1, R2, R3 and R4 in Figure 14) are supported by the architectural bindings of threads to processors.
- maintenance dependency between the two processors that share a repairman that is not simultaneously available for the two components. This maintenance dependency is not visible on the architectural model of Figure 14.
- fault tolerance dependency between the two RDunit threads and the two FPunit threads. If the replica that delivers the service fails but the other one is error free, the two software replicas switch roles. Then, the failed replica is restarted. These dependencies (FT1 and FT2 in Figure 14) are supported by the connections between the replicas of each software unit.
- structural dependency between the bus and the threads of the RDunit. If the bus fails, the broken connections bound to it make the RDunit fail in mode *Nominal* of *Configuration1* and in mode *Reconfigured* of *Configuration2*. This dependency (F3 in Figure 14) is supported by the binding of the connection from the FPunit to the RDunit to the bus.
- functional dependencies between the FPunit and the RDunit. The active FPunit thread may propagate errors to both RDunit threads. These dependencies (F1 and F2 in Figure 14) are supported by the connections of the FPunit replicas to the RDunit replicas. Note that we consider that RDunit errors do not propagate to the FPunit even though there is a connection from the RDunit to the FPunit.

Figure 15 shows the dependency block diagram describing the dependencies between components of the *Configuration1* of the Air Traffic Control System. We built the AADL dependability model iteratively, by integrating first the structural and functional dependencies and then the maintenance, recovery and fault tolerance dependencies. Due to space limitations, we further focus on the two grey-color blocks, which represent the functional dependency between a FPunit replica and both RDunit replicas and the fault tolerance dependency between the FPunit replicas.

The dependency block diagram for *Configuration2* is similar. In *Configuration2*, *Processor1* is linked to *RD_Comp1* via structural and maintenance dependency blocks and *Processor2* is linked to *RD_Comp2* via structural and maintenance

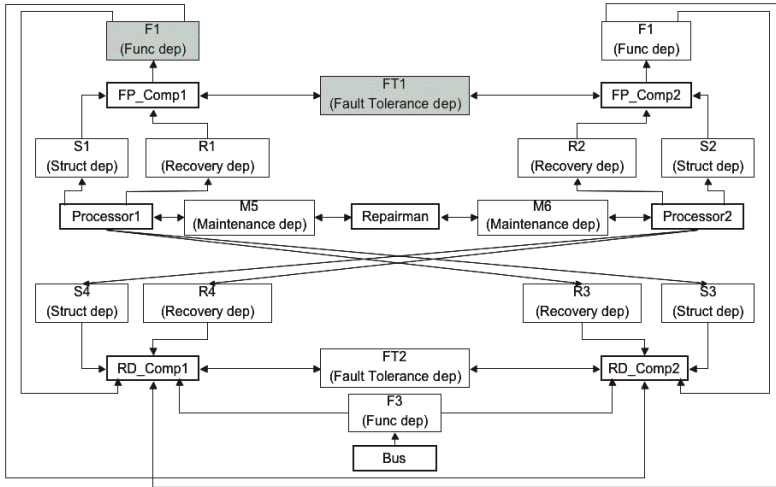


Fig. 15. Description of dependencies between components of *Configuration1* of the Air Traffic Control System

dependency blocks. The functional dependency block (F3) between the bus and the *RD_Comp1* and *RD_Comp2* is internally different (modes are taken into account differently in the two configurations).

6.3 AADL Error Models

We first describe in detail the functional dependency between a FPunit replica and the RDunit replicas and the fault tolerance dependency between the FPunit replicas. Then, we present the corresponding error models.

- **Functional dependency between a FPunit thread and the RDunit threads.** An error propagated from the primary FPunit replica (*FP_Comp1* or *FP_Comp2*) to both RDunit replicas (*RD_Comp1* and *RD_Comp2*) will cause their failure. The FPunit replica will recover without need of restarting. Note that the RDunit replicas do not propagate errors to the FPunit replicas. Also, an error propagated from a FPunit replica does not impact the other FPunit replica. This means that we cannot use the same error model both for FPunit replicas and RDunit replicas. The error model associated with the FPunit replicas must declare an *out Error* propagation without declaring an *in Error* propagation. The error model associated with the RDunit replicas must declare an *in Error* propagation that matches the *out* propagation declared in the error model associated with the FPunit replicas.
- **Fault tolerance dependency between the FPunit threads.** The behavior we intend to model is based on the specification presented in Section 5.3 (for systems with operational modes). In addition to the takeover by the backup replica when the primary replica fails, we add the following assumption. If both components fail one after the other, the first one restarted provides the service

and the FPunit goes to the corresponding mode. To model this behavior, we associate error models with *FP_Comp1* and *FP_Comp2* and we use *Guard_Transition* properties on the out ports *Send* of both components. These *Guard_Transition* properties are extensions of those presented in Figure 11. The behavior in the case of a double failure requires including the notification of the end of the restart procedure before moving to *Error_Free* state.

Figure 16 presents the error model associated with the FPunit threads. Lines f1-f2 correspond to the functional dependency presented above while lines t1-t4 correspond to the fault tolerance dependency. The rest of the error model is similar to the one presented in Figure 1 for isolated components. The component may propagate errors (out propagation *Error*) but it cannot be influenced by *Error* propagations, as it does not declare an in propagation *Error*. The end of the restart procedure is notified (*IAmRestarted* out propagation) before moving to *Error_Free* state.

The only difference between the error model associated with the FPunit threads and the RDunit threads is the direction of the propagation *Error* and the AADL transition triggered by it. In the error model associated with the RDunit threads, *Error* is an in propagation triggering an AADL transition from *Error_Free* to *Failed*.

Error Model Type [forFP_Comp]	
<pre> error model forFP_Comp features Error_Free: initial error state; Erroneous: error state; Restarted: error state; Failed: error state; Temp_Fault: error event {Occurrence => poisson λ1}; Perm_Fault: error event {Occurrence => poisson λ2}; Restart: error event {Occurrence => poisson μ1}; Recover: error event {Occurrence => poisson μ2}; (f1) Error: out error propagation {Occurrence => fixed p}; (t1) FailedVisible: out error propagation {Occurrence=>fixed 1}; (t2) IAmRestarted: out error propagation {Occurrence=> fixed 1}; end forFP_Comp; </pre>	
Error Model Implementation [forFP_Comp.general]	
<pre> error model implementation forFP_Comp.general transitions Error_Free-[Perm_Fault]->Failed; Error_Free-[Temp_Fault]->Erroneous; Failed-[Restart]->Restarted; (f2) Erroneous-[out Error]->Erroneous; (t3) Restarted-[out IAmRestarted]->Error_Free; (t4) Failed-[out FailedVisible]->Failed; Erroneous-[Recover]->Error_Free; end forFP_Comp.general; </pre>	

Fig. 16. Error model forFP_Comp

Figure 17 presents the *Guard_Transition* properties that specify the conditions under which mode transition occur, according to the fault tolerance behavior described above. Mode transitions occur if one of the components sends the *FailedVisible* out propagation while the other one is *Error_Free* or if one of the components sends the *IAMRestarted* out propagation while the other component is not *Error_Free* (meaning that a double failure occurred and the first component has been restarted before the second one).

```

Guard_Transition =>
  (Comp1.Send2[FailedVisible] and Comp2.Send1[Error_Free])
  or (Comp2.Send1[IAMRestarted] and not Comp1.Send2[Error_Free])
  applies to Comp1.Send;
Guard_Transition =>
  (Comp2.Send2[FailedVisible] and Comp1.Send1[Error_Free])
  or (Comp1.Send1[IAMRestarted] and not Comp2.Send2[Error_Free])
  applies to Comp2.Send;

```

Fig. 17. *Guard_Transition* properties associated with Send ports of FPunit threads

6.4 AADL to GSPN Model Transformation

For these two dependencies, we use only the AADL to GSPN transformation rules presented in section 5. We first took into account the functional dependency from *FT_Comp1* to *RD_Comp1* and *RD_Comp2* and then the fault tolerance dependency between *FT_Comp1* and *FT_Comp2*. Before adding the fault tolerance dependency, the GSPN subnet FT1 did not exist and the *FP_Comp1* and *FP_Comp2* subnets were identical to the *RD_Comp1* and *RD_Comp2* subnets.

Figure 18 presents the part of the GSPN corresponding to the functional dependency between the *FP_Comp1* replica of the FPunit and the two replicas of the RDunit and to the fault tolerance dependency between the threads of the FPunit. For clarity reasons, immediate GSPN transition that empty intermediary places corresponding to out propagation are not shown.

6.5 Evaluation of Quantitative Measures

Figure 19 gives the unavailability of the two candidate architectures. Such quantitative measures are obtained from the processing of the GSPN derived from the AADL model. In Figure 19, the varying parameter is the occurrence rate of a bus failure; λ_c . $\lambda_c \leq 10^{-6}/h$ corresponds to a redundant bus. For *Configuration1*, the impact of this parameter is important when $\lambda_c \geq 10^{-5}/h$. *Configuration2* is much less influenced by λ_c , as in *Nominal* mode, the communication between the two units does not go through the bus. From a practical point of view, if $\lambda_c \geq 10^{-5}/h$, *Configuration2* is recommended. Otherwise the two candidate architectures are equivalent.

Other analyses can be carried out on the same model. The results of several analyses allow taking a decision about what candidate architecture best suites the application. For example, performance analyses can be performed to determine the impact of the choices made to achieve dependability goals on a system's performance.

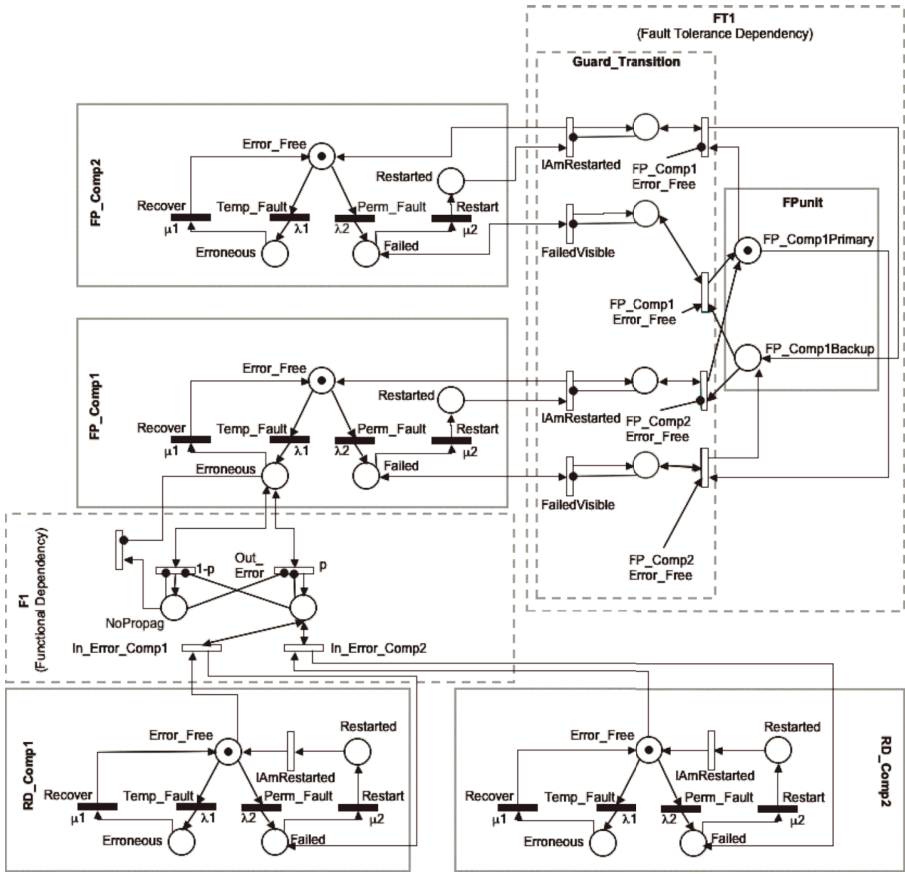


Fig. 18. GSPN model of the Air Traffic Control System – two dependencies

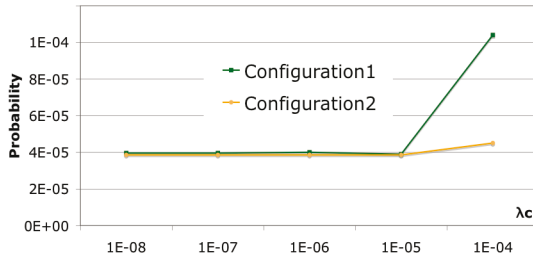


Fig. 19. Unavailability

7 Conclusion

We presented a stepwise approach for system dependability modeling using AADL and GSPNs. The aim of this approach is to hide the complexity of traditional analytical models to end-users acquainted with AADL. In this way, we ease the task of evaluating dependability measures. Our approach assists the user in the structured construction of the AADL dependability model that is transformed into a GSPN to be processed by existing tools. To support and trace model evolution, this approach proposes that the user builds the AADL dependability model iteratively. Components' behaviors in the presence of faults are modeled in the first iteration as if they were isolated. Then, each iteration introduces a new dependency between system's components in the AADL dependability model. The AADL to GSPN model transformation is meant to be transparent to the user. Thus, it is based on rigorous and systematic rules aimed at supporting tool-based transformation automation. The model transformation can be performed iteratively, each time the AADL dependability model is enriched. In this way, the GSPN model can be validated progressively (hence the corresponding AADL architecture and error models can be validated progressively and corrected accordingly, if required). Finally, we illustrated the proposed approach on a subsystem of the French Air Traffic Control System. We have shown the principles of the transformation and some of the rules. The work in progress concerns the implementation of a model transformation tool to be easily integrated into AADL and GSPN based tools.

Acknowledgements. This work is partially supported by 1) the European Commission (ASSERT European IP No. IST 004033 and ReSIST NoE No. IST 026764), 2) the European Social Fund and 3) Zonta International Foundation.

References

1. SAE-AS5506: SAE Architecture Analysis and Design Language (AADL), International Society of Automotive Engineers, Warrendale, PA, USA (November 2004)
2. SAE-AS5506/1: SAE Architecture Analysis and Design Language (AADL) Annex vol. 1, Annex E: Error Model Annex, International Society of Automotive Engineers, Warrendale, PA, USA (June 2006)
3. Bondavalli, A., Chiaradonna, S., Di Giandomenico, F., Mura, I.: Dependability Modeling and Evaluation of multiple-phased systems, using DEEM. *IEEE Transactions on Reliability* 53, 509–522 (2004)
4. Kanoun, K., Borrel, M.: Fault-tolerant systems dependability. Explicit modeling of hardware and software component-interactions. *IEEE Transactions on Reliability* 49, 363–376 (2000)
5. Bernardi, S., Bobbio, A., Donatelli, S.: Petri Nets and Dependability. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) *Lectures on Concurrency and Petri Nets*. LNCS, vol. 3098, pp. 125–179. Springer, Heidelberg (2004)
6. Farines, J.-M., et al.: The Cotre project: rigorous software development for real time systems in avionics. In: 27th IFAC/IFIP/IEEE Workshop on Real Time Programming, Zielona Góra, Poland (2003)
7. Singhoff, F., Legrand, J., Nana, L., Marcé, L.: Scheduling and Memory Requirements Analysis with AADL. In: *SIGAda Int. Conf. on Ada*, Atlanta, GE, USA (2005)

8. Béounes, C., et al.: Surf-2: a program for dependability evaluation of complex hardware and software systems. In: 23rd IEEE Int. Symposium on Fault Tolerant Computing, Toulouse, France, IEEE Computer Society Press, Los Alamitos (1993)
9. Deavours, D.D., et al.: The Mobius Framework and its Implementation. *IEEE Transactions on Software Engineering* 28, 956–969 (2002)
10. Hirel, C., Sahner, R., Zang, X., Trivedi, K.: Reliability and performability modeling using SHARPE 2000. In: 11th Int. Conf. on Computer Performance Evaluation: Modelling Techniques and Tools, Schaumburg, IL, USA (2000)
11. Bernardi, S., Bertinello, C., Donatelli, S., Franceschinis, G., Gaeta, R., Gribaudo, M., Horvath, A.: GreatSPN in the new millenium. In: Tool Session of 9th Int. Workshop on Petri Nets and Performance Models, Aachen, Germany (2001)
12. Ciardo, G., Trivedi, K.S.: SPNP: The Stochastic Petri Net Package (Version 3.1). In: 1st Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, San Diego, CA, USA (1993)
13. Rugina, A.E., Kanoun, K., Kaâniche, M.: An Architecture-based Dependability Modeling Framework using AADL. In: 10th IASTED Int. Conf. on Software Engineering and Applications, Dallas, USA (2006)
14. Rugina, A.E., Kanoun, K., Kaâniche, M.: Modélisation de la sûreté de fonctionnement à partir du langage AADL. In: 15ème Congrès de Maîtrise des Risques et de Sûreté de Fonctionnement, Lille, France (2006)
15. Hugues, J., Kordon, F., Pautet, L., Vergnaud, T.: A Factory To Design and Build Tailorable and Verifiable Middleware. In: Kordon, F., Sztipanovits, J. (eds.) Monterey Workshop 2005. LNCS, vol. 4322, pp. 121–142. Springer, Heidelberg (2007)
16. OMG: Unified Modelling Language Specification (October 2004), <http://www.omg.org>
17. Majzik, I., Bondavalli, A.: Automatic Dependability Modeling of Systems Described in UML. In: Int. Symposium on Software Reliability Engineering (1998)
18. Bondavalli, A., et al.: Dependability Analysis in the Early Phases of UML Based System Design. *Int. Journal of Computer Systems-Science&Engineering* 16, 265–275 (2001)
19. Pai, G.J., Bechta Dugan, J.: Automatic Synthesis of Dynamic Fault Trees from UML System Models. In: 13th Int. Symposium on Software Reliability Engineering, Annapolis, USA (2002)
20. López-Grao, J.P., Merseguer, J., Campos, J.: From UML Activity Diagrams To Stochastic Petri Nets: Application to Software Performance Engineering. In: 4th Int. Workshop on Software and Performance, Redwood City, CA, USA (2004)
21. Bernardi, S., Donatelli, S., Merseguer, J.: From UML Sequence Diagrams and Statecharts to Analysable Petri Net Models. In: 3rd Int. Workshop on Software and Performance, Rome, Italy (2002)
22. Feiler, P.H., Gluch, D.P., Hudak, J.J., Lewis, B.A.: Pattern-Based Analysis of an Embedded Real-time System Architecture. In: 18th IFIP World Computer Congress, ADL Workshop, Toulouse, France (2004)
23. Rugina, A.E., Kanoun, K., Kaâniche, M.: AADL-based Dependability Modelling, LAAS-CNRS Research Report n°06209 (April 2006)
24. Ajmone Marsan, M., et al.: Modelling With Generalized Stochastic Petri Nets. John Wiley & Sons, Chichester (1995)
25. Kanoun, K., Borrel, M., Morteveille, T., Peytavin, A.: Availability of CAUTRA, a Subset of the French Air Traffic Control System. *IEEE Transactions on Computers* 48, 528–535 (1999)